2022 35th International Conference on VLSI Design and 2022 21st International Conference on Embedded Systems (VLSID) | 978-1-6654-8505-0/22/$31.00 ©2022 IEEE | DOI: 10.1109/VLSID2022.2022.00031

MAPPARAT: A Resource Constrained FPGA-Based Accelerator for Sparse-Dense Matrix Multiplication

M. R. Ashuthosh, Santosh Krishna, Vishvas Sudarshan, Srinivasan Subramaniyan and Madhura Purnaprajna
Centre for Heterogeneous and Intelligent Processing Systems,
Dept of Electronics and Communication Engineering, P.E.S. University, Bangalore, India
{ashuthoshmr, santoshkrishna, madhurap}@pes.edu,{vishvas286, srinivasansubramaniam74}@gmail.com

***Abstract*—Matrix Multiplication has gained importance due to its wide usage in Deep Neural Networks. The presence of sparsity in matrices needs special considerations to avoid redundancy in computations and memory accesses. Sparsity becomes relevant in the choice of compression format for storage and memory access. In addition to compression format, the choice of the algorithm also influences the performance of the matrix multiplier. The interplay of algorithm and compression formats results in significant variations in several performance parameters such as execution time, memory, and total energy consumed.**

This paper presents MAPPARAT, a custom FPGA-based hardware accelerator for $sparse \times dense$ matrix multiplication. Our analysis show that the choice of the compression format is heavily dependent on sparsity of the input matrices. We present two variants of MAPPARAT based on the algorithm used for $sparse \times dense$ matrix multiplication, viz., row-wise and column-wise product. A difference in speedup of $2.5\times$ and a difference in energy consumption by about $2.6\times$ is seen between the two variants. We show that an intelligent choice of algorithm and compression format based on the variations in sparsity, matrix dimensions, and device specifications is necessary for performance acceleration. For identical sparse matrices, a speedup of up to $3.6\times$ is observed, when the dense format is chosen for one of the matrices for sparsity in the range of 30% to 90%. MAPPARAT on a resource-constrained device shows performance efficiency of up to 7 GOPs-per-second-per-watt.

I. Introduction

$Sparse \times Dense$ matrix multiplication has become an important component for hardware acceleration and is especially useful in deep-learning applications. Sparsity in the inputs, weights, and activation functions can lead to a large redundancy in computation and memory transfers. Irregular matrix dimensions, sparsity, compression format and algorithmic variations together with device specific constraints can lead to large variations in performance.

Matrix dimension and Sparsity: Exploiting input sparsity to reduce storage and computation is a natural step towards increasing performance and efficiency. Compressed sparse row (CSR) [1], compressed sparse column (CSC) [2], coordinate format (COO) [1], Run-length encoding (RLE) [2] and Bitmap [3] are some of the techniques for sparse matrix representation. Our analysis show that the choice of compression format should be driven by percentage sparsity and dimensions of input matrices, since it affects the memory footprint and the decoding overhead. The COO format provides the row and column addresses directly, whereas CSR, CSC, RLE and Bitmap require additional decoding to obtain the row/column address. This decoder network will likely contain comparators and counters, which will be a critical overhead for area/delay of the design. The irregular nature of sparse matrices where the number of rows are not equal to number of columns results in different overheads if stored in CSR or CSC formats, unlike COO which remains unchanged.

