ASCELLA: Accelerating Sparse Computation by Enabling Stream Accesses to Memory

Bahar Asgari, Ramyad Hadidi, Hyesoon Kim
Georgia Institute of Technology, Atlanta, GA
{bahar.asgari, rhadidi, hyesoon.kim}@gatech.edu

***Abstract*—Sparse computations dominate a wide range of applications from scientific problems to graph analytics. The main characterization of sparse computations, indirect memory accesses, prevents them from effectively achieving high performance on general-purpose processors. Therefore, hardware accelerators have been proposed for sparse problems. For these accelerators, the storage format and the decompression mechanism is crucial but have seen less attention in prior work. To address this gap, we propose Ascella, an accelerator for sparse computations, which besides enabling a smooth stream of data and parallel computation, proposes a fast decompression mechanism. Our implementation on a ZYNQ FPGA shows that on average, Ascella executes sparse problems up to 5.1× as fast as prior work.**

***Index Terms*—Sparse, Stream Memory Access, FPGA.**

I. Introduction

Matrix-vector algebra and more specifically sparse matrix-vector multiplication (SpMV) have a wide range of applications in scientific problems, neural networks, and graph analytics. In most of these fields, the matrices are significantly sparse. Therefore, when running sparse computations on general-purpose processors, most of the time and energy are spent on data movement because of irregular and indirect memory accesses. Several studies have advocated software optimizations for CPUs [1], [2] and GPUs [3]–[5] to facilitate the execution of sparse problems. However, even by employing such optimizations, increasing performance is still limited by power consumption. To address this, hardware accelerators that co-optimize memory accesses and computations are more attractive for accelerating sparse problems [6]–[13].

The source of performance improvements of hardware accelerators for sparse problems is a combination of highly parallel computations and high memory bandwidth utilization. Highly parallel computations are implemented in fine or coarse granularity. For instance, if the matrix operand has blocks of non-zero elements, SpMV operation is divided into many *dense* matrix-vector multiplications, each of which parallelized in fine granularity. Such an approach suites *block-sparse* storage formats. Since such a pattern appears in several applications, the block-sparse storage formats [12], such as blocked compressed sparse row (BCSR), are popular in recent hardware accelerators for sparse problems [9], [10], [12], [13]. Although block-sparse storage formats increase locality in memory accesses, they do not satisfyingly utilize memory bandwidth and footprint, since a percentage of elements in a block are zero. Therefore, alternatively utilizing more aggressive storage formats, such as compressed sparse row and column (CSR and CSC) are used that are favorable for highly scattered sparse data with no pattern in locality. In such a case, the original matrix is parallelized in a coarse-grained manner by splitting into smaller *sparse* operations. While this approach utilizes memory bandwidth more efficiently, it worsens irregular and indirect memory accesses.

The key challenge of preceding approaches is that they either optimize the computation latency or the memory bandwidth in isolation. However, to effectively accelerate sparse problems, *memory streaming and computation must be co-optimized*. To do so, we propose accelerating sparse computation by enabling parallel stream accesses to memory (Ascella). Ascella is the first streaming accelerator for sparse problems, which maintains a balance between compute latency and data transfer by envisioning two insights: (i) using a storage format that from one hand assures streaming only the non-zero values, and from the other hand is easy to decompress; and (ii) proposing a computation engine that follows the speed of memory streaming. To enable the latter, the central building block is optimizing the decompression mechanism, which is often missed by prior work. To address this gap, Ascella *avoids extra accesses* to local buffers, and *provides deterministic parallel accesses to local buffers*. We implement Ascella on a ZYNQ XC7Z020 FPGA. Our results show that the decompression mechanism in Ascella helps executing SpMV up to 5.1× as fast as prior work.

II. Background & Motivation

SpMV can be accelerated by parallelizing its dot products using a multiplier array attached to an adder tree. This paper builds an accelerator based on such a dot-product engine and seeks to efficiently integrate it with stream memory accesses.

Ideal Streaming: To efficiently use a parallel engine to process a stream of data blocks, ideally, the compute time for each block should be the same as data transfer time. In this case, streaming a block and processing it are pipelined by buffering one block. As a result, block $b-1$ (i.e., the previous one) is always processed concurrently while block b is streamed. Therefore, the total latency of SpMV would be:

$$T_{total} = \sum_{b=1}^{blocks} max(T^{b}_{memory}, T^{b-1}_{compute}) \tag{1}$$

in which T^{b}_{memory} is the time to stream a block, and $T^{b-1}_{compute}$ is the time to process the previous block. This paper aims to implement an ideal sparse accelerator on FPGA and uses block RAM (BRAM) for the buffers. However, the proposed ideas are general and not tight to FPGA.

