MatRaptor: A Sparse-Sparse Matrix Multiplication Accelerator Based on Row-Wise Product

Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang
School of ECE, Cornell University
Email: {nks45, hj424, jl3952, dha7, zhiruz}@cornell.edu

***Abstract*—Sparse-sparse matrix multiplication (SpGEMM) is a computation kernel widely used in numerous application domains such as data analytics, graph processing, and scientific computing. In this work we propose MatRaptor, a novel SpGEMM accelerator that is high performance and highly resource efficient. Unlike conventional methods using inner or outer product as the meta operation for matrix multiplication, our approach is based on row-wise product, which offers a better trade-off in terms of data reuse and on-chip memory requirements, and achieves higher performance for large sparse matrices. We further propose a new hardware-friendly sparse storage format, which allows parallel compute engines to access the sparse data in a vectorized and streaming fashion, leading to high utilization of memory bandwidth. We prototype and simulate our accelerator architecture using gem5 on a diverse set of matrices. Our experiments show that MatRaptor achieves $129.2\times$ speedup over single-threaded CPU, $8.8\times$ speedup over GPU and $1.8\times$ speedup over the state-of-the-art SpGEMM accelerator (OuterSPACE). MatRaptor also has $7.2\times$ lower power consumption and $31.3\times$ smaller area compared to OuterSPACE.**

***Index Terms*—sparse matrix multiplication, sparse formats, spatial hardware**

I. Introduction

Sparse-sparse matrix-matrix multiplication (SpGEMM) is a key computational primitive in many important application domains such as graph analytics, machine learning, and scientific computation. More concretely, SpGEMM is a building block for many graph algorithms such as graph contraction [14], recursive formulations of all-pairs shortest-paths algorithms [9], peer pressure clustering [44], cycle detection [60], Markov clustering [50], triangle counting [5], and matching algorithms [41]. It has also been widely used in scientific computing such as multigrid interpolation/restriction [8], Schur complement methods in hybrid linear solvers [57], colored intersection searching [23], finite element simulations based on domain decomposition [19], molecular dynamics [21], and interior point methods [24].

SpGEMM often operates on very sparse matrices. One example is the Amazon co-purchase network [26], which is a graph where each product is represented as a node and the likelihood of two products being bought together is represented as an edge. This network consists of 400K nodes and 3.2M edges forming an adjacency matrix of 400K $\times$ 400K with a density of 0.002%. With such a high sparsity, the SpGEMM computation becomes highly memory-bound and requires effective utilization of memory bandwidth to achieve high performance. Traditionally, SpGEMM computations have been performed on CPUs and GPUs [12], [38], [51], both of which have low energy efficiency as they allocate excessive hardware resources to flexibly support various workloads. Hardware accelerators tackle this energy efficiency problem through specialization. However, there are three key challenges in designing an accelerator for SpGEMM computation: (1) the inner product and outer product algorithms, which perform well for dense matrix multiplication, are not necessarily the ideal algorithms for SpGEMM due to the low densities of the sparse matrices involved in the computation; (2) the SpGEMM computation is highly memory-bound and the traditional sparse storage formats such as CSR, CSC and COO perform random data accesses when used with multiple compute units in parallel, which results in low memory bandwidth utilization; (3) the output of SpGEMM is also a sparse matrix for which the number of non-zeros are not known ahead of the time, and hence contiguous data allocation for different rows/columns that are being computed in parallel requires synchronization.

In this work, we analyze the different dataflows for SpGEMM and compare them in terms of data reuse and on-chip memory requirements. We argue that row-wise product approach has the potential to outperform other approaches for SpGEMM for very sparse matrices. We further propose a new sparse storage format named *cyclic channel sparse row (C^2SR)*, which enables efficient parallel accesses to the main memory with multiple channels. Using the row-wise product approach and the new sparse format, we describe the design of MatRaptor, a highly efficient accelerator architecture for SpGEMM. According to our experiments on a diverse set of sparse matrices, MatRaptor can achieve significant performance improvement over alternative solutions based on CPUs, GPUs, and the state-of-the-art SpGEMM accelerator OuterSPACE [39].

The key technical contributions of this work are as follows:

(1) We systematically analyze different dataflows of SpGEMM by comparing and contrasting them against data reuse and on-chip memory requirements. We show that a *row-wise product* approach, which has not been explored in the design of SpGEMM accelerators, has the potential to outperform the existing approaches.

(2) We introduce C^2SR, a new hardware-friendly sparse storage format that allows different parallel processing engines

(PEs) to access the data in a vectorized and streaming manner leading to high utilization of the available memory bandwidth.

(3) We design a novel SpGEMM accelerator named MatRaptor, which efficiently implements the row-wise product approach and fully exploits the C^2SR format to achieve high performance. Our experiments using gem5 show that MatRaptor is 1.8× faster than OuterSPACE on average with 12.2× higher energy efficiency. Our accelerator is also 31.3× smaller in terms of area and consumes 7.2× less power compared to OuterSPACE.

II. Analysis of SpGEMM Dataflows

In matrix multiplication, since each of the input matrices can be accessed in either a row-major order or column-major order, there are four possible ways to perform matrix multiplication — inner product (row times column), outer product (column times row), row-wise product (row times row), and column-wise product (column times column) as shown in Fig. 1. In the following subsections, we will discuss each of these four approaches in terms of the data reuse and their on-chip memory requirements. We define data reuse as the number of multiply-accumulate (MAC) operations performed when a single byte of data is read/written from/to the memory. For the sake of simplicity, we assume that: (1) each of the input and the output matrices have dimensions of $N \times N$; (2) both the input matrices have nnz number of non-zeros; (3) the output matrix has nnz' number of non-zeros; and (4) the number of non-zero elements for each row/column are the same and equal to $\frac{nnz}{N}$ for input matrices and $\frac{nnz'}{N}$ for output matrix.

A. Inner Product

This is arguably the most widely-known approach for computing matrix multiplication, where a dot product is performed between a sparse row from the first matrix and a sparse column from the second matrix as shown in Eq. (1). With this approach, we can parallelize the computation of multiple dot products across different PEs. Fig. 1a shows the inner product approach for SpGEMM computation and the parallelization strategy.

$$C[i,j] = \sum_{k=0}^{N} A[i,k] * B[k,j] \tag{1}$$

This approach reads a row of sparse matrix A and column of sparse matrix B each of which has $\frac{nnz}{N}$ non-zeros, and performs index matching and MACs. As the number of non-zeros in the output matrix is nnz', the probability that such index matching produces a useful output (i.e., any of the two indices actually matched) is $\frac{nnz'}{N^2}$. Thus the data reuse for the inner product approach is $O(\frac{nnz'/N^2}{nnz/N})$ which is $O(\frac{nnz'}{nnz} \cdot \frac{1}{N})$. Since N can be very large and nnz' is similar to nnz, the data reuse of inner product approach is very low for large matrices. The on-chip memory requirements for this approach is $O(\frac{nnz}{N})$. Since N can be of the order of $100K-10M$ and $\frac{nnz}{N}$ is of the order of $10-100s$ the data reuse for inner product approach is low; however, the on-chip memory requirements are also low.

Thus, inner product approach has three major disadvantages for SpGEMM:

(1) The two input matrices need to be stored in different formats, one row major and another column major.

(2) It attempts to compute each element of the output matrix. However, in case of SpGEMM, the output matrix is typically also sparse, which leads to a significant amount of wasted computation. For example, in Fig. 1 when the inner product is performed between the last row of matrix A and last column of matrix B, none of the indices match and the output is a zero.

(3) It performs a dot product of two sparse vectors, which is very inefficient since it requires index matching where a MAC is performed only when the indices of the non-zero elements from the two vectors match; for example in Fig. 1a the inner product of first row of A and second column of B requires three index matching operations but performs only one MAC.

B. Outer Product

With this approach, an outer product is performed between a sparse column of the first matrix and a sparse row of the second matrix to produce partial sums for the entire output matrix as shown in Eq. (2). The outer product approach parallelizes the computation of different outer products across different PEs. Fig. 1b shows the outer product approach and the parallelization strategy.

$$C[:,:] = \sum_{k=0}^{N} A[:,k] * B[k,:] \tag{2}$$

This approach reads a column and a row of the sparse input matrices A and B, each with $\frac{nnz}{N}$ non-zeros and performs an outer product with $(\frac{nnz}{N})^2$ MAC operations. Thus the data reuse for outer product approach is $O(\frac{nnz}{N})$. The on-chip memory requirement for outer product approach is $O(\frac{nnz}{N} + nnz')$, where the first term is the on-chip storage required for rows/columns of input matrices and the second term is on-chip requirement for the output matrix. Thus, using typical ranges of $\frac{nnz}{N}$ from 10-100 and for nnz' from 100K-10M, the outer product approach reuses the data read/written from/to the memory 10-100 times; but it requires an on-chip memory size of hundreds of mega-bytes.

The outer product algorithm solves two major problems with the inner product approach by computing only non-zero entries of the output matrix and not performing any index matching while multiplying the inputs. However, it has three major disadvantages:

(1) The two input matrices still need to be stored in different formats, one column major and another row major.

(2) Multiple PEs produce the partial sums for the entire output matrix, as shown in Fig. 1b, which requires synchronization among them. This limits the scalability of the hardware.

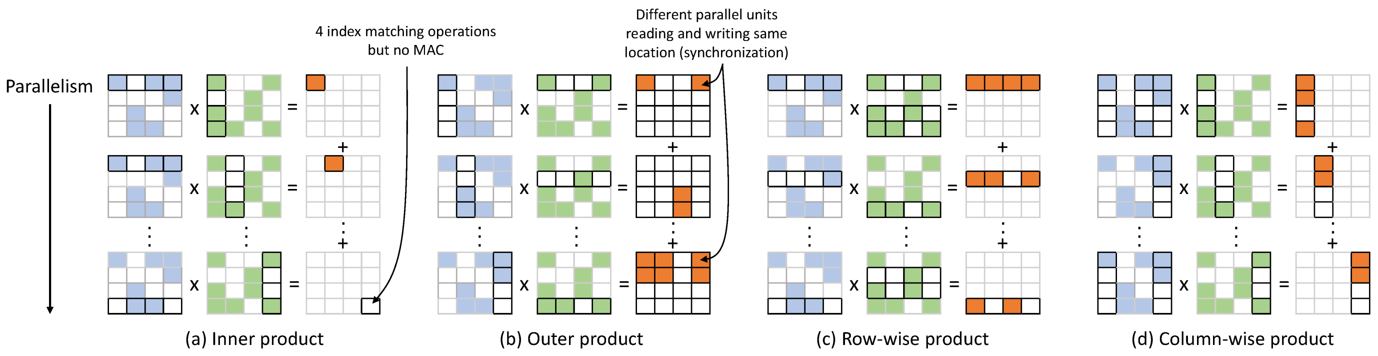

Parallelism
4 index matching operations but no MAC
Different parallel units reading and writing same location (synchronization)
(a) Inner product
(b) Outer product
(c) Row-wise product
(d) Column-wise product

Fig. 1: **Four different ways of computing SpGEMM kernel** — (a) Inner product approach; (b) Outer product approach; (c) Row-wise product approach; and (d) Column-wise product approach. The non-zero elements in the two input matrices are shown in blue and green colors, the non-zero elements in the output matrix are shown in orange color, and the elements of the matrices involved in the computation are shown with dark borders.

