Sparse-TPU: Adapting Systolic Arrays for Sparse Matrices

Xin He, Subhankar Pal, Aporva Amarnath, Siying Feng, Dong-Hyeon Park, Austin Rovinski, Haojie Ye, Yuhan Chen, Ronald Dreslinski, Trevor Mudge
University of Michigan, Ann Arbor, MI

ABSTRACT

While systolic arrays are widely used for dense-matrix operations, they are seldom used for sparse-matrix operations. In this paper, we show how a systolic array of Multiply-and-Accumulate (MAC) units, similar to Google's Tensor Processing Unit (TPU), can be adapted to efficiently handle sparse matrices. TPU-like accelerators are built upon a 2D array of MAC units and have demonstrated high throughput and efficiency for dense matrix multiplication, which is a key kernel in machine learning algorithms and is the target of the TPU. In this work, we employ a co-designed approach of first developing a packing technique to condense a sparse matrix and then propose a systolic array based system, *Sparse-TPU*, abbreviated to STPU, to accommodate the matrix computations for the packed denser matrix counterparts. To demonstrate the efficacy of our co-designed approach, we evaluate sparse matrix-vector multiplication on a broad set of synthetic and real-world sparse matrices. Experimental results show that STPU delivers 16.08× higher performance while consuming 4.39× and 19.79× lower energy for integer (int8) and floating point (float32) implementations, respectively, over a TPU baseline. Meanwhile, STPU has 12.93% area overhead and an average of 4.14% increase in dynamic energy over the TPU baseline for the float32 implementation.

CCS CONCEPTS

• **Computer systems organization → Systolic arrays**.

KEYWORDS

Systolic array, sparse matrix processing, application-specific hardware, hardware-software codesign, hardware accelerators, sparse matrix condensing

ACM Reference Format:
Xin He, Subhankar Pal, Aporva Amarnath, Siying Feng, Dong-Hyeon Park, Austin Rovinski, Haojie Ye, Yuhan Chen, Ronald Dreslinski, Trevor Mudge. 2020. Sparse-TPU: Adapting Systolic Arrays for Sparse Matrices. In *2020 International Conference on Supercomputing (ICS '20), June 29-July 2, 2020, Barcelona, Spain.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3392717.3392751

1 INTRODUCTION

The slowing down of cost-effective scaling in integrated circuits has prompted computer architects to turn to accelerators to deliver improvements in performance and power efficiency [2]. However, this approach, if followed blindly, will ultimately hit the "accelerator wall", where simply adding accelerators is not a feasible solution [9]. The choice and design of accelerators needs to balance power, performance, and area, against the expected usage.

Two-dimensional (2D) systolic arrays have been proposed for energy-efficient execution of dense matrix operations [18]. One state-of-the-art systolic array solution is Google's Tensor Processing Unit (TPU) [18, 33]. Its core is a 2D systolic array of 256×256 identical Processing Elements (PEs) that perform 8-bit Multiply-and-Accumulate (MAC) arithmetic. The latest version (v3) has 128×128 elements that supports floating point arithmetic.

The advent of cloud computing and large datacenters has led to an increasing interest in linear algebra applications that also operate on sparse data structures, such as matrices, where the majority of entries are zero [8]. For instance, one of the most widely adopted applications in datacenters is large-scale graph processing, which is prevalent in fields ranging from social science to machine learning. Most of these problems directly translate into iterative sparse matrix-vector operations [31]. Algorithms that operate on sparse data take advantage of the sparsity by employing data structures that only store the non-zero elements, thus eliminating any redundant operations that involve the zero elements [1, 35]. However, such sparse algorithms have some disadvantages; they typically use data structures (*e.g.* compressed sparse row) that are more complex than simple vectors or matrices, which results in a higher number of memory accesses per useful computation. Also, because the associated data structures are not easily vectorized, these sparse data structures are not compatible with conventional systolic paradigms. Hence, incorporating the ability to handle sparse linear algebra in systolic arrays is both challenging and rewarding, since it extends the use of the same accelerator architecture for both sparse and dense applications.

The root cause of inefficiency in a systolic array when handling sparse matrices stems from the fact that the PEs containing zero-valued matrix entries perform MAC operations that do not contribute to the final result. A recent systolic array based solution was proposed by Kung *et al.* [20], referred to as KMZ in the rest of this paper. Their design is a systolic array that uses packed sparse matrices. The work presented in this paper improves upon this by addressing several limitations. The first is scalability arising from the use of a fixed number of parallel buses through each column of the 2D array. To limit the impact of the buses, they are bit-serial. This in turn leads to the second limitation: the system only supports integer arithmetic, thus failing to cater to applications, *e.g.* scientific computing, that require high numeric precision.

This paper presents *Sparse-TPU*, abbreviated to STPU, a comprehensive framework that maps sparse data structures to a 2D systolic-based processor in a scalable manner. The result is a TPU-like processors [18] that can also handle sparse matrices. To improve

ICS '20, June 29-July 2, 2020, Barcelona, Spain

ACM ISBN 978-1-4503-7983-0/20/06…$15.00
https://doi.org/10.1145/3392717.3392751

the utilization of the systolic array when tackling matrices with a wide range of sparsity and size, we propose a scalable matrix packing algorithm to reduce the number of zero-valued entries mapped into the array. We also explore the trade-off when handling collisions (*i.e.* columns having one/more elements with the same indices) in the packing algorithm. A collision-free constraint could be too strict and prevent compact packing. The resulting algorithm exhibits high *packing efficiency* across a broad set of sparse matrices, where packing efficiency is defined as the ratio of density of the uncompressed sparse matrix to that of the packed sparse matrix. To handle packed matrices we propose an enhanced PE design. It is based on a MAC function unit augmented with simple input matching and holding capabilities to perform conditional MAC operations.

In summary, we make the following contributions.

(1) We propose an algorithm to efficiently pack sparse matrices by merging columns that allows collisions and significantly reduces the number of zero-valued entries mapped to the systolic array.
(2) We present a 2D systolic array design that employs conditional execution to efficiently handle sparse matrices with a high degree of sparsity, while achieving TPU-like performance on dense matrices.
(3) We evaluate our design on a suite of synthetic sparse matrices, as well as real-world matrices spanning multiple domains from the SuiteSparse collection [4].

On average, STPU achieves a speedup of 16.08×, while consuming 4.39× and 19.79× lower energy for integer (int8) and floating point (float32) implementations, respectively, over a TPU baseline.

2 SPARSE MATRIX PACKING

The study of sparse matrices dates to the early days of computing when it was critical to save storage and operation count [7]. More recently, cloud computing and storage which operate on massive datasets have increased the importance of graph algorithms based on sparse matrix vector multiplication [3, 31]. Additionally, the trend of pruning in Deep Neural Networks (DNNs) [15, 16] has resulted in sparse data structures showing up in machine learning applications. These new applications often employ more compact formats of storage to save power as well as off-chip memory bandwidth.

The most widely used sparse matrix format is the Compressed Sparse Row (CSR) format [11], or its counterpart the Compressed Sparse Column (CSC) format. CSR consists of three distinct arrays: *vals*, *cols*, and *row-ptrs*. *vals* is a contiguous array of all the non-zeros in the matrix, organized in row-major order, and *cols* contains the column indices of the corresponding elements in *vals*. The *row-ptrs* hold the start indices of each row of the matrix in the *cols*/*vals* arrays. While CSR allows for fast row accesses and efficient storage of sparse data, it is incompatible with most existing systolic algorithms, since traditional systolic computation is designed around deterministic, fixed-length inputs.

One approach to overcome these shortcomings is to pack the columns of a sparse matrix to create a denser matrix. One of the earliest works that incorporate merging of sparse rows/columns into a denser format was done by Tarjan and Yao [32]. More recently, KMZ proposed packing sparse columns of a filter matrix in a transformed Convolutional Neural Networks (CNNs) to produce denser matrices [20]. We improve upon their solution by providing better *scalability* on sparse matrices with larger sizes, and eliminating the need to have multiple buses through the columns in the systolic array design (see Section 6).

