PrGEMM: A Parallel Reduction SpGEMM Accelerator

ABSTRACT

Due to increasing data sparsity in scientific data sets and pruned neural networks, it becomes more challenging to compute with these kinds of sparse data sets efficiently. Several works discuss efficient sparse matrix-vector multiplication (SpMV). However, because of index irregularity in compact stored matrices, sparse matrix-vector multiplication (SpGEMM) still suffers from the trade-off between space and efficiency of computation.

In this work, we propose PrGEMM, a multiple reduction scheme which (1) computes SpGEMM under compact storage format without expansion of the operands, (2) by using index lookahead, computes and compares multiple index-data pairs at the same time with no order violation of indices. We evaluate our work with the matrices with different sizes in the SuiteSparse data set. Our work can achieve 3.3x of execution cycle improvement compared to the state-of-the-art SpGEMM scheme.

CCS CONCEPTS

• **Hardware → Hardware accelerators**.

KEYWORDS

Accelerator, SpGEMM, Hardware

ACM Reference Format:

. 2022. PrGEMM: A Parallel Reduction SpGEMM Accelerator. In *Proceedings of June 6–8, 2022 (GLSVLSI 2022).* ACM, New York, NY, USA, 6 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Sparse data structures have emerged in many different applications, and computation using these structures is also widely discussed. These applications include the computation and analysis of scientific and graph data ([4],[2],[18]), which contains plenty of data nodes and sparse links between nodes; recommendation systems [10] and natural language processing (NLP), which require large product or word embeddings while the operands are relatively small, like purchase history or a sentence; the pruned weights and activations in sparse deep neural networks ([5], [13]. In this work, we focus on the acceleration of sparse matrix-matrix multiplication (SpGEMM), which is one of the crucial components in the above applications.

To reduce the storage cost of sparse matrices, most sparse matrices are stored in some kind of compressed format, which only stores the non-zero values (nnz) and the address information according to that nnz, resulting in compact storage of sparse matrices. However, the indices stored in memory are irregular, and this irregularity of the sparse matrices makes efficient computation of SpGEMM challenging. First, the memory access pattern related to the indirect access of the compact stored matrix is usually irregular. Thus, when trying to compute the SpGEMM in parallel, the memory requests between different computation units could have little spatial locality, which increases memory traffic. Also, the cache miss rate induced by the irregular access pattern is significant. Second, the index irregularity inside the vector makes the index matching process of two compact stored vectors difficult to parallelize or requires more memory to expand the operands. Besides index irregularity, the vector length irregularity is also an issue in SpGEMM. One of the issues is the load balancing induced by the different vector lengths in each computation unit, which results in each unit's different finishing time. Furthermore, the result of SpGEMM needs to be stored in compressed format; without prior information of the result matrix shape, the irregular vector length of the result matrix prevents the out-of-order storage of the result matrix. To accelerate the SpGEMM operation, we focus on parallelizing the matching process in the computation of two vectors to increase computation efficiency.

In this work, we use the following methods to achieve parallel index matching in the SpGEMM operation and increase computation efficiency:

- We use Gustavson's algorithm-based [9] computation model to compute the SpGEMM, where both vectors are stored in CSR format. Each computation unit computes a row of the result matrix by computing the scalar to vector multiplication, buffering the results as the partial row, and reducing the vector into the complete row of the result matrix. In each reduction operation, 2 partial vectors are merged into a new vector.
- In the reduction network, we compute the index comparison and value reduction in parallel with width 4. We construct a reduction network to achieve this computation. To avoid the out-of-order values and indices appearing in the result matrix, we applied an additional "lookahead index" to block out the invalid output and keep the comparison results correct.
- An accelerator is constructed to compute the SpGEMM. Then, we insert a scheduler between the accelerator and the memory to schedule each computation unit's load/store requests and pack the result matrix into CSR format.

Section II discusses the background of the sparse matrix storage format and the computation method of SpGEMM. Section III explains the challenge of intra-row level parallelization of SpGEMM computation. Section IV discusses the design of PrGEMM, including the parallel reduction network and the peripherals in the accelerator. Section V discusses the experiment setting and results, and Section VI explains the related works. Finally, section VII is the conclusion.