Algorithms: Matrix multiplication iterates over three nested loops to multiply input matrices $A_{M\times K}$ and $B_{K\times N}$ to produce the output $C_{M\times N}$. Reordering of these loops can lead to multiple design alternatives that have a corresponding impact on data reuse and on-chip memory requirement, even though the total number of computations may remain unchanged. The naive inner product version, accounts for a dot product implementation without any data-reuse. The outer-product algorithm produces the output matrix by multiplying a column of $A_{M\times K}$ with the corresponding row of $B_{K\times N}$ to obtain partial products of the size of the whole $C_{M\times N}$ matrices which are accumulated. The row-wise algorithm multiplies each element of a row of $A_{M\times K}$ with all the elements of the corresponding row of $B_{K\times N}$ to get the output row. This process is repeated for all the rows of the $A_{M\times K}$ to get the rows of the $C_{M\times N}$. Column-wise algorithm works in a similar fashion, but in column major order. Here, every element of a column of $B_{K\times N}$ is multiplied with all the elements of the corresponding column of $A_{M\times K}$ to get an output column. This process is then repeated for all the columns of the $B_{K\times N}$ to get the columns of the $C_{M\times N}$. Depending on the choice of data organisation, the number of accesses to the input matrix varies.

Device-specific constraints: An intelligent placement of data (on-chip versus off-chip) can make a major impact on performance. Given the irregular nature of matrix dimensions and sparsities of the input matrices, the matrix with relatively high frequency of accesses is placed on-chip and the one with fewer accesses can be moved off-chip, thereby saving energy. In addition, the I/O interface that facilitates data movement becomes a major factor in the utilisation of compute and memory resources. Consequently, the choice of the algorithm for a given matrix characteristic (dimension and sparsity) can be altered significantly based on device-specific resource constraints.

978-1-6654-8505-0/22/$31.00 ©2022 IEEE
DOI 10.1109/VLSID2022.2022.00031

In this paper, our contributions are as follows.

- Analysis to show variation in performance based on matrix type(square or non-square dimensions), sparsity and impact on the choice of algorithm and device-specific constraints (memory, compute and interface bandwidth).
- Custom hardware design of a $sparse \times dense$ matrix multiplier called MAPPARAT with two algorithmic variants, viz., row-wise and column-wise.

II. MAPPARAT Architecture

In this section we describe the design consideration towards architecting our custom hardware accelerator for $sparse \times dense$ matrix multiplication called MAPPARAT. To accommodate moderate sparsity, irregular matrix dimensions, the COO format is chosen. Data is 32-bit floating point. To validate the impact of algorithmic variations on device-specific resource constraints, two design alternatives, viz., MAPPARAT-RW (with row-wise product) and MAPPARAT-CW (column-wise product) are presented. Through these variants, we show that the choice of the algorithm is a decision based on application-level characteristics (matrix dimension, sparsity) and resource constraints (compute, memory, interface bandwidth). $Sparse \times Dense$ matrix multiplication involves five stages listed below.

Data Fetch: Sparse/Dense input matrices are fetched and forwarded to the address generation unit. The sparse matrix is stored/fetched in COO format if the sparsity is higher than 66.6%. For cases with sparsity less than 66.6%, the matrix is stored/fetched in dense format since the total data movement is less in the former.

Indexing: The address generation unit is responsible for feeding the data through the right set of input indices (one sparse and one dense) to the multiplier, based on the algorithm chosen. Maximising parallelisation relies on the possibility to obtain multiple input data through parallel indexing and parallel computation.

Multiplication: In this step, data available through indexing is multiplied, with the objective to maximise performance. Parallel computation and pipelining is considered, if resource constraints are met.

Accumulation: The partial products generated by the multipliers are accumulated in this step. The difference in the number of multiplications and additions can result in under utilisation of the multipliers. The accumulation unit can become a bottleneck if the right method of accumulation is not chosen. Based on the nature (sparse or dense) of the partial products obtained, the following approaches can be considered.

Approach 1: As seen in Figure 1, addition is performed only if the row-index or column-index of the partial product match, for this approach every index of the partial product generated needs to be compared. This approach can lead to reduction in frequency and increase in latency on account of the additional parallel comparators in the critical path. However, the number of adders used is limited to the number of matched indices.

