Efficient FPGA-based Sparse Matrix-Vector Multiplication with Data Reuse-aware Compression

Shiqing Li, Di Liu, Weichen Liu

***Abstract*—Sparse matrix-vector multiplication (SpMV) on FPGAs has gained much attention. The performance of SpMV is mainly determined by the number of multiplications between non-zero matrix elements and the corresponding vector values per cycle. On the one side, the off-chip memory bandwidth limits the number of non-zero matrix elements transferred from the off-chip DDR to the FPGA chip per cycle. On the other side, the irregular vector access pattern poses challenges to fetch the corresponding vector values. Besides, the read-after-write (RAW) dependency in the accumulation process shall be solved to enable a fully pipelined design. In this work, we propose an efficient FPGA-based sparse matrix-vector multiplication accelerator with data reuse-aware compression. The key observation is that repeated accesses to a vector value can be omitted by reusing the fetched data. Based on the observation, we propose a reordering algorithm to manually exploit the data reuse of fetched vector values. Further, we propose a novel compressed format called data reuse-aware compressed (DRC) to take full advantage of the data reuse and a fast format conversion algorithm to shorten the preprocessing time. Meanwhile, we propose an HLS-friendly accumulator to solve the RAW dependency. Finally, we implement and evaluate our proposed design on the Xilinx Zynq-UltraScale ZCU106 platform with a set of sparse matrices from the SuiteSparse matrix collection. Our proposed design achieves an average 1.18x performance speedup without the DRC format and an average 1.57x performance speedup with the DRC format w.r.t. the state-of-the-art work respectively.**

***Index Terms*—SpMV, FPGA, Data Reuse, Throughput**

I. Introduction

SPARSE matrix-vector multiplication (SpMV) is a crucial primitive in multiple areas such as machine learning and economic modeling [1]. Specifically, SpMV is a kernel widely used in iterative linear solvers [2] and sparse fully connected layers in neural networks [3] [4] [5] [6]. With the increasing problem scale, SpMV accounts for over 75% execution time in these applications [7]. Thus, a high-performance accelerator is required. With high design flexibility and large on-chip memory, FPGA is a promising solution [8] [9] [10] [11] [12] [13] [14] [15]. In this work, we mainly target SpMV used in applications which require data in double-precision floating-point type to achieve high computational precision. The long delay of *double* addition aggravates the read-after-write (RAW) dependence and thus the fully-pipelined accumulator necessitates a dedicated design. Further, the data type also directly affects the final performance. Consequently, we target [8] which uses the same data type as the state-of-the-art work instead of [11] [12] [13] [14] [15]. The dataflow approach [8] efficiently utilize the off-chip bandwidth in which the execution of SpMV is fully pipelined to overlap the computation latency with the off-chip memory access latency. However, it failed to take advantage of large on-chip memories which leads to bad performance.

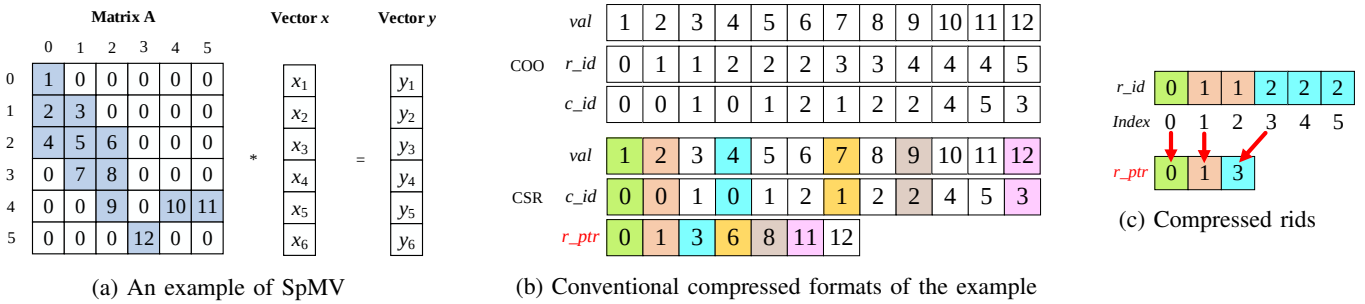
Large on-chip memories of FPGAs can buffer the vector to reduce off-chip traffic and facilitate the reuse of vector values. The corresponding vector values of the transferred non-zero matrix elements per cycle shall be timely fetched to perform multiplications, thus ensuring effective utilization of the off-chip memory bandwidth. However, typical on-chip memories on FPGAs only have two access ports and thus two vector values can be fetched. The throughput mismatch between the off-chip memory bandwidth and the vector buffer stalls the fully-pipelined accelerator, resulting in ineffective utilization of the off-chip memory bandwidth. To solve the throughput mismatch, there are mainly two solutions.

The first solution is to increase the throughput of the vector buffer. To achieve higher throughput, the most straightforward approach is holding multiple copies of the vector on-chip. However, it cannot be applied to applications with large vectors. A better approach is to partition the vector buffer into multiple sub-banks. Each bank holds a part of the vector and processes the corresponding memory accesses. Array partitioning [16] [17] [18] has been widely studied for arrays with regular access patterns. However, it is hard to find a fixed pattern to partition the vector buffer for SpMV with irregular access patterns. Overall, although the throughput mismatch can be solved, data reuse is not exploited and repeated memory accesses to a vector value are performed.

The other solution is that we can exploit the data reuse to reduce the number of memory accesses. In this work, we observe that repeated memory accesses to one vector value can be omitted by reusing the fetched data and thus fewer memory accesses are performed. Following this observation, we propose a reordering algorithm to optimize data reuse. Compared to the previous work [19], we propose a novel data reuse-aware compressed format (DRC) to further exploit the benefits of data reuse and a fast format conversion algorithm. Our key contributions are as follows.

- We observe that repeated memory accesses to one vector value can be omitted by reusing the fetched data and propose a data reordering algorithm to exploit the data reuse of vector values.
- We propose a novel data reuse-aware compressed format (DRC) to take full advantage of data reuse. In addition, a fast format conversion algorithm is proposed to shorten the preprocessing time.

S. Li and W. Liu are with the School of Computer Science and Engineering, Nanyang Technological University, Singapore. W. Liu is the corresponding author. D. Liu is with the Department of Computer Science, Norwegian University of Science and Technology. Email: shiqing.li@ntu.edu.sg, di.liu@ntnu.no, liu@ntu.edu.sg



(a) An example of SpMV

(b) Conventional compressed formats of the example

(c) Compressed rids

Fig. 1: Sparse matrix-vector multiplication and conventional compressed formats

- We propose an HLS-friendly accumulator design to solve the RAW dependency. We implement and evaluate the hardware with a set of matrices from the SuiteSparse matrix collection on Xilinx ZCU106 [20]. The experimental results show that our design achieves an average 1.18x performance speedup without the DRC and an average 1.57x performance speedup with the DRC w.r.t. the state-of-the-art work [8] respectively.

II. Background and Related Work

In this section, we introduce some background about SpMV. Then, we analyze the challenges of DATAFLOW designs for SpMV. At the last of this section, we discuss related work.

A. Sparse matrix-vector multiplication

As shown in Eqn. 1, SpMV refers to the multiplication of a sparse matrix A by a dense vector x to generate a result vector y. To lower storage requirements, sparse matrices are stored in compressed formats which only hold nonzero matrix elements. Since non-zero matrix elements are randomly distributed, their row indices and column indices are required to locate the corresponding result vector values and input vector values separately. Thus, in general, compressed formats have three arrays to store values (*val*), row indices (*r_id*), and column indices (*c_id*) of non-zero matrix elements.

$$y_i = \sum_{i=0}^{Row} \sum_{j=0}^{Col} A_{ij} * x_j \quad if \quad A_{ij} \neq 0 \tag{1}$$

There are three commonly used compressed formats called COOrdinate (COO), Compressed Sparse Row (CSR), and Compressed Sparse Column (CSC). As shown in Fig. 1b, the most straightforward one is COO. In this format, it holds all the values, row indices, and column indices of non-zero matrix elements. To further lower the storage requirement, CSR and CSC are proposed. As shown in Fig. 1c, repeated row indices are compressed in CSR format and the index of the first element in each row is stored in *r_ptr*. For example, in Fig. 1c, the fourth element (whose value is 4) is the first element of the third row and its index is 3. Thus, $r\_ptr[2] = 3$. Further, we need to traverse the compressed *r_ptr* array to retrieve row indices of non-zero matrix elements. For example, in Fig. 1b, the index of the nonzero matrix element whose value is 3 is 2. Since 2 is greater than $r\_ptr[1]$ and is less than $r\_ptr[2]$, the row index of this element is 1. For CSC, nonzero matrix elements follow column-major order and repeated column indices are compressed. Both these two formats require extra control logic to retrieve the target row index or column index.