GLSVLSI 2022, Orange County, CA, USA,

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

Table 1: Summary of the Previous works

Work	Algorithm	Storage format	Transpose	Reduction method	Storage
Matraptor[17]	Gustavson's Algorithm	C^2SR	N/A	Serial Reduction	C^2SR
OuterSPACE[12]	Outer Product	CSR	Matrix A	Sorting first entries	CSR/CSC
SIGMA[15]	Inner Product + bitmap	Run-length compression	N/A	Tree topology	N/A
Gamma[19]	Gustavson's Algorithm	CSR	N/A	Tree reduction	N/A
TensorDash[11]	Inner Product + load balance	Compact	N/A	Inner Product	N/A
PrGEMM	Gustavson's Algorithm	CSR	N/A	4-wide parallel	CSR

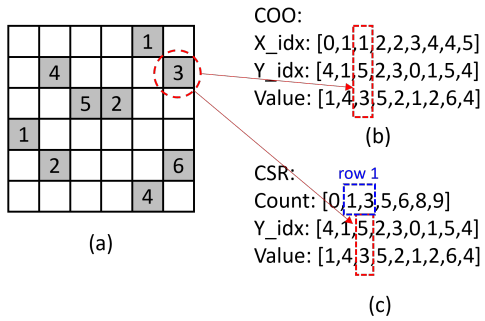

Figure 1: A matrix in the (a) dense matrix (b) Coordinate list (COO) (c) Compress Sparse Row (CSR) representation

2 BACKGROUND

2.1 Sparse Matrix Representation

When storing sparse matrices, the critical point is to hold only the nnzs of the matrix to eliminate the storage cost of 0-storage and encode the address information of the corresponding nnzs. The most straightforward way to encode the address information is by storing the cartesian address of the nnz, called coordinate list (COO) (Fig.1(b)). COO uses 3 vectors to encode the x-address, y-address, and the value; the information in the same index of these three vectors corresponds to the same nnz in the matrix. Compressed Sparse Row/Column (CSR/CSC) is a method to further compress the matrix by only storing the accumulation count of nnzs in the first vector, and the number in the first vector is the pointer to the first element of each row/column. Fig.1(c) shows an example of CSR format, here 1 and 3 in the count vector indicate the accumulated element count of row 0 and row 1, which is equivalent to the start element of row 1 and row 2, respectively. If 2 adjacent numbers of the count vector are the same, there is no element in the corresponding row. The y-indices and value of the elements on a non-empty row are stored in the Y_idx and value vector, respectively. For a sparse matrix, the storage cost of COO representation is 3*nnz, and CSR/CSC is number of rows + 2*nnz. This comparison shows that the CSR and CSC gain more storage efficiency than COO when the number of nnz is greater than the matrix dimension. Moreover, CSR has more benefits from a computation perspective because it is more difficult to find a specific element in the COO than CSR. In COO, randomly accessing an element in the matrix requires the traversal of the matrix; however, in CSR representation, this kind of accessing only needs to traverse the specific row, which increases the flexibility of the computation. In this work, considering the decoding cost of the matrix, the SpGEMM algorithm, and the consistency of the input and output matrices, we choose CSR to be the representation in this work.

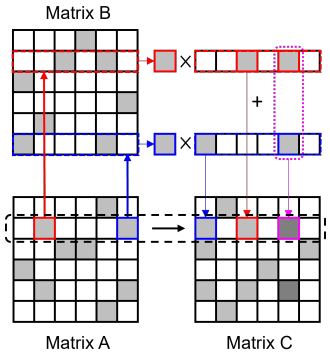

Figure 2: Matrix multiplication using Gustavson's Algorithm

2.2 Sparse matrix multiplication

Sparse matrix-matrix multiplication (SpGEMM) computes $C = A \times B$, which A and B are both sparse matrices. Based on the computation method, there are two different types of operation: The inner product method and the outer product method. In this work, we use an alternative outer product method, called Gustavson's algorithm [9], to only compute on nnzs without needing to transpose either input matrix.