Approach 2: As seen in Figure 2, the sparse partial products are stored in register arrays of the size equal to the number

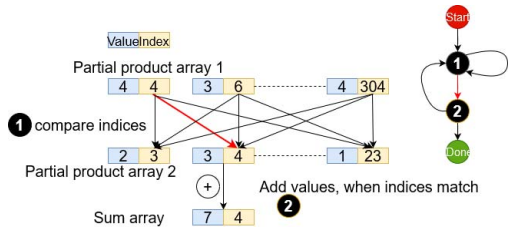

Fig. 1: Approach 1 for sparse partial product accumulation

of rows (or columns) based on the row-index (or column-index) of the sparse partial product. Through this approach the sparse partial products are stored according to the row (or column) indices, along with the zeroes. All the values in this register are accumulated, including the redundant values (addition of zero with zero) in row-wise (or column-wise) fashion. This approach avoids the index matching using a mesh of comparators, as in approach 1. In terms of latency and operational frequency, approach 2 is better than approach 1, but needs more compute and memory resources than the approach 1. The number of adders needed are equal to the number of rows (or columns) of partial products to be added, facilitating parallel addition. Dense partial product that are in the register already in order in row (or column) format are accumulated in row-wise(or column-wise) fashion.

Approach 3: In this approach the row-index (or column-index) of the new set of partial products to be added are used as addresses to obtain the previous partial product from the on-chip memory, as seen in Figure 3. This process is repeated for every set of partial product generated. Each multiplier is connected to an adder with a memory block, permitting independent addition. However after each multiplications for a particular row/column, the contents of the memory used for parallel addition is sent to an adder tree . The level of branches of adder tree is $\log_2$(number of memory blocks in parallel). The multipliers are stalled from generating partial products for next set of inputs until the adder tree completes the accumulation. The final row/column is the output from the adder tree. BRAM is used for partial sum storage instead of flip flops, to reduce logic. With this approach, the additions are stalled until the previous partial product has been added and written into the memory, on account of memory access latency, increasing the total latency of the implementation.

Storage The output of multiply-accumulation is the final output data or partial product. Limiting on-chip data storage increases the number of accesses to off-chip memory, which is undesirable, yet, limiting the partial sum storage will limit the number of parallel computations done every cycle. Depending on the algorithm and the number of compute resources, the matrix could be divided either into chunks of number of rows (or columns), minimising on-chip data storage and off-chip memory access. On-chip storage is a trade-off between stall-free parallel computation and minimal off-chip access.

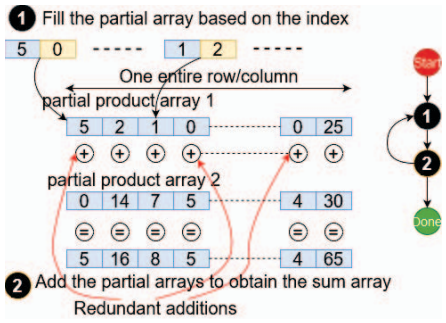

Fig. 2: Approach 2 for sparse partial product accumulation

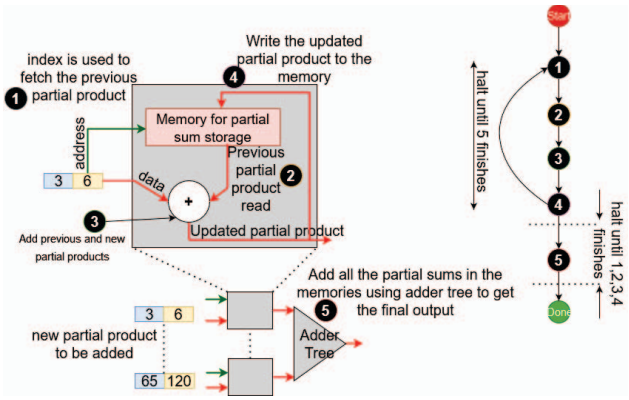

Fig. 3: Approach 3 for sparse partial product accumulation

A. MAPPARAT-RW

Row-wise product algorithm multiples every value in the sparse matrix (A) with all the elements of the corresponding row in the dense matrix (B). The row-wise partial products that are dense in nature are accumulated to obtain the final output as shown in Figure 4. To avoid frequent off-chip memory access, the matrix B is stored on-chip. If the device is resource limited then the B matrix of size *K* x *N* is split up into smaller matrices of size *K* x *N'*, which have the same number of rows but fewer columns. There are *N*/*N'* such sub-matrices.

