Squaring the circle: Executing Sparse Matrix Computations on FlexTPU—a TPU-like processor

Xin He, Kuan-Yu Chen, Siying Feng, Hun-Seok Kim, David Blaauw, Ronald Dreslinski, Trevor Mudge
University of Michigan
Ann Arbor, MI, USA

ABSTRACT

Systolic arrays have been successful to accelerate dense linear algebra for deep neural networks (DNNs), but cannot handle sparse computations efficiently. Though early attempts have been made to perform sparse matrix operations on weight-pruned DNNs, handling highly sparse matrices with skewed nonzero distribution commonly seen in real-world graph analytics remains challenging. In this paper, we propose FlexTPU framework to repurpose tensor processing units (TPUs) to execute sparse matrix-vector operations (SpMV). First, we propose a lightweight *Z-shape mapping* of sparse matrices onto the systolic array to eliminate the processing of zeros as much as possible, regardless of the sparsity and nonzero distribution. On top of the mapping, we devise an SpMV dataflow executed by an array of PEs, which are a slightly modified version of the conventional TPU PE. Second, in contrast to the excess preprocessing mandatory for prior attempts, the *Z-shape mapping* facilitates on-the-fly matrix condensing from the widely-used compressed sparse matrix (e.g. CSR) representation. This is accomplished by a proposed sparse data loader that includes an on-chip row decoder and parallel nonzero loaders. We evaluate FlexTPU on a broad set of synthetic and real-world sparse matrices. The experimental result shows that FlexTPU achieves 3.55× speedup and 3.27× energy saving over a state-of-the-art design, Sparse-TPU. It performs even better on sparse matrices with power-law distributions. Compared to state-of-the-art library implementations on a CPU and a GPU, FlexTPU also achieves an average speedup of 2.4× and 4.3×, and energy saving of 130.4× and 495.3×, respectively. FlexTPU is also evaluated against a recent reconfigurable (chip multi-processor) CMP machine, Transmuter. FlexTPU outperforms Transmuter by achieving 5.12× speedup and 2.65× energy saving.

1 INTRODUCTION

The demise of Dennard's scaling has prompted computer architects to turn to application-specific integrated circuits (ASICs), *i.e.* hardware accelerators, to handle linear algebra, due to the surge of interests in the fields of machine learning, scientific computing, and genomics. Adopting ASICs bridges the gap between the increasing computational demands and the stagnating transistor budgets [7, 11, 29, 42]. Unfortunately, the application space for customized design targeting a specific computational domain is limited. For data centers handling diverse workloads [5, 21, 24], the gain through piling up accelerators will be eventually capped by on-chip resources and hit the "accelerator wall" [8, 13]. In this paper, we show that we can avoid this accelerator bloat by repurposing existing dense matrix accelerators like TPU to execute SpMV.

Cloud computing has raised the interests in *graph analytics*, and example algorithms such as Breadth-First Search, Single Source Shortest Path, PageRank and Collaborative Filtering, can be executed with *sparse matrix-vector multiplication (SpMV)* using sparse matrix represented graphs as inputs [45]. Existing graph frameworks (*e.g.* Ligra) employ compressed sparse data formats, which only store the non-zero entries (NZs) to eliminate any redundant operation involving zero elements [1, 53]. However, the irregular data structure and indirect memory reference pattern make it difficult to vectorize computations for sparse input, and it is made worse if the input sparse data has a skewed distribution. This irregular and high sparsity has become a main obstacle for acceleration on conventional parallel machines, such as GPUs and dense matrix accelerators like Google's TPU.

This paper evaluates repurposing systolic array architectures to execute SpMV efficiently. Two-dimensional systolic arrays, such as Google's TPU, have been proposed for energy-efficient execution of DNNs using dense linear algebra [22, 48]. The advantage of the systolic array is the pipelining of arithmetic operations and the inputs/outputs forwarding as well as the low-overhead PE design. Pipelining and the removal of external memory accesses improve the frequency, while the simplicity of the PE design (mainly a MAC unit with a few registers) lowers the power consumption.

Mapping and executing a sparse matrix on the systolic array is a *compute-latency bound* problem. Naively breaking down a large sparse matrix into tiles of computation tasks and mapping the tasks to the systolic array will inevitably incurs low NZ occupancy, i.e. most PEs are holding zeros which will not contribute to the final results and cannot be bypassed because of the highly regular streaming interconnect. During the execution, the bandwidth and the PEs are not fully utilized, and large amount of PEs only executes a small amount of nonzeros. Therefore, the sparse matrix operation takes an excess number of iterations to complete. In Fig 1, we illustrate the mapping of a 9×9 sparse matrix that has 18 non-zero entries (NZs) onto a 3×3 TPU systolic array. The size of the input matrix is selected to be larger than the array to reflect the real-world large-scale sparse workloads, which often requires multiple iterations ($ITER$) of processing. As shown in Fig 1(b), TPU directly partitions the input matrix based on the shape of the systolic array, and the execution takes nine iterations of processing due to low NZs occupancy. In order to execute sparse operations efficiently,

PACT '22, October 10–12, 2022, Chicago, IL, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9868-8/22/10…$15.00
https://doi.org/10.1145/3559009.3569665

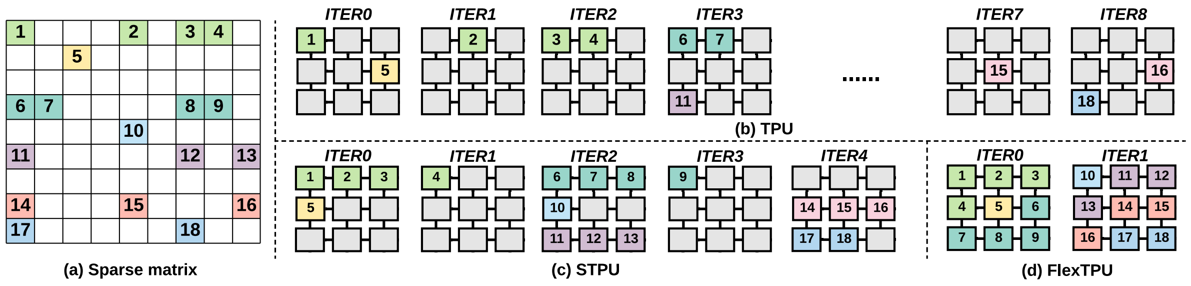


Figure 1: Mapping of (a) a sparse matrix of size 9×9 with 18 NZs to (b) TPU, (c) STPU and (d) conceptual Z-shape mapping by FlexTPU, assuming a 3×3 array size. FlexTPU has the best resource utilization and thus the smallest number of iterations.

the key is an efficient NZ mapping and a dataflow that reduce the number of iterations of execution.

Recent attempts to accelerate sparse operations using systolic arrays include Kung *et al.* [26] (referred to as KMZ) and Sparse-TPU [18] (referred to as STPU). These works preprocess the input sparse matrix and perform sparse column merging (more details are described in Section 2). However, *the major disadvantage* of column merging, which is essential to these prior methods is that the NZs are only condensed in the horizontal direction, leaving PEs still unoccupied when handling very sparse matrices and sparse matrices with skewed NZ distribution. The mapping of STPU on the exemplary matrix is also illustrated in Fig 1(c). STPU performs extensive column merging before the execution, *e.g.* the columns of iteration 0 are composed by the matrix column 0, 2, 4 and 6. STPU improves on TPU and requires five instead of nine iterations to process the matrix. However, as seen in $ITER1$ and $ITER3$, only the top-left PE has a valid NZ because $row0$ and $row3$ of the matrix are the longest rows. And because of the high sparsity, some rows may have fewer NZs which cannot fill a row of PEs in the systolic array, as shown in $ITER0$ and $ITER2$. Power-law matrices/graphs are commonly seen in real world where a small portion of vertices/nodes occupy the most edges in the graph, resulting in an imbalanced row/column length. For example, citation graphs have a small portion of nodes that receive several orders of magnitude more citations than the average nodes. Even with the column merging technique, long columns will delay and dominate the execution time, since STPU can only condense the sparse matrix horizontally.

Another major disadvantage of the column merging is its excessive preprocessing time, because merging sparse columns to minimize the zero-entries is a typical packing problem which has been proved to be NP-complete. For matrices consisting of millions of nodes, iterating and merging numerous columns can only be performed offline, making STPU less appealing to large or non-constant sparse matrices (*e.g.* temporal graphs [39, 50]).