(3) For output reuse the partial sums are typically stored on-chip. This requires a large buffer since the partial sums from the entire output matrix are produced for each outer product. For example, OuterSPACE [39], which employs the outer product approach, uses 0.5MB of on-chip memory (256KB of scratchpads within the PEs, 256KB of L0 caches and 16KB of victim caches). Yet it still cannot hold all the partial sums on the chip as the size of partial sums varies from 10-100MB.

C. Row-Wise Product

In the row-wise product approach (also known as Gustavson's algorithm [16]), all the non-zero elements from a single row of matrix A are multiplied with the non-zero entries from the corresponding rows of matrix B, where the row index of matrix B is determined by the column index of the non-zero value from matrix A. The results are accumulated in the corresponding row of the output matrix as shown in Eq. (3). Multiple rows from the output matrix can be computed in parallel. Fig. 1c illustrates the row-wise product approach and the parallelization strategy. Fig. 2 gives a more concrete example which illustrates the computation of SpGEMM with two PEs using the row-wise product approach. Here, each of the two PEs reads entire rows of A and B matrices and writes entire rows of the output matrix C.

$$C[i,:] = \sum_{k=0}^{N} A[i,k] * B[k,:] \tag{3}$$

With this approach, we read a scalar value from matrix A and a row of matrix B with $\frac{nnz}{N}$ non-zeros and perform an scalar-vector product with $(\frac{nnz}{N})$ MAC operations. While the data reuse is low, the on-chip memory requirement for this approach is only $O(\frac{nnz}{N} + \frac{nnz'}{N})$. Here the two terms represent the on-chip storage required by the non-zeros from a row of B and a row of the output matrix, respectively. Thus, using typical ranges of $\frac{nnz}{N}$ and $\frac{nnz'}{N}$ from 10-100, the row-wise product approach only requires an on-chip buffer size of a few kilo-bytes. The key advantages of using row-wise product are as follows:

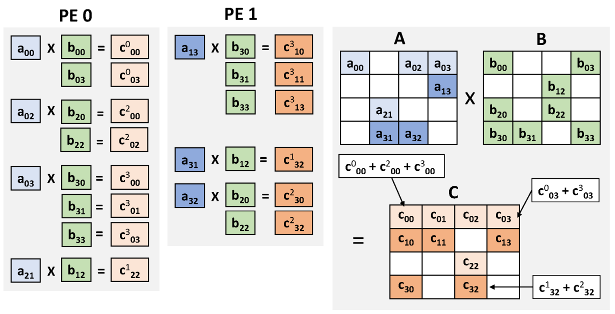

PE 0
PE 1
A
B
C
a_{00} x b_{00} = c^0_{00}
b_{03} = c^0_{03}
a_{02} x b_{20} = c^2_{00}
b_{22} = c^2_{02}
a_{03} x b_{30} = c^3_{00}
b_{31} = c^3_{01}
b_{33} = c^3_{03}
a_{21} x b_{12} = c^1_{22}
a_{13} x b_{30} = c^3_{10}
b_{31} = c^3_{11}
b_{33} = c^3_{13}
a_{31} x b_{12} = c^1_{32}
a_{32} x b_{20} = c^2_{30}
b_{22} = c^2_{32}
$c^0_{00} + c^2_{00} + c^3_{00}$
$c^0_{03} + c^3_{03}$
$c^1_{32} + c^2_{32}$

Fig. 2: **Parallelization of row-wise product on two PEs** — PE0 is assigned to rows 0 and 2 of input matrix A and computes rows 0 and 2 of output matrix C; PE1 is assigned rows 1 and 3 of the matrix A and computes rows 1 and 3 of the matrix C; The matrix B is shared between the two PEs.

- **Consistent Formatting**: Row-wise product accesses both the input matrices and the output matrix in row-major order that allows us to use the same format for the inputs and outputs. Since many algorithms such as graph contractions require a chain of matrix multiplications having a consistent format is essential.
- **Elimination of Index Matching**: The core computation in this approach is scalar-vector product; hence it computes only non-zero entries of the output matrix and does not require inefficient index matching of the inputs as in the case of inner product approach.
- **Elimination of Synchronization**: This approach allows multiple PEs to compute different rows of the sparse output matrix and hence there is no need to synchronize the reads and writes to the output memory.
- **Low On-Chip Memory Requirements** Since a single output row is computed at a time, the required output buffer size is in the order of $O(\frac{nnz'}{N})$. This is contrast with the outer product approach, which requires the entire output matrix to be stored on chip with a buffer size of $O(nnz')$. As N is typically very large (in the order of 100K–10M), the on-chip memory savings from row-

wise product approach over outer product approach are significant.

The row-wise product approach also has some disadvantages: (1) on-chip data-reuse for for matrix `B` is low as compared to outer product; and (2) just like the other approaches row-wise product needs to cope with the load imbalance issue among multiple PEs. The low on-chip data reuse has more impact on the performance when the density of matrix `B` is high. This, however, is not the case with most of the SpGEMM algorithms where both the operand matrices are highly sparse. The load imbalance issue can mostly be solved using a round-robin row allocation strategy discussed in Section IV-A.

D. Column-Wise Product

In column-wise product approach, all the non-zero elements from a single column of matrix B are multiplied with the non-zero entries from the corresponding columns of matrix A and the results are accumulated in the corresponding column of the output matrix as shown in Eq. (4). Fig. 1d shows the column-wise product approach and the parallelization strategy.

$$C[:,j] = \sum_{k=0}^{N} A[:,k] * B[k,j] \tag{4}$$

This approach is similar to the row-wise product approach and has the same data reuse and on-chip memory requirements as in case of row-wise product approach. The rest of the paper focuses on row-wise product approach for SpGEMM as it has low on-chip memory requirements and does not lose much in terms of data reuse compared to the outer product approach, especially for very sparse large matrices.

III. Sparse Matrix Format

By using row-wise product, we can achieve high compute utilization and low on-chip storage requirements. In this section, we further propose a new hardware-friendly sparse storage format to achieve high utilization of the off-chip memory bandwidth, which is essential for high-performance SpGEMM computation. For scalability and high performance, the on/off-chip memories are often divided into smaller memory banks to achieve lower memory latency and higher bandwidth. We abstractly represent such memory banks as *channels* in our discussion and assume that data is interleaved in these channels in a cyclic manner. These channels can be later mapped to different DRAM channels and scratchpad banks.

A. Limitations of CSR Format

Fig. 3 shows the same sparse matrix `A` as in Fig. 2 using its dense form (Fig. 3a) and the conventional compressed sparse row (CSR) format (Fig. 3b). Here the CSR format consists of (1) an array of (value, column id) pairs for the non-zero elements in the matrix and (2) an array of row indices, where the i^{th} index points to the first non-zero element in the i^{th} row of the matrix. In Fig. 3, we illustrate how the (value, column id) array can be mapped to a memory with two channels. Fig. 3e shows how two PEs read the data from matrix `A` and Fig. 3f shows how these PEs write the output matrix `C` using the row-wise product approach depicted in Fig. 2. We assume that the channel interleaving is 4 elements and each PE sends 4-element wide requests (2 non-zero values and 2 column ids) to the memory. As shown in Fig. 3e and Fig. 3f, the CSR format has several major limitations: (1) a vectorized memory request can be split among different channels, leading to non-vectorized memory access within each channel; (2) multiple PEs may send the memory read requests to the same channel resulting in memory channel conflicts; (3) a vectorized memory read can read wasteful data which does not belong to that PE; and (4) for writing the output matrix, each PE writing the i^{th} row to the memory needs to wait for all the PEs writing rows $< i$ to finish, which leads to synchronization issues.

B. The Proposed C^{2}SR Format

To overcome the aforementioned limitations, we propose a new sparse storage called *channel cyclic sparse row* (C^{2}SR), where each row is assigned a fixed channel in a cyclic manner. Fig. 3c shows the cyclic assignment of rows to the channels and Fig. 3d shows the corresponding C^{2}SR format. This new format consists of an array of (row length, row pointer) pairs and an array of (value, column id) pairs. The (value, column id) array stores the non-zero elements from the sparse matrix along with their column ids. The i^{th}entry in the (row length, row pointer) array stores the number of non-zeros in the i^{th} row and the pointer to the first non-zero element from that row in the (value, column id) array. To store a sparse matrix in C^{2}SR format, first each row is assigned to a fixed channel in a cyclic manner and then for each channel all non-zero elements are written serially to the memory locations corresponding to that channel in the (value, column id) array. For example in Fig. 3c, rows 0 and 2 are assigned to channel 0 and hence their non-zero elements are stored at the locations corresponding to channel 0 in the (value, column id) array in Fig. 3d. The reading of matrix `A` and writing to output matrix `C` are shown in Fig. 3e and 3f. The C^{2}SR storage format has the following three key properties:

- **No Channel Conflicts:** Each row is assigned to a unique channel, which means that the rows mapped to different channels do not have memory channel conflicts and can be accessed in parallel. This is in contrast to CSR format, where different rows are not necessarily mapped to distinct channels and result in memory channel conflicts.
- **Vectorized and Streaming Memory Reads:** All the rows mapped to a channel are stored sequentially in that channel, resulting in high spatial locality when accessing these rows in a row major order.
- **Parallel Writes to the Output Matrix:** The rows of the sparse matrix mapped to a channel can be written to that channel without requiring any information about the rows mapped to other channels. For example in Fig. 3f, with C^{2}SR format PE0 and PE1 do not wait for each other and can write to the results to their corresponding channel in parallel. While using CSR format, PE1 needs to wait for

