IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received April 21, 2022, accepted June 8, 2022, date of publication June 17, 2022, date of current version June 22, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3184116

Row-Wise Product-Based Sparse Matrix Multiplication Hardware Accelerator With Optimal Load Balancing

JONG HUN LEE[1], BEOMJIN PARK[2], JOONHO KONG[3,4], (Member, IEEE), AND ARSLAN MUNIR[5], (Senior Member, IEEE)

[1]LX Semicon, Seoul 06763, South Korea
[2]Samsung Electronics, Hwaseong 18448, South Korea
[3]School of Electronic and Electrical Engineering, Kyungpook National University, Daegu 41566, South Korea
[4]School of Electronics Engineering, Kyungpook National University, Daegu 41566, South Korea
[5]Department of Computer Science, Kansas State University, Manhattan, KS 66506, USA

Corresponding author: Joonho Kong (joonho.kong@knu.ac.kr)

This work was supported in part by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education under Grant NRF-2021R1I1A3A04037455, and in part by Samsung Electronics.

ABSTRACT Matrix multiplication is a main computation kernel of emerging workloads, such as deep neural networks and graph analytics. These workloads often exhibit high sparsity in data, which means a large portion of the elements in the data are zero-valued elements. Though systolic arrays have shown a significant performance and energy efficiency improvement over central processing units (CPUs) or graphic processing units (GPUs) when executing matrix multiplications, data sparsity is largely overlooked in the conventional systolic arrays. In this paper, we propose a row-wise product-based sparse matrix multiplication (SpMM) hardware accelerator for compressed sparse row (CSR)-formatted input matrices. Our hardware accelerator leverages row-wise product, which has advantages over inner-product or outer-product when executing the sparse matrix multiplications. As compared to the conventional systolic arrays, which cannot skip the ineffectual operations, our hardware accelerator only performs effectual operations with non-zero elements, improving the performance when executing SpMM. In addition, we also propose an optimal load balancing scheme when using multiple processing elements (PEs). Our load balancing scheme utilizes an operation count-based matrix tiling for parallel execution of the PEs and resource contention avoidance. According to our evaluation, our 32PE-SpMM accelerator shows 13.6× – 47.9× speedup over tensor processing unit (TPU)-like systolic arrays, on average. Furthermore, our operation count-based load balancing scheme shows better performance over the fixed tiling and non-zero element count-based tiling by up to 8.48% and 6.28%, respectively, with only up to 0.06% matrix tiling pre-processing latency overhead.

INDEX TERMS Sparse matrix multiplication, row-wise product, load balancing, matrix tiling, speedup.

I. INTRODUCTION

Matrix multiplication (MM) is a pivotal part of many emerging workloads, such as convolutional neural networks (CNNs), recurrent neural networks (RNNs), and graph analytics. Many of these emerging workloads show the data sparsity, which means most of the elements in input matrices are zero. In MMs, since the zero-valued elements in the operand matrices do not affect the results, removing the multiplications and accumulations (MACs) with zero-valued elements can significantly improve the performance and energy efficiency of the systems.

The sparse matrix multiplication (SpMM) can be performed in typical central processing units (CPUs) or graphic processing units (GPUs). However, CPUs do not perform data parallel workloads well while they have strengths on executing the control-intensive workloads. Though GPUs (e.g., NVIDIA V100 [1]) could be an alternative for executing the data parallel workloads, GPU cannot skip the zero MACs and only provides a rigid support for sparsity, worsening the performance and energy efficiency. In addition, GPUs are typically power-hungry, making them hard to be deployed in

The associate editor coordinating the review of this manuscript and approving it for publication was Songwen Pei.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

the systems that have a tight cost budget. On the other hand, the systolic arrays such as Google tensor processing units (TPUs) [2] have strengths on executing MMs. However, the systolic arrays cannot also skip the zero MACs due to their fixed dataflow. Thus, a specialized accelerator for SpMM will be a key component for performance and power efficiency of modern computer systems.

For an efficient SpMM accelerator, there are two traditional and well-known approaches: inner-product and outer-product. Assuming that we perform the matrix multiplication $A \times B = C$, the inner-product approach performs the dot product between each row of A and each column of B. The outer-product approach performs the cross product between each column of A and each row of B. For sparse MM, to remove the MAC operations with zero elements, the inner-product must match the column indices of the non-zero elements in the row vector of the matrix A and row indices of the non-zero elements in the column vector of the matrix B during the dot-product, making the hardware design complicated. Though the outer-product does not require the index matching, partial results from the cross-product require a large on-chip memory to minimize the amount of data transfer between the accelerator and off-chip memory. In addition, in the cases of inner-product and outer-product, accessing the column of the matrices could be difficult for exploiting the locality. In order to fully exploit the locality from the matrix column, the matrix (i.e., B in the inner-product and A in the outer-product) should be stored in a transposed format, eventually causing the burden of transposing operations. To overcome the drawbacks of the inner- and outer-product, we can utilize a row-wise product-based approach as an alternative. The main advantages of the row-wise product MM can be summarized as follows. First, it does not need to perform index matching. Second, it does not require a huge size of on-chip memory which is necessary for storing the partial results. Third, it does not require column-wise access to the operand matrices, making it advantageous to exploit the locality.

Considering the several advantages of the row-wise product, in this paper, we propose a row-wise product-based SpMM accelerator architecture. In addition, to store the sparse data efficiently, we leverage the compressed sparse row (CSR) format, which is one of the most well-known formats for compressing the sparse matrix. In summary, our accelerator performs the matrix multiplication with CSR-formatted input matrices, requiring much less on-chip memory to store the sparse matrices.

Despite of the advantages of the row-wise product, the efficient parallelization between the row-wise product operations with multiple processing elements (PEs) is challenging. Depending on the non-zero value distributions, the amount of load in each PE would be diverse. Since the entire execution time will be determined by the PE that has the largest load, PE load balancing is a critical factor for performance (and thus, energy efficiency) of SpMM execution with multiple PEs. By considering the number of operations and non-zero elements assigned to each PE, our load balancing technique tries to attain an optimal load balance among PEs, leading to a better performance.

We summarize our contributions as follows:

- We propose a row-wise product-based SpMM accelerator for CSR-formatted input matrices;
- We also propose an operation count-based matrix tiling scheme for optimal load balancing across the multiple PEs;
- Our row-wise product-based SpMM accelerator with the operation count-based matrix tiling scheme shows $13.6\times - 47.9\times$ speedup (on average) over the TPU-like systolic arrays;
- Our operation count-based load balancing scheme shows better performance over the fixed tiling and non-zero element count-based tiling by up to 8.48% and 6.28%, respectively, on average, while only incurring the matrix tiling latency overhead in the host CPU up to 0.06% (on average), which is almost negligible.

II. RELATED WORK

As the SpMM is frequently used in a wide variety of the emerging applications, many SpMM hardware architectures have been recently introduced. One of the most well-known and intuitive approaches for MM is an inner-product approach. In [3], Gondimalla *et al.* have introduced SparTen architecture, which utilizes an inner-product based approach. They have introduced an efficient inner-join with bitmask (to mark non-zero element positions) which eventually are ANDed to identify ineffectual operations. They have also introduced a sorting-based greedy load balancing technique for the PEs. However, the sorting operation is a time-consuming and complicated process, which is not desirable for efficient hardware implementation. Qin *et al.* have also proposed an inner-product based hardware accelerator architecture, SIGMA [4]. They have introduced a dot product engine (a.k.a., Flex-DPE) that exploits tree-based topology and forward adder network in order to support flexible interconnect. It also utilizes a bitmap-based format for compressed data format. However, only loaded operand matrix is represented by the compressed format while the streaming matrix is represented by dense format (i.e., including both all zero and non-zero elements), causing the inefficiency in memory and on-chip storage. Furthermore, the inner-product based approaches inherently requires index matching (or inner-join), which is not desirable for cost-efficient hardware design and implementation.