The first dense sub-matrix (B_1) is fetched from the off-chip storage and stored in BRAMs. Next, matrix A is fed through the input ports. Once the entire matrix A has been operated on with the B_1, then the second sub-matrix B_2 is loaded onto the chip and the process repeats till the multiplication of A against B_m is done where m indicates the number of chunks of B. Column index of the sparse value indicates the row of dense matrix to be multiplied with the sparse value. Address generation unit reads the column index of the sparse value and generates address to the corresponding row of dense values. One input to the multiplier is the sparse value and the second value comes from the BRAMs that store the dense sub-matrix. The number of multipliers is equal to the number of columns in the sub-matrix(*N'*) which is chosen to be maximum number of multipliers available in the target device. Hence, the output of the multipliers is a 1 x *N'* vector.

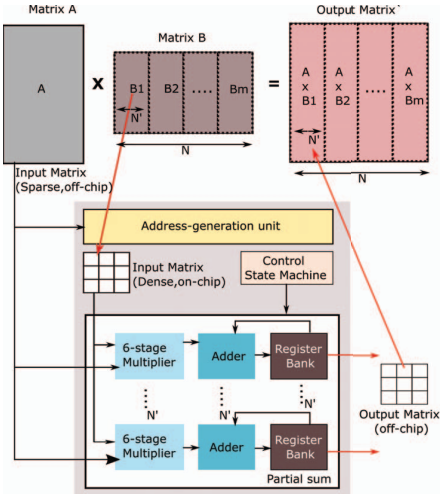

Fig. 4: MAPPARAT-RW: Internal block diagram of $Sparse \times Dense$ matrix multiplication using row-wise product algorithm.

The accumulation of the partial products generated by the multiplier is carried out until the final result of the particular row is obtained. The row index of the sparse value indicates if the accumulation needs to continue or reset for next row output. Since the nature of partial product generated are dense, approach 2 where the number of adders equivalent to the size of row/column (column(*N'*) in this case) is used. There is no requirement of storing the partial sums as every partial product is added with the next value and they need not be indexed before adding. In this implementation, the on-chip storage has a sub-matrix *K* x *N'*, which is replaced with the next chunk after the entire matrix A has been operated on this sub-matrix.

B. MAPPARAT-CW

Column-wise product algorithm multiples every value in the dense matrix (B) with all the non-zero elements of the corresponding column in the sparse matrix (A). The column-wise partial products that are sparse in nature are accumulated to obtain the final output. In the column-wise product algorithm, the matrix A is accessed more frequently compared to the dense matrix B. Hence the matrix A needs to be placed closer to compute. Matrix A is stored on chip in the COO format. The internal block diagram of column-wise implementation is shown in Figure 5. If the device is resource limited then the A matrix is split up into smaller matrices of size *M'*$\times$ K, which have the same number of columns but fewer rows. There are *M*/*M'* such sub-matrices. The number of non-zeros in each sub-matrix is assumed to be equally distributed. This assumption leads to the fact that the column index for each non-zero entry need not be stored since it can be directly inferred from constant number of non-zero entries in each column(also called as column weight NNZ_C). If matrix A is split up into smaller matrices each of size *M'*$\times$ K, then the matrix B has to fully operate on each sub-matrices. Matrix B is brought from off-chip as many times as A is divided into.