The column merging technique is restricted to map one matrix row onto a row of PEs. To overcome this disadvantage, a straightforward mapping to fully occupy the PE array is to store consecutive NZs from different (short) matrix rows to the PEs in one row. This mapping method demonstrates the potential to handle a sparse matrix with even fewer number of iterations compared with TPU and STPU, regardless of the input size, density, and NZ distribution. To achieve this, three essential questions need to be addressed in order to execute SpMV with this mapping. First, how are index matching, mathematical operations and data forwarding performed in a systolic manner. Second, how to develop the dataflow without changing the pipelined and locally connected nature of a systolic array while keeping the PE design simple. Last but not least, how to construct the condensed format from the conventional compressed sparse matrix representations during execution.

In this paper, we propose FlexTPU, an efficient and flexible framework built around the idea of mapping NZs of a sparse matrix consecutively on a systolic array (*i.e.* Z-shape mapping). The core of FlexTPU is an SpMV dataflow utilizing the proposed mapping. In FlexTPU, PE operations are input-driven and the PEs perform index matching, data registering, and data forwarding accordingly. FlexTPU also proposes a system loader design to realize the end-to-end execution of SpMV, which includes a global matrix row decoder and parallel local NZ loaders. The system loader converts inputs in widely-used conventional compress data representations (*i.e.* CSR) to the proposed mapping, eliminating the need for preprocessing and greatly improving the applicability of FlexTPU.

Specifically, this work makes the following contributions:

(1) We propose an efficient Z-shape mapping that maps the NZs of an input matrix compactly onto a systolic array.
(2) We present necessary changes to repurpose TPU PEs to perform SpMV with the condensed matrix on the array.
(3) We describe a sparse input loader featuring a row decoder and parallel NZ loaders which converts the compressed sparse matrix to the condensed form.
(4) We evaluate FlexTPU and show that FlexTPU improves the performance and energy consumption of STPU by 3.55× and 3.27× across a suite of real-world matrices. FlexTPU also exhibits 5.12× speedup and 2.65× energy improvement over Transmuter, a reconfigurable CMP machine targeting both dense and sparse matrices.

2 BACKGROUND AND MOTIVATION

This section describes the dataflow of dense matrix vector multiplication (DMV) on the systolic array architecture, and illustrates the limitation of the same dataflow for SpMV. A few recent solutions are also discussed to motivate our work.

2.1 DMV and SpMV on systolic arrays

Systolic array consists of a 2-dimensional locally-connected PE array of multiply-and-accumulate (MAC) units and the registers feeding it. To compute DMV, *i.e.* $\mathbf{W}\overline{\mathbf{x}} = \overline{\mathbf{y}}$, where $\mathbf{W}$ is the input matrix, $\overline{\mathbf{x}}$ is the input vector, and $\overline{\mathbf{y}}$ is the output vector, the input matrix $\mathbf{W}$ needs to be loaded into the PE array such that each PE holds a matrix element. The mathematical computation starts when the input vector $\overline{\mathbf{x}}$ is streamed into the PE array from the top edge. Each PE multiplies the incoming vector entry with the locally-stored matrix element, accumulates the result with the partial sum from the left neighbor, and passes the new partial sum to the right neighbor. The output value that comes out of the right-most PE corresponds to an element of the intermediate output vector. If the matrix dimension is greater than the PE array dimension, $\mathbf{W}$ will be partitioned. While the existing dataflow can be applied directly to sparse matrix computation, it comes at a cost of poor hardware utilization. When systolic array performs SpMV, any MAC unit with a zero-valued entry remains idle throughout the entire computation and does not contribute to the final result. Since the majority of the values in a sparse matrix are zeros, mapping a sparse matrix onto a systolic array incurs significant waste of computing resources and memory bandwidth.

2.2 Accommodating SpMV on systolic arrays

A common approach targeting the resource under-utilization of systolic arrays when dealing with sparse linear algebra is to convert the matrix from a sparse storage format into a condensed format before mapping to the systolic array. One of the earliest works that adopt such techniques is done by Tarjan and Yao [46]. Tarjan and Yao proposed a row-interleaved compressed format which merges the NZ elements of different rows of a sparse matrix into a single sparse vector by assigning each row a different offset. However, using this format will lead to datapath conflict in the systolic array.

Recent work attribute the inefficiency of performing sparse operations in the systolic array to the excess amount of zeros mapped onto the array. Targeting convolutional neural networks, Kung *et al.* (KMZ) proposed extensive offline column merging of the sparse filter matrices to condense the matrix [26]. This work is specialized for DNNs, which targets a relatively modest density range ($\sim 20\%$) and leverages the error-tolerance nature of DNNs to tolerate possible NZ collisions. In their design, KMZ employs bit-serial buses to limit the negative impact on the power/area of the parallel buses which deliver multiple integer inputs to a column of PEs. Hence KMZ only supports integer arithmetic, and cannot cater to applications such as scientific computing, which requires computations with high numeric precision.

Inspired by KMZ, STPU eliminates the parallel buses by streaming the inputs sequentially. STPU improves the scalability when handling sparse matrices with a wider density range without compromising accuracy. The proposed key techniques are merging columns at partition-wise and with limited collisions, *i.e.* columns that have NZs with the same index will collide during condensing [18]. The former technique reduces the possibility that two sparse columns have NZs with the same row index (*i.e.* collisions), whereas the latter makes it possible to merge columns that only have a limited number of collisions. Both techniques effectively increase the density of the compressed matrix.

Though the above mentioned works effectively reduce the total number of zeros mapped to the systolic array, they still face certain limitations. First, they are not able to convert the sparse matrix into a completely dense representation, *i.e.* there always exist PEs that store zero elements. Second, the irregular nonzero distribution in real-world matrices can significantly affect the effectiveness of the merging algorithms, due to the existence of extremely dense rows/columns. Most importantly, the matrix compression tends to induce a non-negligible preprocessing overhead. Though most works claim that such overhead can be amortized by repeatedly using the same matrix for computation, the preprocessing cost is still significant compared to the potential performance gains.

In this work, we highlight an efficient Z-shape mapping of a sparse matrix to systolic arrays without excessive preprocessing, which enables the efficient execution of SpMV on systolic arrays.

3 EXECUTING SPMV ON FLEXTPU

In this section, we first present the Z-shape mapping of a sparse matrix and the corresponding dataflow for SpMV. Then we detail the necessary modifications to TPU PEs which enable this dataflow, without affecting the ability to handle dense matrix multiplication.

3.1 The proposed Z-shape Mapping

In contrast with STPU, FlexTPU proposes a Z-shape mapping by arranging the NZs consecutively in a row-major order onto the systolic array. Specifically, the NZs are stored continuously from the top-left PE to the bottom-right PE, forming a Z-shape pattern. We name it "Z-shape" because when the mapping of a NZ in a matrix row reaches beyond the right edge of the systolic array, the NZ will wrap around to the leftmost PEs of the next row of the systolic array. Figure 1(d) shows FlexTPU maps the NZs onto the systolic array without incurring any unoccupied PEs, which only requires two iterations of processing. Note that here we make the assumption that the number of iterations to execute SpMV is the dominant factor in the execution time of SpMV, because TPU, STPU, and FlexTPU exhibit a large difference in N_{iter}, overwhelming the difference in single iteration processing time. Also, the Z-shape mapping in Figure 1 is conceptual and does not show the housekeeping PEs for simplicity.

The detailed Z-shape mapping describes how the row/column ids and values of matrix NZs are stored in the systolic array. To fully define a NZ, a PE can be designed to hold *row_id*, *col_id* and *val* of the NZ as a tuple inside three registers. But the PEs holding the NZs from the same matrix row will store identical *row_id*, wasting register storage. To eliminate the wasted register storage, the PEs utilizing the Z-shape mapping is designed to employ only two registers (W_{idx} and W_{val}), and W_{idx} can be interpreted as *col_id* or *row_id*, depending on the content of W_{val}. The PE that stores the *col_id* and *val* of NZs is referred to as a NORMAL PE, whereas the PE that stores the *row_id* is referred to as a SEPARATOR PE. When the matrix NZs are loaded, the PE would be a SEPARATOR when W_{val} is 0, otherwise it is a NORMAL PE.