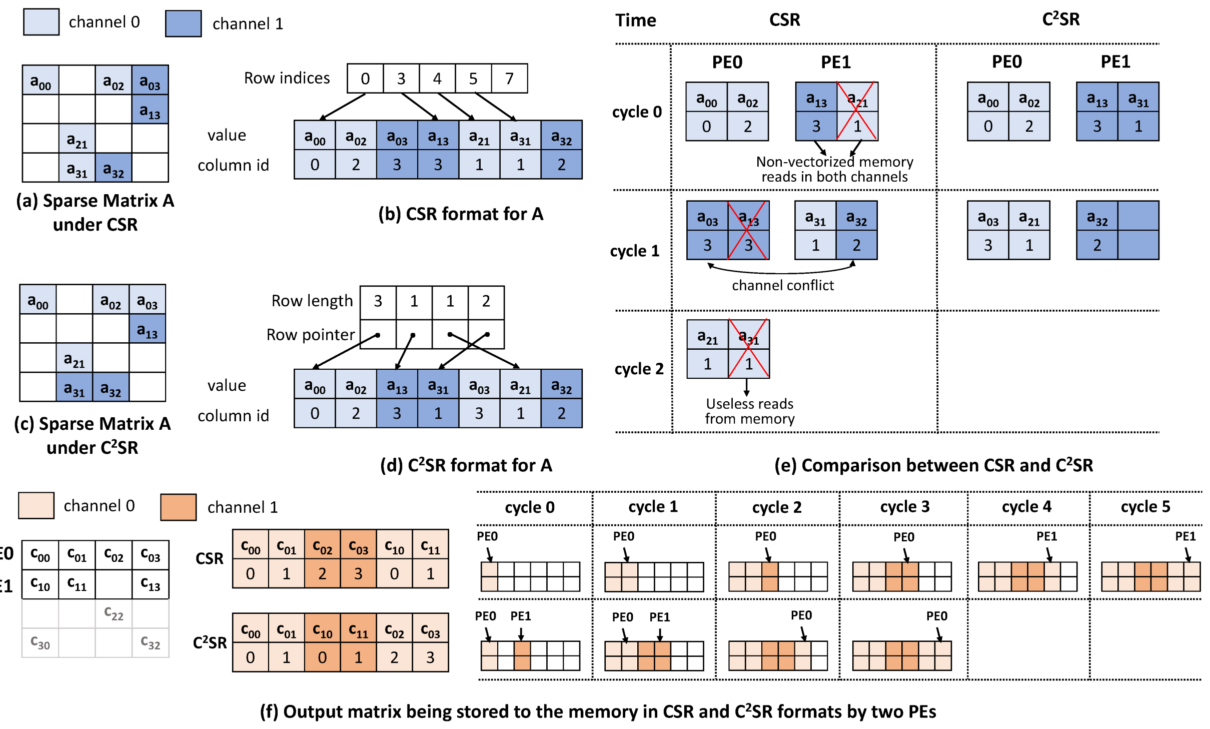

channel 0
channel 1
(a) Sparse Matrix A under CSR
Row indices
value
column id
(b) CSR format for A
(c) Sparse Matrix A under C²SR
Row length
Row pointer
(d) C²SR format for A
Time
CSR
C²SR
PE0
PE1
cycle 0
cycle 1
cycle 2
Non-vectorized memory reads in both channels
channel conflict
Useless reads from memory
(e) Comparison between CSR and C²SR
cycle 3
cycle 4
cycle 5
(f) Output matrix being stored to the memory in CSR and C²SR formats by two PEs

Fig. 3: **Comparison of sparse storage formats** — (a) shows the sparse matrix A in its dense representation; (b) shows matrix A stored in the CSR storage format and assignment of non-zero elements to two channels using 4-element channel interleaving; (c) shows the same sparse matrix A in dense representation where each row is mapped to a unique channel; (d) shows the sparse matrix A in C^2SR format; (e) shows the cycle by cycle execution of two PEs where PE0 and PE1 read rows 0 and 2, and rows 1 and 3 of matrix A, respectively; and (f) shows the output matrix C being written by the two PEs in CSR and C^2SR formats.

PE0 to finish so that it can determine the next available memory location to write the output row.

IV. MatRaptor Architecture

This section details the implementation of row-wise product approach for SpGEMM, which benefits over other conventional SpGEMM methods through: (1) consistent formatting, (2) elimination of index matching, (3) elimination of synchronization, and (4) low on-chip memory requirements. In Section IV-A we first describe the two major operations in the row-wise product implementation: multiplication and merge, where for merge operation an efficient sorting technique using queues is introduced. In Section IV-B, we provide the details of the MatRaptor accelerator architecture.

A. Row-wise Product Implementation

An important consideration for SpGEMM acceleration is to balance load among the PEs. Many real-world sparse matrices follow the power-law distribution, in which some of the rows can be completely dense while others may be almost empty. This can result in load balancing issues in the cases when: (i) different PEs work in lock step, meaning all the PEs finish processing one row each before starting to process the next row; and (ii) the rows of a sparse matrix mapped to one PE have significantly more non-zero elements than the number of non-zeros in the rows mapped to a different PE. MatRaptor solves (i) by allowing all the PEs to work completely asynchronously. Such asynchronous execution is made possible by the C^2SR format, which partitions the address space of the output matrix between different PEs and allows the PEs to independently produce the results of the output matrix. MatRaptor tackles (ii) by doing a round robin allocation of rows to different PEs. This effectively ensures that for the sparse matrices with few high-density regions, the non-zero elements in these regions are approximately uniformly split among different PEs.

Fig. 2 shows the parallelization of SpGEMM in row-wise product approach using two PEs. Here, rows 0 and 2 of the input matrix A are assigned to PE0, which is responsible for computing rows 0 and 2 of output matrix C; rows 1 and 3 of the input matrix A are assigned to PE1, which is responsible for computing the corresponding rows of the output matrix C. The input matrix B is shared between both PEs. The PEs read the input matrices A and B stored in the memory in C^2SR, perform the SpGEMM computation, and write the non-zero elements of the output matrix C to memory using the same C^2SR format.

The SpGEMM computation within a PE consists of two major operations: (a) *multiply operations*, which correspond to the scalar-vector multiplications shown in Fig. 2; (b) *merge operations*, which involves sorting and accumulations. From now onwards, we will use the following notations: (1) The

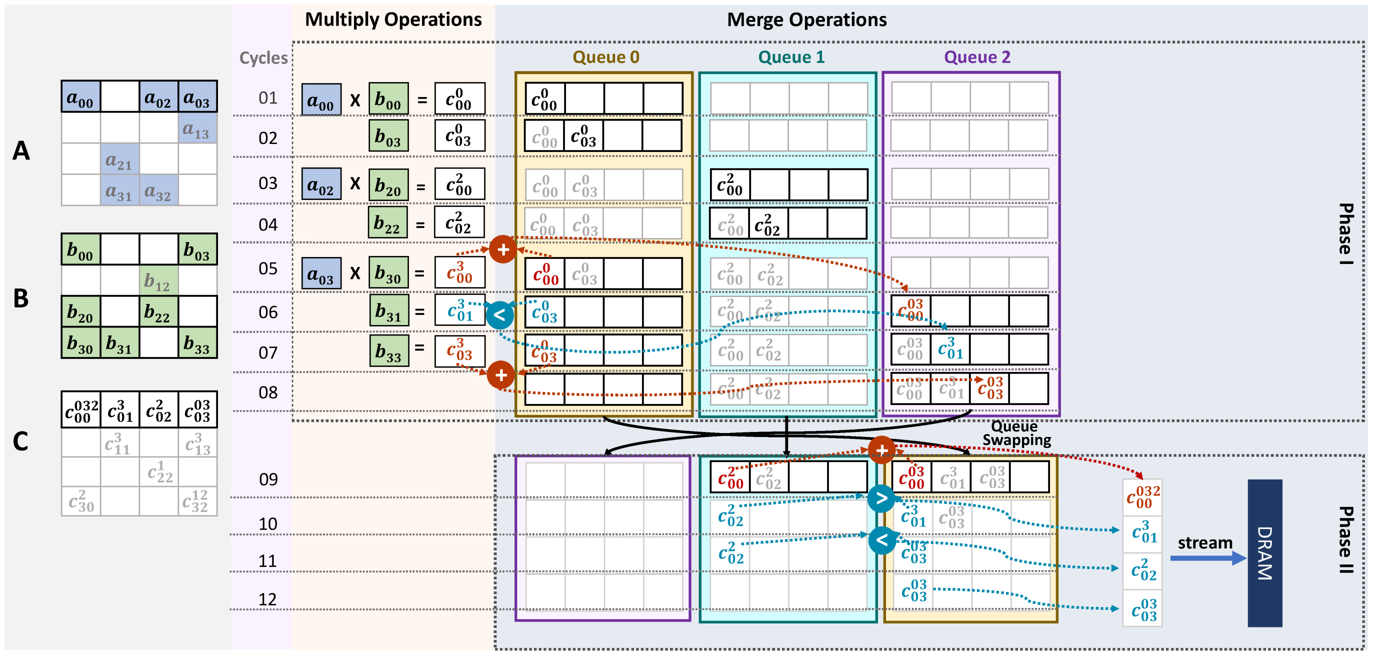


Fig. 4: **Illustration of multiply and merge operations involved in computing the results for a single row of the output matrix** — Phase I corresponds to the cycles when the multiplications are performed and the result of the multiplication is merged with the $(data, col\ id)$ values in one of the queues; and Phase II corresponds to the cycles when the $(data, col\ id)$ values in different queues are merged together and streamed out to the DRAM.

non-zero elements from the input matrices A and B will be represented as a_{ik} and b_{kj} where the subscripts represent the row index and the column index of the matrix elements, respectively; (2) A non-zero element of the output matrix will be represented as either c_{ij}^{k} or $c_{ij}^{k_0,k_1,...k_m}$ where $c_{ij}^{k} = a_{ik} * b_{kj}$ and $c_{ij}^{k_0,k_1,...k_m} = c_{ij}^{k_0} + c_{ij}^{k_1} + ... + c_{ij}^{k_m}$. The subscripts i and j represent the row index and the column index of the non-zero element in matrix C.

Multiply Operations. For the multiply operations, a PE reads non-zero elements a_{ik} from the i^{th} row of matrix A assigned to it. For each a_{ik}, it then reads all the non-zero elements $\{b_{kj_1}, b_{kj_2}, b_{kj_3}, ...\}$ from the k^{th} row of matrix B and performs a scalar-vector multiplication to produce the partial sums $\{c_{ij_1}^{k}, c_{ij_2}^{k}, c_{ij_3}^{k}, ...\}$ for the i^{th} row of matrix C. For example in Fig. 2, PE0 reads the non-zero element a_{00} and the 0^{th} row of matrix B and performs scalar-vector multiplication on $\{b_{00}, b_{03}\}$ to produce the partial sums $\{c_{00}^{0}, c_{03}^{0}\}$ for the 0^{th} row of matrix C.