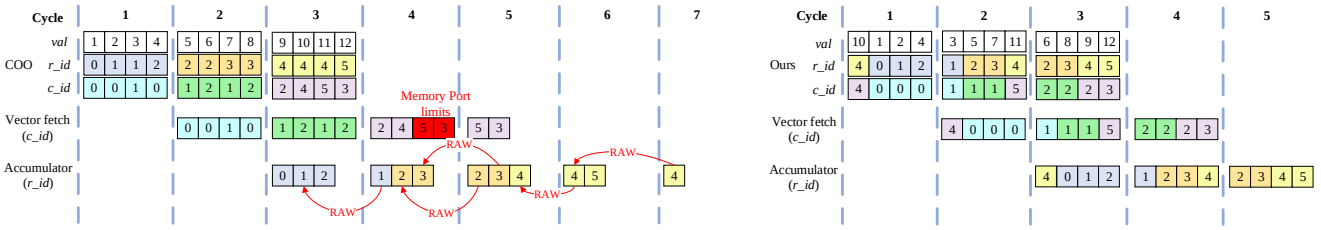
B. DATAFLOW-based design for SpMV and its challenges

The DATAFLOW approach is a task-level optimization directive and sub-modules of a DATAFLOW design run in parallel. Since it can overlap the computation latency with the off-chip memory access latency, it is an ideal approach for memory-intensive applications like SpMV. Although CSR and CSC further reduce the matrix size, additional cycles are required to obtain the target row or column indices. Especially, empty rows or columns make the time unpredictable, which could block the whole pipeline. As shown in Fig. 2, 7 cycles are required to get all the row indices. Thus, COO is preferred for FPGA designs using the DATAFLOW approach since it does not incur any control overhead. To increase the performance of SpMV, the existing challenges mainly include the following.

1) The Throughput Mismatch: Although the vector can be buffered on-chip to reduce the off-chip traffic, the limited throughput of the vector buffer hinders the timely fetching of the corresponding vector values of the transferred non-zero matrix elements per cycle, leading to ineffective utilization of the off-chip memory bandwidth. Thus, the main challenge is to deal with the mismatch between the throughput of the transferred non-zero matrix elements (i.e., the throughput of the off-chip memory bandwidth) and the throughput of the vector buffer. Traditional compressed formats only reduce the storage and ignore the throughput mismatch.

2) The Size of Sparse Matrices: Given the fact that the number of transferred non-zero matrix elements per cycle determines the performance upper bound of SpMV. Consequently, it is important to compress sparse matrices while considering the throughput mismatch issue.

3) Inherent Read-after-write Dependence: The read-after-write (RAW) dependence exists in the accumulation process of SpMV. Due to the long delay of the *double* addition, a dedicated fully pipelined accumulator design is required. Meanwhile, rows that have more non-zero matrix elements take more cycles to perform the accumulation. Although the dependence cannot be eliminated, its impact can be mitigated by reordering non-zero matrix elements.



(a) The execution simulation of COO

(b) The execution simulation of Ours

Fig. 3: The motivational example of intra-group reuse

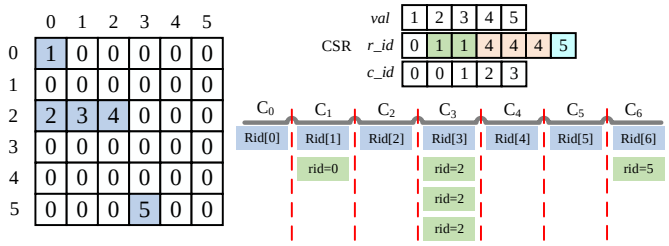


Fig. 2: Irregular control flow of CSR format

C. Related work

The importance of SpMV gains lots of attention. In [21], the authors design an accelerator based on CSC format. However, they assume that the memory access of the vector is sequential. [10] proposes a framework that can generate a specific accelerator for each matrix. However, this framework requires a quite long preprocessing time. In [8], the authors propose a high-performance dataflow engine for CPU-FPGA heterogeneous platform. They utilize the CPU part to fetch the corresponding vector values of non-zero matrix elements sequentially and the FPGA part performs computations. However, this design relies on a high-speed CPU counterpart that matches the speed of the FPGA design. In addition, they transfer repeating 64-bit vector values instead of 32-bit column indices which wastes lots of memory bandwidth. The authors in [9] maximize the data locality by clustering the randomly distributed non-zero matrix elements. However, the proposed design consumes plenty of logic resources and it is hard to apply this solution on a single FPGA board. Further, the preprocessing time is pretty long since it reorders the whole input vector and matrix. On the other side, SpMV is an important application of graph processing. In [11], the authors partition large-scale graphs to fit into on-chip RAMs and exploit BRAM resources from multiple FPGA boards. [12] vertically partitions the large-scale graphs to enlarge the partition size. In [13], the authors propose an HLS-based graph processing framework to facilitate the adoption of FPGAs. However, these designs mainly consider single-cycle addition and cannot be applied to SpMV with double-precision floating-point data.

In this work, we propose to solve the throughput mismatch by exploiting the data reuse of vector values. Further, we propose a novel compressed format called DRC to take full advantage of the data reuse and reduce the matrix size. To the best of our knowledge, this is the first paper that exploits data reuse for SpMV on FPGAs.

III. Motivational Example

A. Definitions

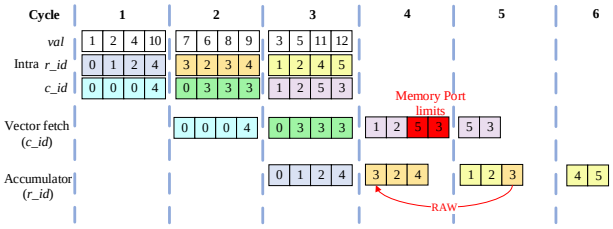
In this subsection, we first define some concepts. We call the non-zero matrix elements transferred in a cycle a *group*. If one *group* has multiple accesses to a vector value, we only perform one memory access and reuse the fetched data for the other accesses. This kind of data reuse is called intra-group reuse. Meanwhile, if the fetched data is reused in different groups, we call the data reuse inter-group reuse. Since a reused vector value is multiplied with non-zero matrix elements with the same column index, a column with multiple non-zero matrix elements is called a reusable column.

B. Motivational Example

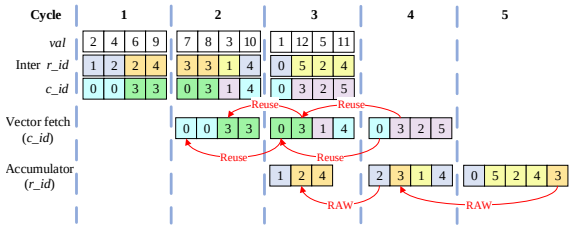
In this subsection, we present motivational examples to show the benefits of data reuse. We use the configuration of the Xilinx UltraScale ZCU106 platform on which four non-zero matrix elements are transferred to the FPGA chip per cycle. Given the fact that the vector buffer has two memory ports and hence up to two vector values can be fetched per cycle. We show simulations of motivational examples to help understand. Due to the limited page space, cycles for multiplications and additions are omitted. Since the design is fully pipelined, the correctness of simulations is ensured.