To remove the complex index matching process, several outer-product based approaches have been introduced. Zhang *et al.* have proposed SpArch [5] that utilizes the outer-product approach for sparse MM. For compressed data format, SpArch utilizes condensed matrix representation that compacts non-zero elements in a row. SpArch also utilizes Huffman tree-based scheduler, which helps efficiently use memory bandwidth. SpArch produces the partial result matrices that have less non-zero elements earlier than those that

have more non-zero elements, so that it improves a reusability of the partial result matrices and reduces a memory bandwidth requirement. This is because the sparser matrix requires less size in memory or storage with condensed matrix representation. Hojabr *et al.* have proposed SPAGHETTI [6], which also utilizes the outer-product approach. SPAGHETTI uses different compressed formats for input matrices (compressed sparse row and compressed sparse column for each operand matrix) and output matrix (COO: coordinate format). Since the COO format is not a sorted format (i.e., the coordinates and values are not sorted in a format of row- or column-major order), it is hard to be utilized in an in-situ manner. In addition, the outer-product based approaches typically require a large storage for partial result matrices.

In order to complement the disadvantages of the inner- and outer-product based approaches, several row-wise product-based approaches have also been proposed. Srivastava *et al.* have proposed MatRaptor [7], which is a row-wise product based approach with a new compressed format, channel cyclic sparse row (C^2SR). Exploiting that the multiplications for each row of the matrix A can be performed in parallel, MatRaptor carries out row-wise multiplication followed by sort and accumulation with the primary and helper queues. Similarly, in [8], another row-wise product-based approach has been introduced by Zhang *et al.*, which is called as Gamma. It also exploits the row-level parallelism while it also employs Fiber cache, a specialized memory structure to store the non-zero elements and their coordinates. Our proposed approach is similar to the above works in that the row-wise product-based approach is used. However, our approach uses a CSR format for compressed matrix format, which is commonly used. In addition, we also propose a novel load distribution and balancing technique for better parallelism between the PEs, which considers the amount of operations assigned to each PE.

III. BACKGROUND

A. ROW-WISE PRODUCT-BASED MATRIX MULTIPLICATION WITH SPARSE INPUT MATRICES

In this subsection, we briefly explain how row-wise product-based MM is performed. Assuming we perform a matrix multiplication A (dimension: M×N) × B (dimension: N×K) = C (dimension: M×K), we perform scalar × vector multiplications between each element (coordinate: (m, n)) in the matrix A and row vector with the row index n in the matrix B. It generates a partial result of coordinate (m, k) in the matrix C where k is the column index in the row vector of the matrix B. In order to generate the complete result matrix C, we should perform the multiplications iteratively for all elements in the matrix A with all the rows in matrix B, meaning that M×N scalar-vector multiplications are required. We then perform accumulations with all partial results for each element in the matrix C.

For SpMM, we should remove ineffectual MACs when generating the row-wise products. We can easily skip the

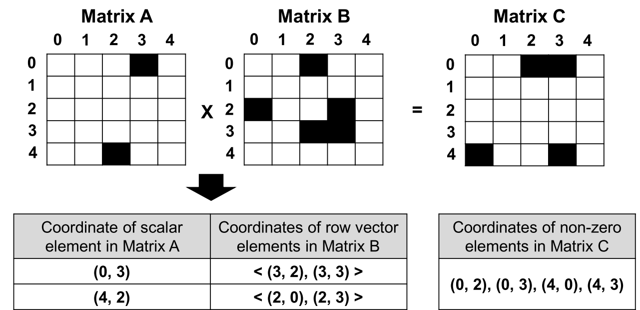


FIGURE 1. An example of row-wise matrix multiplication by exploiting sparsity. The black-colored element indicates a non-zero value while the white-colored element indicates a zero value.

ineffectual operations (i.e., where the elements of operand matrices are zero) in row-wise product-based MM. In the case of a zero-valued scalar element in the matrix A, we can skip a scalar-vector multiplication which involves zero-valued scalar value as an operand. During the scalar-vector multiplication with a non-zero scalar element in the matrix A, we can also skip ineffectual scalar-scalar multiplications with the zero-valued scalar elements in the row vectors of the matrix B.

For an illustrative example, we show a matrix-matrix multiplication (input dimension: 5 × 5) in Figure 1. First, we perform scalar-vector multiplications for each scalar value in each row of matrix A with the row vector in the matrix B of which row index is equal to the column index of the scalar element in the matrix A. At the first row of the matrix A, we can remove four ineffectual scalar-vector multiplications while performing the scalar-vector multiplication only with the scalar value in (0, 3) of matrix A. When performing the scalar-vector multiplication with the scalar value in (0, 3) of the matrix A, we can also skip three scalar-scalar multiplications in (3, 0), (3, 1), and (3, 4) of matrix B, leaving only two scalar-scalar multiplications ((3, 2) and (3, 3) of matrix B). Since the second, third, and fourth row of the matrix A do not have non-zero elements, we can also skip 15 scalar-vector multiplications. For the last row of the matrix A, we only perform one scalar-vector multiplication while removing four scalar-vector multiplications. We perform the scalar-vector multiplication with the element (4, 2) in the matrix A and third row (i.e., row index = 2) of the matrix B. Since there are three zero-valued scalar elements in the row vector of the matrix B, we only perform two scalar-scalar multiplications ((4, 2) of the matrix A with (2, 0) and (2, 3) of the matrix B). As a result, in the matrix C, we have four non-zero elements in the coordinates (0, 2), (0, 3), (4, 0), and (4, 3).

As we explained in Section I, when performing SpMM, row-wise product-based MM has several advantages over inner- or outer-product-based MM approaches. Assuming that we perform the inner-product-based MM with the example shown in Figure 1, inner-product-based MM requires index matching in order to perform multiplication operations between two non-zero scalar elements. To generate the result

element in the coordinate (0, 0) of the matrix C, we need to find whether the indices (3 in Figure 1) of the non-zero elements in the row vector (index = 0) of the matrix A and the indices (2 in Figure 1) of the non-zero elements in the column vector (index = 0) in the matrix B are equal or not, which is referred to as 'index matching'. When performing SpMM with inner-product, we need to perform index matching for each row vector in the matrix A and column vector in the matrix B, which incurs a large overhead when performing SpMM with large input matrices. Similarly, assuming that we perform the outer-product-based MM with the example shown in Figure 1, we perform cross-product with the column vectors in the matrix A and the row vectors in the matrix B. If we perform the cross-product between the column vector (index = 0) in the matrix A and row vector (index = 0) in the matrix B, the partial result matrix will contain the partial results of all the matrix element from the coordinate (0, 0) to (4, 4). Since the column vector with index 0 in the matrix A does not have any non-zero element, all the partial elements in the partial result matrix will be zero. Similarly, we perform cross-products and partial results are accumulated to generate the complete result matrix C. As it can be seen from the example, we need a large memory or storage for temporarily storing the partial matrices. The memory or storage overhead will be exaggerated when performing the SpMM with large input matrices.

B. COMPRESSED SPARSE ROW FORMAT

A CSR format is a well-known compressed format for sparse matrices. Since the dense format (i.e., storing the matrix elements in the order of the coordinates in either row-major or column-major order) is not efficient when storing the sparse matrix due to a huge number of zero-valued elements, the CSR format has been regarded as an efficient way for storing the sparse matrix.

As shown in Figure 2, the CSR format stores three components: values of non-zero elements, column index of each non-zero element, and row pointers. The CSR format removes the zero-valued elements when storing the matrix while only maintaining the non-zero elements and their location information. In the first part, non-zero value (NV), the CSR format contains the non-zero value itself. The second part, column index (CI), contains the index of the column location of the corresponding non-zero elements, which is maintained as a pair with each non-zero element. For a pair of the non-zero elements and column index, the same index value for non-zero element array and column index array is used. The third part, row pointer (RP), maintains how many non-zero elements exist from the first row to the current row (i.e., cumulative number). For example, $RP[x]$ contains how many non-zero elements exist from the row with index 0 to the row with index $x-1$ (thus, $RP[0]$ is always 0).