Gustavson [9] proposed a row-based algorithm to compute the SpGEMM. This method is a rearrangement of the outer product method, and has several variation [1] to meet the requirements of different applications. In this algorithm (Fig.2(c)), to compute a row i of output matrix C, we traverse the row i of A, each nnz in row A[i] is multiplied with the corresponding rows of B and generates partial product vectors. Finally, these vectors are merged to the result matrix row C[i]. The equation can be written as:

$$C[i,:] = \sum_{k=0}^{n} A[i,k] * B[k,:] \quad (1)$$

There are 3 main benefits compared to the original outer product method: (1) Because both matrices are accessed in row orientation, both matrices can directly compute under CSR format without transpose. (2) the partial result is a vector, which reduces the storage cost of the partial result. (3) We know a single row of result matrix C is computed from the row with the same row index of A, which can be highly flexible with the applications that might only need a row of the result matrix.

However, similar to the outer product-based method, with the multiplication only happening on the nnzs and the vectors of B stored in compact form, the partial result vectors are also stored in

compact form. The merge of these partial result vectors is called sparse vector (SV) reduction. Because of the index irregularity of compact stored vectors, the index matching problem is a challenge discussed in the next section.

3 CHALLENGE OF THE SV REDUCTION

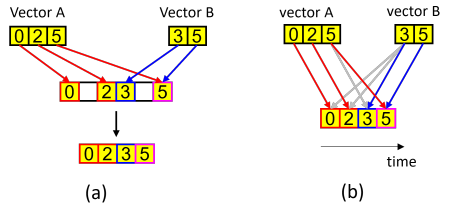

Figure 3: Sparse vector reduction by (a)expending the computation space (b) index matching over time

Due to the index irregularity in the compact stored sparse vector, it is challenging to compute sparse vector reduction on compactly stored vectors efficiently. Currently, there are two approaches for sparse vector reduction. The first is preparing a space according to the size of the vector as the computation space[14] (Fig.3(a)). While operating, the values are passed to the location matching to their indices in the computation space. If multiple values are passed to the same location, they coalesce into a single output value. After the operation, the vector is re-compressed back to the compact format. This method is highly efficient because each value can be correctly applied to the corresponding index without any index comparison or index matching operation. Also, a single computation space can handle all the partial vectors, so there is no need to store each partial vector separately. However, if the matrix or vector is sparse in the output, the space overhead on creating the computation space is still quite significant. To re-compress the vector, it must traverse the computation space to collect all the nnzs, which adds to computation overhead.

Another method is merging both vectors under the compact format with index matching, which has been widely used in many previous works [17]. Fig.3(b) shows an example of the process of index matching. Because CSR stores all the vectors in ascending order, the merged vector's ordering is maintained by continuously comparing the index of the first value of both vectors. In this method, the value with a smaller index is popped out first; if there is a matching index between 2 vectors, the values of these 2 vectors coalesce and both popped out. This method can use the least computation space to achieve the merger of the sparse vectors; however, because of the irregularity of the index, this method is computed in serial method, which results in low computation efficiency.

Parallel SV reduction in compact format?

Multiple index matching has an opportunity to generate more results in a single operation, which can improve the efficiency of sparse vector reduction. However, the main issue that prevents multiple sparse vector reduction is the irregularity of the vector indices. This issue makes the computation more complex within the selected granularity for computation. Furthermore, due to the correctness issue, the indices outside the selected granularity also need to be considered. For example, Fig.4 shows a parallel reduction with granularity = 4. Even though the result between 2 pieces of the vector is correct, it violates the ascending order of the result vector because the last index is greater than an index outside the selected granularity, which has to be reduced in the subsequent reduction step.