Figure 2 shows the detailed mapping, with the SEPARATOR PEs added, of the first iteration of SpMV with the example sparse matrix

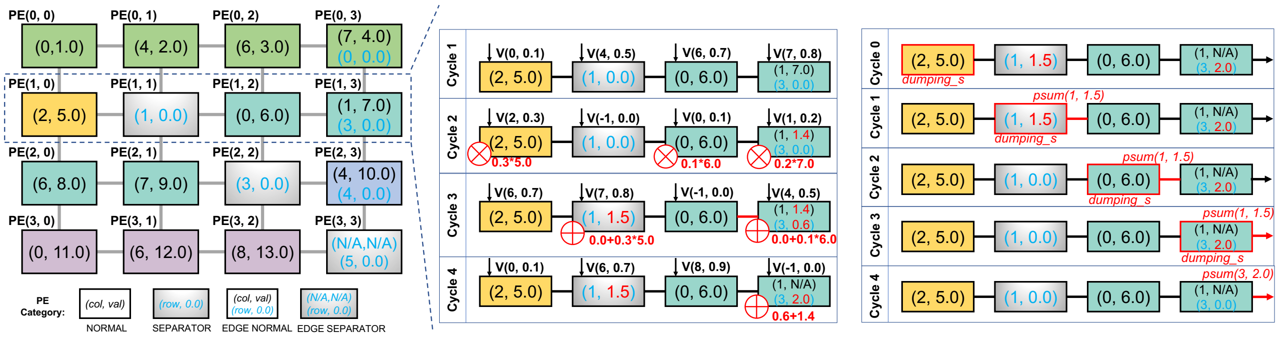


Figure 2: (a) Detailed mapping of the sparse matrix in Figure 1(a) to FlexTPU using a 4×4 systolic array. A NORMAL PE stores the column index and value of an NZ. A SEPARATOR PE stores the row index of the NZ of its left neighbor. The PE at the right edge of the systolic array is referred to as an EDGE PE. It stores an additional copy of the row index of its NZ if it is a NORMAL PE. (b) and (c) show SpMV dataflow of PE(1,0) to PE(1,3) in *computing* phase and *dumping* phase, respectively. In *computing* phase, a NORMAL PE does multiplication upon an index matching and sends the results to its right neighbor. A SEPARATOR (EDGE) PE does accumulation when it receives a value from its left neighbor. In *dumping* phase, SEPARATOR/EDGE PEs send results to their right neighbor but prioritize propagating results from its left neighbor.

mapped on a 4×4 systolic array. The NORMAL PEs holds the *col_id* and *val* of consecutive NZs of the sparse matrix (e.g. the first row of PE), separated by the SEPARATOR PEs to indicate the ending of rows in the Z-shape mapping. The SEPARATOR PE also holds the *p_sum* for accumulating the partial results (e.g. PE (1, 1) with (row_id : 1, p_sum : 0.0)). We also introduce a variant type of PE for the right edge, an EDGE PE, which contains two additional registers, compared to the rest of PEs in the array. The two additional registers keep track of the *row_id*, otherwise FlexTPU will not know which matrix row the NZs stored in PEs left of it belong to, e.g. in the first row of PEs, PE(0, 3) also holds the *row_id* in addition to the *col_id* and *val*. And we refer to this mode or configuration of the EDGE PE as EDGE NORMAL PE. On the other hand, it is also possible that the last NZ of a matrix row is mapped to the second last PE in an array row, in this case the EDGE PE will only record the *row_id* (e.g. PE(3,3)), indicated as the EDGE SEPARATOR PE. In summary, there are two types of PEs STANDARD (not named) and EDGE, each with two modes NORMAL and SEPARATOR.

3.2 SpMV dataflow in a systolic array

This section focuses on the dataflow inside the systolic array. The input/output interface will be presented in the next section. In contrast to the TPU where PEs in the same row perform multiplications and accumulations to generate a single output, FlexTPU can deal with condensed formats and can produce multiple outputs from a single row when NZs from multiple matrix rows are mapped to the same row of PEs. FlexTPU operation consists of two phases: 1) in the *computing* phase PEs perform input matching, conditional nonzero multiplications in the NORMAL PEs, and accumulations of the partial results in the SEPARATOR PEs; and 2) in the *dumping* phase the SEPARATOR PEs forward the results rightward.

***Computing* phase** starts with the input vector elements from the input FIFO streaming down the systolic array. The elements propagate downward one PE per cycle. NORMAL PEs compare the index (X_idx) of the incoming vector element (X_idx, X_val) against its stored *col_id*. When a match occurs, the PE performs multiplication between X_val and its stored *val* and passes the partial result (P_sum) to its right neighboring PE. And the P_sum will be forwarded rightward by other NORMAL PEs and finally get accumulated in the SEPARATOR PE. The *computing* phase is illustrated in Figure 2(b). Taking the second array row in Figure 2(a) as an example, at cycle 2, PE(1, 0), PE(1, 2), and PE(1, 3) receive matching input elements and perform multiplications, and PE(1, 0) and PE(1, 2) pass the products to their rightward neighbors. At cycle 3, PE(1, 1) and PE(1, 3) accumulate and store the incoming products in their *p_sum* registers. At cycle 4, PE(1, 3) accumulates the last partial result. Finally, the partial results are stored in SEPARATOR PE(1, 1) and EDGE PE(1, 3).

***Dumping* phase** is invoked by a *dumping_s* signal to inform the SEPARATOR PEs and EDGE PEs to stream out their stored results. The *dumping_s* signal is propagated like a diagonal wavefront to guarantee the results come out of the systolic array in the correct order. Starting from the top left PE, the *dumping_s* signal is propagated horizontally through PEs in the top row, at the same time the signal is passed down to lower PEs one step per cycle.

When SEPARATOR PEs and EDGE PEs receive the *dumping_s* signal, they send the stored *p_sum* along with *row_idx* rightward. When a PE needs to pass an incoming *p_sum* and its own *p_sum* at the same time, the PE is designed to prioritize passing the neighbor's *p_sum* first to keep the sparse outputs in ascending index order. Figure 2(c) shows the *dumping* phase of the second systolic array row. In cycle 1, PE(1, 1) receives the *dumping_s* signal and sends its *row_id* and *p_sum* tuple (row_id : 1, p_sum : 1.5) rightward. In cycle 2, PE(1, 2) passes the tuple rightward. In cycle 3, the EDGE PE(1, 3) receives the *dumping_s* signal and prioritizes outputting its neighbor's *p_sum* (row_id : 1, val : 1.5). Finally, in cycle 4 EDGE PE(1, 3) outputs its own tuple (row_id : 3, p_sum : 2.0).

Single iteration latency. The dataflow completes when all results are collected from the right edge. The *dumping_s* signal is streamed right after the last vector element from the top-left edge, and the last PE to receive the signal is the bottom right PE. Also, the maximum number of results that reside in a row of PEs is $systolic\_len/2$. Hence the upper bound of single iteration latency is $3 \times systolic\_len + systolic\_len/2$ cycles (448 cycles for a 128×128 systolic array). Since the latency of STPU is 384 cycles per-iteration, FlexTPU has a 16.7% longer latency per-iteration. However, this small overhead is easily offset by the great performance benefits that result from the reduced number of iterations.

3.3 Detailed PE design

Most of FlexTPU's circuitry goes into the array of PEs, and the PE micro-architecture is illustrated in Figure 3. Low-cost PE-to-PE local connections are employed for data passing. The vertical connection transmits the tuple of a vector input (X_idx, X_val) and a 3-bit control signal from top to bottom, whereas one horizontal connection carries the $A_{in}(row\_id, p\_sum)$ rightward while a second horizontal one loads the NZs (tuples of W_{idx_in} and W_{val_in}) from the right edge.

Separate multipliers and adders are used for multiplications and additions in PEs instead of the multiply-and-accumulation (MAC) unit used in the TPU PE. This is because in a single cycle, a PE only performs either a multiplication or an accumulation. Although this separation increases area overhead slightly, it offers a faster maximum operational frequency and lower power consumption. It is worth noting that using the decoupled but pipelined multipliers and adders has negligible impact (one extra cycle overall) on the performance of FlexTPU when executing dense matrix computations like GEMM.

PE control for the sparse and the dense mode. Each PE has a state machine which provides four states to control the actions: *IDLE*, *SPARSE-COMPUTE*, *SPARSE-DUMPING* and *DENSE*. PEs in the *IDLE* state only transmits the partial sum and the matrix NZs. PE in the *IDLE* state will transition to the *SPARSE-COMPUTE* or the *DENSE* state depending on the control signal it receives. When switching from the *IDLE* state to the *SPARSE-COMPUTE* state, the PE checks the received data tuple in the registers to determine the PE mode (NORMAL or SEPARATOR), and operates accordingly. Upon receiving a *dumping_s* signal, NORMAL PEs will transition to *IDLE* state while others will transition to the *SPARSE-DUMPING* state. PEs in the *SPARSE-DUMPING* state will pass the stored p_sum rightward, and once the PEs finish passing their partial sum, they will transition back to the *IDLE* state. PEs in *DENSE* state perform dense matrix multiplication in TPU fashion.

Double buffering. To overlap data loading and execution, after the fashion of the TPU and the STPU, FlexTPU employs double buffering of the matrix NZs inside each PE. Hence, the storage of each PE includes two sets of registers to store NZs, a 2-bit register indicating the validity of each set and an one bit register that indicates which set is being used.

4 SYSTEM ARCHITECTURE

In this section, we demonstrate how FlexTPU functions at system level, *i.e.* how sparse matrices in CSR format can be easily input and output to and from the PE array, which differentiates this work from earlier works that require extensive offline matrix column merging. An overview of FlexTPU is shown in Figure 4(a).