Other than the CSR format, COO, C^2SR [7], and two-step compression [9] can also be used as a compressed format for sparse matrices. However, the CSR format is a classical format for sparse matrices, meaning that many software libraries

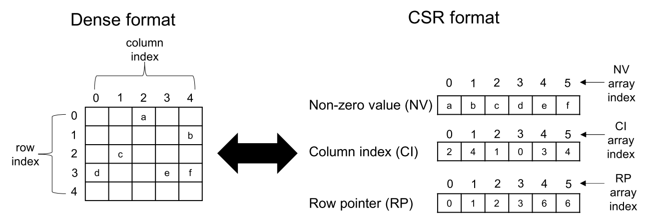


FIGURE 2. A conversion between the dense and compressed sparse row format. The coordinate (x, y) indicates a location of the element in the row index *x* and column index *y* with the dense format matrix. In the CSR format, an array index indicates a location in the NV, CI, or RP array. For CSR format, we follow a general C-style representation for indicating a certain array element. For example, NV[2] and CI[2] are equal to 'c' and '1', respectively.

already support an efficient conversion between the CSR and dense format. One can also argue that the bitmap-based compression can also be used for a compressed matrix format. However, bitmap-based compression requires at least 1-bit for each element to record whether the corresponding element is non-zero or not. It means the bitmap-based format requires at least 1-bit metadata for each element regardless of the value (either non-zero or zero value), which has a disadvantage in compression rate when compressing highly sparse matrix.

In addition, the CSR format is well suitable for the row-wise product-based MM. This is because the CSR format records the non-zero elements in a row-wise fashion with the row pointers. When performing the row-wise product, the scalar value in the matrix A can be easily accessed with the ascending order of the index from the non-zero element array and column index array. The row vector in the matrix B can also be easily accessed by using the column index of the matrix A. In the following section, we will explain how to perform row-wise product-based SpMM with the CSR formatted input matrices.

IV. ROW-WISE PRODUCT-BASED SPARSE MM ACCELERATION WITH OPTIMAL LOAD BALANCING

A. SPARSE ROW-WISE PRODUCT-BASED MM ACCELERATOR WITH CSR FORMAT

1) OVERVIEW

In this subsection, we demonstrate how the sparse MM with CSR-formatted operand matrices can be performed with row-wise product. Figure 3 demonstrates how we perform a scalar-vector multiplication with CSR-formatted input matrices. For the matrix A, we can sequentially pick the element of the CSR-formatted matrix A and search for the matched row vector in the matrix B. Given a non-zero scalar value in the matrix A, we refer to the column index of the selected non-zero element in the matrix A (i.e., CI_A) to find the matched row vector in the matrix B. As shown in Figure 3, when performing a scalar-vector multiplication with $NV_A[x]$, we refer to $CI_A[x]$ to find the matched row vectors in the matrix B where x is a certain array index for NV_A and CI_A. Next, we select the row vector (in the matrix B) whose index is equal to the $CI_A[x]$. It can be accomplished by

accessing the NV_B and CI_B entries starting from the array index $RP_B[CI_A[x]]$ (w in Figure 3). The number of elements in the row vector can be calculated by $RP_B[CI_A[x] + 1] - RP_B[CI_A[x]]$ ($z - w$ in Figure 3). For scalar-vector multiplication, the scalar value from the matrix A is multi-casted to the corresponding elements in the row vector, which enables to perform multiple scalar-scalar multiplications.

Our sparse row-wise product-based MM with CSR format also stores the output matrix (matrix C) in the CSR format. The multiplication operations do not occur in the order of the column index within a row vector in the matrix C. Since the pairs of the non-zero elements and column indices in the CSR format should be already aligned in the ascending order of the column index within the row, we need to find the appropriate array index (i.e., location) of the output elements to accumulate the partial result. Assuming that the currently accessed row index of the matrix A is equal to i (i.e., currently accessing $RP_A[i]$ as shown in Figure 3), a partial result will be accumulated to the coordinate (i, $CI_B[j]$) in the matrix C where j is equal to the currently accessed array index of the CSR-formatted matrix B used as an operand to produce the (partial) result (in Figure 3, j is equal to w). Since we do not know whether the partial result of (i, $CI_B[j]$) in the matrix C already exists or not, we need to search for the corresponding elements in the CSR-formatted matrix C. We sequentially seek for the corresponding element in NV_C and CI_C from the array index $RP_C[i]$ until the empty (==NULL) entry. This is the advantage of the row-wise product-based MM in which the partial results are produced in the ascending order of the row index in the matrix C, implying that we do not need to search for all the elements from the beginning to find the corresponding element. In other words, when constructing the matrix C, the overhead of the sequential search can be relieved by using the row-wise product-based MM.

During idx (currently accessed array index in the CSR-formatted matrix C) search in the case of the partial sum update, there are four possible cases (summarized in Figure 4). In the first case where $CI_C[idx]$ is less than $CI_B[j]$, we need to keep searching for the appropriate array entry by increasing the idx. In the second case where the $CI_C[idx]$ is higher than $CI_B[j]$, we need to allocate a new array entry for $CI_C[idx]$ by right-shifting the right-side array entries in order to maintain the NV_C and CI_C as a sorted order. After allocating a new array entry, we can update the (partial) result to that array entry (NV_C and CI_C). In the third case where we find $CI_C[idx]$ equal to $CI_B[j]$, we can accumulate the partial result to the current array entry $CI_C[idx]$ and $NV_C[idx]$. In the fourth case where the $CI_C[idx]$ is an empty array entry (i.e., it has not been allocated yet), existing array entries do not need to be right-shifted, and we can write the (partial) result to that array entry (NV_C and CI_C). The example shown in Figure 3 corresponds to the third case ($CI_C[idx] == CI_B[j] == m$). Thus, we accumulate the partial result to the $NV_C[idx]$ ($+=$ a×b).

2) ALGORITHM

Figure 5 presents a pseudocode of the algorithm for our row-wise product-based MM with CSR format. Before we describe the algorithm, we define the variables used in our algorithm. The algorithm takes inputs NV, CI, and RP for each operand matrix A and B. The NV_A, CI_A, and RP_A correspond to those of the matrix A while NV_B, CI_B, and RP_B correspond to those of the matrix B. We also require the row height of matrix A (Row_A) and column width of matrix B (Col_B), which can be obtained by measuring the input matrix dimensions. For outputs, we will obtain NV_C, CI_C, and RP_C. There are temporary variables to maintain the current status of the MM operation. idx_{search} and idx_{empty} are used for the indices that indicate the current search and empty array entry (i.e., the array entry next to the last filled entry), respectively, in the output matrix C. idx_A and idx_B are used to indicate the current array indices, which point out the array entry for the MM operations of the matrices A and B, respectively. $colidx_A$ corresponds to the value of the current CI_A entry pointed out by idx_A (i.e., equal to $CI_A[idx_A]$).

At first, the temporary variables, idx_A, idx_B, and idx_{empty} are initialized to 0. We also set the $RP_C[0]$ as 0 because the first array entry of RP_C is always 0 regardless of the matrix elements and dimensions. For the main loops there are three nested loops (loops ①–③). The first loop (①) is iterated by the row height of the matrix A (Row_A). The second loop (②) is iterated by the number of non-zero scalar values in the selected (in the loop ①) row vector of the matrix A (the number of non-zero vector elements is equal to $RP_A[i + 1]$ - $RP_A[i]$). In other words, inside of the second loop, the multiplications of the scalar value in the matrix A and row vector in the matrix B are performed. Before starting the scalar-vector multiplications, we set the multiple variables: $colidx_A$, idx_B, and idx_{search}. The $colidx_A$ is used to maintain the column index of the current scalar value in the matrix A ($CI_A[idx_A]$). To perform the multiplication, we need to match the column index of the matrix A and row index of the matrix B. Thus, we need to set idx_B as $RP_B[colidx_A]$. We also initialize idx_{search} as $RP_C[i]$ because we need to accumulate the partial sum in the row index i where i is equal to the row index of the current scalar element in the matrix A. Since our sparse MM algorithm performs the multiplications only with non-zero operands, we need to perform the iterations for the third loop (③) by the number of required scalar-scalar multiplications, which can be identified with $RP_B[colidx_A+1]$ - $RP_B[colidx_A]$. Inside of the third loop (③), before we perform the MAC operations, we need to identify the location (entry) to write the partial sum in NV_C and CI_C which will be determined in the fourth loop (④).