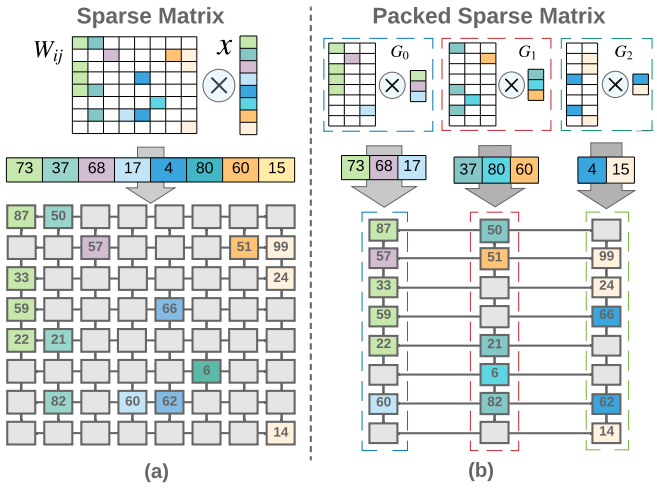

Figure 1: a) Sparse matrix mapping on the TPU and b) sparse matrix packing and mapping on STPU. The offline packing algorithm packs the columns Ci of the original sparse matrix W_{ij} to form the multi-column groups Gi of the packed matrix W_{ij_pack} and the corresponding input vector is packed to form element vector groups. For acceleration, the groups Gi are mapped vertically on STPU.

In the rest of this section, we first describe a naïve algorithm to produce packed matrices. We then augment this algorithm to improve column packing for large sparse matrices with higher density. Note that *the sparse matrix packing is conducted offline*, and loaded into the systolic array in the same manner as the TPU [18].

2.1 Basic Column Packing Algorithm

The most straightforward packing approach is a greedy algorithm. It tries to pack as many columns into the multi-column groups as possible. The algorithm selects a candidate column and packs it into a group as long as there is no *collision* between this column and any other column in the packed group. A *collision* between two columns occurs when they have a non-zero entry at the *same* row index. When a collision occurs between the next column to be packed and all existing multi-column groups, a new group is created to hold the conflicting column. This process stops once every column has been processed.

Fig. 1 illustrates how the columns of an example 8×8 sparse matrix can be packed to form a more compact 8×3 matrix. Each of the three resulting multi-column groups consists of elements from several columns of the original matrix. *E.g.* the leftmost multi-column group, G_0, is the product of packing original columns C_0, C_2, and C_3. While this does not result in a fully dense matrix (*i.e.*

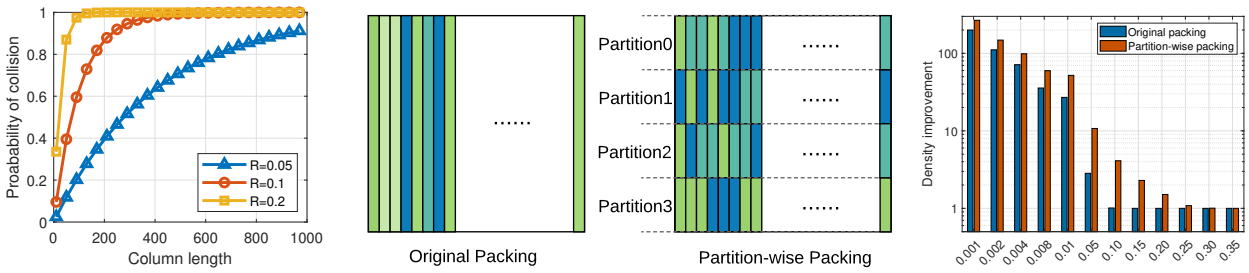

Figure 2: Graphical illustration of the probability of collision with different column size for three densities, (b) Illustration of original packing and partition-wise packing and (c) Density improvement under original packing and partition-wise packing

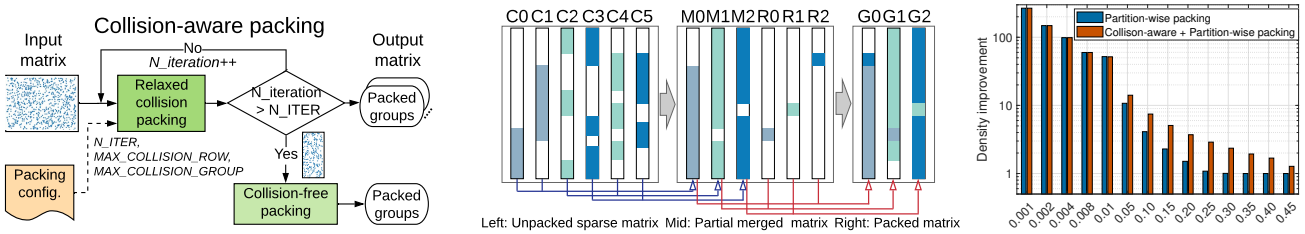

Figure 3: (a) Two-step collision aware packing. Step 1: Relaxed-collision packing is performed iteratively to process the matrix entries producing multi-column groups. Step 2: Collision-free algorithm packs the residual columns/entries to packed multi-column groups. (b) Illustration of the relaxed-collision step. (c) Density improvement with collision-aware packing.

one with *no zeros*), our experiments demonstrate packing efficiency of up to 718× in our example sparse matrices.

2.2 Optimizations For More Efficient Packing

The packing strategy has a direct impact on the overall latency and energy when performing sparse matrix computation on a systolic array. To improve the greedy algorithm, we propose two algorithm-level optimizations to produce a denser packed matrix: *partition-wise packing* and *collision-aware packing*.

Partition-wise Packing. For two columns of a random sparse matrix, the probability of collision is $P = 1 - (1 - R^2)^N$, where R is the density of the sparse matrix and N is the matrix column size. As shown in Fig. 2a, the probability of collision increases logarithmically with matrix column length N. Even a very sparse matrix (*e.g.* 0.05) can have a very high collision rate (91.18%) when the column length exceeds 1,000. The high occurrence of collisions limits the efficiency of the packing algorithm. Employing *partition-wise packing* can improve the packing efficiency further. It is not necessary to pack the whole column for a large sparse matrix, because the number of PEs in the systolic array is limited and the packed sparse matrix has to be split into blocks the size of the systolic array before mapping onto the array, .

Partition-wise packing effectively reduces the length of packing candidates (to the length of the systolic array), leading to improved packing density, because shorter columns reduce the probability of collisions. Fig. 2b illustrates the difference between the original method and the partition-wise method, where columns with matching colors are the candidates to be packed. The sparse matrix is first partitioned vertically by the length of the systolic array, then column packing is performed on each partition. Finally, the packed partitions are further divided horizontally into blocks that are mapped on the systolic array. Fig. 2c compares the density improvement of original packing method and partition-wise packing method for 1,000×1,000 sparse matrices with varying densities. Partition-wise packing achieves 13.83× higher density than the naïve packing method.

To support partition-wise packing, minor changes are made to the way the SpMV vector is input. With naïve packing, the packing strategy of input elements remains the same across all partitions, which means the groups of input elements sent to a column in the systolic array remains the same across all partitions. For partition-wise packing, since each partition is packed independently, the grouping strategy of input elements for one partition will be different from another.

Collision-aware Packing. We notice that when the density is higher than 0.2, it is hard to pack columns due to inevitable collisions (a probability of 0.9946 that collisions will occur even under 128 vertical partitioning). Statistical analysis of occurred collisions shows that *only a handful collisions prevent columns from merging with each other*. To further improve packing efficiency, we relax the collision-free constraint and adopt a less conservative packing method to enable column packing for sparse matrices with higher density. Specifically, when deciding on whether a column can be packed into a multi-column group, we adopt a relaxed requirement: that the total number of collisions should be within a

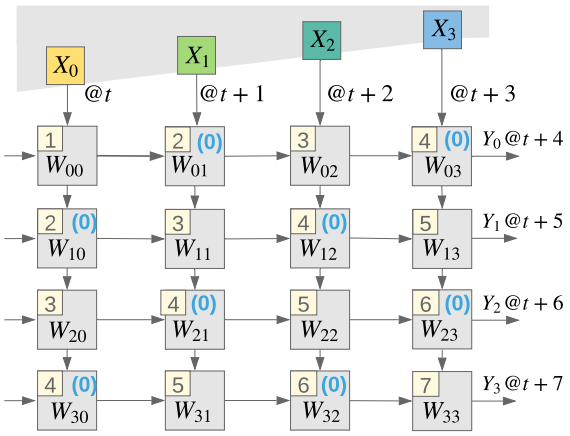

Figure 4: The basic operation of the TPU performing SpMV. The number n in the top left corner indicates that the corresponding PE performs a MAC operation at the nth cycle. Zero entries are directly mapped onto the array in TPU.