We gratefully acknowledge the support of NSF CSR 1526798.

978-3-9819263-4-7/DATE20/©2020 EDAA

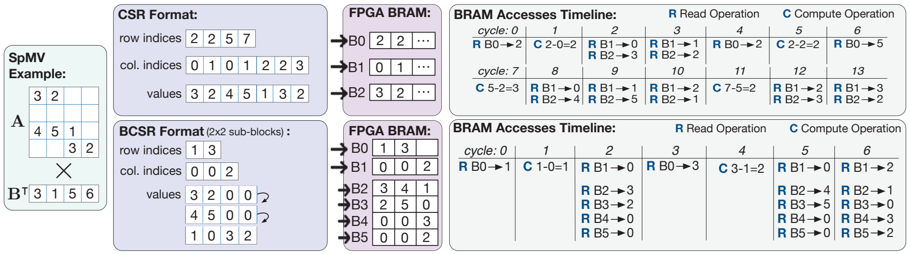

SpMV Example:
A
3 2
4 5 1
3 2
B^T 3 1 5 6
CSR Format:
row indices 2 2 5 7
col. indices 0 1 0 1 2 2 3
values 3 2 4 5 1 3 2
FPGA BRAM:
B0 2 2 ···
B1 0 1 ···
B2 3 2 ···
BRAM Accesses Timeline:
R Read Operation
C Compute Operation
cycle: 0 | 1 | 2 | 3 | 4 | 5 | 6
R B0→2 | C 2-0=2 | R B1→0 R B2→3 | R B1→1 R B2→2 | R B0→2 | C 2-2=2 | R B0→5
cycle: 7 | 8 | 9 | 10 | 11 | 12 | 13
C 5-2=3 | R B1→0 R B2→4 | R B1→1 R B2→5 | R B1→2 R B2→1 | C 7-5=2 | R B1→2 R B2→3 | R B1→3 R B2→2
BCSR Format (2x2 sub-blocks) :
row indices 1 3
col. indices 0 0 2
values 3 2 0 0
4 5 0 0
1 0 3 2
FPGA BRAM:
B0 1 3
B1 0 0 2
B2 3 4 1
B3 2 5 0
B4 0 0 3
B5 0 0 2
BRAM Accesses Timeline:
R Read Operation
C Compute Operation
cycle: 0 | 1 | 2 | 3 | 4 | 5 | 6
R B0→1 | C 1-0=1 | R B1→0 R B2→3 R B3→2 R B4→0 R B5→0 | R B0→3 | C 3-1=2 | R B1→0 R B2→4 R B3→5 R B4→0 R B5→0 | R B1→2 R B2→1 R B3→0 R B4→3 R B5→2

Fig. 1. The time steps for reading a sparse matrix compressed in CSR and BCSR formats and buffered in BRAM. We assume one read per cycle.

Related Work: Storage formats CSR and CSC have been proposed for sparse problems to use memory bandwidth efficiently. However, they result in indirect memory accesses. To relax the indirect memory accesses, BCSR, an extension of CSR is used in recent studies [9]–[13]. BCSR takes advantage of the locality in the non-zero values in the sparse matrix. However, even BCSR cannot satisfy the requirements of an ideal streaming accelerator because its decompression still creates a bottleneck. More specific storage formats such as the diagonal format (DIA) [14] and Ellpack-Itpack (ELL) [15] have been explored for cases when the non-zeros follow a specific pattern. Such formats similarly suffer from the same decompression problem.

The Challenge: Figure 1 uses an example to clarify the decompression mechanisms of CSR and BCSR, the frequently used storage formats. CSR uses three vectors (i.e., row indices, column indices, and values) to represent a matrix. Row indices indicate the number of elements in each row. Thus, for decompressing a non-zero row, we need to first read one element of row indices, and then, we read column indices and values as required. As a result, the key challenge of CSR is that (i) *an overhead of one access to the buffers and one computation is always required for all rows*; and, (ii) to retrieve the column indices and values, *accesses to the buffers are sequential*, because we do not know in advance which elements of column indices and values are going to be accessed. Thus, we cannot partition and allocate those two vectors across the blocks of BRAM to guarantee parallel reads. In summary, the latency to process an $L \times W$ matrix (L: number of rows), is defined by the overhead of accessing the buffer (T_{BRAM}) once per row, and the latency of decompressing the non-zero rows ($nnzr$: # non-zero rows), and applying dot product on them:

$$T^{CSR}_{compute} = L \times T_{BRAM} + \sum_{r=1}^{nnzr} T_{dot} + T^{CSR}_{decomp.}(r) \qquad (2)$$

in which T_{dot} is the latency of the dot product. Because of sequentially reading the elements of a non-zero row, the latency of decompressing a row ($T^{CSR}_{decomp.}(r)$) depends on the number of non-zero elements in that row ($NNZ(r)$) and the latency to decompress a *single* non-zero value ($t^{CSR}_{decomp.}$):

$$T^{CSR}_{decomp.}(r) = t^{CSR}_{decomp.} \times NNZ(r) \qquad (3)$$

If we stream the row indices and column indices using two streamlines in parallel, the longer one defines the latency to stream the $L \times W$ matrix as $T^{CSR}_{memory} = max(NNZ, L) \times t_m$, in which NNZ is the length of column indices, L is the length of row indices, and t_m is the latency of streaming an element.

The decompression mechanism of BCSR is similar to CSR, whereas instead of individual non-zero elements, we decompress the non-zero sub-blocks. In Figure 1, the size of sub-blocks are 2×2. Therefore, the matrix includes three non-zero sub-blocks. Similar to CSR, one access to the buffer per each row of sub-blocks is always required. The advantage of BCSR over CSR is that sub-blocks can be distributed over BRAM blocks to be accessed in parallel. In summary, the latency to process a $L \times W$ matrix partitioned into $l \times w$ sub-blocks is defined by the overhead of accessing the buffer (T_{BRAM}) once per each row of sub-blocks (L/l), and the latency of decompressing all rows of sub-blocks ($NNZR$) and applying dot product to each row of each sub-block:

$$T^{BCSR}_{compute} = (\frac{L}{l}) \times T_{BRAM} + \sum_{R=1}^{NNZR} (l \times T_{dot}) + T^{BCSR}_{decomp.}(R) \quad (4)$$

Note that regardless of the zero rows in the sub-blocks, we always need to perform l dot products for each R. The latency of decompressing a row of sub-blocks ($T^{BCSR}_{decomp.}(R)$) depends on the number of non-zero sub-block in R ($NNZ(R)$) and the latency to decompress a *single* non-zero value ($t^{BCSR}_{decomp.}$):

$$T^{BCSR}_{decomp.}(R) = l \times w \times t^{BCSR}_{decomp.} \times NNZ(R) \qquad (5)$$

As the length of row indices is always shorter than that of column indices and values, the number of sub-block rows ($NNZR$) and the size of sub-blocks define the memory latency as $T^{BCSR}_{memory} = l \times w \times NNZR \times t_m$.

III. Ascella

***Key Insights & Solutions*:** To achieve the ideal streaming accelerator for sparse problems, we propose Ascella, which sustains a balance between computation latency and data transfer rate. To do so, on one hand, Ascella avoids streaming the unnecessary zero elements to efficiently use the memory bandwidth; and, on the other hand, it provides fast computation to keep following the speed of non-stop streaming. To enable the latter, the key insight of Ascella is *avoiding extra accesses* to the buffers, and providing *deterministic parallel accesses* to them, which are the two main obstacles of using typical well-known compressed storage formats (e.g., CSR and BCSR).

To avoid extra accesses to the buffers, and enable deterministic parallel accesses to them, we use list-of-lists (LIL), a storage format supported by SciPy library in Python. Figure 2 clarifies how using LIL reduces the number of cycles to read compressed data and decompress the non-zero rows. As Figure 2 shows, for each column of the original sparse matrix,

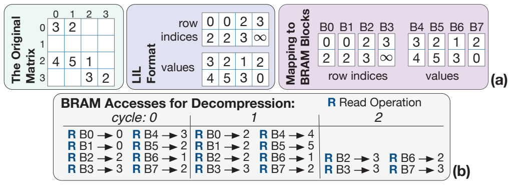


Fig. 2. (a) Compressing a sparse matrix (matrix in Figure 1) using LIL storage format and mapping the row indices and values to BRAM blocks. (b) Time steps of reading row indices and values to decompress the non-zero rows.