Since CSC and CSR require extra logic to get the target column or row index, we use COO as our baseline. The key observation is that repeated memory accesses of one vector value can be omitted by reusing the fetched data. As shown in Fig. 3a, column indices in the first *group* are [0, 0, 1, 0] and thus $x[0]$ are accessed three times in Cycle 2. Only the first memory access is served and the other two accesses are omitted by reusing the fetched data. Thus, the non-zero matrix elements in the first *group* can be paired with the corresponding vector values. However, the COO format does not ensure that all the non-zero matrix elements can be paired. Specifically, $x[5]$ and $x[3]$ cannot be fetched in Cycle 4 as shown in Fig. 3a. To solve this problem, we observe that $x[2]$ is accessed in both Cycle 3 and Cycle 4. In other words, the fetched vector value (i.e., $x[2]$) is not fully reused. Consequently, we reorder the corresponding non-zero matrix elements into one *group*. As shown in Fig. 3b, $x[2]$ is only accessed in the third *group* instead of two *group*s as in Fig. 3a. We also perform the same operations to the corresponding non-zero matrix elements of $x[0]$ and $x[1]$. Further, due to the RAW dependency, Cycle 6 and Cycle 7 are used to consume the remaining products as shown in Fig. 3a. As shown in Fig.

3b, non-zero matrix elements in the fifth row are scheduled to the former *groups* (e.g., the first *group*). After reordering, only four cycles (Cycle 2 to Cycle 5) are required.



(a) The execution simulation of intra-group reuse



(b) The execution simulation of inter-group reuse

Fig. 4: The motivational example of inter-group reuse

Further, only considering intra-group reuse is not enough in some cases. As shown in Fig. 4a, we fully utilize the throughput of the vector buffer and fully reuse the fetched data in each cycle. However, it still cannot pair all the non-zero matrix elements. For instance, the memory accesses in red boxes are not served in Cycle 4 of Fig. 4a. Thus, we consider reusing the fetched data in different groups (i.e., inter-group reuse). For example, $x[3]$ is reused in Cycle 2, Cycle 3, and Cycle 4 in Fig. 4b. However, it is only reused in Cycle 3 and re-accessed in Cycle 5 in Fig. 4a. As a result, only three cycles (Cycle 2 to Cycle 4) are required to fetch all the vector values in Fig. 4b. Meanwhile, we also observe that not all non-zero matrix elements are suitable for inter-group reuse. The reused vector values shall be read at first and then reused in the following groups. To fully utilize the memory bandwidth, two non-zero matrix elements in each reusable column are first scheduled as shown in Cycle 2 of Fig. 4b. If a reusable column only has one extra element to be reused besides the two elements, the final performance is the same as the case that only considers intra-group reuse. For example, if we only look at Cycle 2 and Cycle 3 in Fig. 4b, [0, 0, 3, 3] [0, 3, 1, 4] and [0, 0, 0, 1] [3, 3, 3, 4] are feasible. Thus, reusable columns refer to columns that have more than three elements.

IV. Hardware Architecture

In this section, we first show the overview of our proposed hardware accelerator. Then, we detail each component.

A. Overview

As shown in Fig. 5, the matrix is preprocessed offline and stored in the DDR. When the execution begins, non-zero matrix elements are transferred to the FPGA chip per cycle. After that, the *decoder* decodes and fetches the target vector values. Then, the paired non-zero matrix elements and vector values stream into Processing Elements (PEs). Once PEs receive the paired data, they perform multiplications and the products are pushed into the corresponding *accumulators* according to their row indices. The *accumulators* sum all the products up and write the final results back to the DDR.

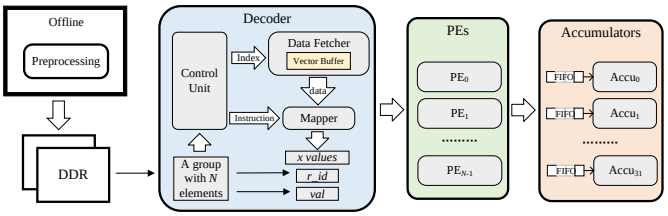


Fig. 5: Hardware Overview

B. Decoder

The *decoder* is the key part to solve the main challenge of SpMV, i.e., the throughput mismatch. It consists of three parts: the control unit, the data fetcher, and the mapper. When a group arrives, the control unit first generates control information according to column indices. The control information includes target memory addresses which are sent to the data fetcher and control instructions which are sent to the mapper. We list control instructions in Table I. When the data fetcher receives the memory addresses, it fetches values from the *Vector Buffer* and streams them to the mapper. When the mapper receives the control instructions and the fetched data, it maps vector values to non-zero matrix elements according to the control instructions. At last, the paired non-zero matrix elements and vector values stream into PEs alongside their row indices.

TABLE I: Control Instructions

Instruction	Definition
REUSE1	Reuse the first data fetched in previous cycles
REUSE2	Reuse the second data fetched in previous cycles
LOAD1	Use the first data fetched in this cycle
LOAD2	Use the second data fetched in this cycle
BUFFER1	Use and buffer the first data fetched in this cycle
BUFFER2	Use and buffer the second data fetched in this cycle

Recall that we consider intra-group reuse and inter-group reuse. For intra-group reuse, control instructions are directly generated. For instance, if the input column indices are [2, 2, 2, 1], the control instructions [`LOAD1`, `LOAD1`, `LOAD1`, `LOAD2`] are generated and memory addresses [2, 1] are sent to the data fetcher. For inter-group reuse, the fetched vector value and its address (i.e., the column index of non-zero matrix elements) shall be buffered. We only buffer one value and its index for each port in the *decoder*. The key reason is that all the elements in reusable columns are reused to solve the throughput mismatch and it is easy to replace the buffered data if we only buffer one reusable column. Specifically, when the current reusable column is exhausted, the buffered vector value and its index are directly replaced by the new data. Meanwhile, we use a fixed pattern of column indices (i.e., reuse_cindex1, reuse_cindex1, reuse_cindex2, reuse_cindex2) to identify reusable columns. Using this pattern, the off-chip memory bandwidth of the current group is fully utilized and the control overhead is little. When the buffered data shall be replaced, control instructions `BUFFER1` and `BUFFER2` are generated.

We use some specific examples to illustrate how the *decoder* works. Assume that three groups sequentially stream into the *decoder* whose column indices are [1, 1, 2, 2], [1, 2, 3, 4], [5, 5, 6, 6]. For the first group, the control unit sends the control instructions [BUFFER1, LOAD1, BUFFER2, LOAD2] to the mapper and memory addresses [1, 2] to the data fetcher. After that, [1, 2] and $[x[1], x[2]]$ are buffered in the control unit and the mapper separately. When the second group arrives, the control instructions [REUSE1, REUSE2, LOAD1, LOAD2] are generated. As a result, $[x[1], x[2], x[3], x[4]]$ are paired with the non-zero matrix elements. When the last group arrives, the same instructions as the first group is generated. Following these instructions, the buffered data is replaced by $[x[5], x[6]]$.

C. Processing Elements

In this work, PEs are double-precision floating point multipliers. Since this design is fully pipelined and the off-chip memory bandwidth limits the number of transferred non-zero matrix elements, the number of PEs equals the maximum number of transferred non-zero matrix elements. Specifically, there are N_{PE} PEs in our design. Meanwhile, deploying more PEs cannot improve the overall performance. In addition, since non-zero matrix elements and vector values have been paired and no dependence exists among different pairs, the workload is equally distributed to the available PEs. Ideally, one PE performs a multiplication per cycle.

```
void accumulator(double* input_stream,
                 double* output)
{
    double buffer[6];
    int index = 0;
    while (!input_stream.empty()){
#pragma HLS PIPELINE II=1
#pragma HLS DEPENDENCE variable=buffer false
        val_t input_data = input_stream.read();
        buffer[index] += input_data;
        index = (index == 5) ? 0 : index + 1;
    }
    *output = sum(buffer);
}
```

Fig. 6: Accumulator Implementation

D. Accumulators

The proposed fully pipelined design requires a dedicated accumulator to solve the RAW dependency. Products belonging to the same row are accumulated. However, the operation delay of double-precision floating-point addition is more than one cycle which means that the partial result is not ready when the new input product arrives in the next cycle. Some RTL-based designs have been proposed [22] [23] [24] and these work focus on the circuit design within the adder. However, for HLS-based designs, the adder cannot be optimized since the addition is a meta operator in high-level languages like C/C++. In this work, we propose an HLS-friendly accumulator.