The iteration count for the fourth loop will vary depending on which case we encounter among the four cases shown in Figure 4. In the first case (corresponding to Figure 4 (a)), we need to keep searching for the appropriate array entry because $CI_C[idx_{search}]$ is less than $CI_B[idx_B + k]$ (the column index of the element for the partial sum update in the

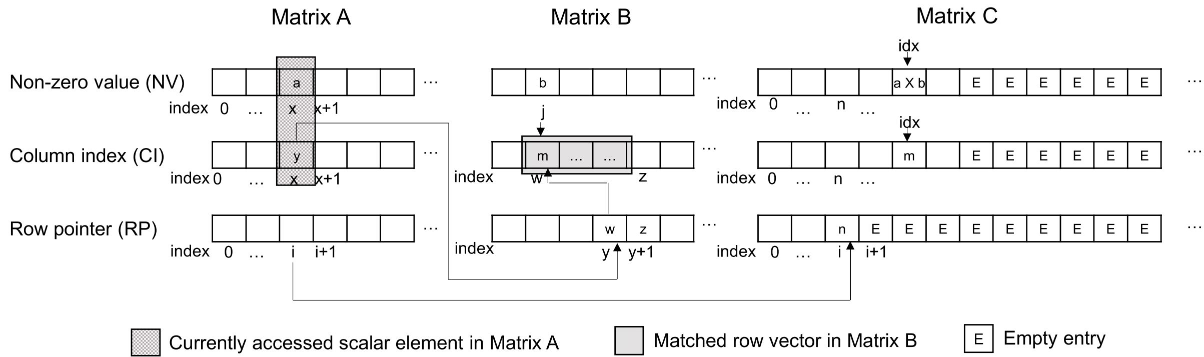


FIGURE 3. An example of a row-vector multiplication with CSR-formatted input matrices.

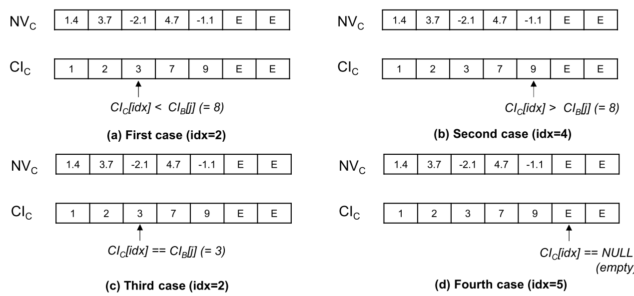


FIGURE 4. Four possible cases when writing the (partial) results in the CSR-formatted output matrix. 'E' in the entry means that the entry is empty.

matrix C). In this case, we need to increment idx_{search} to find the next CI_C array entry without terminating the fourth loop (④). In the second case (corresponding to Figure 4 (b)), the currently searched column index in CI_C ($CI_C[idx_{search}]$) is higher than the $CI_B[idx_B+k]$. In this case, we need to perform right-shift operations (while loop denoted as ⑤) in NV_C and CI_C from the array index idx_{search} to the $idx_{empty} - 1$ to maintain the sorted order of the NV_C and CI_C entries. In the third case (corresponding to Figure 4 (c)), $CI_C[idx_{search}]$ is equal to $CI_B[idx_B + k]$, meaning that we can accumulate the partial sum to the current array index idx_{search} entry location. In the fourth case (corresponding to Figure 4 (d)), idx_{search} is equal to idx_{empty}, meaning that we should allocate a new array entry pointed out by the current array index idx_{search} where we will update the partial sum. For the second, third, and fourth cases, we terminate the fourth loop (④), implying that we are ready for accumulating the partial sum to the NV_C and CI_C (i.e., we have already provisioned the NV_C and CI_C entries for writing the partial sum). Once terminating the fourth loop, we need to perform the actual MAC operation and fill in the (partial) result and column index to the identified (or provisioned) NV_C and CI_C entries pointed out by idx_{search}, respectively (lines 24-25). Once terminating a scalar-vector multiplication, we also increase idx_A to perform the next scalar-vector multiplication (line 27). Before terminating the second loop (②), we also need to update $RP_C[i + 1]$ as

```
Pseudocode for row-wise product-based MM with CSR format
Inputs : RP_A, CI_A, NV_A, RP_B, CI_B, NZ_B, Row_A, Col_B
Outputs : RP_C, CI_C, NV_C
Variables: idx_search, idx_empty, idx_A, idx_B, colidx_A

1  idx_A = 0, idx_B = 0, idx_empty = 0, RP_C[0] = 0
2  for i=0 to (Row_A - 1)
3    for j=0 to (RP_A[i+1] - RP_A[i] - 1)
4      colidx_A = CI_A[idx_A], idx_B = RP_B[colidx_A], idx_search = RP_C[i];
5      for k=0 to (RP_B[colidx_A+1] - RP_B[colidx_A] - 1)
6        for m=0 to (Col_B - 1)
7          if CI_C[idx_search] < CI_B[idx_B + k]
8            idx_search++;
9          else if CI_C[idx_search] > CI_B[idx_B + k]
10           n = idx_empty;
11           while n > idx_search
12             NV_C[n] = NV_C[n-1];
13             CI_C[n] = CI_C[n-1];
14             n--;
15           endwhile
16           idx_empty++;
17           break;
18         else if CI_C[idx_search] == CI_B[idx_B + k]
19           break;
20         else if idx_search == idx_empty
21           idx_empty++; break;
22         endif
23       endfor
24       NV_C[idx_search] += NV_A[idx_A] * NV_B[idx_B + k];
25       CI_C[idx_search] = CI_B[idx_B + k];
26     endfor
27     idx_A++;
28   endfor
29   RP_C[i+1] = idx_empty;
30 endfor

Legends for loops
① ──────  ② - - - - -  ③ -·-·-·-  ④ ◆────◆  ⑤ ◀────▶
```

FIGURE 5. Proposed algorithm (in the form of a pseudocode) for row-wise product-based matrix multiplication with CSR format.

idx_{empty} (line 29) because one row of the output matrix C has been generated.

3) HARDWARE ARCHITECTURE

Figure 6 depicts our hardware architecture for a single PE. We have internal buffers where the CSR-formatted matrices A, B, and C are stored. We have idx_A and idx_B registers which maintain the currently accessed indices of the matrices A and B, respectively. By referring to the current indices,

we multiply the values from $NV_A[idx_A]$ and $NV_B[idx_B]$. The middle part is responsible for writing the result matrix C in the CSR format. The writing control unit determines the corresponding case among four possible cases (see the previous subsection). The writing control unit receives the comparison results of $CI_B[idx_B]$ and $CI_C[idx_{search}]$ that maintains current search point for an array entry in the NV_C and CI_C. The comparison result will trigger the writing control unit, which takes the appropriate action depending on the comparison result (i.e., cases shown in Figure 4). The output is 4-bit one-hot encoded value where each output bit from the writing control unit corresponds to each partial sum update case (from 1st to 4th) shown in Figure 4. In the 1st and 4th case, the idx_{search} and idx_{empty} (maintains the first empty array index of the NV_C and CI_C) are incremented, respectively. In the 2nd case, it triggers NV_C and CI_C shift unit which performs right-shift operations in the NV_C and CI_C entries for an ordered NV_C and CI_C array entries. In the 2nd, 3rd, and 4th cases, the upper OR gate output becomes 1 (logic high), and the multiplied result from the multiplier is forwarded to the accumulator and the result in NV_C is updated. In addition, the lower OR gate output becomes 1 (logic high) for either case 2, case 3, or case 4, and thus $CI_C[idx_{search}]$ is updated to the same value as the current value in the $CI_B[idx_B]$.

Though we only depict a single PE architecture in Figure 6, our SpMM accelerator can also be extended to the version with multiple PEs. In this case, on-chip buffers (NV, CI, and RP) are shared across the multiple PEs while each PE has its own multiplier, accumulator, index registers, and writing control unit.