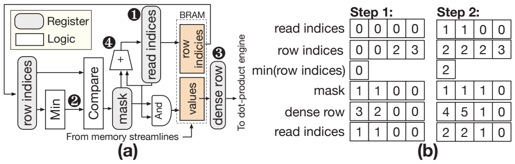
LIL saves a list of row indices corresponding to each non-zero value; as well as a list of all non-zero elements in that column (i.e., *values*). Since the *columns* of values and row indices can always be accessed in parallel, we can map them (the columns) to different blocks of BRAM (e.g., to blocks B0 to B7 in Figure 2). As a result, no extra read is required for determining the number of next read accesses. Thus, the latency of processing an $L \times W$ matrix by using Ascella is:

$$T_{compute}^{Ascella} = nnzr \times (T_{BRAM} + t_{decomp.}^{Ascella} + T_{dot}) + T_{BRAM} \quad (6)$$

in which $nnzr$ is the number of non-zero rows. As the equation shows, creating a non-zero row takes the latency of one BRAM access (since the accesses are inparallel), plus the latency for creating the input of the dot product ($t_{decomp.}^{Ascella}$). To recognize the end of the non-zero rows, the time of one additional BRAM-access is added. Note that $T_{compute}^{Ascella}$ indicates the end-to-end latency, but in reality, T_{dot} can be hidden by pipelining. Memory streaming time for Ascella is defined by the number of non-zero rows ($nnzr$), the size of rows (i.e., W), and transferring one additional row for indicating the end of non-zero rows as $T_{memory}^{Ascella} = (nnzr + 1) \times W \times t_m$.

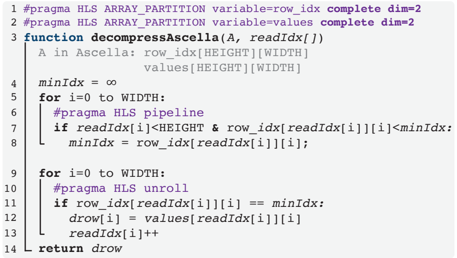
Microarchitecture: To ideally stream data to a parallel dot-product engine, the key component of Ascella is a lightweight microarchitecture for creating a *dense row* (shown in Figure 3a), which connects the streamlines to the dot-product engine. This microarchitecture implements deterministic parallel accesses to BRAM and significantly reduces the decompression latency (i.e., $t_{decomp.}^{Ascella}$ in Equation 6) by just applying a lightweight logical operation (i.e., AND) to generate addresses. At each step of decompression, we use *read indices* to read the *row indices* ❶. The minimum of row indices is used to create a binary *mask*. The *values* corresponding to ones in the *mask* are selected to participate in creating a *dense row* ❸. The mask is also used for updating the *read indices* ❹. Figure 3b illustrates two steps of using the microarchitecture to decompress two dense rows of the example matrix of Figure 2.

Effective Sizing: Because of two reasons, compressing and transferring large units of data (e.g., the entire original matrix) is not beneficial for neither Ascella nor our baselines. First, although all compressed formats eliminate transferring a big portion of zeros, to satisfy the worst case, we need to allocate a sufficient amount of BRAM for the buffers. Second, formats such as CSR transfer one index for all rows, even for the zero rows. Therefore, we apply all the techniques only on the *non-zero blocks* of the original matrices. To choose an appropriated size for such blocks, we explore the percentage of non-zero rows in large (e.g., 8000×8) and small (e.g., 8×8) blocks. Based on our experiments, on average, the percentage of non-zero rows in a 8000×8 is 0.4%, which is approximately 47× as low as that of 8×8 blocks. Therefore, we choose 8×8 (i.e., $W, L = 8$) blocks to apply Ascella, CSR, and BCSR.



	Step 1:				Step 2:			
read indices	0	0	0	0	1	1	0	0
row indices	0	0	2	3	2	2	2	3
min(row indices)	0				2			
mask	1	1	0	0	1	1	1	0
dense row	3	2	0	0	4	5	1	0
read indices	1	1	0	0	2	2	1	0

Fig. 3. The microarchitecture of Ascella for decompression.



```
1 #pragma HLS ARRAY_PARTITION variable=row_idx complete dim=2
2 #pragma HLS ARRAY_PARTITION variable=values complete dim=2
3 function decompressAscella(A, readIdx[])
     A in Ascella: row_idx[HEIGHT][WIDTH]
                   values[HEIGHT][WIDTH]
4    minIdx = ∞
5    for i=0 to WIDTH:
6      #pragma HLS pipeline
7      if readIdx[i]<HEIGHT & row_idx[readIdx[i]][i]<minIdx:
8        minIdx = row_idx[readIdx[i]][i];

9    for i=0 to WIDTH:
10     #pragma HLS unroll
11     if row_idx[readIdx[i]][i] == minIdx:
12       drow[i] = values[readIdx[i]][i]
13       readIdx[i]++
14   return drow
```

Fig. 4. The decompression mechanism of Ascella implemented in HLS.