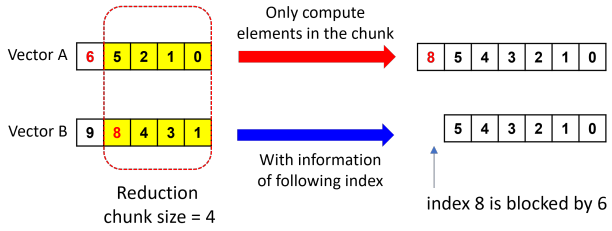

Figure 4: Parallel reduction of 2 SVs with granularity = 4

To solve this issue, we must look ahead at the following indices of each vector to act as the mask index to block out those indices that violate the ascending order of the result vector. With the property of ascending order of the compact sparse vector, it only needs to look ahead to 1 more index to guarantee the correctness of the result vector. Compared to the serial method of sparse vector reduction, which compares 1 index from each vector and generates 1 result, the parallel reduction with granularity = 4 can produce up to 8 results if there is no matching index in both vectors. At a minimum, unless the case that only one side has the input vector (unbalance vector length), or both vectors cannot fill in the input of the reduction engine (usually happen at the tail of the vector), the reduction can produce at least 4 results if both input vectors are full.

Based on these observations, we design an accelerator for computing SpGEMM with a lookahead parallel reduction network to achieve high-efficiency SpGEMM computation.

4 SPGEMM ACCELERATOR DESIGN

In this work, we use the algorithm proposed by Gustavson [9] to compute SpGEMM to utilize better the storage efficiency obtained from the CSR format. To deal with the inefficiency within the index matching problem, we propose a parallel reduction network to merge two in-ordered but irregular indexed vectors in a coarse-grain operation (size = 4). A lookahead index in both vectors is invoked to block out the invalid result generated from the reduction network and output the correct result vector.

4.1 Parallel Reduction network

The parallel reduction network (Fig.5(b)) contains 4 components: The reduction signal generator, reduction adder array, output multiplexers, and the output mask. The reduction signal generator has a comparator array and a set of reduction logic to generate 3 different control signals to control the other 3 blocks in the reduction network. The reduction adder array contains a reduction shifter to redirect the input values from both vectors to the corresponding place, depending on the index matching or not. The output multiplexer is a mux array that sorts the result vector based on the comparison result generated from the reduction signal generator. Finally, the output mask is responsible for blocking out the output values with the indices greater than one of the lookahead indices to generate the output vector correctly.

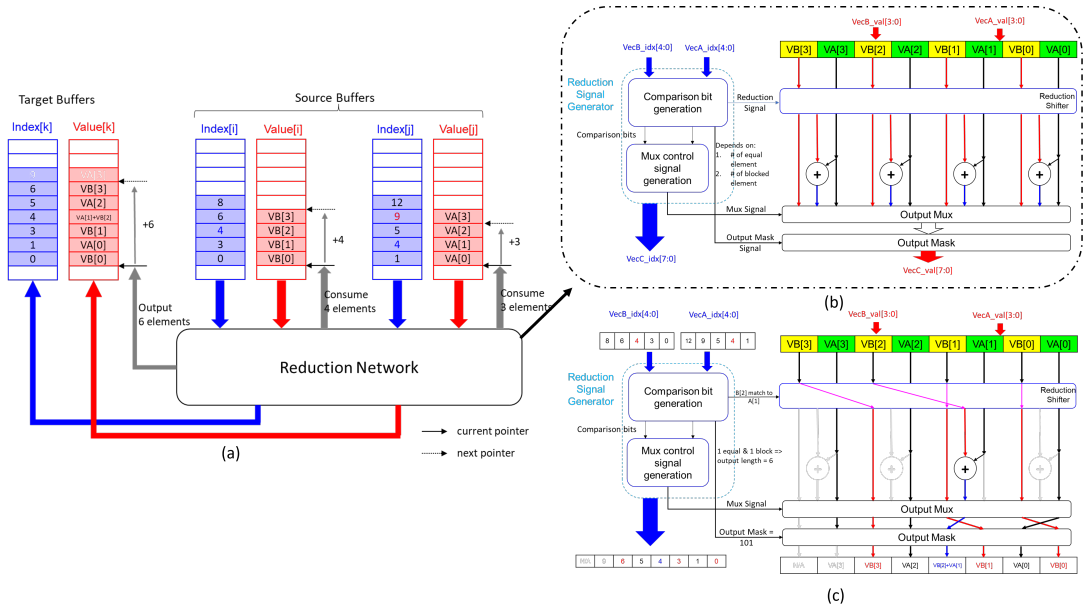

Figure 5: (a) Block diagram of the reduction operation of PrGEMM (b) The structure of the reduction network (c) Example of the reduction operation with block out and reduction happen