We show the pseudo-code of the proposed accumulator in Fig. 6. Since the final output is the sum of all the inputs, we can first get some partial results and then sum these partial results up. Note that the delay of *double* addition is 5 cycles in our implementation. If an input arrives in Cycle 1, the partial result can be read in Cycle 7. During Cycle 2 to Cycle 6, if a new input arrives, another partial result is required. Thus, we allocate a buffer that includes six elements in an accumulator. With this buffer, we implement a fully pipelined accumulator. For example, if three inputs stream into the accumulator from Cycle 1 to Cycle 3, they are added with buffer[0], buffer[1], and buffer[2] separately. When all the products from PEs are consumed, we sum the six elements up to get the final result.

Each row requires one accumulator. However, the fully pipelined accumulator consumes a lot of hardware resources. Constrained by limited resources, we cannot deploy an accumulator for each row. According to the number of accumulators, the matrix is split into multiple sub-matrices along the row and we call each sub-matrix a row batch. By default, we deploy 32 accumulators. Due to the RAW dependence, only one input can be consumed per cycle and a FIFO is required to hold the other possible inputs. By default, we set the depth of FIFOs 64. These two parameters can be adjusted if this design is deployed on other platforms. Besides, a ping-pong buffer is used to hide the computation latency of summing the six elements up.

V. Reordering Algorithm

A. Problem Formulation

Since the design is fully pipelined and the main bottleneck is the throughput mismatch, the execution time is approximately equal to the number of cycles spent on fetching the target vector values in all the groups. Assume one group contains N_g non-zero matrix elements and there are N_G groups. The total execution time of the proposed design can be formulated as Eqn. 2.

$$T_{Total} = T_C + T_{RAW} + T_{PIPE} \tag{2}$$

where T_C refers to the number of cycles of fetching vector values, T_{RAW} refers to the number of cycles of processing the remaining products in accumulators when all the vector values are fetched (i.e., the number of cycles to deal with the RAW dependence), and T_{PIPE} refers to the depth of the whole pipeline which includes data transfer among sub-modules and the latency of each sub-module.

$$T_C = \sum_{i=0}^{N_G} \lceil len(Set(\{c | c \in CI_i, c \notin CI_{i-1}\}))/N_{Ports} \rceil \tag{3}$$

where CI_i refers to the column indices in $group_i$, $len(Set())$ returns the number of unique column indices in $group_i$ except for reused ones and N_{Ports} refers to the number of available memory ports of the vector buffer.

$$T_{RAW} = max(\{FIFODepth_i | i \in N_A\}) \tag{4}$$

where $FIFODepth_i$ refers to the remaining products in $Accumulator_i$ when all the vector values are fetched and N_A refers to the number of accumulators.

For example, for the matrix in COO shown in Fig. 3a, $T_C = \lceil 1 \rceil + \lceil 0.5 \rceil + \lceil 1.5 \rceil = 4$ and $T_{RAW} = 2$. For the matrix reordered following our proposed algorithm shown in Fig. 3b, $T_C = 1 + 1 + 1 = 3$ and $T_{RAW} = 1$. Although this

problem can be formulated, the $len(Set())$ function cannot be converted into mathematical equations. Thus, this problem cannot be solved by mathematical solvers. Meanwhile, the brute-force method is not applicable to large-scale problems. For instance, there are over 10^8 combinations if we partition 100 elements into 25 groups. Thus, we propose a heuristic algorithm to solve this problem.

B. Reordering Algorithm

In order to fully utilize the off-chip memory bandwidth while considering the throughput of the vector buffer, the biggest obstacle is columns that only have one element. For example, the corresponding columns of $x[3]$, $x[4]$, and $x[5]$ shown in Fig. 3a have only one element. Non-zero matrix elements in these columns must occupy one memory port and the fetched data cannot be reused. Based on motivational examples, we define three rules as follows.

- Rule 1 (Intra-group reuse): The available on-chip memory throughput should be fully utilized and the fetched data should be reused as much as possible in each group.
- Rule 2 (Inter-group reuse): The fetched vector values can be reused in different groups which can further increase the reusability of the fetched data.
- Rule 3: Non-zero matrix elements in longer rows should be scheduled as early as possible.

TABLE II: Assignment of *Capacity*

Element number in a column (N)	$N\%3 \neq 1$	$N\%3 = 1$		
		1	4	Others
Capacity	$N/3$	-1	0	$N/3$-1

As shown in motivational examples in Sec. III, Rule 2 is not necessary and thus we should determine whether Rule 2 is required. Given the fact that the on-chip vector buffer only has two memory ports and hence we should select non-zero matrix elements from at most two columns to form a group. We call one non-zero matrix element, two non-zero matrix elements, and three non-zero matrix elements in one column *single element*, *2-element block*, and *3-element block* separately. In order to fully utilize the off-chip memory bandwidth, we should form *single element*s with *3-element block*s. Thus, if all the *single element*s are paired with *3-element block*s, Rule 2 is not required. Otherwise, Rule 2 is required to consume extra *single element*s. In other words, a column with more than three non-zero matrix elements has a bigger capacity to hold more *single element*s following Rule 2. We define a metric called *capacity* to help determine whether Rule 2 is required. The *capacity* can be obtained following Table II. A special case is columns which have four elements. It can be split into two *2-element block*s or a *single element* and a *3-element block*. The total capacities in both cases are 0. Overall, if the sum of all the capacities is greater or equal to 0 which means that all the *single element*s can be consumed, Rule 2 is not required.

As shown in Algorithm 1, we first profile the original sparse matrix to get statistics that include lengths of rows, lengths of columns, and the total number of non-zero matrix elements (line 1). Lengths of rows are used to guide the algorithm to follow Rule 3 and the $Capacity$ of each column is calculated according to its length (line 3). For each row batch, as we have proved, if *totalCapacity* is no less than 0, Rule 2 is not required. Otherwise, we try to find reusable columns and *single element*s (line 5). If we find any reusable column, two elements of the column are scheduled at first to match the pre-defined pattern mentioned in Section IV-B (line 9). Then, each remaining element in the columns is paired with *single element*s to form a group and the group is appended to M_r (line 11-12). When reusable columns are exhausted or *totalCapacity* is no less than 0, we begin to reorder the remaining non-zero matrix elements following Rule 1 and Rule 3 (line 16-20). Following Rule 3, we first pop an element block from the longest row (line 17). Then, we use a greedy algorithm to find its counterpart to efficiently utilize the memory bandwidth (line 18). For example, if the popped element block is *single element*, its counterpart could be *3-element block*, *2-element block*, or another *single element* in order of priority. In addition, once a group is formed, all the statistics (e.g., lengths of rows and the total number of elements) are updated (line 12, line 19).

Algorithm 1 Reordering Algorithm

Input:
Non-zero matrix elements in COO, M_{coo};
Output:
The reordered non-zero matrix elements, M_r;

```
1:  Profile the original M_coo of all the row batches;
2:  for each row batch do
3:    Calculate Capacity of each column and totalCapacity;
4:    while totalCapacity < 0 do
5:      Find reusable columns and single elements;
6:      if reusable columns are not found then
7:        break;
8:      end if
9:      Form the first group using reusable columns;
10:     for each remaining element e in the columns do
11:       Pair single elements with e into a group;
12:       Append the group to M_r and update statistics;
13:       Update reusable columns if necessary;
14:     end for
15:   end while
16:   while the number of matrix elements > 0 do
17:     Pop one element block from the longest row;
18:     Found counterpart of the block;
19:     Append the group to M_r and update statistics;
20:   end while
21: end for
```

C. Discussion

1) Complexity of the algorithm: We first define N, N_{nc}, and N_g to indicate the number of non-zero matrix elements, the number of non-empty columns, and the number of groups separately. The profiling (line 1) iterates all the non-zero matrix elements and the complexity is $O(N)$. After profiling, the statistics are stored in dictionaries, e.g., *row_length*, *elements_in_rows*. For each row batch, finding reusable columns

and *single element*s (line 5) iterates the lengths of columns in the row batch. For the row batch, the complexity is $O(N_{nc})$. In line 17, popping one element block from the longest row finds the longest row in the row batch which includes 32 rows. Note that we have profiled lengths of rows and thus it just finds the maximum value of 32 values. Meanwhile, finding the counterpart (line 18) performs similar operations. Thus, the complexity of these two parts is $O(N_g)$. For all the row batches, the complexity of the algorithm is $O(N) + O(N_g)$.