Implementation: We implement Ascella and the baselines using Xilinx Vivado HLS. We use relevant *#pragrma* as hints to describe our desired microarchitectures in C++. The top functions of all three implementations sequentially stream the compressed non-zero blocks, and for each block, they call SpMV function that iteratively creates dense rows (*drow*) and calls dot product. The decompression function varies based on the storage format. For CSR, since we cannot parallelize the accesses to values and row indices, we apply *pipeline* pragma to their for loop for latency optimization. For BCSR, to make sure that the elements of sub-blocks are accessed in parallel, we completely partition the values across the second dimension of the array before calling the decompression function. By doing that, we unroll the for loop and enable parallel accesses to BRAM blocks. For Ascella (Figure 4), since accesses to the columns of row indices (*row_idx*) and *values* is parallelized, we partition both of them (lines 1 and 2) before calling decompression. The minimum tree for Ascella, and routing *values* from BRAM to *drow* are implemented by *pipeline* (line 6) and *unroll* (line 10) pragma, respectively.

IV. Results

Experimental Setup: We report latency and resource utilization based on the implementation on ZYNQ XC7Z020 FPGA. The baselines and Ascella use similar memory stream interfaces to communicate with an external DDR3 memory, and utilize the same dot-product engine, and, only their decompression logic differ. Inputs and outputs of the accelerators are transferred through the AXI stream interface. The clock frequency is set to 100 MHz. All computations are 32-bit integers. For BCSR, the sub-block size is four (i.e., $l, w = 4$). We run SpMV on various-size matrices from the SuiteSparse matrix collection [16], as listed in Table I, with applications in scientific and graph problems.

TABLE I
MATRICES FROM THE SUITESPARSE MATRIX COLLECTION [16].

Dataset	%Sparsity	Dataset	%Sparsity
2cubes-sphere	0.016	ASIC-100k	0.009
GaAsH6	0.088	hollywood-2009	0.005
kron-g500-logn16	0.11	mono-500Hz	0.05
offshore	0.01	poisson3Db	0.031
road-usa	0.00001	scircuit	0.09
soc-LiveJournal1	0.0029	thermomech-TC	0.07

Performance: The latency of a dot product of size eight ($T_{dot} = 100ns$), streaming four bytes ($t_m = 12ns$), and BRAM access ($T_{BRAM} = 70ns$) are similar for the baselines and Ascella, but, the latency for decompressing a single nonzero differs. Ascella, which concurrently decompresses the elements of a nonzero row, spends $t^{Ascella}_{decomp.} = 15ns$ to decompress an *entire* row, whereas CSR and BCSR spend $t^{CSR}_{decomp.} = t^{BCSR}_{decomp.} = 11ns$ for decompressing a *single* nonzero element, the latency of which are summed up to define the total latency (Equation 3 and 5). Based on the distribution of non-zero values in the input matrices, the latency for streaming various blocks and to process them differs. However, as Figure 5 suggests, the relation between memory latency and compute latency follows a constant pattern, which depends on the decompression mechanism.

As Figure 5a and b show, regardless of the differences in computation and memory access time for CSR and BCSR, their total latency is limited by computation. As Figure 5c shows, using Ascella balances memory and computation latency. Moreover, the storage format of LIL reduces the maximum latency of streaming a block to 800 ns, because it does not transfer extra row indices for zero rows that occur frequently even in 8×8 block size. Note that if we were to simply transfer the dense 8×8 blocks, without any compression, latency would have been dominated by memory latency of $12 \times 64 = 768$ ns per block. Figure 6 shows the total latency of applying SpMV for all datasets normalized to CSR. As Equation 1 shows, the latency of each timestep is defined by the maximum of processing previous block and streaming the current block. Thus, the total latency of BCSR is the highest and that of Ascella is the lowest. On average, Ascella executes SpMV 2.7x and 5.1x as fast as CSR and BCSR, respectively.

Resource Utilization: Figure 7 shows the resource utilization and the absolute size/units for each of the implementations. The utilization of BRAM is defined by the required memory to buffer data and meta-data for one block of size 8×8. In the worst case, CSR requires vectors of size 8, 64, and 64 for buffering row indices, column indices, and values, respectively. Although BCSR in the worst case requires vectors of half size, for row and column indices, and the same-size vector for values, since we partition and map the values to different BRAM blocks, the tool allocates more 18 Kbit blocks for values and one for each of the row and column indices, even though each of the blocks are not fully utilized. Ascella requires two vectors of size 64 to buffer indices and values, both of which are partitioned and allocated to different BRAM blocks. Since CSR does not implement any parallelism, it has the lowest flip-flop and look-up table (LUT) utilization.