Merge Operations. The partial sum vectors for the i^{th} row of the output matrix C need to be merged to produce the final results. This requires sorting all the partial sum vectors with respect to their column indices j and then accumulating the partial sums which have the same column index. One naïve solution is to collect all the partial sum vectors and sort them using a sorting hardware. However, such a sorting hardware would be inefficient as it would not make use of the property that each of the partial sum vectors is already sorted with respect to column indices.

A better approach to solve such a sorting problem is by using multiple queues. In this approach, each PE has Q $>$ 2 queues where each queue maintains the invariant that its entries are always sorted with respect to column indices. Out of all the queues, all queues except one act as primary queues while the remaining one acts as a helper queue. For each row of the output matrix, the first (Q$-$1) partial sum vectors are written to one of the primary queues such that there is only one partial sum vector per queue. After the first (Q$-$1) partial sum vectors, each partial sum vector is merged with the queue with the least number of entries and the results are written to the helper queue. While merging, if the column indices in the partial sum vector and the top of the queue match, the entry is removed from both the partial sum vector and the queue and their sum is pushed to the helper queue. If the column indices do not match, then the one (either partial sum vector or queue) with the smaller column index is popped and the value is pushed to the helper queue. After the merge is complete, the helper queue is swapped with the primary queue involved in the merge, and the primary queue becomes the new helper queue. This process continues until the row index of the non-zero element from A changes.

Fig. 4 shows the merge part of the computation with three queues. Initially, the first two queues are the primary ones and the last is a helper, and the partial sum vectors $\{c_{00}^{0}, c_{03}^{0}\}$ and $\{c_{00}^{2}, c_{02}^{2}\}$ are inserted into queues 0 and 1. Then the partial sum vector $\{c_{00}^{3}, c_{01}^{3}, c_{03}^{3}\}$ is merged with queue 0 as it has the least number of entries and the results are written to the helper queue. When the row index of the non-zero element from A changes, then the entries in all the queues need to be merged and written back to the main memory. To merge the data in all the queues, the queue with the smallest column index is popped and the data is streamed to the main memory. In the case when multiple queues have the same minimum column index at the top of the queue, all such queues are popped and the sum of popped elements is streamed to the main memory, as shown in Fig. 4. After the last non-zero element from the first row of matrix A is processed, queue 0 and queue 1 are merged and the results are streamed to the DRAM.

B. Architectural Details of MatRaptor

Fig. 5a shows the micro-architecture of MatRaptor. It consists of Sparse Matrix A Loaders (SpAL), Sparse Matrix B Loaders (SpBL), and the compute PEs. SpALs, SpBLs, and the PEs implement a one-dimensional systolic array with N rows. The rows of the input matrix A are assigned to the rows of the systolic array in a round-robin fashion.

Each SpAL reads the non-zero elements a_{ik} from a row of matrix A and sends it along with its row and column indices to SpBL. SpBL uses the column index k received from SpAL to fetch the non-zero elements from k^{th} row of matrix B (i.e., b_{kj}), and sends a_{ik}, b_{kj}, i and j to the main compute PEs. The PE performs multiplication and merge computations where it multiplies a_{ik} and b_{kj} and merges the results and writes them to the main memory. A crossbar connects the SpALs, SpBLs and PEs to the main memory. Since each SpAL and PE is connected to only one HBM channel, the crossbar is not a full crossbar and its design is much simplified. The following subsections describe each component of MatRaptor micro-architecture in more detail.

Sparse Matrix A Loader. SpAL is configured with the number of rows N in the sparse matrix A and the pointer to the beginning of the arrays containing the (row length, row pointer) pairs in the C^{2}SR storage of matrix A. SpAL first sends a memory read request to fetch the (row length, row pointer) pair for a row of matrix A. Then it sends multiple memory read requests using the row pointer to fetch the (value, column id) pairs in that row. To achieve high memory bandwidth, SpAL sends wide memory read requests for (value, column id) pairs such that size of the memory request is the same as the channel interleaving size and thus implements vectorized memory reads. SpAL also implements outstanding requests and responses queue to avoid stalling for the memory responses and thus is able send and receive memory requests and responses in a pipelined manner. Once a (value, column id) pair is read from the memory, SpAL sends the values along with its row and column indices, namely, (a_{ik}, i, k) to SpBL.

Sparse Matrix B Loader. SpBL receives (a_{ik}, i, k) values from SpAL and sends a memory read request for the (row length, row pointer) pair in k^{th} row of matrix B. It then uses the row pointer to send multiple memory read requests for the (value, column id) pairs in k^{th} row of the B matrix. Similar to SpAL, SpBL also loads the (value, column id) pairs in vectorized streaming manner and maintains outstanding requests and responses queue. When a (value, column id) pair is read from the memory, SpBL sends the values a_{ik}, b_{kj}, i and j to the PE.

Processing Element. Each PE receives (a_{ik}, b_{kj}, i, j) values from SpBL and performs the multiply and merge operations. Fig. 5b shows the design of a single PE. It consists of a multiplier to calculate the product of a_{ik} and b_{kj} and produce the partial sum c_{ij}^{k}. It also consists of two sets of Q queues, where each queue contains $(data, col\ id)$ pairs. The reason behind having two set of queues is that the merge operations in Fig. 4 can be divided into two phases: *Phase I*, when the multiplications are performed and the result of the multiplication is merged with the $(data, col\ id)$ values in one of the queues; and (b) *Phase II*, when all the partial sums for a single output row have been written to one of the queues and the $(data, col\ id)$ values in different queues are merged together and streamed out to the DRAM.

Since Phase II stalls the multiply operations, this can lead to poor utilization of the multipliers. With two sets of queues, when Phase I is completed, the multipliers can start computing the results for the next output row in a different set of queues while the results from the current queues are merged and written to the DRAM. With this kind of double buffering, Phase I and Phase II can be performed in parallel, which results in higher compute utilization.

All the queues within a set are connected to a multiplexer, which is used to select the queue with least number of entries. The queues within a set are also connected to an adder tree and minimum column index selection logic. The minimum column index logic outputs a Q-bit signal where each bit represents whether the corresponding queue has the minimum column index. The output of minimum column index logic is then sent to the controller which configures the adder tree to accumulate the $data$ values from the selected queues. The controller also pops an element from each of these queues. Fig. 5b shows the PE when the set of the queues on the left are involved in Phase I computation and the set of queues on the right are involved in Phase II of the computation. The inactive components from both the sets are shown with gray color and dotted lines.

If the number of rows of the systolic array is an integer multiple of the number of channels, then each row of the systolic array will read/write the elements of matrix A/C from/to a unique channel; however, multiple rows of the systolic array can access the data from the same channel. If the number of channels is an integer multiple of the number of rows of the systolic array, then no two rows of the systolic array will share a channel while a single row of systolic array will be assigned more than one channel. For the cases when the number of rows in the systolic array and the number of channels are the same, each row of the systolic array is assigned one channel.

V. Experimental Setup

To evaluate the performance of MatRaptor, we model our architecture consisting of SpALs, SpBLs, PEs, and HBM using the gem5 simulator [7]. We implement the systolic array with

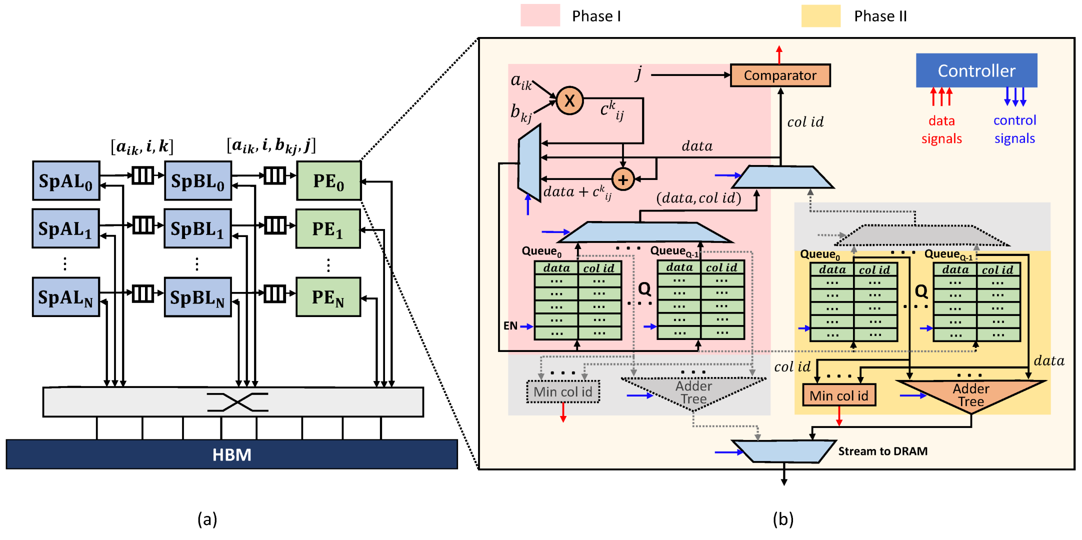


Fig. 5: **MatRaptor architecture** — (a) shows the MatRaptor microarchitecture consisting of Sparse A Loaders (SpALs), Sparse B Loaders (SpBLs), Processing Elements (PEs), system crossbar and high-bandwidth memory (HBM); (b) shows the microarchitecture of a single PE consisting of multipliers, adders, comparator and queues on the left performing phase I of the multiply and merge operations, and the queues on the right, adder tree and minimum column index logic performing phase II of the merge operations.

eight rows to match the number of channels in the HBM. Each PE consists of ten queues which are implemented as SRAMs and where each queue is 4KB in size. We implemented the memory request and response queues with 64 entries. We used the gem5 memory model for HBM, which supports up to eight 128-bit physical channels, runs at 1GHz clock frequency and provides a peak memory bandwidth of 128 GB/s.

We attach MatRaptor to a host CPU as a co-processor where both the CPU and MatRaptor share the same off-chip memory. For the host CPU we use the RISCV minor CPU model in gem5. We add support for a custom instruction `mtx` (move to accelerator) to send messages from host CPU to MatRaptor in the RISCV gcc compiler and gem5 CPU model. The host CPU first sends the pointers of the sparse storage arrays for matrices A and B, and the pointers to an empty storage location for C to the accelerator and then starts the accelerator by writing `1` in x0 configuration register and waits for the accelerator to finish.