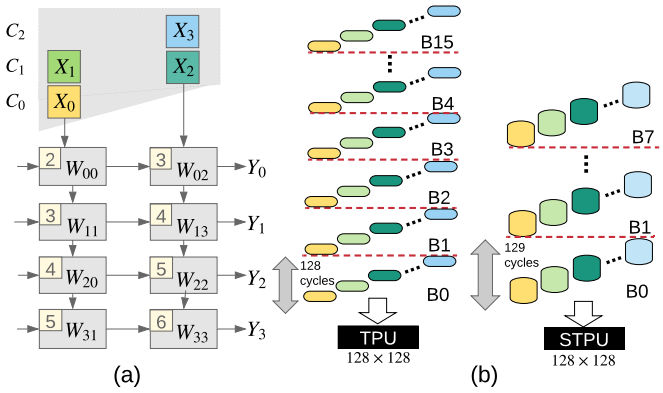

Figure 5: (a) SpMV on STPU with a packed matrix. STPU accommodates for maximum overlap of input vector groups. (b) SpMV input dataflow on TPU and STPU with original sparse matrix and its packed counterpart, assuming a 512×512 sparse matrix can be packed into 512×256. STPU outperforms TPU due to less iterations and small per-iteration latency overhead.

pre-specified bound, and that only one collision is allowed per row in a multi-column group.

In the proposed collision-aware scheme, we propose a two-step packing algorithm (Fig. 3a). In the first step, relaxed-collision packing is performed in an iterative way (*MAX_COLLISION_ROW* sets the maximum number of allowed collisions per row of the multi-column group and *N_ITER* indicates the number of iterations that relaxed collision packing method is invoked, and the two parameters are set to one and two, respectively, based on empirical observations in our experiments). The iterations are performed with tighter requirement. We use 15 and 4 as *MAX_COLLISION_GROUP*, the maximum number of collisions per group, for the first and the second iterations, respectively. In the first step, the entries that collided in the first iteration are used as additional packing candidates for the second iteration. Even after two iterations, there could still be residual candidate entries. The second step employs conventional collision-free packing to pack these entries. Since the total number of residual entries from the first step is small, they can be packed into a limited number of groups. Fig. 3b shows a simplified illustration of the relaxed-collision method iterated in step 1. The original matrix cannot be packed under the collision-free constraint. As a result of the relaxed constraint, $C0, C1, C2, C4$ and $C3, C5$ can be first packed into three intermediate groups $M0, M1, M2$ and three collided columns $R0, R1, R2$. These groups can then be packed into $G0, G1, G2$. Fig. 3c shows that using collision-aware packing on top of partition-wise packing achieves 1.86× higher packing efficiency than solely adopting partition-wise packing for matrices with moderate (0.05) to high (0.45) densities.

To support collision-aware packing, the SpMV vector is input so that one input element may be streamed into multiple rows of the array (similar to multi-casting), because different parts of the column could be packed together into different groups.

3 STPU OPERATION

Fig. 4 illustrates the basic operations of a systolic array network performing SpMV (zero entries are directly mapped onto the array) [18]. The input vector, $\overline{\mathbf{x}}$, is streamed into the network from the top. The partial results then propagate through the network cycle-by-cycle, starting from the top left corner and proceeding downward and to the right in a diagonal wave as represented by the different colors. Unlike the one-to-one mapping between vector elements to PE columns employed in the TPU, in STPU, the input vector elements need to be grouped and streamed into their corresponding PE column as the sparse matrix is in a packed format. There are several challenges to designing an interface that delivers a group of input elements to each array column.

The work described in KMZ used parallel bit-serial buses to stream the input groups, targeting matrices with densities between 12.6% and 33.3%. While their approach achieved higher energy-efficiency over SC-DCNN [28] with minimal hardware overhead, it does not scale for large, highly sparse matrices, even with bit-serial buses. For instance, using a simple greedy bin-packing algorithm on a 10,000×10,000 sparse matrix with 0.08% density results in a packed matrix of 28 multi-column groups, each consisting of 357 columns on an average. This will require the system to not only have hundreds of buses (eight maximum in Kung's approach) going through each column of the systolic array, but also a large MUX in each PE to select the matching input element for the MAC operation. In addition, any multi-column group with total number of columns less than the equipped buses will suffer under-utilization of resources.

To solve this scalability issue, we employ sequential streaming of the input vector elements to PE columns instead of parallel streaming. The sequential streaming eliminates the overhead incurred by parallel buses and large MUXes to select the inputs. Fig. 5a illustrates the operation of STPU executing sparse matrix-vector multiplication on a compact matrix, which eliminates the zeros

from the example in Fig. 4. Fig. 5b shows an example input for the TPU and STPU performing SpMV with a 512×512 sparse matrix. Without loss of generality, we assume the sparse matrix is packed into a 512×256 compacted matrix. On a 128×128 TPU, performing SpMV with the original matrix takes 16 iterations, each of which has a 128-cycle latency. As shown, STPU outperforms the TPU in terms of latency (since the matrix is packed) by performing SpMV in fewer iterations (8 in this example) while incurring minimal overhead from overlapping–see Fig. 5b.

We show three key differences between the operation of STPU and the TPU-like systolic array: *staggered group-wise input loading*, *input and partial-sum forwarding*, and *data-driven MAC computing*. In the TPU, the input elements are staggered to form a diagonal flow into the systolic array, whereas in STPU, the input elements are grouped due to matrix packing so that group-wise input staggering is performed. Then the input elements from a group are sequentially forwarded from the north edge to the south edge cycle-by-cycle, and each PE captures a copy of the input element which matches the column index of the stored matrix entry. Unlike the TPU PE, which performs MAC operation upon the arrival of an input element, STPU PE performs MAC operations only after 1) obtaining a matching element (which is guaranteed when all the elements from a group passed through it) and 2) its neighboring/left PEs have already performed a MAC and produced a partial-sum. To meet these requirements, the STPU PE performs the MAC *after* it has seen all the input elements of a group. The input staggering ensures that the MAC operation in a PE always happens later than its left neighbor PE. By doing this, the partial-sum can be correctly accumulated from the west edge to the east edge. The detailed design of the STPU PE is presented in Section 3.2.

3.1 SpMV with Large Matrices on STPU

In this section we examine the case of packed matrices that are much larger than the systolic array. A common solution is to employ blocking algorithms on the packed matrices [19, 21]. Blocking algorithms partition large matrices into multiple sub-matrices of size $SA_H \times SA_W$, where SA_H and SA_W are the height and width of the systolic array. For instance, a randomly generated sparse matrix W of size 1,000×1,000 and density of 0.08 is packed into a denser matrix W_{pack} of size 1,000×221 with a density of 0.34. In this example, both the number of rows and columns/groups of the matrix W_{pack} are larger than the dimensions of the underlying 128×128 systolic array. Therefore, the matrix W_{pack} must be divided into 16 blocks of size 128×128.

Before diving into the mapping strategy, we first detail the process of loading matrix blocks into the systolic array to expose the timing constraint. In a conventional blocked matrix vector multiplication as shown in Fig. 5, if matrix *block i* is already loaded in the array, the elements of the vector are input into the array from *cycle 0* to *cycle* $SA_H - 1$. Meanwhile, each row of *block i+1* is loaded one after another from the left edge of the systolic array. For *block i+1*, the vector can be input starting from *cycle* $SA_H - 1$, the time when the first row of *block i+1* is fully loaded and propagated to the destination. In this case, the scheduling of input elements in both TPU and STPU is straightforward, as shown in Fig. 6a and c.

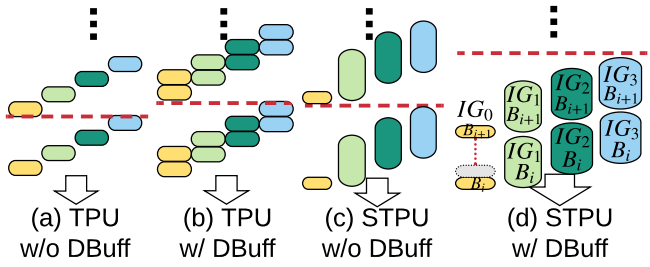

Figure 6: Input dataflow for TPU and STPU with and without double buffering. In (d), if IG_0 for B_{i+1} is scheduled as the grey box, the previous results will be overwritten before being used by the rightward PEs in the systolic array.