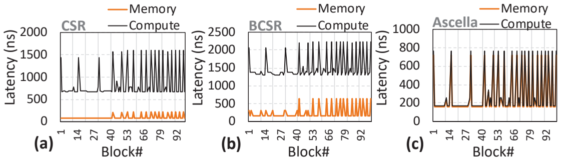


Fig. 5. The latency of memory and computation for a snapshot of non-zero blocks for thermomech-TC dataset: (a) CSR, (b) BCSR, and (c) Ascella.

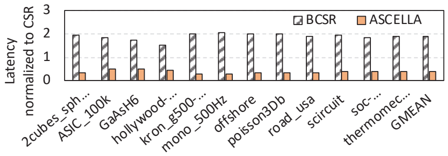


Fig. 6. Total latency normalized to CSR (lower is better).

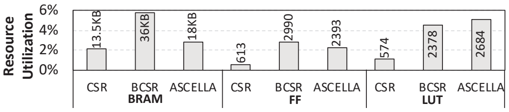


Fig. 7. Resource utilization.

V. CONCLUSIONS & FUTURE WORK

We proposed Ascella, the first streaming accelerator for sparse problems that streams the non-zero rows of sparse matrices and processes them as they come at the same pace. Ascella is a significant step towards accelerating larger scale sparse problems as its storage format facilitates partitioning large matrices and more importantly, it is supported in Python libraries, which makes the implementation straightforward.

REFERENCES

[1] K. Akbudak *et al.*, "Exploiting locality in sparse matrix-matrix multiplication on many-core architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 8, pp. 2258–2271, 2017.

[2] E. Saule *et al.*, "Performance evaluation of sparse matrix multiplication kernels on intel xeon phi," in *PPAM*. Springer, 2013, pp. 559–570.

[3] S. Dalton *et al.*, "Optimizing sparse matrixmatrix multiplication for the gpu," *ACM TOMS*, vol. 41, no. 4, p. 25, 2015.

[4] F. Gremse *et al.*, "Gpu-accelerated sparse matrix-matrix multiplication by iterative row merging," *SIAM*, vol. 37, no. 1, pp. C54–C71, 2015.

[5] W. Liu *et al.*, "An efficient gpu general sparse matrix-matrix multiplication for irregular data," in *IPDPS*. IEEE, 2014, pp. 370–381.

[6] C. Y. Lin *et al.*, "Design space exploration for sparse matrix-matrix multiplication on fpgas," *International Journal of Circuit Theory and Applications*, vol. 41, no. 2, pp. 205–219, 2013.

[7] Q. Zhu *et al.*, "Accelerating sparse matrix-matrix multiplication with 3d-stacked logic-in-memory hardware," in *HPEC*. IEEE, 2013.

[8] S. Pal *et al.*, "Outerspace: An outer product based sparse matrix multiplication accelerator," in *HPCA'18*. IEEE, 2018, pp. 724–736.

[9] B. Asgari *et al.*, "Lodestar: Creating locally-dense cnns for efficient inference on systolic arrays," in *DAC*. ACM, 2019, p. 233.

[10] L. Song *et al.*, "Graphr: Accelerating graph processing using reram," in *HPCA*. IEEE, 2018, pp. 531–543.

[11] B. Asgari *et al.*, "Alrescha: A lightweight reconfigurable sparse-computation accelerator," in *HPCA'20*. IEEE, 2020.

[12] R. W. Vuduc *et al.*, "Fast sparse matrix-vector multiplication by exploiting variable block structure," in *HPCC*. Springer, 2005, pp. 807–816.

[13] B. Asgari *et al.*, "Eridanus: Efficiently running inference of dnns using systolic arrays," *IEEE Micro*, 2019.

[14] Y. Saad, *Iterative methods for sparse linear systems*. siam, 2003.

[15] D. R. Kincaid *et al.*, "Itpackv 2d user's guide," Texas Univ., Austin, TX (USA). Center for Numerical Analysis, Tech. Rep., 1989.

[16] T. A. Davis *et al.*, "The university of florida sparse matrix collection," *ACM TOMS*, vol. 38, no. 1, p. 1, 2011.