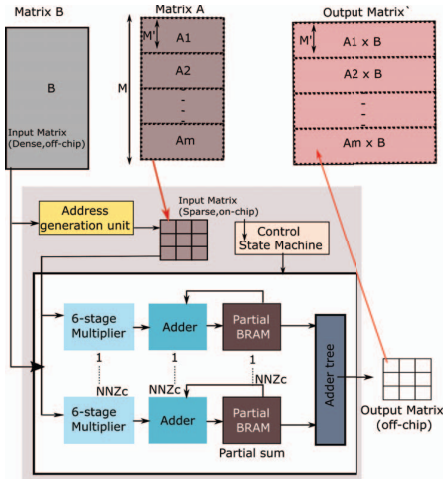

Fig. 5: MAPPARAT-CW: Internal block diagram of $Sparse \times Dense$ matrix multiplication using column-wise product algorithm.

Matrix B is read in the column-major order from the off-chip memory. For every dense value that is fetched, the address generation unit sends the address to the on-chip memory to obtain the corresponding column of matrix A. Each dense value is multiplied with all the non-zero entries present in that column of matrix A. Number of multipliers used is equal to the number of non zero entries present in that column of matrix A. The multipliers generate the partial products that are sparse in nature that requires index matching for accumulation. Approach 3 from section II is used for this case. Each of the partial products produced is then fed as one of the inputs to an adder. The second input is read from a BRAM memory associated with the adder. The produced sum is written back in the same BRAM location. Once the entire dense column has been processed, the contents of the BRAMs holding the partial sums are accumulated using an adder tree and sent out as the final output. The BRAMs holding partial sums are reset and the process is repeated for the next column of matrix B until all of matrix B is operated on matrix A.

III. Experimental Setup and Results

MAPPARAT was implemented on the Arty$-A7\ 100T$ board with the FPGA $XC7A100TCSG324-1$. The matrix used for implementation was taken from SuiteSparse [4] matrix collection. The verilog design was synthesized and validated functionally through post-synthesis simulations in Vivado 2019.2.

A. Resource constraints

The design has two 32-bit input interfaces and two 64-bit output interfaces to off-chip memory. The off-chip frequency and the on-chip frequency are the same. The total number of 32-bit multipliers that can be accommodated are 60, each utilising 4 DSP slices. The available on-chip memory is 607 KB.

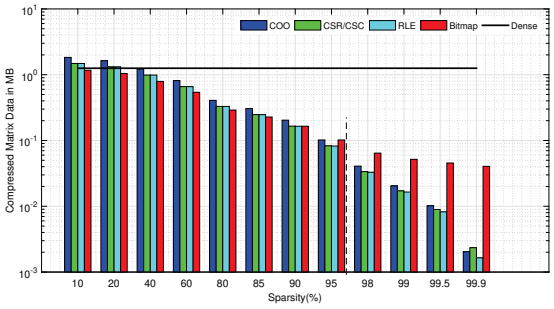

Fig. 6: Comparison of compressed matrix sizes with varying sparsity for different compression techniques for a matrix of dimension 560×560. Data used is 32-bit floating point. Since this is a square matrix, CSR and CSC have identical sizes. Bitmap has the smallest memory footprint for sparsity up to 95% and has higher memory overheads for matrices with sparsity above 95%. COO has a marginally lower footprint compared to CSR/CSC for sparsity above 99.9%.

B. Sparsity versus Meta-data

We compare the impact of meta-data (overheads of compression) with varying sparsity levels. Figure 6 compares Dense, CSR, CSC, COO, RLE and Bitmap. Each metadata is 32-bits, except for the case of Bitmap where the meta data is a 1-bit matrix. RLE is better in all cases, but decoding the RLE format consumes extra processing time. For sparsity ranging from 10% to 95%, Bitmap has the lowest memory footprint, followed by CSR/CSC and then COO. The Bitmap format is good at compression for the matrices of sparsity less than 95%, resulting in at least 50% less memory overhead than other formats. Beyond 99% sparsity, Bitmap has a very large overhead of over 100%. COO, CSR, CSC requires fewer bytes than Bitmap, beyond 95% sparsity. CSC and CSR have identical memory footprints for square matrices. For non-square matrices where the number of rows is greater than the number of columns, CSC format occupies less space than CSR, and vice versa. COO format depends only on the number of non-zeros and is not effected by the change in dimension of the matrix as in the case of CSR/CSC. Overall, COO is a balanced compromise between decoding overhead and memory footprint.