A. Measurements

We implemented the PEs and crossbar using PyMTL [32], and performed RTL simulations to validate our gem5 model. We then translated them to Verilog, synthesized them using the Synopsys Design Compiler for synthesis and Cadence Innovus for place-and-route, targeting a TSMC 28nm library. We modeled the latency, area and power of the queues in the merge logic and outstanding request and response queues using CACTI 7.0 [35]. For SpALs and SpBLs, since the area and power are dominated by outstanding memory requests and response queues, we use the area and power numbers for these queues from CACTI and add 10% overhead for the control logic. Table I shows the area and power breakdown of different components of the design. For HBM we use the energy numbers from [45]. Overall the area of our accelerator is $2.2mm^2$, which is $31.3\times$ smaller than the area of OuterSPACE ($70.2mm^2$ after technology scaling). The main reason behind this is our PEs and on-chip memory are much simpler than the PEs, scratchpads and caches in OuterSPACE.

TABLE I: Area and power breakdown of MatRaptor.

Component	**Area** (mm^2)	**%**	**Power (mW)**	**%**
PE	1.981	87.77 %	1050.57	78.11 %
– Logic	0.080	3.54 %	43.08	3.20 %
– Sorting Queues	1.901	84.22 %	1007.49	74.90 %
SpAL	0.129	5.71 %	144.15	10.71 %
SpBL	0.129	5.71 %	144.15	10.71 %
Crossbars	0.016	0.7 %	6.067	0.45 %
Total	2.257	100 %	1344.95	100 %

B. Baselines

We compare our design against three baselines: CPU, GPU, and OuterSPACE [39].

CPU: We use Intel Math Kernel Library (MKL) to evaluate our benchmarks on both single thread and multiple threads (12 threads) of Intel Xeon E5-2699 v4 server-class CPU, which is manufactured using 14nm technology node, runs at 2.20 GHz and has 32 KB L1 cache per core, 256 KB shared L2 and 55 MB of shared L3 caches. We kept the number of CPU threads in the multi-threaded version the same as the one used in OuterSPACE [39] for their CPU baseline.

The CPU uses DDR4 with 4 DRAM channels and supports a peak memory bandwidth of 76.8 GB/s. Since, SpGEMM is primarily memory-bound and the Intel CPU supports a peak memory bandwidth of only 76.8 GB/s while HBM used for MatRaptor has a peak bandwidth of 128 GB/s, we also scale the performance and energy efficiency of the CPU accordingly to 128 GB/s for comparison purposes. For energy estimation of CPU and DRAM, we use the energy numbers from McPAT [28] and [6].

GPU: We use cuSPARSE [38] to evaluate the benchmarks on an NVIDIA Titan Xp GPU, which is manufactured using a 16nm technology node; it uses GDDR5x DRAM with a peak bandwidth of 547.6 GB/s and has a peak 32-bit performance of 12.15 TFLOP/s. We use CUDA 9.1 to program the GPU. Similar to the CPU, we scale the performance and energy numbers of the GPU to 128 GB/s of peak memory bandwidth and use both scaled and unscaled versions for comparisons. We use `nvidia-smi` to measure the power consumption while running SpGEMM benchmarks on the GPU and estimate the power consumption of GDDR5x using [3].

Accelerator: We also compare our work against OuterSPACE, the state-of-the-art SpGEMM accelerator, which uses the outer product approach. We obtained the performance numbers for all benchmarks from the authors of OuterSPACE and used those numbers for comparison. For energy comparisons, we used the power numbers from [39] which are in 32nm technology node and scale them to 28nm.

TABLE II: Matrices from SuiteSparse [11] with their dimensions, number of non-zeros (nnz), density and problem domain.

Matrix	**Dim**	nnz	$\frac{nnz}{N}$	**Density**
web-Google (wg)	916K $\times$ 916K	5.1M	5.6	6.1e-6
mario002 (m2)	390K $\times$ 390K	2.1M	5.4	1.3e-5
amazon0312 (az)	401K $\times$ 401K	3.2M	8.0	1.9e-5
m133-b3 (mb)	200K $\times$ 200K	801K	4.0	2.0e-5
scircuit (sc)	171K $\times$ 171K	959K	5.6	3.2e-5
p2pGnutella31 (pg)	63K $\times$ 63K	148K	2.4	3.7e-5
offshore (of)	260K $\times$ 260K	4.2M	16.3	6.2e-5
cage12 (cg)	130K $\times$ 130K	2.0M	15.6	1.1e-4
2cubes-sphere (cs)	101K $\times$ 101K	1.6M	16.2	1.5e-4
filter3D (f3)	106K $\times$ 106K	2.7M	25.4	2.4e-4
ca-CondMat (cc)	23K $\times$ 23K	187K	8.1	3.5e-4
wikiVote (wv)	8.3K $\times$ 8.3K	104K	12.5	1.5e-3
poisson3Da (p3)	14K $\times$ 14K	353K	26.1	1.8e-3
facebook (fb)	4K $\times$ 4K	176K	43.7	1.1e-2

C. Technology Node Scaling

We scale the energy numbers for all these baselines to 28nm technology node. The dynamic power can be estimated as $\alpha f C V_{dd}^2$, where α is the switching activity, f is the clock frequency, C is the total capacitance and V_{dd} is the supply voltage. As α remains the same between technology nodes, capacitance C scales proportional to the square of Contacted Gate Poly Pitch (CPP) and V_{dd} is different for different technology nodes, we use the ratio of the square of CPP as a scaling factor for C and the ratio of V_{dd} to scale the power and energy numbers. We obtain the CPP and V_{dd} values for different technology nodes from their technical specifications [52]–[55].

D. Datasets

For benchmarks, we used the same matrices as in OuterSPACE [39] which are taken from SuiteSparse [11] as shown in Table II. Since OuterSPACE evaluates the performance of SpGEMM by multiplying a sparse matrix with itself (`C = A×A`), we used the same approach for our evaluation to perform a fair comparison. However, since many of the real-world applications such as graph contractions involve the sparse matrix multiplication of two different matrices, we did a performance evaluation by using different combinations of `A` and `B` matrices from Table II. We selected the top-left 10K $\times$ 10K submatrices from all the matrices in Table II so that the two matrices have same size but different sparsity structure representative of the real matrices. This technique has been adopted from a prior work on characterizing SpGEMM performance on GPUs [25].

VI. Evaluation

A. Bandwidth Utilization

To compare the bandwidth utilization of CSR and C^{2}SR we simulated 2, 4 and 8 PEs reading a sparse matrix from the memory in a streaming fashion. For CSR, we assumed that each PE reads 8-byte data elements from the memory to avoid sending vectorized memory requests which map to different channels and cause memory alignment problem. For C^{2}SR, each PE sends a 64-byte wide streaming memory requests. For all the simulations we assumed the number of PEs to be the same as the number of DRAM channels. Fig. 6 shows the achieved bandwidth with CSR and C^{2}SR formats. As it can be seen from the figure, the achieved bandwidth from C^{2}SR format is higher than the achieved bandwidth from CSR and is also close to the theoretical peak memory bandwidth.

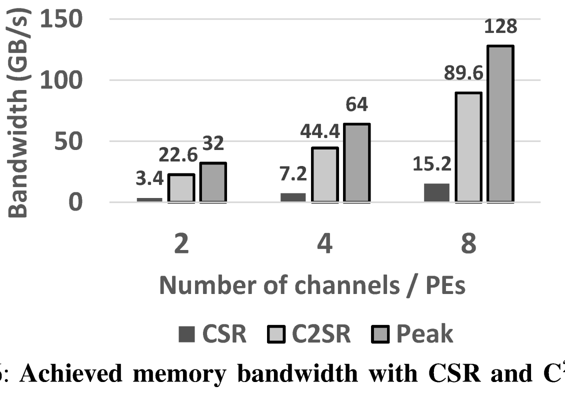


Fig. 6: **Achieved memory bandwidth with CSR and C^{2}SR** — here the PE count equals the number of channels.

B. Roofline Evaluation

Fig. 7 shows the throughput of SpGEMM under the roofline [56] of MatRaptor. The horizontal line towards the

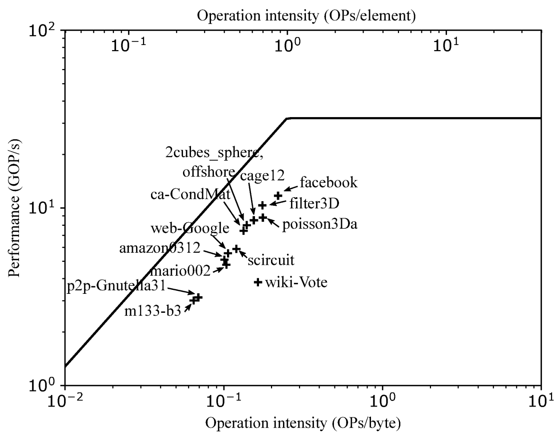


Fig. 7: **Performance of SpGEMM under the roofline of MatRaptor for A×A** – Throughput(T) = Operation Intensity(OI) × Bandwidth(B). Thus, log(T) = log(OI) + log(B), which is the equation of a line, y = mx + c with y = log(T), x = log(OI), m = 1 and c = log(B). Thus, the y-axis in the roofline is throughput in log domain (log(T)), x-axis is operation intensity in log domain (log(OI)), the slope of the slanted line is 1 (m=1) and the intercept of the slanted line (value at OI = 1) is the off-chip memory bandwidth in GB/s i.e. 128). The horizontal line on the right is of the form y = log(Tmax) where Tmax is the maximum attainable throughput i.e. 32 GOP/s in our case.

right of the plot shows the peak attainable performance from the design when the operation intensity is high (kernel is compute bound) and the inclined line (with slope 1) towards the left shows the peak attainable performance when the operation intensity is low (kernel is memory bound). The gap between the roofline and the achieved performance of a kernel indicates the inefficiencies within the hardware and the algorithm. Our design consists of 8 PEs, each with one MAC unit and hence our design has $8 \times 2 = 16$ multipliers and adders. Since we simulate our design for a 2GHz clock frequency the peak attainable throughput is $16 \times 2 = 32$ GOP/s. For peak memory bandwidth, we use the peak bandwidth of HBM which is 128 GB/s.