To accelerate matrix computation, TPU employs double buffering. We also employ this in STPU. With double buffering, a PE can hold two entries, one from each of two blocks. While the array is performing matrix-vector multiplication for a block, the other block is being loaded without any interference. Since double buffering allows two matrix entries to reside in a PE, it can seamlessly switch from one block to another block for computation. Loading future blocks (*e.g. block i+1*) does not block the execution of an existing block (*e.g. block i*). Double buffering improves the throughput by loading and computing simultaneously. For TPU with double buffering, two diagonal streams of input elements comes with a one cycle gap, and the next two streams are scheduled SA_H cycles later. This then repeats, as shown in Fig. 6b.

For the mapping of packed sparse matrices on STPU with double buffering, the input vector elements need to be carefully coordinated to ensure correctness and maximized performance. Unlike TPU, the matrix packing algorithm can lead to an imbalanced distribution of packed columns into the groups. Without proper input scheduling, a PE may not be able to fetch the correct partial-sum from its left PE, because the partial-sum may be overwritten.

When mapping a large packed sparse matrix onto a double-buffered STPU, a set of rules are enforced on the input vectors. In particular, the elements of the vector for block B_{i+1} cannot be input into the corresponding column of the systolic array until the following conditions are met, as shown in Fig. 6(d): 1) to avoid datapath conflicts, an input group IG_n for block B_{i+1} has to follow the previous group IG_n for block B_i; 2) to maintain correctness, the last element of IG_n for block B_{i+1} should be input after the last element of IG_{n+1} for block B_i streams into the neighboring PE on the right. Fig. 6(d) shows the constraints on the input vectors (formed by input element groups IG_0, IG_1, IG_2, IG_3 from left to right) into STPU with a double buffer that holds both block B_i and block $i+1$. First, streaming element group IG_n for block $i+1$ must wait until IG_n for block B_i is totally input, else there would be datapath conflicts. Second, as indicated by the vertical dashed line in Fig. 6d, streaming IG_0 for block B_{i+1} needs to wait until IG_1 for block B_i is fully streamed, else PEs in *Col*1 processing block B_i would accumulate the modified partial sum from PEs in *Col*0 processing block $i+1$. Note that the timing for streaming in the elements of input vector is determined offline by the compiler.

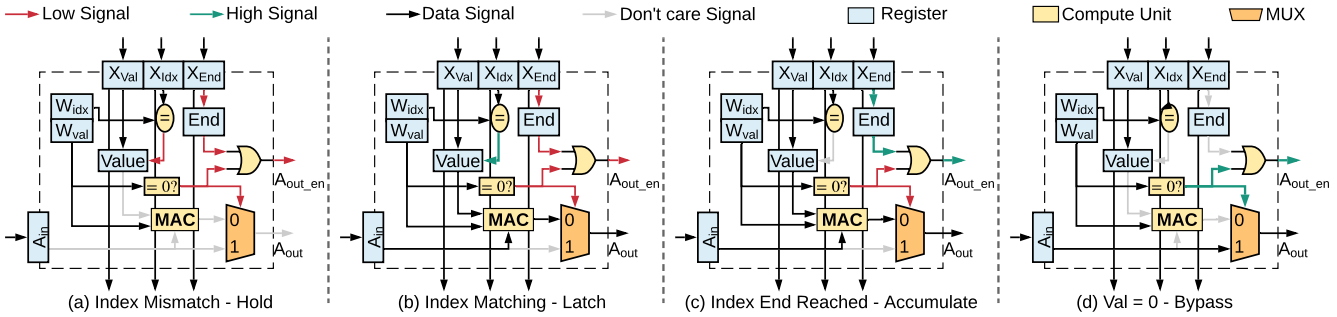

Figure 7: Microarchitecture and dataflow within a PE in different modes of operation. (a) Hold: The PE holds the accumulated result when the index mismatches. (b) Latch: The PE latches the input vector element when the input index matches (c) Accumulate: When the end of a vector group arrives, the PE calculates the MAC result and updates the Accu register of the rightward PE. (d) Bypass: If the matrix value held in the PE is 0, the data coming from leftward is bypassed rightward.

3.2 STPU Architecture

STPU comprises of a 128×128 systolic array attached to a multi-module high-bandwidth memory (HBM) subsystem with 300 GB/s bandwidth per HBM module, which feeds the matrix and vector inputs into STPU. Although TPU v1 uses an off-chip DDR3 for this purpose [18], TPU v2-3 use 8 GBs of HBM per systolic array pod [5]. The increased bandwidth is utilized in STPU to load the indices of the matrix entries as well as to support float32 and int8 values. Each node of the systolic array is a Processing Element (PE) and each matrix entry loaded into the systolic array is held by the PE in its internal registers. PEs in both STPU and TPU have double buffers to store the values and indices of the matrix entries for two matrix blocks. The following subsections presents the architecture and operation modes of the STPU PE as well as the memory interface used in STPU.

3.2.1 Processing Element Architecture. Fig. 7 shows the architecture of a PE in the systolic array of STPU. The PE is an energy-efficient hardware block designed around a floating-point MAC unit. In this work, we consider two data formats for the input data values — 32-bit floating point (float32) and 8-bit signed integer (int8). The sparse matrix column-index is represented as a 16-bit unsigned integer. The PE block consists of registers that hold the matrix index (W_{idx}) and value (W_{val}), and a 16-bit comparator that compares the input vector element's index (X_{idx}) with the matrix index that it holds. There is also a comparator that checks if the held matrix value, W_{val}, is zero. A register (A_{in}) holds the accumulated result coming from the left stream. At the beginning of computation, the PE receives its assigned matrix {value, index} entry from the left stream and holds it in its registers. For clarity, we omit the double buffering logic and connections for matrix loading in the figure.

The elements of the input vector are streamed from the top edge of the array. The PE also receives an accumulated value (A_{in}) from the left neighboring PE. To shorten the critical path of the matrix computation, which depends highly on the delay between streaming in the *first* and *last* element groups, we perform a delayed accumulation. This feature allows the input element groups to be streamed sequentially into the systolic array with overlaps, as shown in Fig. 5a. The key is that instead of performing the MAC and sending the result downstream as soon as a vector element with a matching index arrives, the PE latches-in the element value upon an index match. The PE does not compute the MAC until the last element in the vector group is received, in order to ensure functional correctness. As shown in Fig. 7, the PE requires two additional registers to support this optimization, one for holding the vector element (*Value*), and the other, *End* , for holding a 1-bit signal indicating the end of a vector group to ensure correct timing of operations. While this feature of the PE delays the computation in a few PEs, it allows faster streaming of the vector groups, thus significantly reducing the total latency of the computation.

Table 1: PE modes for different control signals.

$W_{val} == 0$	X_{End}	$W_{idx} == X_{idx}$	Mode
0	0	0	Hold
0	0	1	Latch
0	1	×	Accumulate
1	×	×	Bypass

*×: Don't care

3.2.2 Modes of Operation. Based on the input element's index, value of the matrix entry stored in the PE and the end signal of an element group, the PE can be operating in one of the four modes: *hold*, *latch*, *accumulate* or *bypass*, as illustrated in Fig. 7 and Tab. 1. The input vector elements, *i.e.* X_{val}, X_{idx}, and X_{End}, are always passed to the downward PE regardless of the mode of the PE.

Hold: The PE selects the matching vector element from a group of streaming input vector elements. If the input index X_{idx} does not match W_{idx} and W_{val} is non-zero, the PE retains the value of A_{in}.

Latch: When the PE encounters an element that matches the index it held, *i.e.* when the input index X_{idx} matches W_{idx}, and W_{val} is non-zero, the input value X_{val} is copied and stored until the end signal arrives.

Accumulate: The PE starts performing MAC operations after all input elements are streamed through it. When a delayed end signal X_{End} corresponding to the end of the input vector group arrives, the stored value X_{val} is multiplied with W_{val}, summed with A_{in} and stores the partial-sum into A_{in} of the rightward PE.

Table 2: PE evaluation of the double-buffered (DBuff) TPU and STPU, and single-buffered (SBuff) KMZ.

	Mode	Float32 Power (mW)	Float Area (μm^2)	Int8 Power (mW)	Int Area (μm^2)
STPU (DBuff)	ACCU	4.31 (198.6%)	6726.6 (112.93%)	0.51 (269.1%)	650.2 (211.8%)
	HOLD/ LATCH	1.23 (56.7%)		0.48 (251.8%)	
	BYPASS	1.24 (57.1%)		0.48 (250.8%)	
	IDLE	0.05		<0.01	
KMZ [20] (SBuff)	ACCU	N/A	N/A	0.34 (179.6%)	506.5 (165.0%)
	IDLE	N/A		<0.01	
TPU [18] (DBuff)	ACCU	2.17 (100%)	5956.4 (100%)	0.19 (100%)	306.9 (100%)
	IDLE	0.05		<0.01	

Bypass: Generally, a sparse matrix can not be packed into a completely dense matrix. Thus, the compressed matrix is still sparse and has zero values. To conserve power consumed by a PE in this scenario (W_{val} is zero), we bypass A_{in} from a PE into the A_{in} of its right neighbor, while powering off all the compute elements.