For matrix elements, the system controller instructs the global matrix row pointer decoder and local NZ loaders (*e.g.* 128 in this paper) to read the NZs directly from a CSR-format sparse matrix. Whereas for the vector inputs, the local NZ loaders also request the multi-bank vector buffer to load the vector elements to the vector FIFOs which are attached to every column of PEs. Due to the finite size of the vector buffer which can only store a limited number of inputs, a common practice to deal with very large matrices is partitioning [2, 12, 32, 40, 52]. In this work, FlexTPU repeatedly processes each column partition so that the vector inputs used by each partition can fit into the on-chip buffer.

4.1 System-level overview of FlexTPU

To start data loading (see Figure 4(a)), the controller signals the row decoder and matrix NZ loaders to load NZs into the systolic array and vector FIFOs until the two blocks/iterations of the sparse matrix are loaded. Once one block/iteration of the systolic array and vector FIFOs are loaded, the system controller starts the SpMV computation by instructing the FIFOs to stream in the input elements. The controller waits for the *load_done* signal indicating that the FIFOs have finished loading data into the systolic array. After receiving the *load_done* signal from the vector FIFOs, the controller sends the *dumping_s* signal to inform the SEPARATOR PEs and EDGE PEs to stream out the results.

4.2 Loading a CSR matrix into the systolic array

Matrix row decoder: The matrix row decoder (Figure4(b)) is responsible for specifying the starting address and the number of NZs for the NZ loader by parsing the row pointer array (*indptr*) of the CSR matrix. In the decoder, a row pointer buffer keeps sending memory load requests to fetch the next memory block in the row pointer array as long as there is an empty slot in the buffer. The row decoder works on memory blocks of the row pointer array, and when it finishes using the current block, it will hold the last row pointer and request a new block from the row pointer buffer.

The row decoder monitors the number of PEs (*curPEs*) that have been occupied in the current systolic array row. It can process up to N row pointer values in each cycle, starting from the row indicated by the PTR register, and calculates the total numbers of PEs needed to accommodate each of the N rows. Specifically, for the ith row, we need to consider all of the first i rows in the N rows. The total number of PEs needed are the sum of the number of NZs (NNZ) in row 0 to row $i - 1$ plus the number of non-empty rows to account for the SEPARATOR PEs.

There are three possible conditions when processing the N rows. 1) If all N rows can fit into the current systolic row of PEs, *i.e.* $curPEs + \sum_0^{N-1} NNZ_n + \#rows_N < systolic\_len$, the row decoder will push the row indices and the NNZs of the N rows to the corresponding row NZ loader, increment PTR by N, and accumulate *curPEs* by the number of PEs occupied. 2) If i out of N rows fit and saturate the current row of PEs, *i.e.* $curPEs + \sum_0^{i-1} NNZ_n + \#rows_i == systolic\_len$, the row decoder will push the row indices and the NNZs of the first i rows to the NZ loader, move PTR to row $(i + 1)$, and reset *curPEs*. 3) If row i partially fits

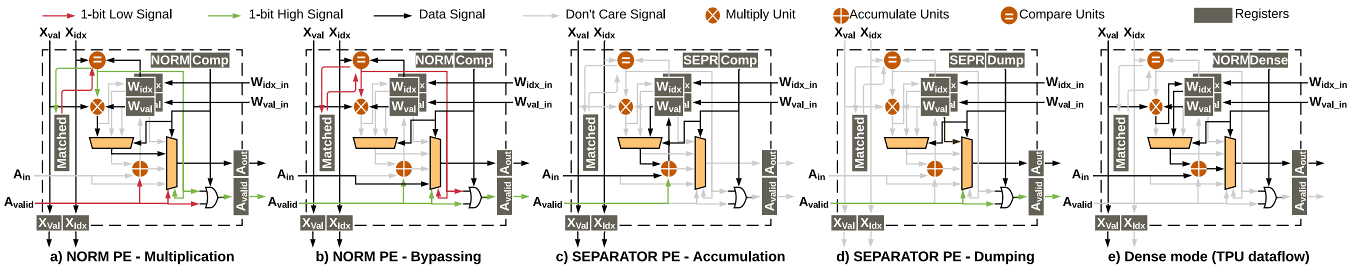


Figure 3: Microarchitecture of a PE and its dataflow in different phases. (a) NORMAL PE doing multiplication upon index matching in *computing* phase. (b) NORMAL PE bypassing values upon index mismatching in *computing* phase. (c) SEPARATOR PE doing accumulation in *computing* phase. (d) SEPARATOR PE propagating results in *dumping* phase. (e) Similar to TPU dataflow, PE computing in dense mode is retained in FlexTPU and the added logic can be clock-gated.

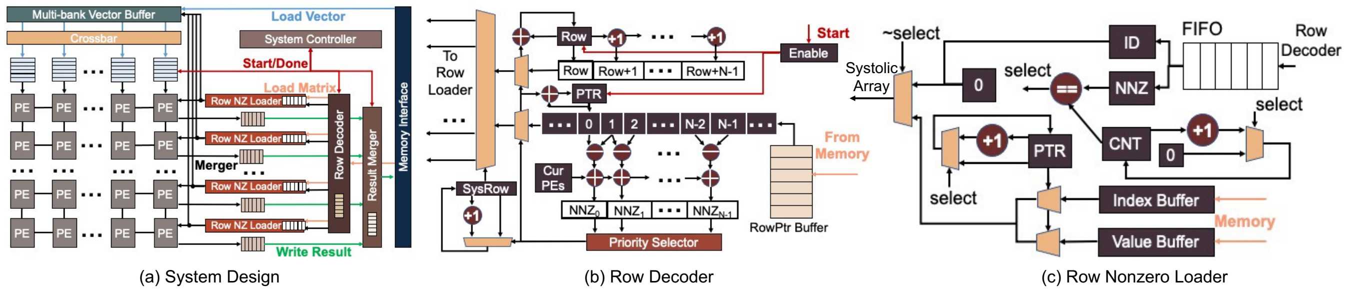


Figure 4: Design of (a) FlexTPU system, (b) the matrix row decoder and (c) the matrix NZ loader. The matrix row decoder and the matrix NZ loaders read the input sparse matrix stored in CSR format off-chip and form the packets on the fly to feed the systolic array. The needed vector elements are loaded into the vector buffer beforehand. The matrix NZ loaders populate the vector FIFOs using the column indices of matrix NZs. The vector FIFOs stream the vector elements downward. The calculated results are sent rightward to the result merger, which merges the partial sums and writes back to the main memory.

in the current row of PEs and overflows to next row of PEs, *i.e.* $curPEs + \sum_0^{i-2} NNZ_n + \#rows_{i-1} < systolic\_len$ and $curPEs + \sum_0^{i-1} NNZ_n + \#rows_i > systolic\_len$, the row decoder will push the row indices and NNZs of row 0 to row $i-1$ to the NZ loader and reset $curPEs$. The row decoder will also move PTR to matrix row $i-1$ and update the row pointer of row $i-1$ to offset the number of NZs used in the current systolic row. In 1), the NZ decoder keeps working for the same NZ loader in the next cycle, whereas in 2) and 3) the NZ decoder starts working for the next NZ loader.

Finally, whenever the row decoder progresses to next systolic row, the updated row pointer index will also be sent to the corresponding NZ loader to form a start address for index/value array.

Matrix NZ loader: A matrix NZ loader is employed for every systolic row and is responsible for populating the PEs in that row and fetching the vector inputs from on-chip buffer. The detailed structure of a matrix NZ loader is shown in Figure 4(c). The NZ loader keeps the row indices and the SEPARATOR information from the matrix decoder in a FIFO, and also uses the address provided by the matrix row decoder to read continuous memory blocks of column indices and values to the index/value buffer. To decide if the next output is for a SEPARATOR or NORMAL PE, the NZ loader also tracks the total number of NZs (CNT in the Figure4(c)) that have been sent to the PE array in the current matrix row. When CNT is equal to the NNZ of the current matrix row, this means all NZs of the current matrix row has been sent and the matrix loader can output the row_idx and p_sum for the SEPARATOR PE, reset the CNT, and pop the FIFO; otherwise the matrix loader will output the col_idx and val from the index/value buffer pointed by the PTR register, and increment CNT and PTR by one.

To make sure the PEs get the correct NZ, the matrix NZ loader use a counter based method to pass data. The destination counter value in the NZ loader is initialized to be $systolic\_len - 1$. Each time the loader sends a matrix tuple, the counter is decremented by one. Along with the matrix tuple, the loader also passes the counter value to the right edge of the systolic array. The PEs will check the counter value when it receives a valid matrix tuple. If the counter value is greater than 0, the PE decrements the counter value and passes the counter value and the matrix tuple to the left neighbor; otherwise, the PE stores the data.