C. MAPPARAT-RW versus MAPPARAT-CW

Table I shows the results obtained for the row-wise and column-wise implementation for $sparse \times dense$ matrix multiplication with the matrix size of 560×560.

For the row-wise implementation, as mentioned in Section III-A, the total available storage on-chip is $607KB$. Since the entire matrix B cannot be stored ion-chip, the 560×60 chunk of matrix B was stored. With this chunking, the entire sparse matrix (A) can be stored on-chip. All the DSP slices of the device were completely utilised since there is enough dense data to operate on without any stalls. The maximum frequency of operation of this implementation is limited by the 64-bit additions of the partial product every cycle. The total latency is the sum of the total latency spent in multiplication

TABLE I: MAPPARAT: Resource utilisation on XC7A100TCSG324

Algorithm	BRAM	LUTs	FFs	DSPs	Frequency (MHz)	Latency (cycles)	Energy (mJ)
Row-wise	75(55%)	13830(21.81%)	7680(6.06%)	240(100%)	240.44	342720	2.48
Column-wise	97(71.22%)	9216(14.53%)	12288(9.69%)	192(80%)	226.19	853720	6.57

of A with all the chunks of B and the latency spent in refilling the storage with the chunks of matrix B.

For the column-wise implementation, the utilization of DSPs is lower than that of row-wise implementation. The entire sparse matrix A is stored on-chip. There is still memory on-chip enough to fit a column of dense matrix. Hence, a column of the dense matrix B is stored on-chip. The on-chip storage is also used to store partial products as described in approach 3 of Figure 3. For every dense value of matrix B there exists 16 non-zero entries in every column of the sparse matrix A which leads to 16 multiplications. This way the multiplications are limited to multiples of the number of non-zero entries per column without exceeding the maximum number of multipliers present in the target device. Hence the current implementation uses 48 multipliers every cycle to multiply 3 dense values with their corresponding column values of matrix A. The maximum frequency of operation of this implementation is limited by the accumulation unit that computes 64-bit additions of the partial product every cycle. The latency of this implementation is higher than that of the row-wise implementation since, all the 60 multipliers on-chip are not being utilized and the stall in the multiplication until the adder tree completes the accumulation of the sparse partial products. The accumulation overhead is avoided in row-wise implementation due to the dense nature of the partial products and hence performs better.

D. MAPPARAT for varying dimensions and sparsity

Based on the row-wise and column-wise implementation with a fixed sparsity, latency was also estimated for the same dimensions with varying sparsity as shown in the Table III. It can be observed that the trend of row-wise being better than the column-wise in terms of speed for $sparse \times dense$ matrix multiplication remained the same for variations of sparsity. The reason for the same is due to the stalls caused by the resource limited approach in accumulation of the sparse partial products. For the $sparse \times sparse$ matrix multiplication, the row-wise and column-wise implementation both result in same latency since identical sparsity is considered for both the matrices. For the range of sparsity as mentioned in the Table III a speedup is observed when one of the matrices is chosen to be dense instead of sparse for sparsities between 30% and 90%.

For a fixed sparsity of matrix A and matrix B, the latency were estimated for different dimensions including both square and non-square irregular matrices that are found in modern deep learning workloads. Matrix A was chosen to be 80% sparse. Matrix B was chosen to be 30% sparse, which had more data in compressed format than in the dense format. With the same target resource constraints, both $sparse \times dense$ and $sparse \times sparse$ latency estimations were tabulated in the table II. For the case when the matrix B is less sparse than matrix A and when the data to store matrix B in compressed format is more than the normal dense format, $sparse \times dense$ multiplication outperforms $sparse \times sparse$ multiplication in terms of speed. For $sparse \times dense$ multiplication, row-wise approach is faster than the column-wise approach. For $dense \times sparse$ matrix multiplication this situation is reversed as row-wise algorithm now leads to partial products that have fewer non-zero entries and hence column-wise is better suited. From the results of Table II, we conclude that exploitation of sparsity of matrices necessarily translates to better performance indicating design space exploration is important to make right decisions based on the algorithm, sparsity and target device.