3.2.3 STPU Memory Interface. Though the focus of the paper is the systolic array architecture and dataflow, in this subsection we present a memory interface solution to feed the input vector elements to STPU.

On-chip memory, *i.e.* a multi-bank scratchpad is used to store the input vector and allows parallel access to the vector elements so that data streaming could match the speed of processing. The scratchpad stores tuples of (column index, value) corresponding to each input element. To distribute the elements from the on-chip memory, a swizzle-switch network-based crossbar (XBar) is employed which supports point-to-point and multi-cast communication [30]. We augment the crossbar design with a Crosspoint Control Unit (XCU) that enables reconfiguration by programming the crosspoints. The XCU also stores the reconfiguration information describing the XBar connections between SRAM banks and FIFOs which stores the grouped vector elements. Each FIFO pops the elements in a staggering manner as described in Section III. FIFOs are double-buffered to hide data-fetch latency so that one set of FIFOs can be updated by incoming vector elements while the second set of FIFOs are drained out by the STPU array.

Discussions. Adopting XBars for data shuffling introduces extra area overhead. In the proposed STPU, which has a 128×128 sized systolic array, a 128×128 crossbar with a bit-width of 49 (16 column index bits, 32 data bits width and 1 bit for the end signal) is used to feed the FIFOs. Our conservative estimate of the crossbar area based on [30] is $0.6724mm^2$, which is a negligible overhead of only 0.201% over the TPU chip area. Incorporating on-chip 128 FIFOs with 128 depth consumes just 128 kB, which is far less than the 25 MB buffer space used in the TPU.

4 METHODOLOGY

Simulator and Physical Design. We compared our proposed architecture, *STPU*, with Google's *TPU* and *KMZ*. For fairness, we assume a 128×128 PEs systolic array across all designs. We implemented a custom cycle-accurate high-level language simulator for the three designs, aimed to model the RTL behavior of synchronous circuits. Each STPU PE cell implements *hold*, *bypass*, *accumulate* and *idle* states.

In order to accurately measure both area and power, we modeled the PE designs in the proposed STPU as well as the baseline TPU and KMZ using System Verilog. For a thorough comparison of STPU and TPU, we implemented PEs of two arithmetic precision, int8 and float32 for both the proposed STPU and the baseline TPU. For the KMZ design, we only implement int8 PEs, because their bit-serial design does not support floating point arithmetic. We synthesized each design using the Synopsys Design Compiler with a 28 nm CMOS standard cell library and a target clock frequency of 700 MHz.

The power and area estimates and the relative area/power costs of the STPU and KMZ PE to the TPU PE are listed in Tab. 2. The overhead of 2.11× area and 2.55× average power of STPU compared to the TPU in the int8-format becomes much smaller in the float32 counterpart (1.13× area and 1.04× average power increase). The overhead of int8-format is a result of the input/matrix index registers being comparable to the int8 multipliers. In the float32-format implementation, the energy/area cost of the float32 multiplier dominates, hence the additional registers and logic only introduced a limited overhead. In Section 5, we show how this difference affects the energy consumption of STPU running SpMV. Note that in the KMZ design, the PE has a smaller area compared with STPU because it only implements single-buffering.

Datasets. We used two datasets to evaluate the proposed hardware/software co-design scheme. The first set comprises 15 synthetic sparse matrices that each have a uniformly random distribution of non-zeros. The sizes of the matrices range from 500 to 10,000, while the density is chosen between 0.001 and 0.3. The second set is a collection of 13 real-world sparse matrices from the SuiteSparse collection [4], including *M0: bwm200, M1: California, M2: circuit204, M3: fpga_dcop_26, M4: freeFlyingRobot_1, M5: grid2, M6: hydr1c_A_01, M7: laser, M8: piston, M9: poisson2D, M10: problem1, M11: spaceStation_4, M12: young3c.*

5 EVALUATION

We evaluate the efficiency of both our packing scheme and the STPU design. For the offline packing algorithm, the packing efficiencies of the two different strategies are compared with KMZ [20]. We further compare the energy and performance over the TPU and KMZ designs executing SpMV with a random dense vector.

For STPU, we evaluate two packing strategies, STPU-C0 and STPU-C1, on the two sets of sparse matrices. Our proposed *partition-wise packing* scheme is used in both strategies for its significant reduction in collision probability and improved number of non-zero (NNZ) element density. The difference between the two schemes is that STPU-C0 maintains a mandatory collision-free constraint while STPU-C1 relaxes the constraint by using collision-aware packing (detailed in Section 2).

5.1 Packing Algorithm Efficiency

We define *packing efficiency* as D_m/D_p. The density of an unpacked matrix D_m is calculated as the NNZs divided by the size of the matrix, *i.e.* $\frac{NNZs}{matrix\_h \times matrix\_w}$. Since we adopt partition-wise packing in both cases for the packed matrix, the density D_p is computed as $\frac{\sum NNZ_i}{\sum (block\_w_i \times block\_h)}$, where $block\_h$ is the height of partition for packing (128 in this paper to match the systolic array

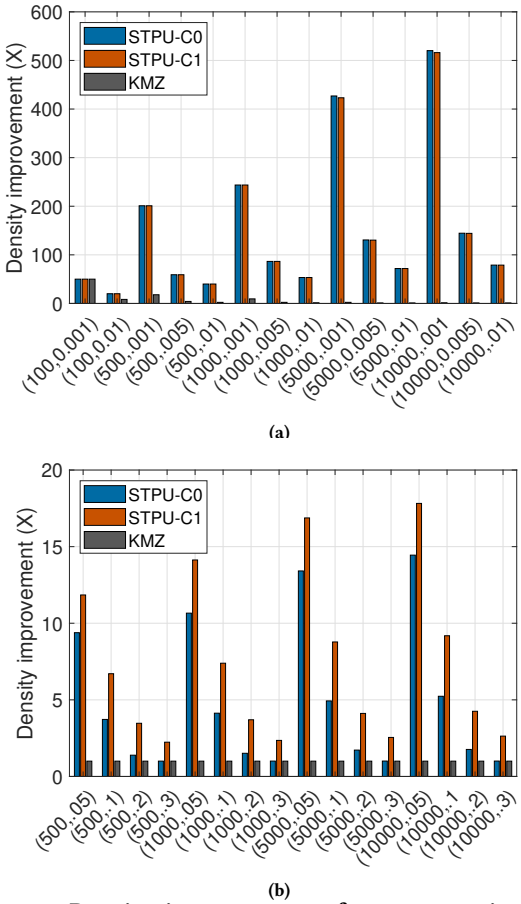

Figure 8: Density improvement of sparse matrices using STPU-C0, STPU-C1 and KMZ packing schemes on (a) ultra sparse matrices (density <=0.01) and (b) moderately sparse matrices (density >=0.05).

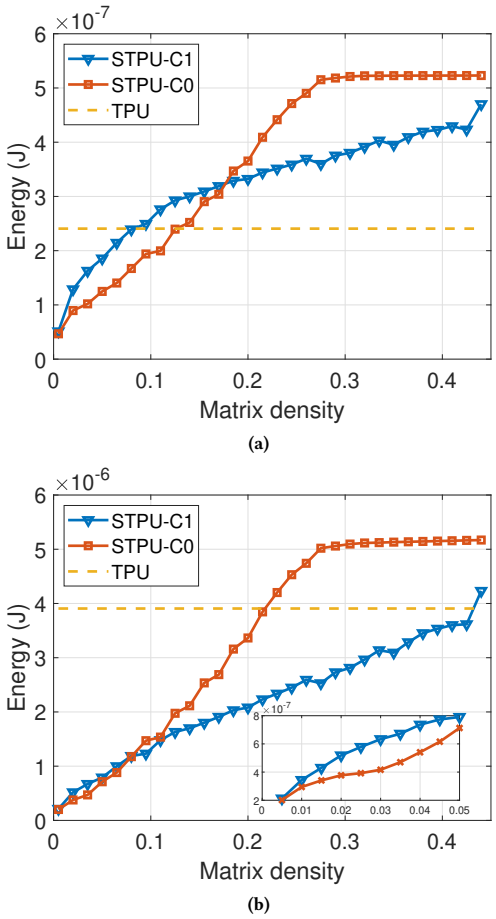

Figure 9: Energy evaluation of STPU and TPU running SpMV with fixed size sparse matrices with different densities. (a) int8 implementation, (b) float32 implementation.

dimension), i is the partition ID for the packing algorithm, NNZ_i is NNZs in partition i and $block\_w_i$ is the width (the number of multi-column groups) of the ith partition after packing.