Loading vector FIFOs: To prepare the vector inputs, each time a matrix NZ loader sends a NZ tuple (col_idx, val) to the PE array, it also uses col_idx to request the multi-bank vector buffer to send the vector element into the target vector FIFOs starting from leftmost one. The column coordinate of the destination PE is also sent along with the request to fill the corresponding FIFO. Specifically, the vector buffer is designed to be multi-banked to provides enough bandwidth that match the number of requests received per cycle, which can be as many as the dimension of the systolic array.

4.3 Merging and outputting the results

As shown in Figure 4, the output interface includes the row-wise output buffers and a global result merger. The calculated results will be first sent to the corresponding row-wise output buffer, and then gathered in the global result merger which writes the final results to the main memory.

The size of each output buffer is half the width of the array, matching the maximum number of rows that can be fit in a row of PEs. When an output buffer receives the propagated *dumping_s* signal, it also receives and stores a continuous stream of valid results due to the systolic nature. When an output buffer receives a *popping* signal from its upper fully-drained buffer, it pops the stored results to the accumulator through a pipelined multiplexer tree connected to all the output buffers. This *popping* mechanism starts from the top buffer to the bottom one, which guarantees the results are in ascending order of the indices. Since only one output buffer would be actively popping data, there would be no collision in the multiplexer tree which makes it light-weighted and scalable.

The adoption of the result merger is because, first, a matrix row can span multiple rows of PEs, which produces multiple partial results to be merged, and second, since the input matrix is partitioned vertically for better reuse of the vector elements, the output partial results across different partitions need to be merged. To merge results across consecutive rows of PEs (*i.e.* local merging), the merger checks the incoming outputs with the prior results and performs accumulation if the consecutive results have the same *row_id*. To merge the results across different partitions (*i.e.* global merging), the result merger fetches the previous results from the main memory based on the *row_id* and accumulates the value. After merging, the memory block is written back to the off-chip memory through the memory interface. Since the incoming data is in an ascending order of the indices, a memory block would only be referenced at most once when processing a partition. Thus, no cache is needed and a memory block buffer is enough. Each output buffer is also double-buffered to hide the write back latency.

Table 1: Specifications of CPU/GPU/Transmuter [36].

Platform	Specifications	Library
CPU	Intel i7-6700K, 4 cores/8 threads at 4.2 GHz, 16 GB DDR3 memory @ 34.1 GB/s, 122 mm^2 (14 nm)	MKL 2018.3.222
GPU	NVIDIA Tesla V100, 5120 CUDA cores@1.25 GHz, 16 GB HBM2 memory at 900 GB/s, 815 mm^2 (12 nm)	cuSPARSE [33]
Transmuter	16 tiles of 16 ARM Cortex M4F cores interconnected with non-coherent crossbar at 1 GHz (16 nm)	CoSPARSE [12]

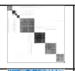
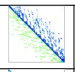
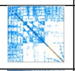
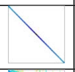
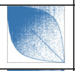
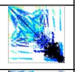
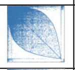
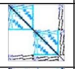
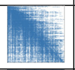
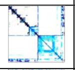
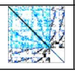
5 EXPERIMENTAL METHODOLOGY

5.1 Simulator and Physical Design

We first compared our proposed FlexTPU against STPU and Google's TPU, assuming a 128 × 128 PE systolic array across all designs. We implemented a custom cycle-accurate C++ simulator for all three designs to model the RTL behavior. To accurately compare both area and power, we modeled the PE designs in FlexTPU, TPU, and STPU using System Verilog. Specifically, we implemented the float32 arithmetic precision for all three types of PEs. And 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. To accurately measure the power consumption, we make extensive use of test vectors to obtain the Switching Activity Interchange Format (SAIF) as input for the gate level synthesized designs. For a fair comparison, we focus on the evaluation of the systolic array, because the memory-array interface design and on-chip memory organization are not detailed in either TPU or STPU.

We also compared FlexTPU against the CPU, the GPU and a recent reconfigurable accelerator, Transmuter [36]. For fairness, we performed a full system implementation of the FlexTPU framework using System Verilog. In addition to the systolic array, this includes the global row decoder (with processing granularity $N = 16$), row loaders, swizzle-switch network-based crossbar distribution networks (128 in × 128 out) [10, 41], on-chip memory with a double buffered vector buffer (128 banks of 256 × 32bit) and vector FIFOs (128 FIFOs of 128 × 64bit), an output unit which has 128 64 × 64bit buffers, and a system controller to orchestrate the execution. An HBM system of 16 channels that is able to deliver up to 128GB/s bandwidth is used. We applied the same methodology to obtain the additional area and power of the full FlexTPU system as above. Area numbers for SRAM buffers/FIFOs are obtained from the TSMC 28nm CMOS LOGIC High Performance Single Port SRAM Compiler. We compare FlexTPU against state-of-the-art library packages on commercial systems, namely, Intel MKL (Version 2018.3.222) on the CPU and cuSPARSE (Version 11.7) on the GPU. We also compared FlexTPU with Transmuter running the CoSPARSE SpMV Framework. Transmuter is a programmable and reconfigurable many-core accelerator, whereas CoSPARSE is an efficient SpMV framework which exploits Transmuter's reconfigurability. The specifications are summarized in Table 1.

Table 2: Matrices from SuiteSparse [9] and SNAP database [4] with their plots, dimensions, number of non-zeros (*nnz*), average nnz per row/column and problem domain.

Matrix	Plot	Dim. *nnz* nnz_{row}	Kind	Matrix	Plot	Dim. *nnz* nnz_{row}	Kind
facebook		4K 88K 21.8	Friend- ship network	cage12		130K 2.0M 15.6	Genomics
nopoly		10.7K 70.8K 6.6	Undirected graph	af23560		23.5K 460K 19.6	Math-matics
slashdot-0902		82.2K 948K 11.5	Friend- ship network	offshore		259K 4.2M 16.3	Electro-magnetic
soc-Epinions1		75.9K 508K 6.7	Trusting network	bcircuit		68.9K 375K 5.45	Circuit simulation
wiki-vote		7.1K 103K 14.6	Voting graph	rajat28		87.2K 606K 7.0	Circuit simulation
flicker-Edges		105K 2.3M 21.9	Patent citation network	scircuit		171K 958K 5.6	Circuit simulation

5.2 Datasets

We used three datasets to evaluate our proposed FlexTPU. The first set comprises synthetic sparse matrices taken from two distinct distributions. One group of matrices are taken from a uniformly random distribution of nonzero elements, and the sizes of the matrices range from 1,024 to 32,768, while the density range is between 3.125E-4 and 0.01. The other group are taken from the power-law distribution, and the matrices are generated by the Stanford Network Analysis Project (SNAP) R-MAT data generator [4]. The generator is configured with default parameters (a: 0.6, b: 0.1, c: 0.15) to generate undirected power-law graphs. The second set is a collection of 12 real-world sparse matrices from SNAP [28] and the SuiteSparse Matrix Collection [9], which covers a wide spectrum of real-world sparse matrices from diverse domains such as social networks, fluid dynamics problems, circuit simulation problems, etc, as detailed in Table 2. The last set includes eight NetworkX-generated synthetic sparse matrices used in CoSparse [12, 16]. Each matrix follows uniformly a random distribution or the power-law distribution and has the same NNZ ($\sim 4M$).

6 EVALUATION

This section first provides a detailed evaluation of the systolic array and compares FlexTPU to STPU and TPU. For the full system implementation, FlexTPU is compared against the state-of-the-art library implementations of SpMV on a CPU and a GPU, and CoSPARSE SpMV framework on Transmuter.

6.1 Power and area analysis of FlexTPU

We report the PE power for each working mode and the PE area of FlexTPU, STPU, and TPU in Table 3. Though the FlexTPU PE introduces additional control logic and registers, the area overhead of FlexTPU compared to that of TPU is 11%, similar to the 12% overhead reported in the STPU. The low overhead is because FlexTPU separates the floating point multiplier and adder with a register rather than using a monolithic multiply-and-accumulate (MAC) unit like STPU and TPU, creating a shorter critical path. This enables the use of slower logic gates to meet the same target frequency, and these slower logic gates incur smaller area and lower power consumption. As shown in the area breakdown, a multiplier and an adder take $600um^2$ less area than a MAC unit. Although the additional registers and logic require $549um^2$ extra area, using a separated multiplier and adder more than compensates for this overhead. We also observe reduced power consumption when the PE performs computations and passes data to its neighbors, which also stems from the shortened critical path of the separated multiplier and adder. It is worth noting that in *DENSE* mode FlexTPU exhibits 5.07% power overhead compared to TPU in dense mode, because of the clock-gating on the added logic, indicating that FlexTPU is still capable to handle dense matrix operations efficiently. We also analyze the power and area of different modules in FlexTPU, as shown in Table 4. Most of the energy and area is consumed by the systolic array, which is the core part of FlexTPU.