2) Direct array partition: Although array partitioning can increase the throughput of the vector buffer, it fails to exploit the data reuse which can further compress sparse matrices mentioned in Section VI. Besides, since the access pattern is irregular, a scheduling algorithm is required to solve the throughput mismatch for each sub-bank.

VI. Data reuse-aware Compression

Although the proposed reordering algorithm solves the throughput mismatch, it fails to fully exploit the benefits of data reuse. In this section, we first introduce the proposed data reuse-aware compressed (DRC) format. Then, we show the format conversion algorithm and discuss its complexity.

A. DRC Format

With the exploration towards the data reuse of vector values, we can further compress the data to transfer more non-zero matrix elements via the limited off-chip memory bandwidth and thus improve the performance. Following the proposed reordering algorithm, fetched vector values are reused and the data reuse is known before the accelerator starts. Consequently, column indices of non-zero matrix elements whose corresponding vector values are reused can be compressed. Besides, constrained by the number of accumulators, 32 rows are processed per batch. Thus, the bitwidth of row indices can be reduced from 32 to 5.

	0	1	2	3	4	5
0	1	0	0	0	0	0
1	2	3	0	0	0	0
2	4	5	6	0	7	8
3	0	9	10	11	0	0
4	0	0	12	13	14	15
5	0	0	0	16	17	18

	N	M	Val (N+M)*64	Rid (N+M)*5	Cid M*32	Map (N+M)*2
Group 1	4	2	1,2,4,3,5,9	0,1,1,2,2,3	0,1	0,0,0,1,1,1
Group 2	4	2	6,10,12,11,13,16	2,3,3,4,4,5	2,3	0,0,0,1,1,1
Group 3	4	2	7,14,17,8,15,18	2,4,5,2,4,5	4,5	0,0,0,1,1,1

Fig. 7: DRC Format

As shown in Fig. 7, the proposed DRC format consists of groups. The length of a group equals the available off-chip memory bandwidth. Within each group, it mainly consists of the following data segments:

- N holds to the number of reused vector values
- M holds to the number of vector values to be fetched
- Val holds $N+M$ values of non-zero matrix elements in the group
- Rid holds $N + M$ row indices of non-zero matrix elements in the group
- Cid holds M memory addresses of vector values to be fetched
- Map holds $N + M$ mappings between non-zero matrix elements and vector values

Since we manually exploit the data reuse, we can further compress the data. For the reused vector values, we replace their 32-bit addresses (i.e., column indices of the corresponding non-zero matrix elements) with 2-bit mapping information. As shown in Fig. 7, we hold two 32-bit addresses and six 2-bit mapping information instead of six 32-bit addresses in each group. As a result, one group can hold up to 6 non-zero matrix elements and up to 2 vector values are fetched per cycle. Thus, N is 3-bit and M is 2-bit in this design. Each row index is a 5-bit data. Meanwhile, since we buffer two vector values, the mapping information has four possible cases: the first unbuffered vector value, the second unbuffered vector value, the first buffered vector value, and the second buffered vector value. Thus, each item in Map is 2-bit. For example, the *Map* in Group 1 is [0, 0, 0, 1, 1, 1] which means that the first three non-zero matrix elements are paired with the first unbuffered vector value and the last three non-zero matrix elements are paired with the second unbuffered vector value. As shown in Fig. 7, eighteen elements in DRC format can be reordered into three groups and three cycles are required to pair these elements with their corresponding vector values. Instead, the process takes five cycles without DRC. When the *decoder* receives a group, it first reads N and M. Then, it reads the following segments according to N and M.

Algorithm 2 Format Conversion

Input:
Non-zero matrix elements in each row batch, M_{ori};
Output:
The matrix in DRC format, M_{drc};
1: Profile M_{ori} to get the length of each column;
2: Sort columns according to their lengths;
3: buffered_value = false, buffered_cid = C_B;
4: longest_cid = C_L, shortest_cid = C_S;
5: **while** there are remaining matrix elements **do**
6: **if** not buffered_value **then**
7: **if** $N_{C_L} + N_{C_S}$>=6 **then**
8: Select as many elements as possible from C_S;
9: Form a group with elements from C_L;
10: C_B = C_L;
11: **else**
12: Form a group with elements from C_L and C_S;
13: **end if**
14: **else**
15: **if** $N_{C_B} + N_{C_S}$>=6 **then**
16: Select as many elements as possible from C_S;
17: Form a group with elements from C_B;
18: **else**
19: Form a group with elements from C_B and C_S;
20: Get the current shortest column C_S;
21: Fill the group with elements in C_S;
22: buffered_value = false;
23: **end if**
24: **end if**
25: **end while**

B. Format Conversion

In this subsection, we propose a fast format conversion algorithm. We observe that Rule 3 mentioned in Section V achieves little performance speedup while increasing the execution time of the preprocessing algorithm. The fast format conversion algorithm aims to consume columns with fewer non-zero matrix elements to solve the throughput mismatch. Note that a group holds up to 6 non-zero matrix elements. As shown in Algorithm. 2, we first profile and sort columns according to their lengths (line 1-2). If there is no buffered vector value, we form a group with non-zero matrix elements from the longest column and the shortest column (line 7-13). In order to consume short columns, we select as many elements as possible from the shortest column and buffer the corresponding vector value of the longest column if the longest column has remaining elements (line 10). If the buffered vector value is available, we try to form a group with elements from the buffered column and the shortest column (line 15-23). If the group has not been fulfilled, we use elements from the current shortest column to fill it (line 21-22).

For the complexity of this algorithm, we first traverse all the non-zero matrix elements in a row batch and the complexity is $O(N_{nnz})$ where N_{nnz} refers to the number of non-zero matrix elements in the row batch. Then, we sort non-empty columns and the complexity is $O(N_{nc}logN_{nc})$ where N_{nc} refers to the number of non-empty columns. Since we access the longest and shortest columns in the remaining execution, the complexity is $O(1)$. Overall, the complexity of the algorithm is $O(N_{nnz} + N_{ne}logN_{ne})$.

C. Discussion

1) Application to ASIC Designs: This work first exploits the data reuse to solve the throughput mismatch between the on-chip vector buffer and the off-chip memory bandwidth. Although the throughput mismatch can be solved by increasing the throughput of the on-chip vector buffer in ASIC designs, it fails to further compress the data. With the proposed DRC format, the matrix is further compressed compared to the CSR format as shown in Fig. 14 and more non-zero matrix elements can be transferred via the available off-chip memory bandwidth. Meanwhile, the data reuse still exists even if the throughput mismatch is solved. Consequently, the proposed idea can be applied to ASIC designs to speed up performance.

2) Limitations: Since we exploit the data reuse to solve the throughput mismatch and compress the matrix size, the performance of the proposed idea depends on the reusability of vector values. As shown in experimental results, sparser matrices have less performance improvement.

VII. Experimental Results

In this section, we first detail our experimental setup. Then, we show the experimental results and compare them with the related work.