Fig. 7 shows the achieved throughput for SpGEMM for A × A computation for all the matrices in Table II. It can be seen from the roofline, the throughput for each of the benchmark is close to the peak performance and all the benchmarks lie in the memory bound region. The gap between the peak performance and the attained performance is due to the memory accesses to the matrix B. As in row-wise product approach only matrix A and the output matrix C are partitioned among different PEs while the matrix B is shared between different PEs, this results in memory channels conflicts and lowers the achieved memory bandwidth.

C. Performance Evaluation

Fig. 8a shows the performance comparison of CPU (single-/multi-threaded and without/with bandwidth normalization), GPU (without/with bandwidth normalization), OuterSPACE [39] and MatRaptor over the single-threaded CPU baseline for A × A SpGEMM computation. As it can be seen from the figure MatRaptor outperforms CPU and GPU for all the benchmarks. Compared to OuterSPACE the performance of MatRaptor is better for all the benchmarks except Wiki-Vote, for which the performance of the two are very similar. The main reason behind this is that Wiki-Vote has smaller size compared to other matrices and the on-chip memory for the output matrix in OuterSPACE is sufficient to store the partial sums, which results in similar performance of OuterSPACE and MatRaptor. In terms of geometric mean speedup, MatRaptor achieves 129.2×, 77.5×, 12.9×, 7.9×, 8.8×, 37.6 and 1.8× speedup over single-threaded CPU without and with bandwidth normalization, multi-threaded CPU without and with bandwidth normalization, GPU without and with bandwidth normalization, and OuterSPACE, respectively.

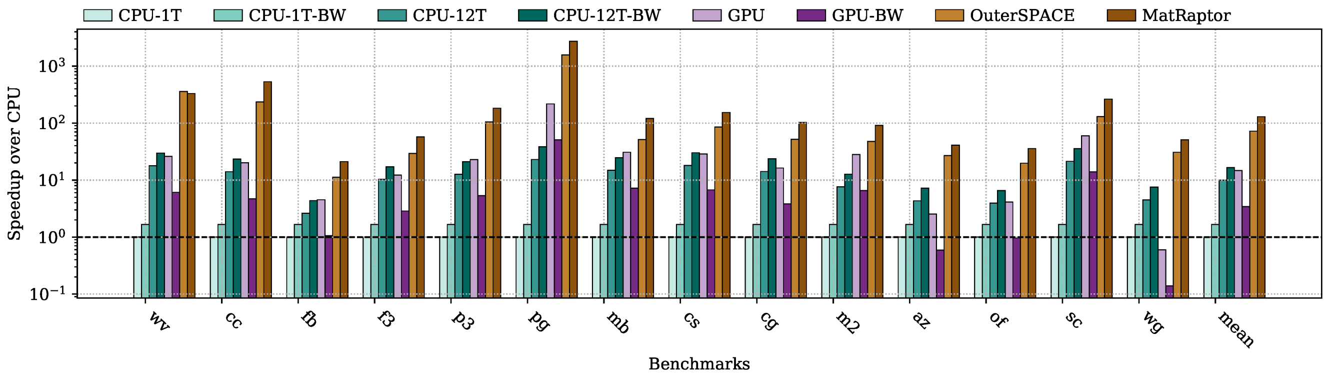
Fig. 9 provides a breakdown that shows (1) the number of cycles when the multipliers in the PEs are active, and (2) the stall cycles due to merge and memory access. For almost all the benchmarks there are stalls due to merge, which is expected as the merging takes more time than multiplications. We also measured the ratio of cycles spent in phases I and II. This ratio varied in the range of [2, 15], as most of the merge logic is in phase I. But since in some cases, phase II takes as long as 50% of the time spent in phase I, we opt to pipeline these two phases using a double buffer. Fig. 10a shows the speedup of MatRaptor over GPU with bandwidth normalization for A × B SpGEMM computation. It can be seen from these figures that for GPU the performance of MatRaptor is better in all the cases. Overall MatRaptor achieves 26.8× speedup over GPU with bandwidth normalization for A × B computation.

D. Energy

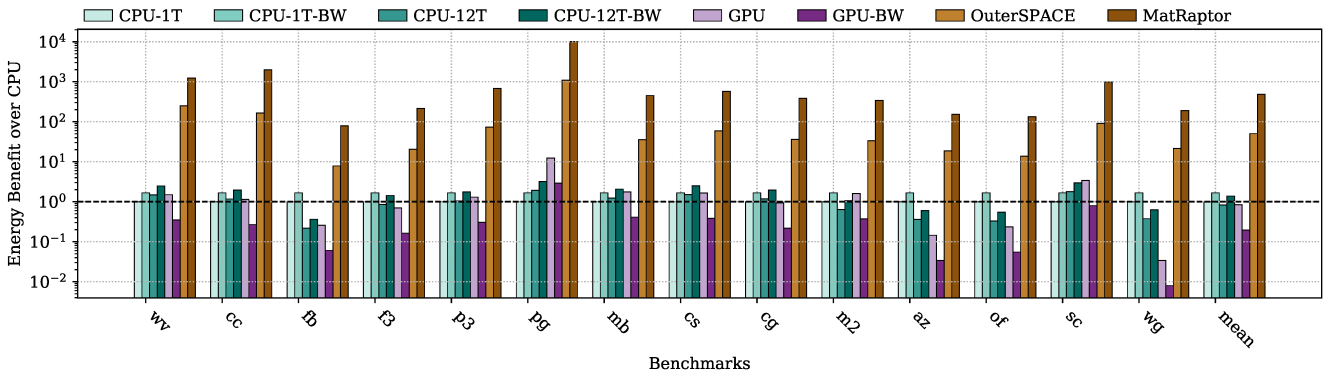
Fig. 8b shows the energy comparison of CPU (single-/multi-threaded and without/with bandwidth normalization), GPU (without/with bandwidth normalization), OuterSPACE [39] and MatRaptor for A × A SpGEMM computation. In terms of geometric mean energy benefit, MatRaptor achieves 482.5×, 289.6×, 581.5×, 348.9×, 574.8×, 2458.9× and 12.2× energy benefit over single-threaded CPU without and with bandwidth normalization, multi-threaded CPU without and with bandwidth normalization, GPU without and with bandwidth normalization, and OuterSPACE, respectively. Fig. 10b shows the energy benefit of MatRaptor over GPU with bandwidth normalization for A × B SpGEMM computation, where MatRaptor achieves 1756.5× improvement in energy.

E. Load Imbalance

To measure the load imbalance due to the power-law distribution of the sparse matrices as discussed in Section IV-A, we determine the total number of non-zeros of matrix A assigned to each PE by C^2SR format and plot the ratio of maximum and minimum number of non-zeros in these PEs. The minimum value of such ratio is 1, which means no load imbalance and a higher ratio means a higher load imbalance. Fig. 11 shows the



(a) Speedup



(b) Energy benefit

Fig. 8: **Speedup and energy comparison for A×A** — CPU-1T = single-thread CPU; CPU-1T-BW = single-thread CPU with bandwidth normalization; CPU-12T = CPU with 12 threads; CPU-1T-BW = CPU with 12 threads and bandwidth normalization; GPU; GPU-BW = GPU with bandwidth normalization; OuterSPACE and MatRaptor. All the speedup and energy benefit numbers are relative to the CPU-1T baseline. The `mean` is the geometric mean of the speedups and energy benefits for different benchmarks.

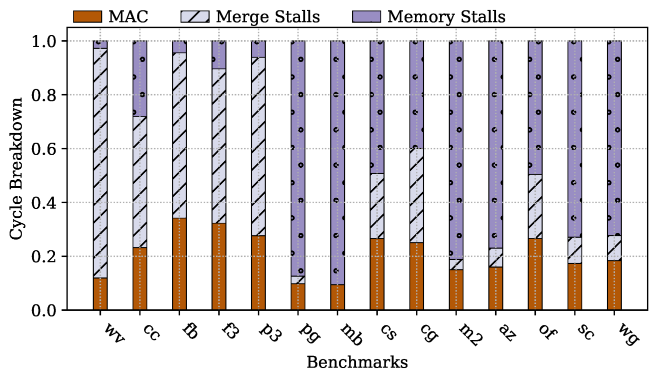
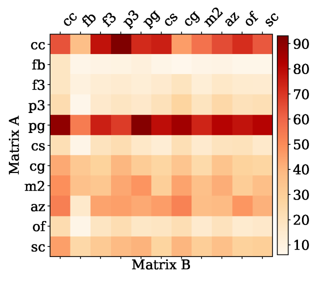
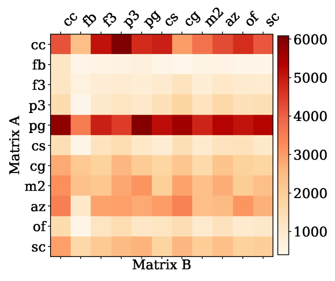


Fig. 9: **Performance breakdown** – plotted as a fraction of total cycles when multipliers are busy and the cycle breakdown when they are stalled due to merge and memory accesses.



(a) Speedup



(b) Energy benefit

Fig. 10: **Speedup and energy benefit of MatRaptor over GPU-CuSPARSE for A×B** — `A` and `B` are top-left 10K×10K tiles of different matrices from the dataset. All the performance and energy results are normalized to 128 GB/s off-chip memory bandwidth and 28nm technology node.

load imbalance; except for `wv` and `fb` the load imbalance for all the benchmarks is less than 5%. For `wv` and `fb` the load imbalance is higher because these matrices are small and thus a round-robin row assignment to PEs is not very effective.

VII. Discussion

Format Conversion – The current implementation assumes that the number of virtual channels used to create C^2SR matches the number of physical channels in the DRAM, which results in highly efficient SpGEMM processing. To

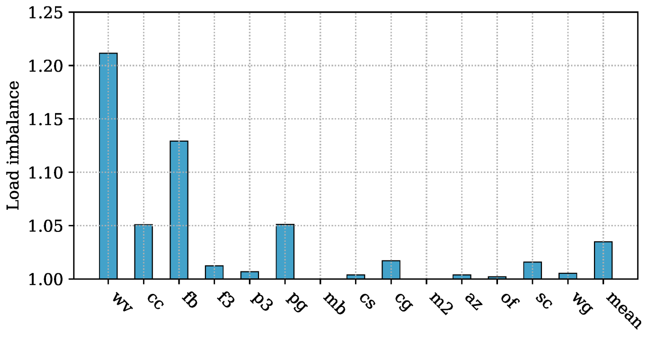


Fig. 11: **Load Imbalance** — measured as the ratio of the maximum and minimum number of non-zeros of matrix A assigned to the PEs.

make the sparse format portable across different platforms, the sparse matrix can be stored in CSR format and converted to C^{2}SR (or vice versa) by a target-specific software library or dedicated logic in the accelerator. The complexity of such conversion is O(nnz) which is much lower than that of SpGEMM O(nnz*nnz/N). More importantly, the cost of format conversion gets amortized due to: (1) In SpGEMM, the rows of matrix B are read multiple times while for the format conversion they are read only once; (2) For algorithms like local graph clustering, the same sparse matrix is reused multiple times; (3) Many other algorithms such as graph contractions perform a chain of matrix multiplications and thus the output matrix becomes the input of another SpGEMM without requiring additional format conversion.