Takeaways. FlexTPU applies non-intrusive and low overhead modifications to PEs. FlexTPU can also execute dense operations with minor overhead by clock-gating the inactive logic.

6.2 Comparison against TPU and STPU

6.2.1 Evaluation on scalability. We conduct a thorough evaluation of SpMV with a set of synthetic power-law matrices and uniformly-random matrices. In this scalability evaluation, we fix the size of sparse matrices at 16, 384 and sweep the density from 3.125×10^{-4} to 1×10^{-2}. Also, we fix the total matrix NNZ to be 167, 772 and sweeps the matrix dimension from 1, 024 to 32, 768. The speedup

Table 3: Power and area analysis of PEs in TPU, STPU and FlexTPU (double buffering enabled). FlexTPU introduces less power/area overhead than STPU.

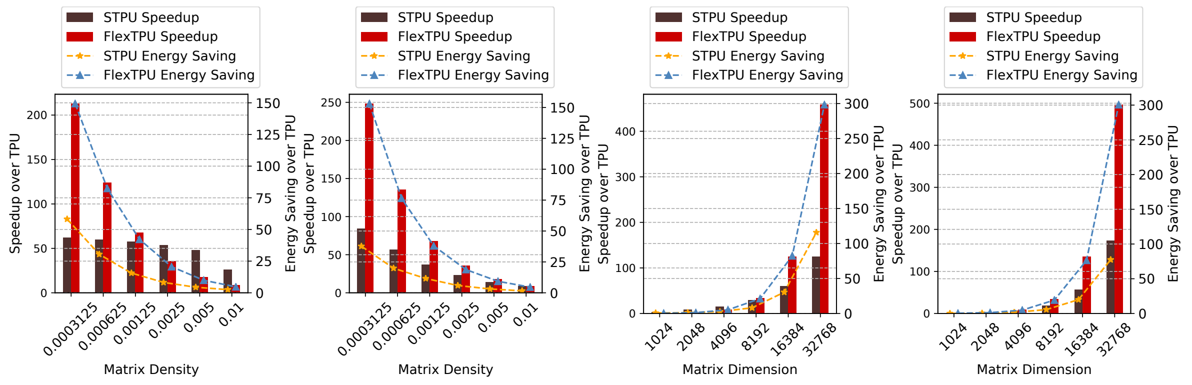
PE Type	Mode	Power (mW)	Area (um^2)	
FlexTPU	*Mult PASS*	1.38	Registers:	889.6
	Add PASS	0.98	MUL+ADD:	3,598.3
	PASS	0.41	Other logic:	717.9
	IDLE	0.04	Total:	5,205.8
	DENSE	2.28		
STPU	*ACCU*	4.31	Registers:	841.6
	HOLD/LATCH	1.23	MAC:	4,197.6
	BYPASS	1.24	Other logic:	216.6
	IDLE	0.05	Total:	5,255.2
TPU	*ACCU*	2.17	Registers:	560.6
	IDLE	0.05	Comb:	4,135.0
			Total:	4,695.6

Table 4: Power and area of main modules in FlexTPU

Components	Area (mm^2)	Power (W)
Systolic Array	85.4 (93.6%)	7.46 (88.4%)
Output Merger/Buffer	1.13 (1.24%)	0.06 (0.71%)
Input Loader	1.19 (1.30%)	0.11 (1.30%)
Vector FIFO	1.24 (1.36%)	0.16 (1.90%)
Vector Buffer	0.90 (0.99%)	0.09 (1.07%)
Input crossbar	1.34 (1.47%)	0.56 (6.64%)
Total	91.2	8.45

and energy saving of STPU and FlexTPU over TPU are shown in Figure 5. For the uniformly random sparse matrices, when fixing the matrix dimensions in Figure 5a, generally both FlexTPU and STPU outperform TPU by achieving 122.3× and 51.3× speedup, and 51.5× and 19.9× energy reduction, respectively. It is worth noting that at a low density level (3.125×10^{-4}) FlexTPU shows 5.37× speedup over STPU. When the density increases to 1.25×10^{-3}, the advantage diminishes to 1.85× over STPU. Eventually, FlexTPU shows 1.84× slowdown to STPU at 0.01 density level. The benefits of FlexTPU over STPU diminishes as matrix density increases due to the following reason. Although STPU performs column packing, the condensed multi-column group can only make use of a limited number of PEs in the systolic array, leading to more processing iterations. In other words, the matrix is only condensed horizontally. In contrast, FlexTPU maps different matrix rows onto the same systolic array row, so FlexTPU actually applies "two-dimensional packing" to an input sparse matrix. But when the density increases the “horizontal” packing in STPU will utilize most PEs in a row. At a 5×10^{-3} density level, 83.5% of the PEs in a row are used on average in STPU. We see a similar trend when fixing the total NNZ in the uniformly random matrices,. FlexTPU and STPU outperforms TPU by achieving 164.3× and 39.9× speedup, and 68.1× and 26.5× energy reduction, respectively. At a low density level, *i.e.* 1.56×10^{-4}, FlexTPU outperforms STPU with 5.80× lower execution time, whereas at a higher density level, *i.e.* 0.01, FlexTPU is 1.12× slower than STPU.

6.2.2 Evaluation on sensitivity to matrix distribution. Figure 5b and 5d show the results for fixing the matrix size and fixing the



(a) Uniformly random mats (Dimension = 16,384) **(b) Power-law mats (Dimension = 16,384)** **(c) Uniformly random mats (# NZ = 167,772)** **(d) Power-law mats (# NZ = 167,772)**

Figure 5: Speedup and Energy saving over TPU of STPU and FlexTPU for sparse matrices with varying size and densities.

total NNZ for power-law matrices. When fixing the size of the matrices and sweeping the density as shown in Figure 5b, FlexTPU and STPU outperforms TPU by achieving 134.6× and 37.4× speedup, and 50.0× and 13.3× energy reduction, respectively. We spot two key differences between the results of uniformly-random matrices and that of the power-law matrices. First, compared to uniformly-random matrices, the speedup of STPU over TPU drops from 51.3× to 37.4× when dealing with the power-law matrices, whereas the speedup of FlexTPU increases from 122.3× to 134.6×. Second, the cut-off density where FlexTPU starts to lose against STPU is shifted to a much higher density level, *e.g.* from around 1.25×10^{-3} to around 0.01. This is because for sparse matrices of the same size and NNZ, power-law graphs usually have a small number of extremely long columns and rows which prevent STPU from mapping the NZs efficiently to the systolic array. By contrast, FlexTPU actually benefits from long rows because fewer SEPARATOR PEs are used. We conclude that FlexTPU handles power-law matrices more efficiently than STPU. Figure 5d shows the results when fixing the total matrix NNZ and sweeping the matrix dimension for the power-law matrices. In this figure, FlexTPU and STPU outperform TPU by a speedup of 177.3× and 43.0×, and an energy reduction of 67.1× and 17.5×, respectively. Similarly, compared to the uniformly random matrices, the cut-off density for power-law matrices shifts from 0.01 to 0.16. To conclude, for very sparse matrices, FlexTPU outperforms STPU in both execution time and energy. In addition, FlexTPU works better for power-law matrices than STPU. These two advantages make FlexTPU a more suitable solution for handling real-world sparse matrices, for example, large-scale graphs that are very sparse and commonly have the power-law distribution.

Takeaways. FlexTPU benefits from condensing the sparse matrix both vertically and horizontally. The results show that FlexTPU outperforms STPU for very sparse matrices. Specifically, for uniformly random matrices with density lower than 1%, FlexTPU achieves higher performance and efficiency. Compared to STPU, FlexTPU also handles power-law matrices more efficiently, as the breakeven density increases by 8-16×.

6.2.3 Evaluation on real-world datasets. Next, we evaluate the speedup and energy saving of FlexTPU and STPU compared to TPU on the set of real-world datasets, which is shown in Figure 6. On average, FlexTPU achieves 531.6× speedup and 202.4× energy saving, and STPU reports 149.8× speedup and 61.8× energy saving over TPU. FlexTPU outperforms STPU by a speedup of 3.55× and an energy reduction of 3.27×. We notice that in the *facebook* matrix, FlexTPU and STPU achieve the smallest speedup (24.38× and 9.42×, respectively) and energy reduction (9.66× and 2.70×, respectively). And compared to STPU, FlexTPU only achieves 5x reduction in the number of iterations, whereas the average reduction of all matrices is 9.56×. This is because *facebook* has a 5.4×10^{-3} density, making it the densest matrix among all the real-world matrices.