Fig. 8 shows the packing efficiency of the proposed STPU packing strategies and KMZ across a set of uniform-random sparse matrices. Fig. 8a and Fig. 8b show the resulting packing efficiency for ultra sparse matrices (*i.e.* with density ≤0.01) and moderately sparse matrices (*i.e.* with density ≥0.05) for both STPU and KMZ. With STPU, for the ultra sparse matrices, STPU-C0 and STPU-C1 respectively achieve 144.0× and 143.4× packing efficiency. For the moderately sparse matrices, STPU-C0 and STPU-C1 achieve 4.48× and 6.92× better density, respectively. For the most sparse case (*i.e.* dimension of 10,000 and density of 0.001), the packing efficiency is as large as 520.2× with the STPU-C0 scheme.

STPU-C0 performs slightly better than STPU-C1 while packing ultra sparse matrices, as collisions are rarely seen in these matrices, STPU-C0 can already reach a high density level without allowing collision, while STPU-C1 unnecessarily relaxes the constraint and creates redundant groups. On the other hand, when handling moderately sparse matrices, STPU-C1 outperforms STPU-C0. The reason is that even if collisions exist, STPU-C1 enables the packing of columns with limited number of collisions that cannot be packed by STPU-C0. This difference demonstrates that STPU-C1 is more effective for less sparse matrices. For the packing algorithm proposed in KMZ [20], as shown in the figure for the ultra sparse matrices, on average 8.0× packing efficiency is seen, whereas for the moderately sparse matrices 1.07× improvement is achieved. These results illustrate that our proposed packing algorithm outperforms the prior work. Even though KMZ's method achieves the same packing efficiency as us for ultra sparse matrix of small sizes (*e.g.* with a dimension of 100 and density of 0.001), with increasing size or density of the sparse matrix, the packing efficiency decreases significantly because the packing method in KMZ cannot handle the increased number of collisions efficiently.

5.2 Performance and Energy for SpMV

Scalability Analysis. In the first experiment, we fix the size of the sparse matrices (*i.e.* 1,000×1,000) and vary the density, showing energy comparisons between STPU and the TPU running SpMV on sparse matrices of increasing density (Fig. 9). The goal of this

Table 3: Performance scaling with fixed number of nonzeros: latency of SpMV execution on STPU compared to the TPU. The dimension of the sparse matrices range from 1000 to 10,000 each having 200,000 non-zero elements.

Matrix Size	Density (r)	Elapsed Time (# cycles)			
		STPU-C0	STPU-C1	TPU	KMZ
1,000	0.2000	2,887	1,481	4,129	18,240
2,000	0.0500	1,833	1,798	16,129	72,960
3,000	0.0220	2,088	2,510	36,129	164,160
4,000	0.0125	2,965	3,552	64,129	291,840
5,000	0.0080	4,071	4,689	100,129	456,000

experiment is to explore the relationship between the density of the sparse matrix and energy consumption. We also show the crossover point of performance between STPU and the TPU. Fig. 9a shows the results for the int8 implementation.

As expected, the energy consumption increases with matrix density. Up to a certain density, the energy consumption remains constant. This is because as the density grows, the packing algorithm encounters higher probability of collision and eventually fails to pack any columns (*e.g.* densities >0.3 in STPU-C0). We also notice an interesting trade-off — for lower densities, STPU-C0 outperforms STPU-C1, and *vice versa* for high densities. This is because in the low density range, though both STPU-C0 and STPU-C1 achieve similar density improvement and elapsed latency, STPU-C1 relaxes the collision-free constraint, allowing entries of a column to be packed into different multi-column groups. As a result, during the execution of STPU-C1 an input vector element may need to be streamed to multiple columns of the systolic array, leading to an increased number of index comparisons and thus higher energy consumption. Taking a sparse matrix with 0.1 density as an example, the matching energy in STPU-C1 is 5.25× greater than STPU-C0. At the high density end, STPU-C1 achieves a higher packing efficiency than STPU-C0 and results in a shorter elapsed latency (2.00× improvement for a density of 0.3) and reduced leakage energy (2.18×).

When comparing with the baseline TPU, STPU handles low density matrices efficiently by consuming lower energy than the TPU. Beyond a certain point (0.12 density for the int8 implementation), the TPU exhibits better energy consumption, because STPU has additional energy overhead, which is comparable to the TPU PE energy in the int8 version. When the density increases to a certain level, the effectiveness of matrix packing diminishes. Fig. 9b shows the results for the float32 implementations. Since the overhead of a STPU PE with float32 is relatively small compared to the TPU PE, the crossover point is shifted towards the high density end (0.20 for STPU-C0 and 0.44 for STPU-C1). These results demonstrate the potential of adopting STPU for applications with high-precision requirements. In this experiment, we also estimate the energy of the KMZ int8 design. The KMZ design consumes 1.70×10^{-5} J of energy, which is 4.35× larger than that for the int8 TPU design. The reason for the low efficiency is two-fold. First, their packing algorithm cannot pack any sparse columns due to collisions. Second, the overhead of bit-serial design incurs longer latency, leading to large leakage energy.

For the second experiment, we fix the total NNZs (*i.e.* 200,000) and report the performance comparison between STPU and the TPU running SpMV on sparse matrices of different sizes, as shown in Tab. 3. This experiment explores the scalability of STPU with increasing problem size while keeping the total number of non-zero MAC operations constant. As observed from Tab. 3, the latency increases linearly with increasing matrix size. This is because the opportunity for packing columns increases linearly with larger matrix dimension (sparser matrix) while the total size of matrix increases quadratically. In contrast, the TPU latency increases quadratically since the entire sparse matrix (with zeros) is mapped onto the systolic array. For KMZ's design, the latency is much larger than the TPU. This is also because their packing algorithm cannot pack as many columns, and thus zero entries are directly mapped on the systolic array. For the same reason, the latency induced by bit-serial operation is exacerbated. To conclude, these results indicate STPU has better scalability when handling very sparse matrices. Note that for STPU-C0, the latency initially reduces with increased matrix size, *e.g.*, 2,887 cycles for a matrix with size 1,000 and density 0.20, to 1,883 cycles for one with size 2,000 and density 0.05. This is because the packing efficiency improves from 1.51× to 11.95× as a larger sparsity gives STPU-C0 more opportunities to pack columns.

Performance and Energy Analysis. For the evaluation of SpMV on STPU, we report the performance improvement and energy reduction over TPU on a broad set of sparse matrices. Fig. 10a and 10b show the results for randomly generated sparse matrices. In terms of performance, STPU-C0 outperforms STPU-C1 for very sparse matrices (density ≤ 0.01) since STPU-C0 achieves 1.48× higher speedup than STPU-C1. While for moderately sparse matrices (density ≥ 0.1), STPU-C1 achieves 1.24× higher speedup than STPU-C0. For energy consumption of int8 implementations, STPU-C1 and STPU-C0 achieve on average 1.66× and 1.46× reduction, respectively. When targeting a high-precision float32 implementation, since the overhead of STPU is relatively small, STPU-C0 and STPU-C1 reduce energy consumption by 8.83× and 8.20× on average, respectively, over the float32 TPU implementation. Fig. 10c and 10d show the results for the SuiteSparse matrices. STPU-C0 and STPU-C1 achieve 24.14× and 21.50× speedup over the TPU, respectively. In terms of energy consumption of int8 implementations, STPU-C0 and STPU-C1 achieve 7.83× and 6.60×, on average reductions. For the float32 counterparts, STPU-C0 and STPU-C1 reduce energy consumption by 33.58× and 28.54× on average, respectively.

In addition, we compare STPU performance against a CPU (4.00GHz Intel Core i7-6700K Processor with MKL) and a GPU (GeForce RTX 2080 @ 1515 MHz with cuSPARSE). On average, STPU achieves 5.5x higher performance over the GPU and 436.0x higher performance over the CPU.