TABLE II: MAPPARAT for varying dimensions: Latency versus varying dimensions (square and non-square matrices) of Matrix A with 80% sparsity and Matrix B with 30% in cyles.

Algorithm	Matrix A	Matrix B	Sparse×Sparse	Sparse×Dense
Row-wise	560×560	560×560	5244960	**1422400**
Col-wise			4842880	3588480
Row-wise	256×2048	2048×256	10893348	**1355780**
Col-wise			11338764	5588788
Row-wise	2048×32	32×4096	6626688	**1875144**
Col-wise			15395114	13560064

TABLE III: MAPPARAT: Variation in latency for varying sparsity for the matrix dimensions of 560×560 in cycles

Sparsity	Algorithm	Sparse × Dense	Sparse × Sparse
30%	Row-wise	**4558400**	16220960
	Column-wise	11775680	
70%	Row-wise	**2049600**	3380160
	Column-wise	5224800	
80%	Row-wise	**1422400**	1715840
	Column-wise	3588480	
90%	Row-wise	795200	**701120**
	Column-wise	1951040	

E. Comparison with other state-of-the-art accelerators

Table IV compares MAPPARAT with the state-of-the-art accelerators in terms of GOP/s/W (Giga-Operations/Second/Watt). The total number of operations for architectures that carry out $sparse \times sparse$ matrix multiplication report peak performance for the respective implementations. Since MAPPARAT is resource-constrained, ASICs (Application Specific Integrated Circuits) tend to have higher performance efficiency. From Table IV it can be seen that MAPPARAT's performance lies close to MatRaptor [5], even with fewer resources and operating frequency on account of carefully selecting the compression format and algorithm used. As compared to other implementations that outperform MAPPARAT, the resources are significantly higher. With more resources such as multipliers, on-chip memory and higher memory bandwidth, the performance efficiency of MAPPARAT can come closer to these custom implementations.

TABLE IV: Comparison of MAPPARAT with other accelerators in terms of performance efficiency.

	[3]	[6]	[7]	[2]	[5]	[This work]
Multipliers	16384	16384	256	64	16	60
Frequency (MHz)	500	500	1500	800	1000	240
Memory bandwidth (GB/s)	1024	1024	128	-	128	2
GOP/s	16000	16000	384	102	16	14
Power (W)	22.3	12.5	24	0.6	1.3	2
GOP/s/W	717.5	1280	16	170	12.3	7

IV. Related Work

Most of the recent accelerators focus on performance improvement by relying on one algorithm and a fixed compression technique. The impact of matrix dimensions and sparsity on the choice of algorithm and compression format has not been studied in any of the accelerators. The role of resource constraints in the design decision is entirely overlooked since they are mainly custom accelerators.

TPUs [6] rely on the inner-product algorithm for multiplication, with high-performance systolic arrays, performing well only when the sparsity levels are low for both matrices. Sparse-TPU [8] extends the TPU to perform sparse matrix-dense vector multiplication. It outperforms TPU by factor $16.08\times$ by compressing the sparse matrix to having fewer columns. This improvement in the performance comes at a cost of 12.93% area overhead over a TPU baseline. SpArch [9] and OuterSPACE [7] use the outer-product algorithm for its $sparse \times sparse$ matrix multiplication, where the first matrix is stored in CSR format and second matrix in stored in CSC format. One of the unnecessary dimensions of both the CSR and CSC formats is reduced into CR and CC due to the nature of algorithm (outer-product) used. However, huge on-chip memory requirement for partial product storage and fixed format for compression chosen can limit performance. SpArch achieves $4\times$ improvement in the performance as compared OuterSPACE by reducing the number of partial matrices generated by outer-product algorithm with a technique called *matrix condensing*. MatRaptor and SIGMA [3] rely on row-wise product as a balance between the data re-use and on-chip memory. MatRaptor looks at matrices with very high sparsity, using the C^2SR format. Sigma on the other hand, uses Bitmap as the compression format for small matrices with moderate sparsity. Eyeriss [10] and EIE [2] use column-wise product for computations and CSC as the compression format. Since there are no resource constraints, the on-chip storage is enough to accommodate the first matrix, facilitating complete reuse of the values read from the off-chip memory. All the above mentioned implementations are custom ASICs, where resource requirement are well-suited to the algorithm under consideration.