In each operation, the reduction network takes 4 values and up to 5 indices (4 reduction indices according to the loaded value, and 1 additional index acts as the lookahead index) from both vectors. Here, a and b denote the reduction network's input vectors, and idx_a, idx_b are the column indices of the input vectors. The comparison result between 2 sets of reduction indices generates two sets of 16 bits comparison signals, and these signals are sent to the reduction logic to create the mux control signals. Also, These comparisons generate the shift signals to redirect the values that need to be reduced in the corresponding adder. The shift signals are only used to shift the values in vector b to make the logic of the reduction shifter simple. Finally, the lookahead indices are compared to those in another vector and generate two 4-bits pre-merged masks separately. These 2 masks are merged into a single 8-bits output mask and adjusted by the index matching information to keep the output mask's correctness. These mask bits are not only for generating the output mask but also for updating the input pointer to correctly fetch the following operands from each vector. The computed result vector is stored in the output buffer and waiting to be merged to the result matrix. Based on the input pattern, several cases are possible in the reduction network:

No reduction and no block out: In this case , only the output mux control signals are generated to merge two input vectors. Because no matching signals are generated from the index comparison, the values from vector n do not need to redirect to the adder array, and all the input values are passed to the mux array. This case can generate the reduction network's maximum output (8 outputs).

Block out: The comparison results from the lookahead index and reduction indices generate 2 sets of 4-bits signals. These signals coalesce into an output mask, and each bit in the mask decides the result from the output mux needs to be blocked out or not. In this case, no reduction happens, and the result length only depends on how many indices been blocked out by both lookahead indices.

Both reduction and block out: In this case, the index matching signals pass to the reduction shifter in the reduction adder, and each value from vector b is redirected to the corresponding line of operation. Fig.5(c) shows the operation of 2 vectors with a matched index. the shifter shift VB[2] to the adder to compute with VA[1] and shift VB[3] to the original location of VB[2], to preserve the format of both vector.Because of the property of CSR format, the lookahead index cannot block out the reduced value (The index of reduced value must be smaller than both lookahead indices), which makes all the reduction operations valid and no conflict exists between the reduction operation and the output mask. The index matching signals also modifies the output mask to guarantee the correctness of output length. In Fig.5(c) case, the output length is 6 with 1 block out index and 1 index been reduced.

Input indices size smaller than 5: This case usually happens when both vectors almost reach the end of the input vector, the size of the input vector is too small, or the imbalance length between 2 input vectors. In this case, the input mask, a 5-bit mask to point out how many inputs are involved in the reduction, is considered to generate the correct reduction signals and output mask. Therefore, the 0 in the input mask is treated as an "always greater index" to be used to obtain the correct reduction signals and output masks.

After the reduction, the 4-bits pre-merged masks are used to update the pointer of the input vectors, and the merged mask is used to update the tail pointer of the result buffer. Fig.5(a) shows the block diagram of the reduction process and the update of the pointers. Once both pointers of the input vectors reach the end of the vector, the reduction is finished. This process continues until there is only one result vector in the buffer, which means the result vector is ready to be merged into the result matrix.