Bandwidth Analysis. As opposed to the TPU, the STPU PE requires the knowledge of the column index to perform index matching, thus the indices have to be streamed in the STPU array along with the values, incurring higher bandwidth requirement. Hence, we conduct design space exploration with the random-generated matrices set and compare the bandwidth requirement of SpMV between TPU and STPU. The results are shown in Fig. 11. The float32-version TPU is simulated for a direct comparison while the bfloat16-version halves the bandwidth requirement. We also

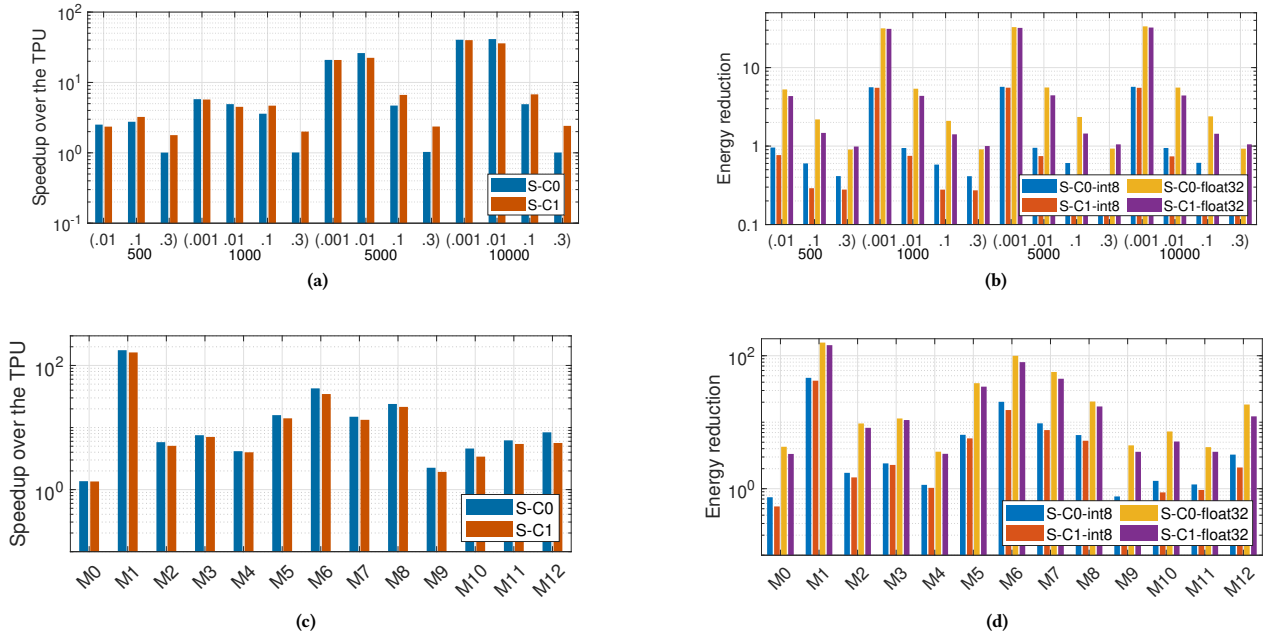

Figure 10: Evaluation of SpMV on synthetic uniform-random sparse matrices (a, b) and SuiteSparse matrices (c, d). (a) and (c) report the speedup (*x*-axis has matrix densities in parentheses and dimensions below them). (b) and (d) report the energy reduction for both the int8 and float32 implementations. S-C0(1) is a shorthand for the STPU-C0(1) scheme.

estimate the bandwidth of STPU-C1 in both float32- and bfloat16-versions. STPU-C1 exhibits higher packing efficiency than STPU-C0, which relieves the bandwidth requirement. As shown, TPU-float32 achieves similar bandwidth, *i.e.* around 633.51 GB/s across different matrices. In contrast, STPU-C1 achieves an average 486.3 GB/s and 327.55 GB/s in the float32 and bfloat16 implementations, respectively. Using the bfloat16 representation reduces the amount of matrix storage by half, as compared with float32, thus leading to a reduced overall bandwidth. For matrices with the same dimensions, larger density leads to higher bandwidth requirement due to the increased difficulty of column packing. We also notice that in some cases the bandwidth requirement exceeds 600 GB/s which is much higher than TPU. A possible solution to address this is using relative column indices of a packing group for the matrix to reduce the bit length of the indexes, rather than the absolute index (16 bits in this paper). Another orthogonal solution is leveraging multiple HBM modules in the memory system like TPUv3 and commodity FPGAs [5, 22]. Besides, HBM3 is expected to be able to provide TB/s-level bandwidth which can easily satisfy the bandwidth requirement of STPU.

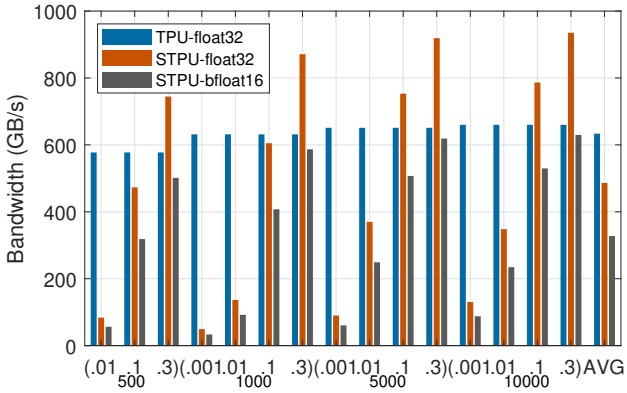

Figure 11: Bandwidth of TPU and STPU performing SpMV on a set of random-generated matrices.

6 RELATED WORK

Sparse DNNs and Matrix Acceleration. There has been significant interest in accelerating machine learning workloads, especially DNNs. One of the main approaches to improve DNN performance is to exploit the high degree of redundancy in the weight matrices in DNNs [6]. Deep Compression [14] and Dynamic Network Surgery [12] compress a DNN to a fraction of its original size through efficient pruning techniques. These pruning techniques have led to an increased interest in sparse matrix algorithms, as the dense weight matrices are pruned to sparse matrices. While conventional architectures, such as CPUs and GPUs, achieve high performance on dense DNNs, they struggle to achieve a similar level of performance on sparse matrices, due to irregular memory accesses that are inherent in sparse data structures. Thus, there has been growing interest in accelerating sparse matrix operations through custom hardware accelerators.

Early implementations of sparse matrix accelerators, such as the DAP processor [27], focused primarily on scientific computation workloads. These scientific and engineering applications operate on single or double precision floating point data, which are much

wider than the formats used in DNN applications. Efficient Inference Engine (EIE) [13] is a custom hardware accelerator specific to DNNs. EIE utilizes Deep Compression [14] to prune the network, then uses a custom pipeline to accelerate the matrix-vector multiplication. EIE operates on compressed data for both activation values and the weights, but relies on complex central scheduling units to coordinate the processing elements. Cambricon-X [34] is a sparse neural network accelerator with a fat-tree interconnect to alleviate network congestion. However, it is only concerned with compressing the activation values (*i.e.* the vector), and does not compress weights. OuterSPACE [23, 24, 26] is a co-designed solution that accelerates sparse matrix multiplication using an outer product algorithm on an architecture that reconfigures the on-chip memory. ExTensor [17] uses hierarchical intersection detection to eliminate unnecessary computations in sparse matrix-matrix multiplications and tensor algebra. SparTen [10] accelerates CNNs by exploiting both the sparsity in feature maps as well as the filter values of the network, and utilize asynchronous compute units for computation. Instead of designing custom accelerator hardware, STPU supports efficient sparse matrix vector multiplication on systolic arrays with minor modifications on TPU PEs [18].

Systolic Architecture for DNNs. Systolic arrays are simple, energy-efficient architectures that minimize data transfer overheads by having each PE forward data directly to the neighboring PE that consumes it. Systolic array based accelerators have been a popular choice for accelerating dense matrix multiplication workloads in DNNs. The Tensor Processing Unit (TPU) [18], which we use for comparison, is a systolic DNN accelerator that utilizes narrow, 8-bit MAC units and a custom 16-bit floating point format to reduce power consumption with minimal loss in accuracy. SCALE-Sim [29] is a simulation infrastructure for exploring design space of systolic array accelerators for DNNs. However, both the TPU and SCALE-Sim focus on dense matrix multiplication, and do not take advantage of weight sparsity resulting from pruning techniques that have emerged in recent years. SCNN [25] is a compressed neural network accelerator with a 2D mesh interconnect, where each PE propagates its data to the neighbor in a similar manner as a systolic array. SCNN operates on sparse matrix data and uses compressed sparse encoding of weights and inputs to operate only on non-zero elements. Kung *et al.* [20] employ a packing strategy similar to ours to accelerate DNNs, but their solution requires serialized parallel buses and re-training after collision handling. STPU proposes an efficient packing algorithm that allows collisions to greatly increase the density of the packed matrix, which leads to improvements in both latency and energy consumption.