A. Experimental Setup

The experimental results are obtained on the Xilinx Zynq UltraScale ZCU106 platform, which integrates a quad-core ARM Cortex-A53 application processor, a dual-core Cortex-R5 real-time processor, and an XCZU7EV-2FFVC1156 FPGA chip. We design the accelerator using C++ and use Vivado HLS v2018.3 to convert the C++ codes into an RTL design. Vivado Design Suite v2018.3 [25] is used to generate the final bitstream. The clock frequency in our design is 100 MHz and the available off-chip memory bandwidth is 6.4GB/s. Table III shows the resource consumption. The number of PEs (i.e., N_{PE}) equals the maximum number of non-zero matrix elements in each group. Thus, N_{PE} is 6 if DRC is deployed. Otherwise, N_{PE} is 4. Meanwhile, we allocate the vector into Ultra RAMs (URAM). Since the size of vectors is different in each benchmark, the URAM consumption is not shown in Table III.

TABLE III: Resource Consumption

	LUTs(%)	FFs(%)	DSPs(%)	BRAMs(%)
Without DRC	65.76	33.49	13.66	49.04
With DRC	76.26	38.72	14.93	49.04

We use the same benchmarks in [8] to ensure a fair comparison and show them in Table IV. The selected benchmarks can all be found on the SuiteSparse matrix collection [20].

TABLE IV: Selected Benchmarks

Benchmark	Cols/Rows	Nonzero	Density	GFLOPs	BU
raefsky1	3242	293409	2.79%	1.1796	0.18
consph	83334	6010480	0.09%	1.1820	0.18
cant	62451	4007383	0.10%	1.1825	0.18
pwtk	217918	11524432	0.02%	1.1796	0.18
rma10	46835	2329092	0.11%	1.1771	0.18
torso2	115967	1033473	0.01%	1.0050	0.16
t2d_q9	9801	87025	0.09%	1.0142	0.16
epb1	14734	95053	0.04%	0.8782	0.14
lns_3937	3937	25407	0.04%	0.6672	0.1
mac_econ	206500	1273389	0.003%	0.6621	0.1
dw8192	8192	41746	0.06%	0.5770	0.09

Since the off-chip memory bandwidth determines the upper bound of the overall performance, we use the bandwidth utilization [8] (BU) as the performance metric to make a fair comparison. BU is formulated by the throughput (GFLOPs) over the off-chip memory bandwidth (GB/s). The throughput is calculated by $2 * N_{nonzero}$ over the execution time T. We combine the two equations $T = C_{exec}/Freq$ and $GB/s = Bandwidth_{cycle} * Freq$ to get the final equation of BU as shown in Eqn. 5. Given a fully pipelined design, $\frac{N_{nonzero}}{C_{exec}}$ in Eqn. 5 approximately equals the number of non-zero matrix elements transferred to the FPGA chip. Consequently, the theoretical upper limit of BU is $2 * N_{epc}/Bandwidth_{cycle}$ where N_{epc} refers to the maximum number of non-zero matrix elements processed per cycle.

$$BU = \frac{GFLOPs}{GB/s} = \frac{N_{nonzero} * 2}{Bandwidth_{cycle} * C_{exec}} \quad (5)$$

where $N_{nonzero}$ refers to the number of nonzero matrix elements, $Bandwidth_{cycle}$ refers to the available memory bandwidth in bytes per cycle which is 64 on Xilinx ZCU106 and 60 on [8], and C_{exec} refers to the execution time in cycles.

B. Performance Analysis

Fig. 8 shows the performance comparison between the proposed design and the state-of-the-art work [8] (refer as *Base* in Fig. 8). We summarize the comparison of off-chip memory transactions in Table V. In the table, N and N_v refer to the number of non-zero matrix elements and the number of vector values. Since read transactions with DRC cannot be formulated with N and N_v, we compare the overall matrix size using DRC with that of CSR in Fig. 14.

Since at most four elements are processed per cycle without DRC, the theoretical peak BU without DRC is around 0.125 (2*4/64). As we can see in Fig. 8, some benchmarks (e.g., `raefsky1`) achieve almost the peak performance. In [8], three elements are processed per cycle and its peak performance is 0.1 (2*3/60). Compared to the baseline, we reduce the off-chip memory traffic (which is around $4N$-$8N_v$) by transferring column indices instead of vector values and achieve an average 1.18x performance speedup. Besides, at most six elements can be processed with DRC per cycle and thus the theoretical peak BU is around 0.1875 (2*6/64). As shown in Fig. 8, denser matrices can achieve higher BU with DRC. Overall, the performance speedup is around 1.57x compared to the baseline.

TABLE V: Comparison of off-chip memory transactions

	[8]	Wo DRC	With DRC
read (bytes)	$20N$	$16N + 8N_v$	-
write (bytes)	$8N_v$	$8N_v$	$8N_v$
read + write (bytes)	$20N + 8N_v$	$16N + 16N_v$	-
elements per cycle	3	at most 4	at most 6

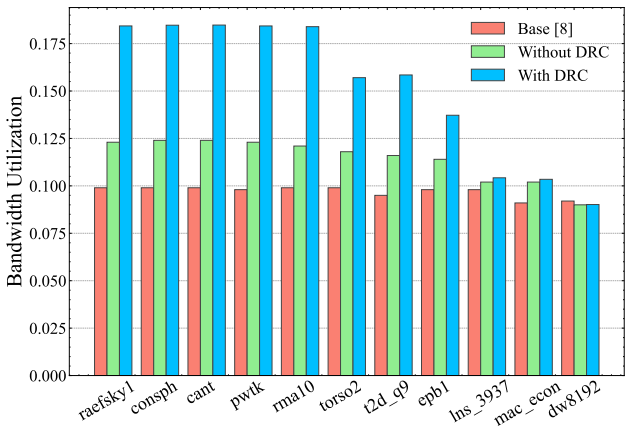


Fig. 8: Performance comparison on ZCU106

To help illustrate the performance speedup, we show the percentage of non-zero matrix elements processed at different speeds in Fig. 9. We can transfer up to 6 non-zero matrix elements with DRC format. The benefits of DRC are fully utilized in denser matrices. For example, almost all the non-zero matrix elements are processed at a rate of 6 per cycle in the left five benchmarks. As a result, the performance speedup is high. For sparser matrices like `dw8192`, the benefit of data reuse is not fully utilized and the performance improvement is little.

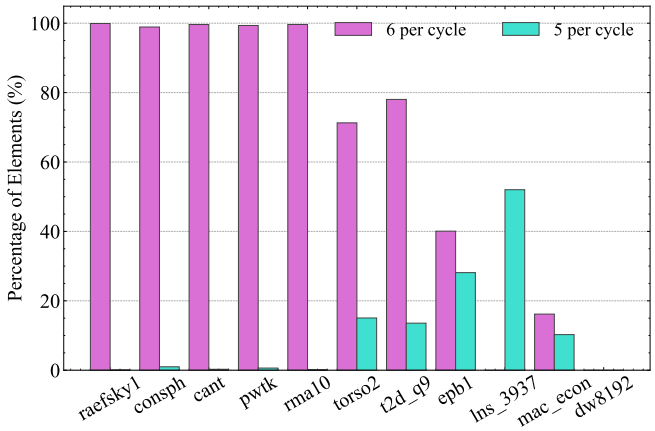


Fig. 9: Percentage of elements processed at higher speeds

Our performance is lower than the baseline on `dw8192`. We show parts of the elements in the 250th row batch of `dw8192` in Fig. 11. All the non-zero matrix elements belong to different columns. Limited by the on-chip memory throughput and no reusable columns exist, only two non-zero matrix elements can be processed. Besides, [8] achieves the performance with the help of a CPU counterpart. For standalone FPGAs, it is quite hard to optimize these kernels without increasing the throughput of on-chip memories.