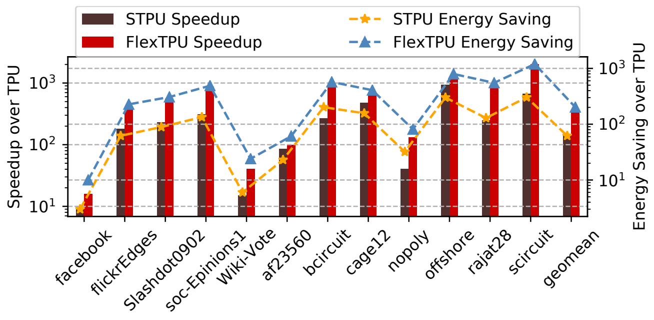


Figure 6: Speedup and Energy saving over TPU of STPU and FlexTPU for real-world sparse matrices.

6.3 Comparison with CPU, GPU and Transmuter

We evaluate the speedup and energy saving of the full system implementation of FlexTPU to a CPU and GPU executing SpMV, as shown in Figure 7. On average, FlexTPU achieves 2.4× speedup and 130.4× energy saving over CPU and achieves 4.3× speedup and 495.3× energy saving over GPU. We also observe that for larger matrices, e.g., *slashdot-0902*, *soc-Epinions1*, *flicker-Edge* and *cage12*, GPU shows higher performance than CPU because higher degree of parallelism can be exploited despite the irregular data accessing/processing pattern and thread divergence. CPU also shows

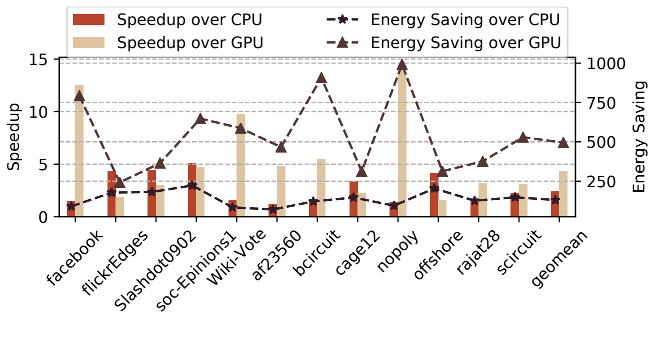


Figure 7: Speedup and Energy saving of FlexTPU over CPU and GPU for matrices in SNAP dataset.

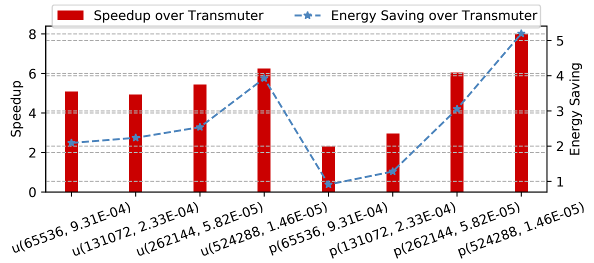


Figure 8: Speedup and Energy saving of FlexTPU against Transmuter (with CoSPARSE SpMV framework) .

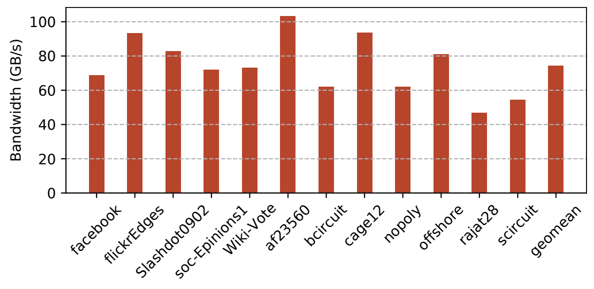


Figure 9: Bandwidth usage of FlexTPU.

high performance on *scircuit* because the structural sparsity can be exploited. For both small graphs like *facebook* and *wiki-Vote* and diagonal matrices like *af23560*, FlexTPU achieves 1.49×, 1.63×, and 1.23× speedup over the CPU, respectively. This is because *facebook* and *wiki-Vote* have the highest density and the smallest size among all the real-world matrices, while *af23560* has NZs concentrated on the diagonal. A CPU with a large on-chip cache can fully exploit the locality to access the sparse matrix and the input vector.

We also compare against the CoSPARSE SpMV framework executed on Transmuter reconfigurable architecture, as shown in Figure 8. We use the same NetworkX synthetic matrices evaluated in CoSPARSE to perform SpMV [12, 16]. A set of uniformly-random matrices (left half) and power-law matrices (right half) with same NNZ (i.e. 4M) are evaluated. On average, FlexTPU achieves 5.12× speedup and 2.65× energy reduction compared to Transmuter. We notice that the advantage of FlexTPU over Transmuter reduces when handling power-law matrices. This is because the existence of dense rows/columns in power-law matrices results in fewer non-empty matrix rows/columns. For Transmuter, fewer input elements are accessed and fewer output vector elements are generated, which are more likely to fit in its on-chip cache.

Finally, we also evaluate the bandwidth usage of FlexTPU across the sparse graphs, as shown in Figure 9. The average bandwidth usage is 74.4GB/s, while the maximum is 103.2GB/s, which can be accommodated by the HBM memory. Compared to STPU, which utilizes around 500GB/s bandwidth on average, FlexTPU's high performance is much less dependent on bandwidth.

7 RELATED WORKS

Though the inner product is the conventional way for matrix multiplication, many alternatives have been explored for sparse matrix multiplications. OuterSPACE [35] accelerates sparse matrix multiplication using an outer product dataflow on a reconfigurable architecture that reconfigures the on-chip memory subsystem based on the memory access pattern for the multiplication and the merge phase of the computation. Sparch [52] improves Outerspace by reducing the off-chip memory transactions for partial sums and augmenting input data reuse. Sadi *et al.* [40] builds a high-performance SpMV accelerator featuring a high-throughput multi-way merge network, a data compression scheme to reduce off-chip traffic, and optimizations for power-law matrices. Matraptor [43] uses row-wise products for SpMM and introduces a new storage format to allow parallel input data streaming. FlexTPU uses TPU as the hardware substrate with minor modification to hardware to relieve the pressure of the accelerator wall. Accordingly, the inner product is used because the dataflow fits naturally on systolic arrays.

Novel sparse formats have been proposed to assist sparse linear algebra acceleration. ExTensor [19] uses hierarchical intersection detection with skip mechanism for fast index matching in sparse tensor algebra. SMASH [23] employs a hierarchical bitmap compression of sparse matrices and designs a lightweight hardware unit for indexing. Tensaurus [44] supports efficient processing of mixed sparse/dense tensor computations by accelerating a common compute pattern with a co-design of hardware and sparse storage format. Alrescha [3] breaks down data-dependent computation in sparse workloads into parallel operations and remaining data dependencies are handled by reconfigurable computation units. Sparse matrices are stored in a custom dense format compatible with the data access pattern. Instead of introducing a new sparse format, FlexTPU directly loads from CSR format and sends the condensed NZs to the systolic array, eliminating expensive preprocessing.

Handling Sparsity on Systolic Arrays. By allowing PEs to pass data directly to neighbors, systolic arrays enable efficient data communication and sharing, and thus have been widely adopted for deep neural network (DNN) acceleration. Recent years have witnessed a growing interest in DNN pruning, which aims to improve DNN performance by getting rid of redundant weight elements to reduce the amount of both data and computation. Much effort has been made in exploiting the sparsity in DNNs [6, 14, 17, 27, 30, 49, 51]. The sparsity and irregularity in DNNs have posed challenges for systolic array based accelerators. Due to the rigid shape and interconnect, systolic arrays suffered from severe resource underutilization and inefficient data loading and accumulation overhead for sparse, irregular DNNs [38]. Kung *et al.* [26] proposed a convolutional neural network (CNN) packing strategy to create a denser format for efficient implementation on a bit-serial systolic array. STPU [18] targets sparse matrix multiplication on systolic arrays with a co-designed approach of first introducing a novel sparse matrix compressing algorithm and then designing a systolic array based architecture to accommodate the computation. Compared to Kung *et al.* and STPU, FlexTPU has negligible preprocessing overhead and focuses on exploring an efficient dataflow to take advantage of the data layout of the CSR format.

Reconfigurable Architecture. Recent reconfigurable accelerators employ a massive amount of PEs interfaced with multi-cast interconnects and reconfigurable on-chip memory to adapt to different applications, incurring performance/power overhead to gain the extra flexibility [8, 15, 20, 25, 31, 34, 36, 37, 47].