7 CONCLUSION

In this paper, we proposed a co-designed framework to adapt systolic arrays to handle sparse matrices, specifically sparse matrix dense vector multiplication (SpMV). Instead of mapping a sparse matrix directly onto the systolic array, we developed a novel packing algorithm to pack matrix columns to reduce the occurrence of zeros. To make use of this column packed format, we presented the STPU design—a 2D systolic array of simple PEs. The PEs incorporates index matching and value holding functionalities, in addition to MAC units. These modifications support the direct processing of sparse matrices in the packed format. The packing algorithm effectively improves the density of the sparse matrices by 74.71× on average across a set of random generated matrices. For SpMV, STPU also exhibits performance improvement of 16.08× over the TPU baselines, on average, for uniform-random matrices and matrices from the SuiteSparse collection. In terms of energy efficiency, STPU achieves energy savings of 4.39× for the int8 implementation and 19.79× for the float32 implementation. Our future work involves expanding the STPU framework to handle SpMV with sparse vectors, as well as sparse matrix-matrix multiplication.

ACKNOWLEDGEMENT

The material is based on research sponsored by Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement number FA8650-18-2-7864. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

REFERENCES

[1] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2016. A scalable processing-in-memory accelerator for parallel graph processing. *ACM SIGARCH Computer Architecture News* 43, 3 (2016), 105–117.

[2] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, Karthik Gururaj, and Glenn Reinman. 2014. Accelerator-rich architectures: Opportunities and progresses. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[3] Alfredo Cuzzocrea, Domenico Saccà, and Jeffrey D Ullman. 2013. Big data: a research agenda. In *Proceedings of the 17th International Database Engineering & Applications Symposium*. ACM, 198–203.

[4] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1.

[5] Jeff Dean. 2017. Machine learning for systems and systems for machine learning. In *Presentation at 2017 Conference on Neural Information Processing Systems*.

[6] Misha Denil, Babak Shakibi, Laurent Dinh, Marc'Aurelio Ranzato, and Nando de Freitas. 2013. Predicting Parameters in Deep Learning. *CoRR* abs/1306.0543 (2013). arXiv:1306.0543 http://arxiv.org/abs/1306.0543

[7] Iain S Duff, Albert Maurice Erisman, and John Ker Reid. 2017. *Direct methods for sparse matrices*. Oxford University Press.

[8] Iain S. Duff, Michael A. Heroux, and Roldan Pozo. 2002. An Overview of the Sparse Basic Linear Algebra Subprograms: The New Standard from the BLAS Technical Forum. *ACM Trans. Math. Softw.* 28, 2 (June 2002), 239–267. https://doi.org/10.1145/567806.567810

[9] Adi Fuchs and David Wentzlaff. 2019. The accelerator wall: Limits of chip specialization. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 1–14.

[10] 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

[11] J. L. Greathouse and M. Daga. 2014. Efficient Sparse Matrix-Vector Multiplication on GPUs Using the CSR Storage Format. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 769–780. https://doi.org/10.1109/SC.2014.68

[12] Yiwen Guo, Anbang Yao, and Yurong Chen. 2016. Dynamic Network Surgery for Efficient DNNs. *CoRR* abs/1608.04493 (2016). arXiv:1608.04493 http://arxiv.org/abs/1608.04493

[13] 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. *CoRR* abs/1602.01528 (2016). arXiv:1602.01528 http://arxiv.org/abs/1602.01528

[14] Song Han, Huizi Mao, and William Dally. 2016. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding.
[15] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).
[16] Xin He, Guihai Yan, Wenyan Lu, Xuan Zhang, and Ke Liu. 2019. A Quantitative Exploration of Collaborative Pruning and Approximation Computing Towards Energy Efficient Neural Networks. *IEEE Design & Test* 37, 1 (2019), 36–45.
[17] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W. Fletcher. 2019. ExTensor: An Accelerator for Sparse Tensor Algebra. In *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. ACM, New York, NY, USA, 319–333. https://doi.org/10.1145/3352460.3358275
[18] 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
[19] Bo Kågström, Per Ling, and Charles Van Loan. 1998. GEMM-based level 3 BLAS: High-performance model implementations and performance evaluation benchmark. *ACM Transactions on Mathematical Software (TOMS)* 24, 3 (1998), 268–302.
[20] 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 *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, New York, NY, USA, 13. https://doi.org/10.1145/3297858.3304028
[21] Jiajia Li, Xingjian Li, Guangming Tan, Mingyu Chen, and Ninghui Sun. 2012. An optimized large-scale hybrid DGEMM design for CPUs and ATI GPUs. In *Proceedings of the 26th ACM international conference on Supercomputing*. ACM, 377–386.
[22] Nallatech. 2018. OpenCAPI enabled FPGAs—the perfect bridge to a data centricworld. https://openpowerfoundation.org/wp-content/uploads/2018/10/Allan-Cantle.Nallatech-Presentation-2018-OPF-Summit_Amsterdam-presentation.pdf. [Online].
[23] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. 2018. Outerspace: An outer product based sparse matrix multiplication accelerator. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 724–736.
[24] Subhankar Pal, Dong-hyeon Park, Siying Feng, Paul Gao, Jielun Tan, Austin Rovinski, Shaolin Xie, Chun Zhao, Aporva Amarnath, Timothy Wesley, et al. 2019. A 7.3 M Output Non-Zeros/J Sparse Matrix-Matrix Multiplication Accelerator using Memory Reconfiguration in 40 nm. In *2019 Symposium on VLSI Technology*. IEEE, C150–C151.
[25] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel S. Emer, Stephen W. Keckler, and William J. Dally. 2017. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. *CoRR* abs/1708.04485 (2017). arXiv:1708.04485 http://arxiv.org/abs/1708.04485
[26] Dong-Hyeon Park, Subhankar Pal, Siying Feng, Paul Gao, Jielun Tan, Austin Rovinski, Shaolin Xie, Chun Zhao, Aporva Amarnath, Timothy Wesley, et al. 2020. A 7.3 M Output Non-Zeros/J, 11.7 M Output Non-Zeros/GB Reconfigurable Sparse Matrix-Matrix Multiplication Accelerator. *IEEE Journal of Solid-State Circuits* (2020).
[27] S. F. Reddaway. 1973. DAP&Mdash;a Distributed Array Processor. In *Proceedings of the 1st Annual Symposium on Computer Architecture (ISCA '73)*. ACM, New York, NY, USA, 61–65. https://doi.org/10.1145/800123.803971
[28] Ao Ren, Zhe Li, Caiwen Ding, Qinru Qiu, Yanzhi Wang, Ji Li, Xuehai Qian, and Bo Yuan. 2017. Sc-dcnn: Highly-scalable deep convolutional neural network using stochastic computing. *ACM SIGOPS Operating Systems Review* 51, 2 (2017), 405–418.
[29] Ananda Samajdar, Yuhao Zhu, Paul N. Whatmough, Matthew Mattina, and Tushar Krishna. 2018. SCALE-Sim: Systolic CNN Accelerator. *CoRR* abs/1811.02883 (2018). arXiv:1811.02883 http://arxiv.org/abs/1811.02883
[30] Korey Sewell, Ronald G Dreslinski, Thomas Manville, Sudhir Satpathy, Nathaniel Pinckney, Geoffrey Blake, Michael Cieslak, Reetuparna Das, Thomas F Wenisch, Dennis Sylvester, et al. 2012. Swizzle-switch networks for many-core systems. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 2, 2 (2012), 278–294.
[31] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R Dulloor, Michael J Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. 2015. Graphmat: High performance graph analytics made productive. *Proceedings of the VLDB Endowment* 8, 11 (2015), 1214–1225.
[32] Robert Tarjan and Andrew Yao. 1979. Storing a Sparse Table. *Commun. ACM* 22, 11 (1979), 606–611.
[33] Paul Teich. 2018. Tear Apart Google's TPU 3.0 AI Coprocessor. https://www.nextplatform.com/2018/05/10/tearing-apart-googles-tpu-3-0-ai-coprocessor/
[34] 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
[35] Youwei Zhuo, Chao Wang, Mingxing Zhang, Rui Wang, Dimin Niu, Yanzhi Wang, and Xuehai Qian. 2019. GraphQ: Scalable PIM-Based Graph Processing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 712–725.