B. PE LOAD BALANCING VIA MATRIX TILING AND TILE PAIR SCHEDULING

Though a single PE would be sufficient in resource-constrained systems, we can also employ multiple PEs to improve the throughput of the accelerator. When using multiple PEs, we can divide operand matrices A and B into multiple tiles while supplying a pair of the tiles (each from A and B matrices) to each PE. In this case, load balancing between the PEs will be a crucial factor for performance since the overall execution time will be dependent on the longest execution time among the PEs. For optimal load balancing, we exploit the following characteristics of SpMM with CSR format: we can calculate the load from the input CSR-formatted matrices in advance. Before executing the SpMM in the accelerator, the host CPU calculates the amount of the loads which will be assigned to each PE. The host CPU then triggers the transfer of the input matrices (from the main memory to the accelerator on-chip buffer) to execute SpMM.

For optimal load balancing, we sample[1] the operation count (i.e., loads counted by the number of required MAC operations) from the matrices A and B. We then divide the A and B matrices into multiple tiles and make pairs of the tiles so that the required operation count with each tile pair (each tile from the matrix A and B) can be as even as possible. In addition, to avoid the stalls due to resource contentions during the SpMM execution, we also need to make the assigned loads mutually exclusive to each PE, which in turn eliminates the resource contention (e.g., on-chip buffer bank conflict). Consequently, our tile pair scheduling scheme makes each PE access different scalar elements and row vectors in a single scheduling round. For accomplishing both load balancing and stall avoidance, when tiling the input matrices (A and B), we cut the matrix A both horizontally and vertically (i.e., tiled by two dimensions) while cutting the matrix B horizontally (i.e., tiled by only one dimension).

For horizontal division of the matrix A, we make the counts of non-zero elements in each horizontally divided tile as even as possible. It makes the assigned non-zero element count evenly distributed to each PE. For the second step, we vertically divide the matrix A so that the number of operation counts can be evenly distributed to each PE. It makes the assigned load to each PE as balanced as possible, leading to the high utilization rate of the PEs (i.e., the idle time of each PE can be minimized). For vertical division of the matrix A, we first calculate the total operation count (ops_{total}) as follows:

$$ops_{total} = \sum_{i=0}^{Col_A - 1} nzC_A^i \times nzR_B^i \tag{1}$$

where nzC_A^i and nzR_B^i correspond to the number of non-zero elements in column index i of matrix A and row index i of matrix B, respectively. Col_A means the column width of the matrix A. We vertically divide the matrix A so that each tile has $\lceil \frac{ops_{total}}{PE} \rceil$ operation count. Since we sample 10% of the rows in the matrix A, we scale the number of the assigned rows to each PE by 10×.

Figure 7 describes our matrix tiling under N PEs. As a result of the division of the matrix A, we have N × N tiles (from $tile_A$ (0, 0) to $tile_A$ (N-1, N-1)) in the case where N PEs are available. When vertically dividing the matrix A into the tiles, we also horizontally divide the matrix B so that the row height of the divided matrix B (TR_0 to TR_{N-1}) can be identical to the column width of the divided matrix A.

Figure 8 demonstrates a tile pairing (each from matrix A and B) and PE scheduling algorithm. When scheduling the divided tiles into the multiple PEs, we use the following method. Firstly, we pair the $tile_A$ (i, $(i + k - 1)\%N$) and $tile_B$ ($(i + k - 1)\%N$), which will be assigned to the same PE in a single scheduling round where i is the assigned PE number (PE0–PE(N-1)) and k is the scheduling round (from 1 to N). In this way, we can assign different scalar values in the matrix A and different row vectors in the matrix B to multiple PEs, which can avoid resource contention (i.e., the scalar values and row vectors are not shared between the PEs in the same scheduling round).

Figure 9 illustrates an example of our matrix tiling and scheduling to two PEs (PE0 and PE1). Please note that the

[1]Please note that we only sample 10% of the rows from the matrix A to calculate the load.

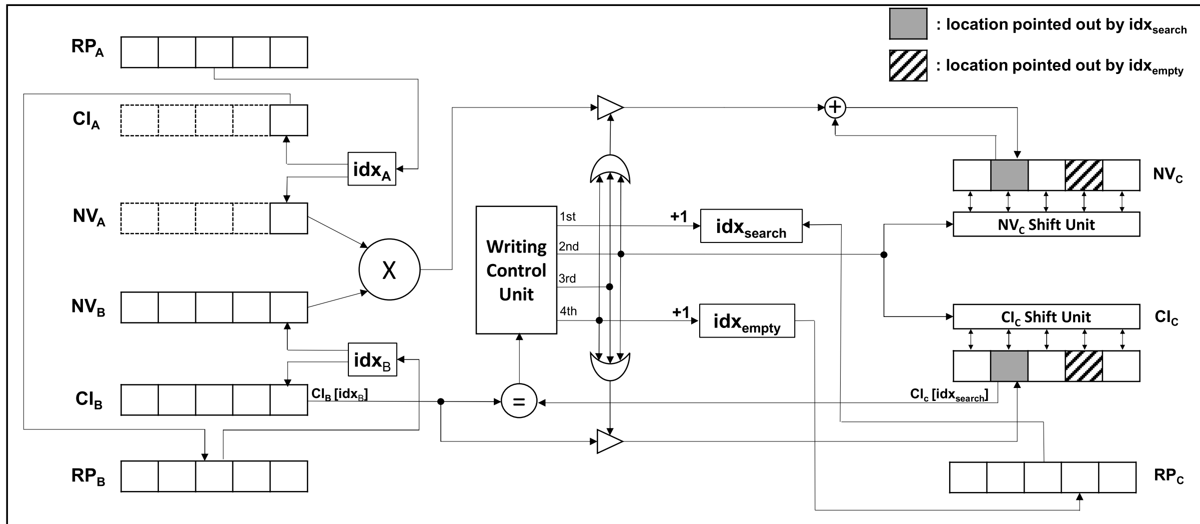


FIGURE 6. Hardware architecture for a single processing element (PE) of our row-wise product-based matrix multiplication with CSR format.

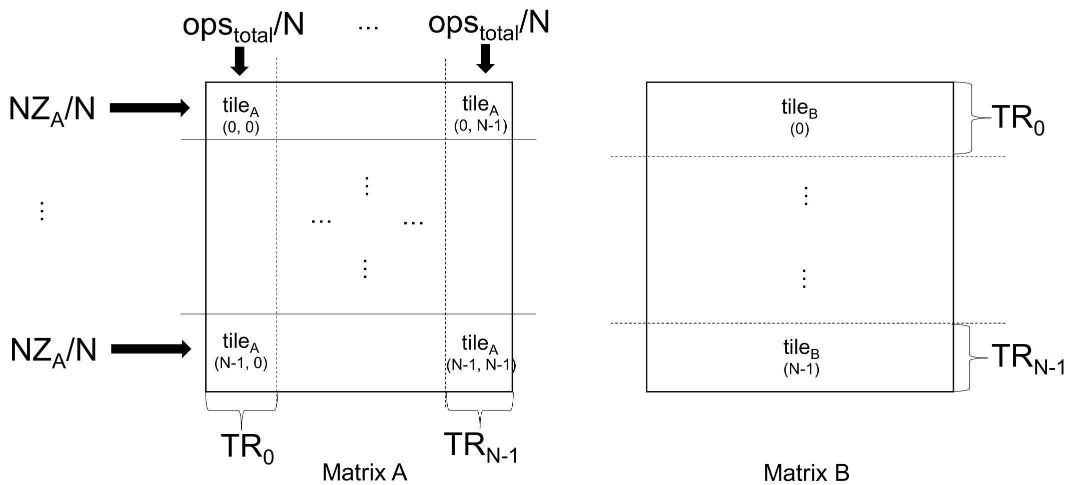


FIGURE 7. Operation count-based matrix tiling.