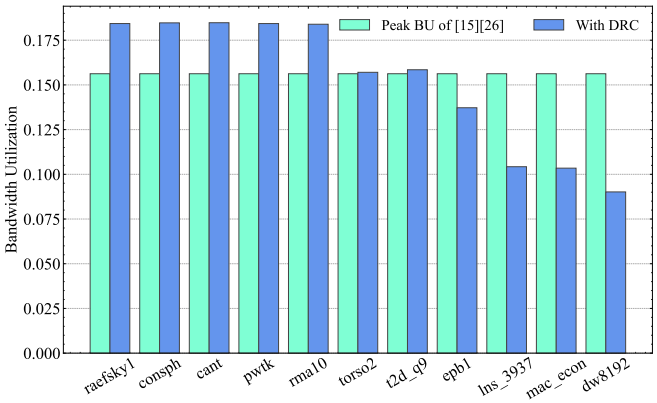


Fig. 10: Comparison with peak BU of [15] [26]

We also compare the performance with HiSparse [15] and Serpens [26] that target HBM-based FPGAs. Both two work hold multiple copies of the vector to solve the throughput mismatch and fail to exploit the benefits of reusing fetched vector values. It is hard to deploy their designs since they do not support data in *double*. For example, HiSparse [15] proposes row interleaving to support data in *float* and the throughput degrades. Consequently, we compare with their peak BU. According to their data formats, up to 5 non-zero matrix elements with data in *double* can be transferred to the FPGA chip via a 512-bit off-chip memory bandwidth. As shown in Fig. 10, we achieve an average 1.18x performance speedup for denser matrices. Since the platform is the same, the comparison of BU is equivalent to that of the performance. Sparser matrices do not fully utilize the benefits of the DRC format due to native sparsity. In the future, we can extend to consider array partition and replication to boost the performance of sparser ones.

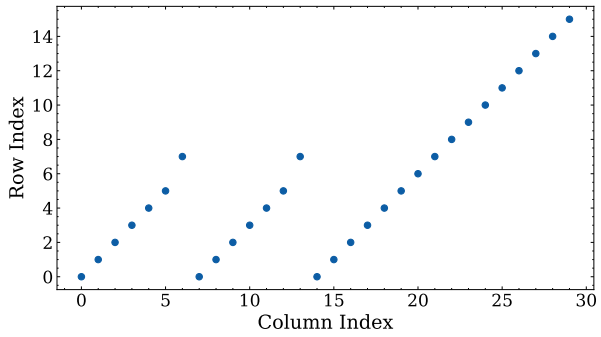


Fig. 11: Distribution of dw8192's partial nonzero elements

We further show the percentage of reused vector values to help understand the performance speedup. As shown in Fig. 12, around 48% of vector values can be reused by reordering on average. Further, around 65% of vector values can be reused with DRC as shown in Fig. 13. Since the proposed format conversion algorithm always follows Rule 2, more vector values are reused following inter-group reuse for denser matrices. Meanwhile, since more non-zero matrix elements can be transferred per cycle with DRC, the number of reused vector values following intra-group reuse increases and the number of reused vector values following inter-group reuse decreases for sparser matrices (e.g., lns_3937).

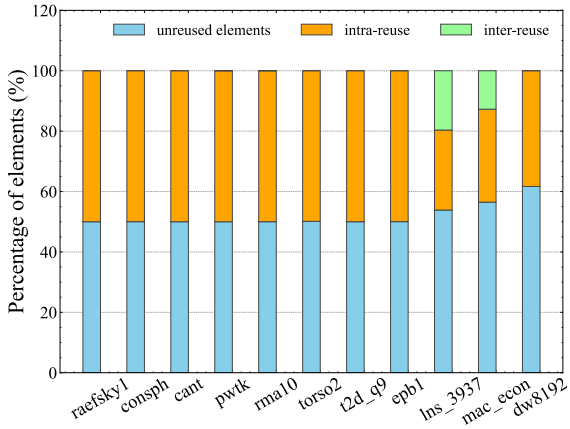


Fig. 12: The percentage of the reused elements w/o DRC

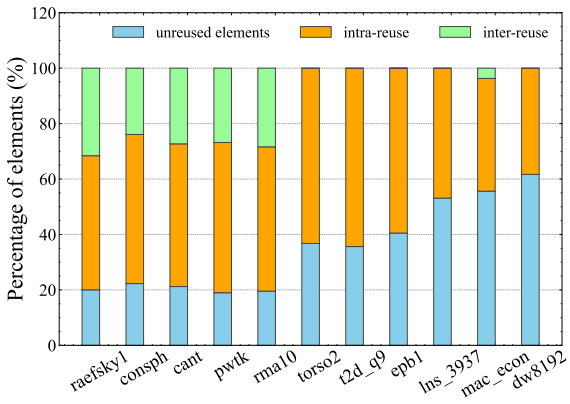


Fig. 13: The percentage of the reused elements with DRC

C. Size of DRC

Without DRC, the matrix size after reordering is the same as the COO format. With DRC, we compress addresses of reused vector values (i.e., 32-bit column indices of non-zero matrix elements) to the 2-bit mapping information. Meanwhile, we reduce the bitwidth of row indices from 32 to 5. As shown in Fig. 14, DRC reduces an average 15.64% matrix size compared to CSR. Concretely, around 65% fetched vector values are reused as shown in Fig. 13. Assume there are $N_{nonzero}$ non-zero matrix elements. For the COO format. the matrix size is around $128*N_{nonzero}$ ($64*N_{nonzero}+32*N_{nonzero}+32*N_{nonzero}$). The matrix size in CSR format is around $96 * N_{nonzero}$ ($64 * N_{nonzero} + 32 * N_{nonzero}$) + $32 * N_r$ where N_r refers the number of rows. Instead, the matrix size in DRC format is around $83 * N_{nonzero}$ ($64 * N_{nonzero} + 5 * N_{nonzero} + 2 * N_{nonzero} + 0.35 * 32 * N_{nonzero} + 5/6 * N_{nonzero}$).

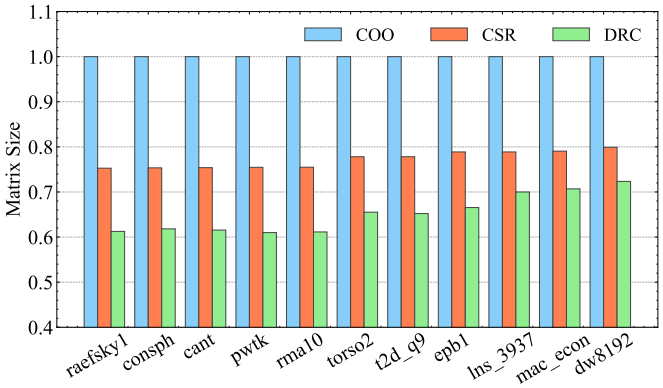


Fig. 14: Matrix size after compression

D. Preprocessing Time

We list the preprocessing time of Algorithm 1 and Algorithm 2 in Table VI. Without considering Rule 3 in Section V which incurs multiple sorts, the execution time significantly reduces. Further, the algorithm can be optimized using multiple threads. Although the complexity of Algorithm 1 is $O(N)$, lots of sorts in each row batch are required to find non-zero matrix elements in the longest row.

TABLE VI: Preprocessing time of each benchmark

Benchmark	cant	t2d_q9	torso2	pwtk	raefsky1	lns.
Algo. 1(s)	167	2	16	512	20	1
Algo. 2(s)	5	<1	1	10	<1	<1
Benchmark	rma10	epb1	consph	dw8192	mac.	
Algo. 1(s)	107	1	330	<1	15	
Algo. 2(s)	3	<1	9	<1	2	

E. Summary of FPGA-based SpMV Accelerator

In summary, we show a holistic comparison between this work and related literature in terms of system specification and performance in Table VII. The main challenge of SpMV is the throughput mismatch of the off-chip memory and on-chip memories. To solve the throughput mismatch, [9], [15], [26], and [27] hold multiple copies on their platforms to increase the throughput of on-chip memories. Meanwhile, [8] deploys the

CPU host to fetch vector values from caches and then transfers them with the corresponding matrix elements to the FPGA chip. However, they fail to explore data reuse of fetched vector values. In this work, we explore the data reuse and propose to compress addresses of reused vector values (i.e., column indices of the corresponding matrix elements). Specifically, a 32-bit column index is replaced by one 2-bit mapping information. As shown in Fig. 14, the proposed DRC format reduces an average 15.64% matrix size compared to CSR. Instead, [26] simply compresses the row index and column index into one 32-bit data and the reduction is little. Meanwhile, thanks to the DRC format, we can transfer up to 6 elements per cycle via a 512-bit memory bandwidth. As a result, we can achieve the highest 0.152 BU (i.e. GFLOPs/GB/s) which is 1.61x $\sim$ 2.53x compared to others. The absolute performance (i.e. GFLOPs) is determined by the available memory bandwidth. Although our available memory bandwidth is lower than [8], we can achieve up to 1.18 GFLOPs as shown in Table IV which is comparable to [8]. Besides, although [8] does not need preprocessing, the solution requires a CPU counterpart that matches the speed of the FPGA accelerator and can support an efficient data transfer between them. In the future, we plan to hold multiple copies of the vector to process sparse matrices with low data reuse, e.g., `dw8192`.