To evaluate the performance overhead of CSR to C^{2}SR conversion and vice versa, we designed a simple hardware unit that reads the sparse matrix in CSR format and stores it back to memory in C^{2}SR. According to our results, the format conversion takes on average 12% of the SpGEMM execution time on MatRaptor.

Buffer Overflow – Since the sorting queues are used to store the partial sums for an output row and the size of the queues is limited, the partial sums may not always fit in the queues. Whenever such a condition arises for an output row, the hardware raises an exception and the SpGEMM computation will fall back to the CPU. This ensures the correctness of MatRaptor irrespective of the size and density of the input matrices.

We can further optimize the performance by having the CPU only handle the rows that cause the overflow. In this scheme, whenever a PE encounters an output row i that will not fit in the queues, MatRaptor simply leaves an "empty" row of size $\sum_{k\epsilon\{A[i,k]\neq 0\}} nnz(B[k,:])$ in the output matrix and continues to compute the next row. This expression represents the sum of the row lengths of matrix B whose row indices are the same as the column indices k of the non-zero elements in $A[i.\ :]$. This serves as an upper bound on the number of non-zeros in $C[i,:]$. Since SpBL is responsible for loading these rows of matrix B, a simple accumulator that sums up the "row length" values in C^{2}SR will be sufficient. When the accelerator has finished computation, it will send to the CPU the indices of the incomplete output rows (if there are any). The CPU will then finish the rest of the computation.

Note that there might be some empty space for padding at the end of the output row computed by the CPU; but this slight storage overhead will not affect the compute or bandwidth requirement, even if the output matrix C is used right away in a subsequent computation. Since the C^{2}SR format encodes the row length of each row, the padding will not be read from memory.

VIII. Related Work

Sparse Storage Formats. Many sparse storage formats have been proposed in the literature. CSR (Compressed Sparse Row), CSC (Compressed Sparse Column) and COO (Co-ordinate) are the most commonly used sparse storage formats for CPUs. Liu *et al.* [30] proposed a sparse tensor storage format F-COO, which is similar to the co-ordinate format and used it for GPUs. CSF [46] and Hi-COO [27] are other sparse tensor storage formats that are based on CSR and COO, respectively. Unlike C^{2}SR these formats do not support efficient parallel accesses to multiple memory channels and thus achieve low bandwidth utilization. For machine learning hardware, researchers have proposed multiple variants of CSR and CSC formats. For example, Cambricon-X [61] proposed a modification of CSR format where the non-zeros are are compressed and stored in contiguous memory and index vectors are used to decode the row and column indices. EIE [18] uses a variant of CSC storage format where instead of storing the row indices they store the number of zeros before a non-zero element. Since these works focus on deep learning, especially CNNs, their sparse storage format is specialized for low sparsity (high density) and is not suitable for SpGEMM computation where the matrices have high sparsity. For SpGEMM involving matrices with high sparsity, OuterSPACE [39] uses a variant of CSR and CSC formats called CR and CC. However, to solve the issues related to channel conflicts and memory misalignment, it uses caches instead to directly accessing the DRAM from the hardware and thus spends 18× more area for on-chip memory compare to MatRaptor. Fowers *et al.* [13] and Tensaurus [47] proposed sparse storage formats called *compressed interleaved sparse row* (CISR) and *compressed interleaved sparse slice* (CISS) which also maps different PEs to different DRAM channels for sparse tensor kernels. In contrast to C^{2}SR, these formats can only be used for a static input matrix and not for the output as the coordinates of all non-zero elements must be known a priori.

CPU/GPU Acceleration. Akbudak *et al.* [1] proposed hypergraph and bipartite graph models for 1D row-wise partitioning of matrix A in SpGEMM to evenly partition the work across threads. Saule *et al.* [43] investigated the performance of the Xeon Phi coprocessor for SpMV computation. Sulatycke *et al.* [49] proposed a sequential cache-efficient algorithm and illustrated high performance than existing algorithms for sparse matrix multiplication for CPUs. Nagasaka *et al.* [36] mitigates multiple bottlenecks with memory management and thread scheduling for SpGEMM kernel on Intel Xeon Phi. The works involving GPU acceleration of SpGEMM computation include [10], [15], [31], [34], [37]. Kiran *et al.* [34] explore the load-balancing problem that only considers the band matrices. Weifeng and Brian [31] apply the techniques such as GPU merge path algorithm and memory pre-allocation to improve the performance and the storage issue. Felix *et al.* [15] reduce the overhead of memory access by merging several sparse rows using the main kernel. Steven *et al.* [10] decompose the SpGEMM operations and leverage bandwidth saving operations like layered graph model. They also perform the SpGEMM in a row-wise product method to balance the workload and improve the performance. Nagasaka *et al.* [37] proposed a fast SpGEMM algorithm that has small memory footprints and achieves high performance.

Custom Accelerators. For sparse-dense and sparse-sparse matrix-matrix and matrix-vector accelerators, prior works involving FPGA implementations include [33], ESE [17], [65] and [13]. Lu *et al.* [33] proposed a CNN accelerator with sparse weights. ESE [17] proposed an FPGA-accelerator for SpMV in LSTMs. Prasanna *et al.* [65] and Fowers *et al.* [13] proposed SpMV accelerators for very sparse matrices. Lin *et al.* [29] proposed an FPGA-based architecture for sparse matrix-matrix multiplication. T2S-Tensor [48] proposed a language and compilation framework to generate high performance hardware for dense tensor computations such as GEMM. Rong *et al.* [42] extended this language to add support for SpMV.

Several prior efforts involved ASIC implementations. Examples include Cambricon-S [63], Cnvlutin [2], SCNN [40], [4], OuterSPACE [39] and ExTensor [20]. Cambricon-S [63] implements hardware accelerator for SpGEMM in CNNs where both weight matrices and neurons are sparse. SCNN [40] proposes a SpGEMM accelerator for CNNs which can also exploit the sparsity in both weights and neurons. Anders *et al.* [4] proposed accelerator designs for SpGEMM. EIE [18] proposes SpMSpV (sparse matrix sparse vector multiplication) accelerator for fully connected layers in CNN and show significant performance gains over CPU and GPU. However, all these works focused on deep learning application where the density is really high. TPU [22] implemented a 2-d systolic array for GEMM. Tensaurus [47] proposed a hardware accelerator for sparse-dense tensor computations such as SpMV and SpMM.

OuterSPACE [39], ExTensor [20] and SpArch [62] are few recent works that propose hardware accelerators for SpGEMM computation on very sparse matrices. However, OuterSPACE applies the outer product approach and ExTensor applies the inner product approach for SpGEMM, the inefficiencies of which have been discussed in Section II. SpArch attempts to improve the outer product approach by matrix-condensing and Huffman trees. However, this results in a complicated design that has more area and power, and lower performance/watt, compared to our approach based on row-wise product. Their simulation infrastructure is also different from OuterSPACE and ours where they use custom models for HBM instead of open-source gem5 HBM memory model. In this work, we do not perform detailed performance comparison with SpArch because of difference in our HBM models.

Yavits and Ginosar [59] and [58] explored content addressable memory (CAM) and RAM-based compute for SpMSpV and SpGEMM. One of the major limitations of the CAM-based approach is that the output elements are not produced in a sorted order of their indices and thus require extra sorting hardware. We conjecture that although the CAM itself might be more efficient, CAM along with the sorting hardware will be more expensive in terms of both area and energy compared to MatRaptor. Zhu *et al.* [64] introduced a 3D-stacked logic-in-memory system by placing logic layers between DRAM dies to accelerate a 3D-DRAM system for sparse data access and built a custom CAM architecture to speed-up the index-alignment process of column-wise product approach.

IX. Conclusion

In this work, we propose a novel row-wise product based accelerator (MatRaptor) for SpGEMM which achieves high performance and energy-efficiency over CPU, GPU and state-of-the-art SpGEMM accelerator OuterSPACE. It also has $7.2\times$ lower power consumption and $31.3\times$ smaller area compared to OuterSPACE. To achieve this, we introduce a new hardware-friendly sparse storage format named C^{2}SR, which improves the memory bandwidth utilization by enabling vectorized and streaming memory accesses. We also implement a novel sorting hardware to merge the partial sums in the SpGEMM computation. We prototype and simulate our MatRaptor using gem5 on a diverse set of matrices.

Acknowledgement

This research was funded in part by CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, under NSF Awards #1453378, #1909661, and by AFRL and DARPA under agreement number FA8650-18-2-7863. 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 AFRL and DARPA or the U.S. Government.

References

[1] K. Akbudak and C. Aykanat, “Exploiting locality in sparse matrix-matrix multiplication on many-core architectures,” *Trans. on Parallel and Distributed Systems*, 2017.

[2] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, “Cnvlutin: Ineffectual-neuron-free deep neural network computing,” *ACM SIGARCH Computer Architecture News*, 2016.

[3] AnandTech. https://www.anandtech.com/show/9883/gddr5x-standard-jedec-new-gpu-memory-14-gbps.

[4] M. Anders, H. Kaul, S. Mathew, V. Suresh, S. Satpathy, A. Agarwal, S. Hsu, and R. Krishnamurthy, “2.9 TOPS/W Reconfigurable Dense/Sparse Matrix-Multiply Accelerator with Unified INT8/INTI6/FP16 Datapath in 14NM Tri-Gate CMOS,” 2018.

[5] A. Azad, A. Buluç, and J. Gilbert, “Parallel triangle counting and enumeration using matrix algebra,” *Workshop in Int'l Symp. on Parallel and Distributed Processing*, 2015.

[6] R. Balasubramonian. (2014) Lecture on memory wall. https://my.eng.utah.edu/~cs7810/pres/14-7810-02.pdf.

[7] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, “The gem5 simulator,” *ACM SIGARCH Computer Architecture News*, 2011.

[8] W. L. Briggs, S. F. McCormick *et al.*, “A multigrid tutorial,” 2000.