8 CONCLUSION

Systolic arrays have been widely adopted for dense linear algebra, but are also known to encounter significant resource underutilization when handling sparse linear algebra. Recent works have proposed several matrix compression algorithms to convert sparse matrices to a denser format and thus improve the utilization rate of PEs. However, such approaches not only induce non-negligible preprocessing overhead, but are also vulnerable to irregular matrix distribution. In this work, we propose a novel mapping of sparse matrices to the systolic array that requires minimal preprocessing and guarantees full utilization of the available PEs regardless of the matrix size, density and distribution. An efficient dataflow and a low-cost PE design are further explored to support sparse matrix computations. The evaluation shows that across a suite of real-world benchmarks, the proposed design, FlexTPU achieves an average speedup of 2.4× and 4.3× and an average energy saving of 130.4× and 495.3× compared to state-of-the-art libraries on a CPU and a GPU, respectively.

ACKNOWLEDGEMENT

The material is based on 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 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] Kadir Akbudak, Oguz Selvitopi, and Cevdet Aykanat. 2018. Partitioning models for scaling parallel sparse matrix-matrix multiplication. *ACM Transactions on Parallel Computing (TOPC)* 4, 3 (2018), 1–34.

[3] Bahar Asgari, Ramyad Hadidi, Tushar Krishna, Hyesoon Kim, and Sudhakar Yalamanchili. 2020. Alrescha: A lightweight reconfigurable sparse-computation accelerator. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 249–260.

[4] D. A. Bader and K. Madduri. [n. d.]. Snap: small-world network analysis and partitioning. http://snap-graph.sourceforge.net

[5] Maria Carla Calzarossa, Luisa Massari, and Daniele Tessera. 2016. Workload characterization: A survey revisited. *ACM Computing Surveys (CSUR)* 48, 3 (2016), 1–43.

[6] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. 2019. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9, 2 (2019), 292–308.

[7] 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.

[8] Vidushi Dadu, Jian Weng, Sihao Liu, and Tony Nowatzki. 2019. Towards General Purpose Acceleration by Exploiting Common Data-Dependence Forms. In *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. ACM, 924–939.

[9] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1–25.

[10] Ronald Dreslinski, Korey Sewell, Thomas Manville, Sudhir Satpathy, Nathaniel Pinckney, Geoff Blake, Michael Cieslak, Reetuparna Das, Thomas Wenisch, Dennis Sylvester, et al. 2012. Swizzle switch: A self-arbitrating high-radix crossbar for noc systems. In *2012 IEEE Hot Chips 24 Symposium (HCS)*. IEEE, 1–44.

[11] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. 2012. Dark silicon and the end of multicore scaling. *IEEE Micro* 32, 3 (2012), 122–134.

[12] Siying Feng, Jiawen Sun, Subhankar Pal, Xin He, Kuba Kaszyk, Dong-hyeon Park, Magnus Morton, Trevor Mudge, Murray Cole, Michael FP O'Boyle, et al. 2021. CoSPARSE: A Software and Hardware Reconfigurable SpMV Framework for Graph Analytics. In *58th Design Automation Conference*. ACM Association for Computing Machinery.

[13] 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.

[14] 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

[15] Venkatraman Govindaraju, Chen-Han Ho, and Karthikeyan Sankaralingam. 2011. Dynamically specialized datapaths for energy efficient computing. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. IEEE, 503–514.

[16] Aric Hagberg, Pieter Swart, and Daniel S Chult. 2008. *Exploring network structure, dynamics, and function using NetworkX*. Technical Report. Los Alamos National Lab.(LANL), Los Alamos, NM (United States).

[17] 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

[18] Xin He, Subhankar Pal, Aporva Amarnath, Siying Feng, Dong-Hyeon Park, Austin Rovinski, Haojie Ye, Yuhan Chen, Ronald Dreslinski, and Trevor Mudge. 2020. Sparse-TPU: Adapting Systolic Arrays for Sparse Matrices. In *Proceedings of the 34th ACM International Conference on Supercomputing (ICS '20)*. ACM, 12 pages.

[19] 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

[20] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F. Martinez. 2007. Core Fusion: Accommodating Software Diversity in Chip Multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture* (San Diego, California, USA) *(ISCA '07)*. ACM, 186–197.

[21] Zhen Jia, Lei Wang, Jianfeng Zhan, Lixin Zhang, and Chunjie Luo. 2013. Characterizing data analysis workloads in data centers. In *2013 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 66–76.

[22] 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

[23] Konstantinos Kanellopoulos, Nandita Vijaykumar, Christina Giannoula, Roknoddin Azizi, Skanda Koppula, Nika Mansouri Ghiasi, Taha Shahroodi, Juan Gomez Luna, and Onur Mutlu. 2019. Smash: Co-designing software compression and hardware-accelerated indexing for efficient sparse matrix operations. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 600–614.

[24] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 158–169.

[25] Martha Mercaldi Kim, John D. Davis, Mark Oskin, and Todd Austin. 2008. Polymorphic On-Chip Networks. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA '08)*. IEEE Computer Society, 101–112.

[26] 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

[27] Ching-En Lee, Yakun Sophia Shao, Jie-Fang Zhang, Angshuman Parashar, Joel Emer, Stephen W Keckler, and Zhengya Zhang. 2018. Stitch-x: An accelerator architecture for exploiting unstructured sparsity in deep neural networks. In *SysML Conference*, Vol. 120.

[28] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[29] Ikuo Magaki, Moein Khazraee, Luis Vega Gutierrez, and Michael Bedford Taylor. 2016. Asic clouds: Specializing the datacenter. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 178–190.

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

[31] Ken Mai, Tim Paaske, Nuwan Jayasena, Ron Ho, William J. Dally, and Mark Horowitz. 2000. Smart Memories: A Modular Reconfigurable Architecture. In *Proceedings of the 27th Annual International Symposium on Computer Architecture* (Vancouver, British Columbia, Canada) *(ISCA '00)*. ACM, New York, NY, USA, 161–171.

[32] Yusuke Nagasaka, Akira Nukada, and Satoshi Matsuoka. 2017. High-performance and memory-saving sparse general matrix-matrix multiplication for nvidia pascal gpu. In *2017 46th International Conference on Parallel Processing (ICPP)*. IEEE, 101–110.

[33] Maxim Naumov, L Chien, Philippe Vandermersch, and Ujval Kapasi. [n. d.]. Cusparse library.

[34] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. 2017. Stream-Dataflow Acceleration. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. ACM, 416–429.

[35] 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.

[36] Subhankar Pal, Siying Feng, Dong-hyeon Park, Sung Kim, Aporva Amarnath, Chi-Sheng Yang, Xin He, Jonathan Beaumont, Kyle May, Yan Xiong, et al. 2020. Transmuter: Bridging the efficiency gap using memory and dataflow reconfiguration. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. 175–190.

[37] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2018. Plasticine: a reconfigurable accelerator for parallel patterns. *IEEE Micro* 38, 3 (2018), 20–31.

[38] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna. 2020. SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 58–70. https://doi.org/10.1109/HPCA47549.2020.00015

[39] Luis EC Rocha and Naoki Masuda. 2014. Random walk centrality for temporal networks. *New Journal of Physics* 16, 6 (2014), 063023.

[40] Fazle Sadi, Joe Sweeney, Tze Meng Low, James C. Hoe, Larry Pileggi, and Franz Franchetti. 2019. Efficient SpMV Operation for Large and Highly Sparse Matrices Using Scalable Multi-Way Merge Parallelization. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 347–358. https://doi.org/10.1145/3352460.3358330

[41] 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.

[42] Yakun Sophia Shao and David Brooks. 2015. Research infrastructures for hardware accelerators. *Synthesis Lectures on Computer Architecture* 10, 4 (2015), 1–99.

[43] 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.

[44] Nitish Srivastava, Hanchen Jin, Shaden Smith, Hongbo Rong, David Albonesi, and Zhiru Zhang. 2020. Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 689–702.

[45] 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.

[46] Robert Tarjan and Andrew Yao. 1979. Storing a Sparse Table. *Commun. ACM* 22, 11 (1979), 606–611.

[47] Michael Bedford Taylor, Jason Sungtae Kim, Jason E. Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffmann, Paul R. Johnson, Jae W. Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matthew I. Frank, Saman P. Amarasinghe, and Anant Agarwal. 2002. The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs. *IEEE Micro* 22, 2 (2002), 25–35.

[48] 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/

[49] Paul N Whatmough, Sae Kyu Lee, David Brooks, and Gu-Yeon Wei. 2018. DNN engine: A 28-nm timing-error tolerant sparse deep neural network processor for IoT applications. *IEEE Journal of Solid-State Circuits* 53, 9 (2018), 2722–2731.

[50] Sijie Yan, Yuanjun Xiong, and Dahua Lin. 2018. Spatial temporal graph convolutional networks for skeleton-based action recognition. In *Thirty-second AAAI conference on artificial intelligence*.

[51] 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

[52] Zhekai Zhang, Hanrui Wang, Song Han, and William J Dally. 2020. Sparch: Efficient architecture for sparse matrix multiplication. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 261–274.

[53] 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.