TABLE VII: Summary of FPGA-based SpMV Accelerator

	[9]	[8]	[27]	[26]	[15]	Ours
Platform	multi-F[1]	C[!]F[1]	F[1]	F[1]	F[1]	F[1]
Main Memory	DDR	DDR	DDR	HBM	HBM	DDR
Bandwidth (GB/s)	19.2	12	6.4	273	268	6.4
Preprocessing?	Yes	No	Yes	Yes	Yes	Yes
Size vs CSR	=	>	>	<	>	<
#Vector Copy	Multi	No	Multi	Multi	Multi	**One**
Perf. (GFLOPs)	1.16	1.13	0.76	18.15	16.7	0.97
#Elements[2]	5	3	4	5	5	**6**
BU	0.06	0.094	0.118	0.066	0.062	**0.152**

[1] C refers to CPU and F refers to FPGA.
[2] The maximum number of elements transferred via a 512-bit memory bandwidth.

VIII. Conclusion

In this work, we observe that repeated memory accesses of vector values can be omitted by reusing the fetched data and propose a data reordering algorithm to exploit the reuse of vector value. Further, we propose a novel data reuse-aware compressed format (DRC) to take full advantage of the data reuse. In addition, a fast format conversion algorithm is proposed which shortens the preprocessing time. Finally, we design a customized hardware accelerator and the experimental results show that our design achieves an average 1.18x performance speedup without DRC and 1.57x performance speedup with DRC w.r.t. the state-of-the-art work respectively.

Acknowledgement

This work is partially supported by the Ministry of Education, Singapore, under its Academic Research Fund Tier 2 (MOE2019-T2-1-071), and Nanyang Technological University, Singapore, under its NAP (M4082282/04INS000515C130).

References

[1] P. Sonneveld *et al.*, "Idr (s): A family of simple and fast algorithms for solving large nonsymmetric systems of linear equations," *SIAM Journal on Scientific Computing*, 2009.

[2] K. Akbudak, E. Kayaaslan, and C. Aykanat, "Hypergraph partitioning based models and methods for exploiting cache locality in sparse matrix-vector multiplication," *SIAM Journal on Scientific Computing*, vol. 35, no. 3, pp. C237–C262, 2013.

[3] H. Shuo, Z. Lei, L. Di, L. Weichen, and R. Subramaniam, "Zerobn: Learning compact neural networks for latency-critical edge systems," *2021 58th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2021.

[4] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," *arXiv preprint arXiv:1608.08710*, 2016.

[5] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[6] S. Cao *et al.*, "Efficient and effective sparse lstm on fpga with bank-balanced sparsity," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2019.

[7] M. Veen, "Sparse matrix vector multiplication on a field programmable gate array," Master's thesis, University of Twente, 2007.

[8] K. Lu *et al.*, "Redesk: A reconfigurable dataflow engine for sparse kernels on heterogeneous platforms," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2019.

[9] S. Li *et al.*, "A data locality-aware design framework for reconfigurable sparse matrix-vector multiplication kernel," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2016.

[10] P. Grigoras *et al.*, "Cask: Open-source custom architectures for sparse kernels," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2016.

[11] G. Dai, T. Huang, Y. Chi, N. Xu, Y. Wang, and H. Yang, "Foregraph: Exploring large-scale graph processing on multi-fpga architecture," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 217–226.

[12] S. Zhou, R. Kannan, V. K. Prasanna, G. Seetharaman, and Q. Wu, "Hitgraph: High-throughput graph processing framework on fpga," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 10, pp. 2249–2264, 2019.

[13] X. Chen, H. Tan, Y. Chen, B. He, W.-F. Wong, and D. Chen, "Thundergp: Hls-based graph processing framework on fpgas," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021, pp. 69–80.

[14] Y. Hu, Y. Du, E. Ustun, and Z. Zhang, "Graphlily: Accelerating graph linear algebra on hbm-equipped fpgas," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 2021, pp. 1–9.

[15] Y. Du, Y. Hu, Z. Zhou, and Z. Zhang, "High-performance sparse linear algebra on hbm-equipped fpgas using hls: A case study on spmv," in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2022, pp. 54–64.

[16] C. Meng, S. Yin, P. Ouyang, L. Liu, and S. Wei, "Efficient memory partitioning for parallel data access in multidimensional arrays," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2015, pp. 1–6.

[17] J. Su, F. Yang, X. Zeng, and D. Zhou, "Efficient memory partitioning for parallel data access via data reuse," in *Proceedings of the 2016 ACM/SIGDA international symposium on field-programmable gate arrays*, 2016, pp. 138–147.

[18] Y. Wang, P. Li, P. Zhang, C. Zhang, and J. Cong, "Memory partitioning for multidimensional arrays in high-level synthesis," in *Proceedings of the 50th Annual Design Automation Conference*, 2013, pp. 1–8.

[19] S. Li *et al.*, "Optimized data reuse via reordering for sparse matrix-vector multiplication on fpgas," in *2021 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2021.

[20] T. A. Davis *et al.*, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, 2011.

[21] R. Dorrance *et al.*, "A scalable sparse matrix-vector multiplication kernel for energy-efficient sparse-blas on fpgas," in *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, 2014.

[22] L. Zhuo, G. R. Morris, and V. K. Prasanna, "High-performance reduction circuits using deeply pipelined operators on fpgas," *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, no. 10, pp. 1377–1392, 2007.

[23] M. Huang and D. Andrews, "Modular design of fully pipelined reduction circuits on fpgas," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 9, pp. 1818–1826, 2012.
[24] S. Sun and J. Zambreno, "A floating-point accumulator for fpga-based high performance computing applications," in *2009 International Conference on Field-Programmable Technology*. IEEE, 2009, pp. 493–499.
[25] Xilinx. Vivado design suite. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html
[26] L. Song, Y. Chi, L. Guo, and J. Cong, "Serpens: A high bandwidth memory based accelerator for general-purpose sparse matrix-vector multiplication," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, pp. 211–216.
[27] B. Liu *et al.*, "Towards high-bandwidth-utilization spmv on fpgas via partial vector duplication," in *Proceedings of the 28th Asia and South Pacific Design Automation Conference*, 2023.

Shiqing Li received the B.S. and M.S. degrees in computer science and technology from Shandong University, Jinan, China, in 2016 and 2019, respectively. He is currently pursuing a Ph.D. degree with the School of Computer Science and Engineering at Nanyang Technological University, Singapore. His current research interests include sparse matrix operations, FPGAs, and high-level synthesis.

Di Liu is an Associate Professor at Department of Computer Science, Norwegian University of Science and Technology (NTNU), Norway. In prior to that, he was a research fellow at Nanyang Technological University, Singapore and a faculty member at Yunnan University, China. He received his PhD degree from Leiden University, The Netherlands, and both his MEng and BEng degrees from Northwestern Polytechnical University, China.

Weichen Liu received his BEng and MEng degrees from Harbin Institute of Technology, China, and PhD degree from the Hong Kong University of Science and Technology, Hong Kong SAR. He is currently an Associate Professor with the School of Computer Science and Engineering, Nanyang Technological University, Singapore. He authored and co-authored more than 140 research papers in peer-reviewed journals, conferences, and books. His research interests include embedded and real-time systems, multiprocessor systems, network-on-chip, and machine learning acceleration.