FPGA-based sparse-matrix multiplication accelerators have custom design considerations for very sparse matrices [11], [12]. ESE [11] uses CSC format and relies on quantisation as additional method of compression for efficiency. Authors Fowers et al., propose a new compression format called CISR (Compressed Interleaved Sparse Row) primarily for parallel sparse-matrix vector multiplication. However, access to high-speed memory gives an added advantage of reducing data movement overheads. In contrast, MAPPARAT is a custom FPGA-based hardware accelerator with two variants (CW and RW) to suit sparsity (moderate and high) and matrix dimensions (small and large). Bringing in the dimension of resource constraint (device-specific), introduces additional constraints in data movement and storage that varies with compression format and matrix dimensions.

V. Conclusions and Future Work

The inter-play of algorithm, compression formats, resource constraints and the overall impact on performance on the design of $sparse \times dense$ matrix multiplication have been discussed in this paper. MAPPARAT is our custom FPGA-based hardware accelerator for sparse-dense matrix multiplication implemented on the ARTY-7 board. Two variants of MAPPARAT, viz., with row-wise and column-wise implementations have been compared for varying dimensions, sparsities and resource constraints. MAPPARAT-RW has a speedup of $2.5\times$ and reduction in energy consumption of $2.6\times$ as compared to MAPPARAT-CW.

References

[1] O. Hamdi and E. Rim, "Performance evaluation of algorithms for sparse-dense matrix product," *International Journal of Scientific and Research Publications (IJSRP)*, vol. 10, pp. 348–358, 10 2020.

[2] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient Inference Engine on Compressed Deep Neural Network," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 243–254.

[3] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, "SIGMA: A sparse and irregular GEMM accelerator with flexible interconnects for DNN training," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 58–70.

[4] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, Dec. 2011.

[5] N. Srivastava, H. Jin, J. Liu, D. Albonesi, and Z. Zhang, "MatRaptor: A sparse-sparse matrix multiplication accelerator based on row-wise product," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 766–780.

[6] "Cloud TPU," https://cloud.google.com/tpu, accessed: 2021-03-29.

[7] S. Pal, J. Beaumont, D. Park, A. Amarnath, S. Feng, C. Chakrabarti, H. Kim, D. Blaauw, T. Mudge, and R. Dreslinski, "OuterSPACE: An Outer Product Based Sparse Matrix Multiplication Accelerator," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 724–736.

[8] X. He, S. Pal, A. Amarnath, S. Feng, D.-H. Park, A. Rovinski, H. Ye, Y. Chen, R. Dreslinski, and T. Mudge, "Sparse-TPU: Adapting Systolic Arrays for Sparse Matrices," in *Proceedings of the 34th ACM International Conference on Supercomputing*, 2020.

[9] Z. Zhang, H. Wang, S. Han, and W. J. Dally, "SpArch: Efficient architecture for sparse matrix multiplication," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 261–274.

[10] Y. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2017.

[11] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, H. Yang, and W. B. J. Dally, "ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA," ser. FPGA 2017.

[12] J. Fowers, K. Ovtcharov, K. Strauss, E. S. Chung, and G. Stitt, "A High Memory Bandwidth FPGA Accelerator for Sparse Matrix-Vector Multiplication," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, 2014, pp. 36–43.