[9] P. D'alberto and A. Nicolau, “R-Kleene: A high-performance divide-and-conquer algorithm for the all-pair shortest path for densely connected networks,” *Algorithmica*, 2007.

[10] S. Dalton, L. Olson, and N. Bell, “Optimizing sparse matrix—matrix multiplication for the gpu,” *Trans. on Mathematical Software (TOMS)*, 2015.

[11] T. A. Davis and Y. Hu, “The University of Florida sparse matrix collection,” *Trans. on Mathematical Software (TOMS)*, 2011.

[12] I. S. Duff, M. A. Heroux, and R. Pozo, “An overview of the sparse basic linear algebra subprograms: The new standard from the BLAS technical forum,” *Trans. on Mathematical Software (TOMS)*, 2002.

[13] J. Fowers, K. Ovtcharov, K. Strauss, E. S. Chung, and G. Stitt, “A high memory bandwidth fpga accelerator for sparse matrix-vector multiplication,” *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, 2014.

[14] J. R. Gilbert, S. Reinhardt, and V. B. Shah, “A unified framework for numerical and combinatorial computing,” *Computing in Science & Engineering*, 2008.

[15] F. Gremse, A. Hofter, L. O. Schwen, F. Kiessling, and U. Naumann, “GPU-accelerated sparse matrix-matrix multiplication by iterative row merging,” *SIAM Journal on Scientific Computing*, 2015.

[16] F. G. Gustavson, “Two fast algorithms for sparse matrices: Multiplication and permuted transposition,” *Trans. on Mathematical Software (TOMS)*, 1978.

[17] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang *et al.*, “Ese: Efficient speech recognition engine with sparse lstm on fpga,” *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2017.

[18] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, “EIE: efficient inference engine on compressed deep neural network,” *Int'l Symp. on Computer Architecture (ISCA)*, 2016.

[19] V. Hapla, D. Horák, and M. Merta, “Use of direct solvers in TFETI massively parallel implementation,” *Int'l Workshop on Applied Parallel Computing*, 2012.

[20] K. Hegde, H. Asghari-Moghaddam, M. Pellauer, N. Crago, A. Jaleel, E. Solomonik, J. Emer, and C. W. Fletcher, “ExTensor: An Accelerator for Sparse Tensor Algebra,” *Int'l Symp. on Microarchitecture (MICRO)*, 2019.

[21] S. Itoh, P. Ordejón, and R. M. Martin, “Order-N tight-binding molecular dynamics on parallel computers,” *Computer physics communications*, 1995.

[22] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, “In-datacenter performance analysis of a tensor processing unit,” *Int'l Symp. on Computer Architecture (ISCA)*, 2017.

[23] H. Kaplan, M. Sharir, and E. Verbin, “Colored intersection searching via sparse rectangular matrix multiplication,” *Int'l Symp. on Computational Geometry*, 2006.

[24] G. Karypis, A. Gupta, and V. Kumar, “A parallel formulation of interior point algorithms,” *Int'l Conf. on Supercomputing*, 1994.

[25] S. E. Kurt, V. Thumma, C. Hong, A. Sukumaran-Rajam, and P. Sadayappan, “Characterization of data movement requirements for sparse matrix computations on gpus,” *Int'l Conf. on High Performance Computing (HiPC)*, 2017.

[26] J. Leskovec, L. A. Adamic, and B. A. Huberman, “The Dynamics of Viral Marketing,” *Trans. on the Web (TWEB)*, 2007.

[27] J. Li, J. Sun, and R. Vuduc, “HiCOO: Hierarchical storage of sparse tensors,” *Int'l Conf. for High Performance Computing, Networking, Storage and Analysis*, 2018.

[28] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, “McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures,” *Int'l Symp. on Microarchitecture (MICRO)*, 2009.

[29] C. Y. Lin, N. Wong, and H. K.-H. So, “Design space exploration for sparse matrix-matrix multiplication on FPGAs,” *International Journal of Circuit Theory and Applications*, 2013.

[30] B. Liu, C. Wen, A. D. Sarwate, and M. M. Dehnavi, “A unified optimization approach for sparse tensor operations on gpus,” *Int'l Conf. on Cluster Computing (CLUSTER)*, 2017.

[31] W. Liu and B. Vinter, “An efficient GPU general sparse matrix-matrix multiplication for irregular data,” *Int'l Parallel and Distributed Processing Symposium*, 2014.

[32] D. Lockhart, G. Zibrat, and C. Batten, “PyMTL: A unified framework for vertically integrated computer architecture research,” *Int'l Symp. on Microarchitecture (MICRO)*, 2014.

[33] L. Lu, J. Xie, R. Huang, J. Zhang, W. Lin, and Y. Liang, “An Efficient Hardware Accelerator for Sparse Convolutional Neural Networks on FPGAs,” *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, 2019.

[34] K. Matam, S. R. K. B. Indarapu, and K. Kothapalli, “Sparse matrix-matrix multiplication on modern architectures,” *Int'l Conf. on High Performance Computing*, 2012.

[35] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, “CACTI 6.0: A tool to model large caches,” *HP laboratories*, 2009.

[36] Y. Nagasaka, S. Matsuoka, A. Azad, and A. Buluç, “High-performance sparse matrix-matrix products on Intel KNL and multicore architectures,” *Int'l Conf. on Parallel Processing Companion*, 2018.

[37] Y. Nagasaka, A. Nukada, and S. Matsuoka, “High-performance and memory-saving sparse general matrix-matrix multiplication for nvidia pascal gpu,” *Int'l Conf. on Parallel Processing (ICPP)*, 2017.

[38] M. Naumov, L. Chien, P. Vandermersch, and U. Kapasi, “Cusparse library,” *GPU Technology Conference*, 2010.

[39] S. Pal, J. Beaumont, D.-H. Park, A. Amarnath, S. Feng, C. Chakrabarti, H.-S. Kim, D. Blaauw, T. Mudge, and R. Dreslinski, “OuterSPACE: An outer product based sparse matrix multiplication accelerator,” *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, 2018.

[40] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, “Scnn: An accelerator for compressed-sparse convolutional neural networks,” *Int'l Symp. on Computer Architecture (ISCA)*, 2017.

[41] M. O. Rabin and V. V. Vazirani, “Maximum matchings in general graphs through randomization,” *Journal of Algorithms*, 1989.

[42] H. Rong, “Expressing Sparse Matrix Computations for Productive Performance on Spatial Architectures,” *arXiv preprint arXiv:1810.07517*, 2018.

[43] E. Saule, K. Kaya, and Ü. V. Çatalyürek, “Performance evaluation of sparse matrix multiplication kernels on intel xeon phi,” *Int'l Conf. on Parallel Processing and Applied Mathematics*, 2013.

[44] V. B. Shah, “An interactive system for combinatorial scientific computing with an emphasis on programmer productivity,” *University of California, Santa Barbara*, 2007.

[45] A. Shilov. (2016) Jedec publishes hbm2 specification. http://www.anandtech.com/show/9969/jedec-publisheshbm2-specification.

[46] S. Smith, N. Ravindran, N. D. Sidiropoulos, and G. Karypis, “SPLATT: Efficient and parallel sparse tensor-matrix multiplication,” *Int'l Symp. on Parallel and Distributed Processing*, 2015.

[47] N. Srivastava, H. Jin, S. Smith, H. Rong, D. Albonesi, and Z. Zhang, “Tensaurus: A Versatile Accelerator for Mixed Sparse-Dense Tensor Computations,” *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, 2020.

[48] N. Srivastava, H. Rong, P. Barua, G. Feng, H. Cao, Z. Zhang, D. Albonesi, V. Sarkar, W. Chen, P. Petersen, G. Lowney, A. H. Herr, C. Hughes, T. Mattson, and P. Dubey, “T2S-Tensor: Productively Generating High-Performance Spatial Hardware for Dense Tensor Com-

putations," *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, 2019.

[49] P. D. Sulatycke and K. Ghose, "Caching-efficient multithreaded fast multiplication of sparse matrices," *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, 1998.

[50] S. M. Van Dongen, "Graph clustering by flow simulation," 2000.

[51] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang, "Intel math kernel library," *High-Performance Computing on the Intel® Xeon Phi™*, 2014.

[52] WikiChip, "14 nm lithography process," https://en.wikichip.org/wiki/14_nm_lithography_process.

[53] WikiChip, "16 nm lithography process," https://en.wikichip.org/wiki/16_nm_lithography_process.

[54] WikiChip, "28 nm lithography process," https://en.wikichip.org/wiki/28_nm_lithography_process.

[55] WikiChip, "32 nm lithography process," https://en.wikichip.org/wiki/32_nm_lithography_process.

[56] S. Williams, A. Waterman, and D. Patterson, "Roofline: An Insightful Visual Performance Model for Multicore Architectures," *Commun. ACM*, 2009.

[57] I. Yamazaki and X. S. Li, "On techniques to improve robustness and scalability of a parallel hybrid linear solver," *Int'l Conf. on High Performance Computing for Computational Science*, 2010.

[58] L. Yavits and R. Ginosar, "Accelerator for sparse machine learning," *IEEE Computer Architecture Letters*, 2017.

[59] L. Yavits and R. Ginosar, "Sparse matrix multiplication on CAM based accelerator," *arXiv preprint arXiv:1705.09937*, 2017.

[60] R. Yuster and U. Zwick, "Detecting short directed cycles using rectangular matrix multiplication and dynamic programming," *SIAM Symp. on Discrete Algorithms*, 2004.

[61] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-x: An accelerator for sparse neural networks," *Int'l Symp. on Microarchitecture (MICRO)*, 2016.

[62] Z. Zhang, H. Wang, S. Han, and W. J. Dally, "SpArch: Efficient Architecture for Sparse Matrix Multiplication," *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, 2020.

[63] X. Zhou, Z. Du, Q. Guo, S. Liu, C. Liu, C. Wang, X. Zhou, L. Li, T. Chen, and Y. Chen, "Cambricon-S: Addressing Irregularity in Sparse Neural Networks through A Cooperative Software/Hardware Approach," *Int'l Symp. on Microarchitecture (MICRO)*, 2018.

[64] Q. Zhu, T. Graf, H. E. Sumbul, L. Pileggi, and F. Franchetti, "Accelerating sparse matrix-matrix multiplication with 3D-stacked logic-in-memory hardware," *Int'l Conf. on High Performance Extreme Computing (HPEC)*, 2013.

[65] L. Zhuo and V. K. Prasanna, "Sparse matrix-vector multiplication on FPGAs," *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2005.