4.2 Accelerator design

Outside the reduction network, each processing element (PE) contains 2 data loaders used to load and buffer the cache line with the

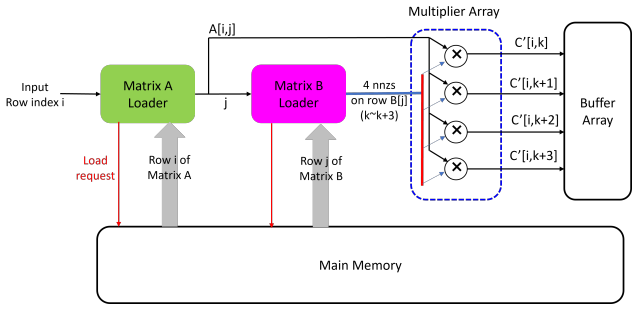

Figure 6: Gustavson's Algorithm based load unit and multiplication array

data part of matrix A and matrix B and a set of buffer arrays to buffer the partially completed row of the output matrix. Each PE operates 1 row of the result matrix at the same time. When the request for matrix multiplication arrives, the scheduler outside the PE can distribute the requested row to each PE. Once the matrix A loader receives the row index i of matrix A, it accesses the count vector to get the pointer information of the index and value vectors. After applying pointer to the metadata encodes in the physical address of these two vectors, matrix A loader directly accesses the memory to load the corresponding cache line. While receiving the cache line, matrix A loader sends out a pair of column index j and value A[i,j] to matrix B loader and the multiplier array, respectively. The matrix B loader does the same thing as matrix loader A to fetch the indices and values according to j. One thing different from the matrix A loader is that the matrix B loader needs to send out at most 4 pairs of indices and values. Fig.6 shows the block diagram of the loader and the multiplier array, here k to k+3 does not mean the continuous indices in matrix B. Because of the irregular length of each sparse vector and the cache line alignment, the matrix B loader might need to pre-load an additional cache line to generate 4 pairs of indices and values correctly. After the matrix B loader sends its data to the multiplier array, the results are computed and sent to the temporary buffer to be reduced.

The buffer setting is similar to the design in Matraptor [17], which maintains n+1 buffer arrays with n partial vector arrays and 1 destination array. If one of the buffer arrays is empty, the partial vector after multiplication is directly sent to one of the empty buffers. Once all n buffers are filled, the following partial vector will compute sparse vector reduction with the vector in one of the buffers. The result is sent to the destination array, and the buffer which participates in the operation will become the new destination buffer. To simplify the logic, we use a round-robin policy to decide which buffer will participate in the reduction process. When no result is sent from the multiplier, the partial vectors will merge into a single result vector and send the signal to the scheduler. The scheduler keeps tracking the finished row to guarantee the output matrix can be tightly packed to the CSR format. Because we can't know the exact size of the result matrix, 3 tightly stored CSR vectors are not packed tightly, however, the header line is responsible to keep the overall structure correct by storing the address of each vector in CSR format.

Table 2: Data Set for Experiment

Matrix	Dimension	NNZs
raefsky3	21.2k	1.49M
p2p-Gnutella31	62.6k	147.9k
2cubes-sphere	101.5k	1.65M
m133-b3	200.2k	800.8k
offshore	259.8k	4.24M
mario002	389.9k	2.1M
roadNet-CA	1.97M	5.53M

5 EXPERIMENT

Environment: To evaluate our work, we use intel OPAE [8] and the hardware abstraction layer (HAL) introduced by [16][7] to simulate the behavior of the communication between the accelerator and the main memory. The setting of PrGEMM uses 4-wide (4 entries + 1 lookahead index), same as the example used in the previous sections. The interface between the main memory and the accelerator is 1 cache line size (512-bit wide). The area analysis is based on Synopsys Design Compiler with 32nm technology node, and the double-precision multiplier/adder in the area analysis uses open-source circuits.

DataSet: The data set used in the simulation and implementation is a subset of the dataset in [17] and [12], which are selected from the SuiteSparse data set [3]. The table 2 shows the matrices used in the experiment. All the operations are computed with $C = A \times A$. Before the operation, all the matrices are converted to CSR format with a header cache line that stores the matrix's size and the offsets of each CSR vector. The counts and indices are stored in INT-32 format (16 entries/ cache line), and values are stored in FLOAT-64 format (8 entries/ cache line).

Baseline: In this work, we implement serial reduction as our baseline. This implementation mimics the reduction method in MatRaptor [17]. To simplify the comparison and focus on matrix multiplication, the peripheral circuits and buffer settings are almost identical.