	Pseudocode for Tile Pairing and PE Scheduling
	Inputs : Sched_round, N, $tile_A$ (0, 0) – (N-1, N-1), $tile_B$ (0) – (N-1)
1	**for** k = 1 to (Sched_round)
2	**for** i = 0 to (N - 1)
3	schedule $tile_A$ (i, $(i+k-1)\%N$) and $tile_B$ ($(i+k-1)\%N$) to PE i
4	**endfor**
5	**endfor**

FIGURE 8. Pseudocode for tile pairing and PE scheduling. Sched_round means the number of scheduling round.

row sampling in the matrix A is omitted in the example shown in Figure 9. We have 5×5 input matrices A and B, generating the 5×5 output matrix C. For horizontal division of the matrix A, we divide the matrix so that each tile has equal (or as equal as possible) number of the non-zero elements. Thus, we horizontally divide the matrix A with the solid line, making three non-zero elements exist in each divided sub-matrix ({a, b, c} and {d, e, f}). When vertically dividing the matrix A, we calculate the ops_{total} when performing A $\times$ B, which results in the ops_{total}=8. Since we have two PEs, we need to vertically cut the matrix A so that each PE has as even operation counts as possible. As a result, when including the d, c, f, and a in the left-side sub-matrix, the operation count of the left-side sub-matrix becomes 4 which is equal to $\lceil \frac{ops_{total}}{PE} \rceil$. Obviously, the operation count of the right-side sub-matrix is also equal to 4, implying that the required MAC operations are evenly distributed to each PE. Thus, we vertically divide the matrix A with the dotted line so that the left-side sub-matrix has d, c, f, and a while the right-side sub-matrix has b and e. The horizontal division of

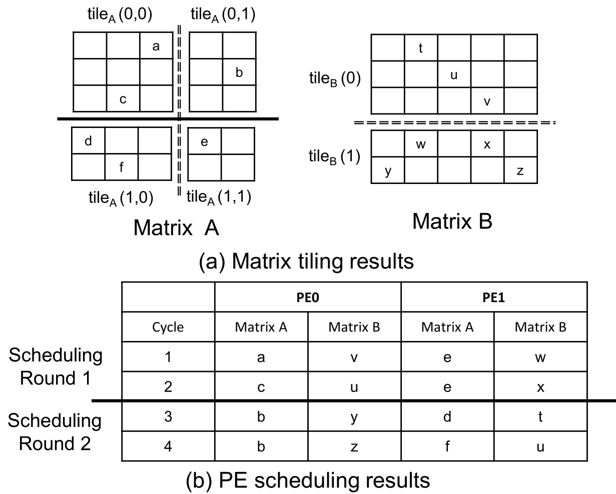


FIGURE 9. Operation count-based matrix tiling. For better understanding, we illustrate the input matrices as dense format. Please note that the matrix tiling is performed with the CSR-formatted operand matrices.

the matrix B follows the vertical division of the matrix A. Thus, we horizontally divide the matrix B so that the upper and lower tiles have 3 (TR_0) and 2 (TR_1) rows, respectively. When assigning the tiles to each PE, we assign $tile_A$ (0, 0) and $tile_A$ (0, 1) to the PE0 while $tile_A$ (1, 0) and $tile_A$ (1, 1) to the PE1. To avoid resource contention, in the first scheduling round, we first schedule the $tile_A$ (0, 0) with $tile_B$ (0) to the PE0 and the $tile_A$ (1, 1) with $tile_B$ (1) to the PE1. In the second scheduling round, the remaining tile pairs are scheduled: the $tile_A$ (0, 1) with $tile_B$ (1) to the PE0 and the $tile_A$ (1, 0) with $tile_B$ (0) to the PE1.

Though we have illustrated a matrix tiling and scheduling with a dense format for better understanding, it can also be applied to CSR formatted matrices. When tiling the matrices, the horizontal division of the matrix A can be done by referring to the RP_A with counting the number of non-zero elements for each row. For the vertical division, we can also refer to the CI_A (by counting the number of array elements in a certain column) and RP_B (by calculating $RP_B[i+1] - RP_B[i]$ where i is a matched row index of the matrix B) to calculate the operation counts. When scheduling the tiles into the multiple PEs, we can set different initial values for idx_A, i, and idx_{empty} (variables shown in Figure 5) of each PE.

Calculation of the ops_{total} and matrix tiling and pairing are done outside of the accelerator. Before we execute the SpMM accelerator, matrix tiling and tile pair scheduling are performed in the host CPU. When executing the SpMM accelerator, the host CPU can send the information of the tile pair and scheduling information (along with the NV_A, CI_A, RP_A, NV_B, CI_B, and RP_B) to the SpMM accelerator so that each PE can have different initial values for the registers in the PE. We have also evaluated the latency overhead of the matrix tiling and tile pair scheduling, which can be performed with a negligible latency overhead (for details, see Section V-D).

TABLE 1. Our 4-PE sparse MM hardware implementation results on Xilinx ZCU106 FPGA platform. Our implementation is verified with 64 × 64 input matrices (for both A and B) while varying the input sparsity.

	Used	Available	Utilization (%)
LUT as Logic	90,151	230,400	39.13
LUT as Distributed RAM	16,426	101,760	16.14
CARRY8	4,321	28,800	15.00
LUT as Shift Register	798	101,760	0.78
F7/F8 Muxes	189	230,400	0.08
BUFG	1	64	1.56
Register	107,513	460,800	23.33
Block RAM	93	312	29.81
DSPs	335	1,728	19.39

C. FPGA PROTOTYPING AND IMPLEMENTATION

We have implemented our SpMM hardware accelerator on Xilinx ZCU106 FPGA platform. Our accelerator can perform the matrix multiplication with 32-bit floating point elements. Table 1 summarizes the implementation results of our hardware. Our 4-PE SpMM hardware is synthesized as 214.270 MHz clock frequency. Our 4-PE hardware implementation shows moderate resource utilization on Xilinx ZCU106 while power consumption is 6.18 W in ZCU106. Please note that logic implementation can further be optimized when our hardware is implemented as application specific integrated circuits (ASICs), resulting in higher clock frequency and lower power consumption.

V. EVALUATIONS

A. METHODOLOGY

For evaluations of our SpMM hardware accelerator, we have implemented a cycle-accurate simulator. The cycle-level behavior of our hardware architecture is based on our FPGA prototype implementation. We compare performance of our hardware architecture with that of the systolic arrays (128 × 128 and 256 × 256) with 1 GHz clock frequency, which is simulated with Scale-SIM [10]. Please note that 128×128 and 256 × 256 systolic arrays are widely used in contemporary DNN accelerators (e.g., Google TPUs [2]). For execution of matrix multiplication in systolic arrays, we cut the input matrices A and B into multiple tiles so that the tile from the matrix A is streamed into the systolic array and the tile from the matrix B is stationary in the systolic array (i.e., weight stationary).

We demonstrate the results by varying the number of PEs in our design. In this paper, we use three PE configurations: 4PE, 16PE, and 32PE. The 4PE[2] design is our FPGA prototype version. 16PE[3] and 32PE[4] designs are iso-power designs against one matrix multiply unit (MXU) in Google TPUv4i and TPUv1 [2], respectively. Considering our SpMM accelerator performs matrix multiplication with 32-bit floating

[2]A single PE power consumption of our SpMM accelerator is ≈1.55W.

[3]TPUv4i MXU power is about 23.3W at 1GHz clock frequency with a utilization rate of 0.53 according to [2]. The iso-power PE count is 15.1≈16.

[4]TPUv1 MXU power is about 57.1W at 1GHz clock frequency with a utilization rate of 0.53 according to [2]. The iso-power PE count is 36.9≈32 (power-of-two).

TABLE 2. Matrices used for our evaluations [7], [11].

Matrix	Matrix abbreviation	Dimension	Density
web-Google	wg	916k × 916k	6.10E-06
mario002	m2	390k × 390k	1.30E-05
amazon0312	az	401k × 401k	1.90E-05
m133-b3	mb	200k × 200k	2.00E-05
scircuit	sc	171k × 171k	3.20E-05
p2pGnutella	pg	63k × 63k	3.70E-05
offshore	of	260k × 260k	6.20E-05
cage12	cg	130k × 130k	1.10E-04
2cubes-sphere	cs	101k × 101k	1.50E-04
filter3D	f3	106k × 106k	2.40E-04
ca-CondMat	cc	23k × 23k	3.50E-04
wikiVote	wv	8.3k × 8.3k	1.50E-03
poisson3Da	p3	14k × 14k	1.80E-03
facebook	fb	4k × 4k	1.10E-02

point while TPUs only support 8-bit integer or 16-bit bfloat format, our iso-power designs in the evaluation is based on the conservative assumption.