5.1 Performance and Area

Performance: Figure 7 shows the performance comparison of SpGEMM between 4-wide PrGEMM and the serial reduction of sparse vectors. The experiment use 1 PE for both PrGEMM and the serial SpGEMM unit. Due to the limitation of the simulation platform (150 cycle for each cache line request) and the irregular access of the sparse matrices, the load operation dominates the total execution cycle of the SpGEMM operation. Therefore, the improvement of the total cycle is about 10%. In the computationally intense case (ex. raefsky3), the improvement on the total time can reach 30%.

If we only consider the cycles spent on the SpGEMM execution, including the time spent on scalar-to-vector multiplication and the sparse vector reduction, the overall improvement can reach 3.3x compared to the serial reduction. Across different cases, the improvement varied from 2.7x to 4.4x. This difference is generated by (1) The utilization of the multiplier (if the vector length is smaller than 4, the performance gain in the multiplication part is lower than 4) and (2) The reduction rate in the reduction stage. Because the floating-point adder in the reduction network only activates when

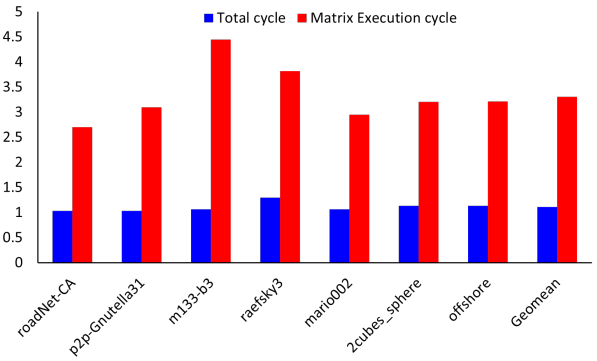

Figure 7: Ratio of execution cycle between 4-wide PrGEMM and serial reduction of SpGEMM

the indices match, the improvement is varied in different matching cases. For example, suppose there is only one matching index in both input chunks of the vector. In that case, the time consumption on the floating-point adder is dominated, and the improvement of these specific chunks of vector might decrease to 2x. On the other hand, if there are no matching indices, the improvement is only dominated by the lookahead indices, which can reach above 4x improvement. In the case of roadNet-CA, because most of the vectors are short, both the multiplier and the reduction network cannot be fully utilized, so the improvement is only 2.7x. In contrast, m133-b3 is a regular matrix with all the vector sizes =4, and the match indices in operation are relevantly small; therefore, the improvement can be 4.4x.

Area penalty: The primary source of the area penalty is from the floating-point multiplier, adder, and the additional comparison unit in the reduction network. We do not take the on-PE buffer into account to fairly compare the area. Using 32nm standard cell synthesis, the area increase between 4-wide PrGEMM and serial SpGEMM units is about 25%. Compared to the performance gain, this area penalty is reasonable.

6 RELATED WORKS

Several previous works proposed hardware acceleration for SpGEMM computation. ExTensor [6] and TensorDash use the inner product to compute SpGEMM. For ExTensor, it used the skip method to speed up the index matching in the inner product operation. For TensorDash, it uses index look-aside and look-ahead for early detection of multiply to zero. OuterSPACE[12] uses the outer product algorithm to compute SpGEMM, and keeps a list of the first element of each partial vector to enable a tree- or tournament-style reduction operation. However, these methods have fundamental issues, like 0-outputs (inner product), storage of partial matrices (outer product) and transpose of one of the operands, which make these method inefficient in space or in time.

MatRaptor[17] and Gamma[19] are the prior works that also used Gustavson's algorithm to build the accelerator. To the best of our knowledge, MatRaptor might be the first hardware accelerator which exploit the Gustavson's algorithm, and it is also the baseline of PrGEMM. MatRaptor proposed C^2SR to better utilize the memory bandwidth. In vector reduction, Matraptor computes 2 partial vector in a single operation with serial reduction. The main difference of MatRaptor and PrGEMM is the granularity of vector reduction, which MatRaptor is 1 and PrGEMM is 4. Also, PrGEMM does not need the preprocessing of the input matrix. Gamma uses a 64 wide tree type reduction network to compute the sparse vector reduction. Even though the output of this network is 1 element per reduction operation, the tree structure avoids computing similar partial vectors many times, which is suitable for denser matrices with higher nnz per row.

7 CONCLUSION

In this work, we construct an accelerator for sparse matrix-matrix multiplication, where the inputs and output remain in compact (CSR) format. To avoid incorrect ordering of indices and values in the result matrix, which prevents the parallel reduction of the sparse vectors, we used lookahead indices to block out the out-of-order indices and keep the result matrix in the correct format with parallel reduction. By using the reduction network with the granularity = 4, we can reach about 3.3x improvement in the SpGEMM operation with only 25% area penalty in the computation datapath.

REFERENCES

[1] Aydin Buluc and John Gilbert. 2012. Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments. *SIAM Journal on Scientific Computing* 34, 4 (2012), C170–C191.

[2] Paolo D'Alberto and Alexandru Nicolau. 2007. R-Kleene: A High-Performance Divide-and-Conquer Algorithm for the All-Pair Shortest Path for Densely Connected Networks. *Algorithmica* 47, 2 (feb 2007), 203–213.

[3] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (dec 2011), 25 pages.

[4] John R. Gilbert et al. 2008. A Unified Framework for Numerical and Combinatorial Computing. *Computing in Science Engineering* 10, 2 (2008), 20–25.

[5] Song Han, Huizi Mao, and William J. Dally. 2016. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. *arXiv: Computer Vision and Pattern Recognition* (2016).

[6] Kartik Hegde et al. 2019. ExTensor: An Accelerator for Sparse Tensor Algebra. In *MICRO-52*. Association for Computing Machinery, New York, NY, USA, 319–333.

[7] intel. 2015. *intel Hardware Abstraction Layer.* Technical Report. intel. https://www.intel.com/content/www/us/en/docs/programmable/683282/current/hardware-abstraction-layer.html

[8] intel OPAE. 2017. *intel OPAE.* Technical Report. intel. https://github.com/OPAE

[9] Shiyu Li et al. 2021. *ESCALATE: Boosting the Efficiency of Sparse CNN Accelerator with Kernel Decomposition.* Association for Computing Machinery, 992–1004.

[10] Ke Liu et al. 2020. RecNMP: Accelerating Personalized Recommendation with near-Memory Processing. In *ISCA-20*. IEEE Press, 790–803.

[11] Mostafa Mahmoud et al. 2020. TensorDash: Exploiting Sparsity to Accelerate Deep Neural Network Training. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 781–795.

[12] Subhankar Pal et al. 2018. OuterSPACE: An Outer Product Based Sparse Matrix Multiplication Accelerator. In *HPCA-2018*. 724–736.

[13] Angshuman Parashar et al. 2017. Scnn: An accelerator for compressed-sparse convolutional neural networks. *ACM SIGARCH Computer Architecture News* 45, 2 (2017), 27–40.

[14] Md. Mostofa Ali Patwary et al. 2015. Parallel Efficient Sparse Matrix-Matrix Multiplication on Multicore Platforms. In *High Performance Computing*. Springer International Publishing, 48–57.

[15] Eric Qin et al. 2020. SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 58–70.

[16] ARC Research. 2020. *intel-training-modules.* Technical Report. University of Florida. https://github.com/ARC-Lab-UF/intel-training-modules

[17] Nitish Srivastava et al. 2020. MatRaptor: A Sparse-Sparse Matrix Multiplication Accelerator Based on Row-Wise Product. In *MICRO-53*. 766–780.

[18] Raphael Yuster and Uri Zwick. 2004. Detecting Short Directed Cycles Using Rectangular Matrix Multiplication and Dynamic Programming. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '04)*. Society for Industrial and Applied Mathematics, USA, 254–260.

[19] Guowei Zhang et al. 2021. Gamma: Leveraging Gustavson's Algorithm to Accelerate Sparse Matrix Multiplication. In *ASPLOS 2021*. Association for Computing Machinery, New York, NY, USA, 687–701.