For benchmarking our SpMM accelerator versus systolic arrays, we use matrices from SuiteSparse [11], which is composed of 14 various matrix dimensions and densities as shown in Table 2. For simulations, we have synthetically generated input matrices (A and B with the same dimension) by referring to the dimension and non-zero densities of 14 benchmarks in SuiteSparse. Please note that we use an evaluation methodology similar to that presented in [7].

B. PERFORMANCE

Table 3 summarizes the latency comparison between our SpMM hardware accelerator and the systolic arrays. For our SpMM results shown in Table 3, we have employed our operation count-based load balancing scheme.

In the case of 32PE-SpMM hardware accelerator, one can obtain 47.9× and 13.6× speedups, on average (geometric mean), as compared to the cases of 128 × 128 and 256 × 256 systolic arrays, respectively. Even when using 16PE-SpMM hardware accelerator, one can still gain 8.8× and 2.5× speedups, on average, as compared to the cases of 128 × 128 and 256 × 256 systolic arrays, respectively. In the case of 4PE-SpMM hardware accelerator, though the latency is increased (i.e., performance loss) as compared to the cases of 128 × 128 and 256 × 256 systolic arrays (on average), our SpMM hardware accelerator still leads to better performance when running the matrix multiplications with large matrices (e.g., wg, m2, az, mb, and pg).

When using the systolic arrays, as the input matrix size increases, the latency also significantly increases. This is because the systolic array cannot remove or skip the ineffectual operations with the zero-valued elements. On the contrary, our SpMM hardware accelerator performs MAC operations only with non-zero elements by exploiting the row-wise product-based MM, resulting in a proportional increase in the latency as the number of non-zero elements in the input matrices increases. It also means our SpMM accelerator is much more scalable as compared to the systolic arrays when running the SpMM.

While our SpMM accelerator shows much higher effectiveness in the case of large input matrices than the case of small ones, our SpMM shows worse performance when the input size is small (e.g., wv, p3, and fb). This is because our SpMM hardware should be implemented with a lower clock frequency (due to higher logic complexity) as compared to the systolic arrays. In addition, although the systolic arrays cannot skip the zero-valued elements, smaller input matrices will have less number of tiles (thus, less number of iterations and cycles for loading and streaming in the systolic array) and lower input matrix streaming latency for the systolic array. It leads to better performance when using the systolic array (with higher clock frequency) as compared to using our SpMM accelerator.

C. EFFECTIVENESS OF OPERATION COUNT-BASED MATRIX TILING FOR LOAD BALANCING

Our operation count-based matrix tiling results in further performance improvement as compared to the fixed tiling (FT: matrix tiling with the regular row and column length) and element count-based tiling (ET: matrix tiling based only on the non-zero element count while not considering the operation count). Please note that the non-zero element count-based tiling only considers the number of non-zero elements within each tile (as evenly as possible) while our operation count-based matrix tiling considers how many MAC operations are performed within each PE. Since a single element can be used for MAC operations with different elements (i.e., each element will have different MAC operation counts), only considering the non-zero element count results in sub-optimal results for load balancing between PEs.

Table 4 summarizes performance improvements of our operation count-based tiling over ET and FT across three SpMM configurations (32PE, 16PE, and 4PE). In the case of 4PE-SpMM, the performance improvement of our operation count-based tiling over the FT and ET is 5.7% and 5.8%, respectively, on average (geometric mean). When employing our operation count-based tiling to the 16PE-SpMM, one can obtain performance improvements by 4.0% and 3.9%, on average, as compared to the FT and ET, respectively. In the case of 32PE-SpMM, our operation count-based tiling brings performance improvements of 8.5% and 6.3% compared to the FT and ET, respectively. It means the load balancing can greatly affect performance when employing a large number of PEs.

In the case of *pg*, our operation count-based tiling leads to performance improvement by up to 25.1% as compared to the FT. While performance of the FT can vary depending on the non-zero element distribution patterns, our operation count-based tiling can show the stable performance without being significantly affected by the data distribution patterns.

On the contrary, in the case of *f3* with 4PE-SpMM, the fixed tiling shows better performance over the ET and our operation count-based tiling. Depending on the non-zero

TABLE 3. Speedup results of our SpMM hardware compared to the 128 × 128 and 256 × 256 systolic arrays. The value over 1.0 means that the our SpMM hardware shows better performance compared to the systolic arrays, and vice versa.

	vs. 128×128 Systolic Array			vs. 256×256 Systolic Array		
	32PE-SpMM	16PE-SpMM	4PE-SpMM	32PE-SpMM	16PE-SpMM	4PE-SpMM
wg	14674.02379	2118.43207	49.42830	4119.74459	594.75160	13.87704
m2	2087.52538	294.69626	8.63732	586.34404	82.77427	2.42605
az	1199.00807	188.34333	6.64816	336.73590	52.89536	1.86710
mb	1560.02422	241.85169	6.30393	438.35629	67.95869	1.77136
sc	574.82223	89.84256	2.16457	161.43533	25.23174	0.60791
pg	657.22355	117.85340	3.61265	184.61512	33.10520	1.01480
of	137.88166	21.22312	0.64881	38.72968	5.96138	0.18225
cg	39.59426	7.12442	0.28410	11.14203	2.00485	0.07995
cs	19.64543	3.79751	0.16945	5.52568	1.06813	0.04766
f3	10.63863	1.99471	0.10005	2.99278	0.56114	0.02815
cc	5.28550	1.21433	0.04076	1.49588	0.34367	0.01153
wv	0.48767	0.11992	0.00490	0.14389	0.03538	0.00144
p3	0.29143	0.06947	0.00353	0.08347	0.01990	0.00101
fb	0.01821	0.00453	0.00009	0.00541	0.00135	0.00003
GEOMEAN	47.93147	8.76579	0.28873	13.59559	2.48639	0.08190

TABLE 4. Performance improvement of our matrix tiling scheme over fixed tiling (FT) and element count-based tiling (ET).

	32PE-SpMM		16PE-SpMM		4PE-SpMM	
	over FT	over ET	over FT	over ET	over FT	over ET
wg	1.940%	0.893%	5.301%	2.644%	19.467%	6.125%
m2	5.596%	0.999%	1.264%	0.836%	2.739%	10.865%
az	0.260%	1.165%	1.689%	3.085%	6.668%	5.573%
mb	4.738%	3.612%	0.189%	1.878%	1.360%	5.885%
sc	24.464%	26.310%	21.906%	22.555%	24.225%	10.108%
pg	25.099%	4.545%	1.555%	2.384%	6.884%	11.969%
of	8.282%	5.893%	2.646%	0.565%	1.266%	7.913%
cg	2.099%	0.832%	2.071%	1.085%	1.581%	3.666%
cs	4.185%	10.165%	1.155%	2.520%	0.462%	0.563%
f3	4.287%	8.931%	4.781%	9.299%	-0.282%	0.531%
cc	2.007%	0.782%	8.413%	7.574%	14.096%	11.219%
wv	17.143%	8.571%	1.769%	0.448%	3.931%	2.498%
p3	18.504%	13.529%	0.324%	0.767%	0.815%	4.053%
fb	4.454%	4.471%	4.490%	0.644%	0.445%	0.889%
GEOMEAN	8.484%	6.278%	3.982%	3.874%	5.726%	5.775%

data distribution patterns, the performance results could be diverse across the ET, FT, and our operation count-based tiling. We leave the detailed investigation on the impact of data distribution patterns across the ET, FT, and our operation count-based tiling on performance as our future work.

D. MATRIX TILING AND TILE PAIR SCHEDULING OVERHEAD

Our operation count-based matrix tiling and tile pair scheduling require an additional latency (referred to as 'tiling latency') to calculate the operation counts that will be assigned to each PE and divide the CSR-formatted input matrices into multiple tiles. We have measured the tiling latency by using gem5 architectural simulation tool [12] and calculated the matrix tiling overhead. Matrix tiling overhead can be given as L_{tile}^{CPU}/L_{Acc}, where L_{tile}^{CPU} denotes tiling latency in the host CPU and L_{Acc} denotes execution latency of the SpMM accelerator. We set the cache parameters of gem5 to Intel Xeon Gold [13] (which is commonly used CPU in servers or datacenters) as similar as possible. The core parameters follow the gem5 O3 model parameters.

Table 5 shows the matrix tiling overhead of our operation count-based matrix tiling across various SpMM configurations. As we have more number of PEs in the SpMM, the tiling overhead will increase because we need to tile the A and B matrices in a finer granularity. When using 4PE-SpMM and 16PE-SpMM configurations, the tiling overheads are almost negligible (0.00033% and 0.01015% when using 4PE and 16PE configurations, respectively). In the case of 32PE-SpMM, latency overhead is only 0.05504%. It implies that even considering the latency overhead of the matrix tiling, our operation count-based matrix tiling can be effectively deployed in real servers and datacenters.

TABLE 5. Matrix tiling overhead of our operation count-based matrix tiling across various SpMM configurations.

	32PE-SpMM	16PE-SpMM	4PE-SpMM
wg	0.05918%	0.00887%	0.00024%
m2	0.05975%	0.00862%	0.00028%
az	0.03066%	0.00473%	0.00014%
mb	0.23290%	0.03568%	0.00099%
sc	0.12057%	0.01921%	0.00043%
pg	2.38530%	0.42445%	0.01136%
of	0.00679%	0.00110%	0.00003%
cg	0.01265%	0.00231%	0.00009%
cs	0.01208%	0.00230%	0.00010%
f3	0.00432%	0.00084%	0.00004%
cc	0.17457%	0.04062%	0.00135%
wv	0.22454%	0.05548%	0.00225%
p3	0.02852%	0.00693%	0.00035%
fb	0.06416%	0.01529%	0.00030%
GEOMEAN	0.05504%	0.01015%	0.00033%

VI. CONCLUSION

Sparse matrix multiplication is a key computation kernel for many real-world applications. In this paper, we have proposed a sparse matrix multiplication (SpMM) accelerator that receives compressed sparse row (CSR) formatted input matrices. Our SpMM accelerator exploits a row-wise product algorithm for MM, which has several advantages over the inner product and outer product. In addition, we have also proposed an operation count-based load balancing scheme that divides input matrices into multiple tiles (and also schedules these tiles) for parallel execution on the multiple processing elements (PEs). According to our evaluations, our 32PE-SpMM accelerator shows $13.6\times - 47.9\times$ speedup over TPU-like systolic arrays, on average. Moreover, our operation count-based load balancing scheme shows better performance over the fixed tiling and non-zero element count-based tiling by up to 8.5% and 6.3%, respectively, while only incurring the matrix tiling overhead in the host CPU up to 0.06%.

We also summarize our future work as follows:

- We will quantitatively evaluate our accelerator and load balancing scheme with the state-of-the-art accelerators;
- We will further investigate the impact of non-zero data distribution patterns of the matrix A and B on performance across the ET, FT, and our operation count-based tiling;
- We will extend our hardware accelerator to enable the multiplication and accumulation operations with various sizes of the matrix elements such as 8-bit, 16-bit, and 64-bit.

REFERENCES

[1] *Nvidia v100*. Accessed: Jan. 27, 2022. [Online]. Available: https://www.nvidia.com/en-us/data-center/v100/

[2] N. P. Jouppi, D. H. Yoon, M. Ashcraft, M. Gottscho, T. B. Jablin, G. Kurian, J. Laudon, S. Li, P. Ma, X. Ma, T. Norrie, N. Patil, S. Prasad, C. Young, Z. Zhou, and D. Patterson, ''Ten lessons from three generations shaped Google's TPUv4i: Industrial product,'' in *Proc. ACM/IEEE 48th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2021, pp. 1–14.

[3] A. Gondimalla, N. Chesnut, M. Thottethodi, and T. N. Vijaykumar, ''SparTen: A sparse tensor accelerator for convolutional neural networks,'' in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2019, pp. 151–165.

[4] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, ''SIGMA: A sparse and irregular GEMM accelerator with flexible interconnects for DNN training,'' in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2020, pp. 58–70.

[5] Z. Zhang, H. Wang, S. Han, and W. J. Dally, ''SpArch: Efficient architecture for sparse matrix multiplication,'' in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2020, pp. 261–274.

[6] R. Hojabr, A. Sedaghati, A. Sharifian, A. Khonsari, and A. Shriraman, ''SPAGHETTI: Streaming accelerators for highly sparse GEMM on FPGAs,'' in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit. (HPCA)*, Feb. 2021, pp. 84–96.

[7] N. Srivastava, H. Jin, J. Liu, D. Albonesi, and Z. Zhang, ''MatRaptor: A sparse-sparse matrix multiplication accelerator based on row-wise product,'' in *Proc. 53rd Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2020, pp. 766–780.

[8] G. Zhang, N. Attaluri, J. S. Emer, and D. Sanchez, ''Gamma: Leveraging Gustavson's algorithm to accelerate sparse matrix multiplication,'' in *Proc. 26th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2021, p. 687—701.

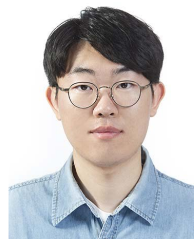
[9] J. Kwon, J. Kong, and A. Munir, ''Sparse convolutional neural network acceleration with lossless input feature map compression for resource-constrained systems,'' *IET Comput. Digit. Techn.*, vol. 16, no. 1, pp. 29–43, Jan. 2022.

[10] A. Samajdar, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna, ''SCALE-sim: Systolic CNN accelerator simulator,'' 2018, *arXiv:1811.02883*.

[11] T. A. Davis and Y. Hu, ''The university of Florida sparse matrix collection,'' *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1–25, Nov. 2011.

[12] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, ''The gem5 simulator,'' *SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 1—7, 2011.

[13] *Intel Xeon Gold Processors*. Accessed: Dec. 23, 2021. [Online]. Available: https://www.intel.com/content/www/us/en/products/details/processors/xeon/scalable/gold.html



JONG HUN LEE received the B.S. and M.S. degrees in electronics engineering from Kyungpook National University, in 2020 and 2022, respectively. He is currently with LX Semicon. His research interests include convolutional neural network acceleration, data compression, and FPGA-based design.

BEOMJIN PARK received the B.S. degree in electronics engineering from Kyungpook National University, in 2022. He is currently with Samsung Electronics. His research interests include machine learning system, deep neural network acceleration, and FPGA-based design.

JOONHO KONG (Member, IEEE) received the B.S. degree in computer science from Korea University, in 2007. He also received the MS and PhD degrees in computer science and engineering from Korea University, in 2009 and 2011, respectively. He was a Postdoctoral Research Associate at the Department of Electrical and Computer Engineering, Rice University, from 2012 to 2014. He was a Senior Engineer at Samsung Electronics, from 2014 to 2015. He is currently an Associate Professor with the School of Electronics Engineering, Kyungpook National University. His research interests include computer architecture, heterogeneous computing, embedded systems, and hardware/software co-design.

ARSLAN MUNIR (Senior Member, IEEE) received the M.A.Sc. degree in electrical and computer engineering from the University of British Columbia, Vancouver, Canada, in 2007, and the Ph.D. degree in electrical and computer engineering from the University of Florida, Gainesville, FL, USA, in 2012.

From 2007 to 2008, he worked as a software development engineer at the Embedded Systems Division, Mentor Graphics Corporation. He was a Postdoctoral Research Associate at the Electrical and Computer Engineering Department, Rice University, Houston, TX, USA, from May 2012 to June 2014. He is currently an Associate Professor with the Department of Computer Science, Kansas State University. His current research interests include embedded and cyber-physical systems, secure and trustworthy systems, parallel computing, artificial intelligence, and computer vision. He received many academic awards, including the Doctoral Fellowship from the Natural Sciences and Engineering Research Council (NSERC), Canada. He was a recipient of the Gold Medals for the best performance in electrical engineering, the Gold Medals and the Academic Roll of Honor for securing rank one in pre-engineering provincial examinations (out of approximately 300,000 candidates).