eSSpMV: An Embedded-FPGA-based Hardware Accelerator for Symmetric Sparse Matrix-Vector Multiplication

Ruiqi Chen[1], Haoyang Zhang[1], Yuhanxiao Ma[2], Jianli Chen[1], Jun Yu[1], Kun Wang[1,*]
[1]State Key Lab of ASIC & System, Fudan University, Shanghai, China
[2]Gallatin School of Individualized Study, New York University, New York, USA
kun.wang@ieee.org

***Abstract*—Symmetric Sparse Matrix-Vector Multiplication (SSpMV) is a prevalent operation in numerous application domains ($e.g.$, physical simulations, machine learning, and graph processing). Existing researches focus on the SSpMV implementation and its improvement on high-performance computing platforms but ignore the resource-limited edge platforms due to the main challenges: memory access overload and limited computing parallelism feasibility. To this end, this paper proposes an embedded-FPGA-based hardware accelerator for SSpMV, called eSSpMV. We first propose an optimized data format, named Symmetric Compressed Sparse Row (SCSR), to reduce memory consumption. Moreover, a fully-pipelined computation unit is proposed to be compatible with the optimized data format. Experimental results show that eSSpMV outperforms the state-of-the-art FPGA implementation for $2.9\times$ speedup, while still achieving a computing resource reduction of 39.3% and 32.3% for LUT and DSP, respectively. As for edge CPU and GPU implementations, eSSpMV achieves $9.3\times$ speedup over CPU while acquiring $13.1\times$ better power latency product than GPU.**

I. Introduction

Symmetric Sparse Matrix-Vector Multiplication (SSpMV), the computational core of various algorithms ($e.g.$, SVM [1], tensor-train decomposition [2], and quasi-newton method [3]), is successfully improved on high-performance computing (HPC) platforms [4, 5] for physical simulation, machine learning, and graph processing. However, directly deploying this HPC-targeting unit on edge devices is not ideal due to limited resources. Therefore, in order to deploy SSpMV in a manner of energy-efficient, embedded FPGA (eFPGA) is the ideal candidate platform among various resource-limited edge devices due to the flexibility and parallelism [6, 7].

Previous FPGA-based studies usually aim to improve data access efficiency. In [8], the sparse matrix is encoded in Sparse Symmetric Skyline (SSS) format to achieve data compression. Other researches rely on the high-bandwidth memory (HBM) technology [9–11], available only on advanced FPGA but not for edge-computing scenarios, to increase memory access for parallelism demands. However, the above works are far from meeting the demands of efficient deployment of SSpMV. The key challenges can thus be derived as: (1) traditional sparse storage formats ($e.g.$, CSR, CSC, and COO) have low utilization of memory bandwidth, due to the random data accesses in parallel SSpMV computation [12]; (2) Designing efficient parallelism is difficult on resource-limited edge devices, for the existing works of SSpMV rely on HBM or other complex load balancing modules.

To this end, we propose an eFPGA-based hardware accelerator for symmetric sparse matrix-vector multiplication (**eSSpMV**), in order to make full use of the restricted memory bandwidth, and to maximize parallelism within limited computation resources to gain high utilization ratio. Our contributions are listed as follows:

Efficient Architecture: We first propose an optimized Symmetric Compressed Sparse Row (SCSR) storage format to reduce the overload of memory access. An eFPGA-based hardware accelerator with computation unit (CU) for SSpMV is then designed, based on SCSR, with multiple hardware threads to further explore parallelism design, which leads to a high DSP efficiency of up to 97.94%.

High Performance: Experimental results show that the **eSSpMV** achieves $2.9\times$ speedup compared with the state-of-the-art (SOTA) FPGA implementation, and still obtains a resource reduction of 39.3% and 32.3% for LUT and DSP, respectively. **eSSpMV** is also $9.3\times$ faster than edge CPU implementation and $13.1\times$ energy-efficient than edge GPU implementation on average.

II. Background and Motivation

A. Background

Symmetric Sparse-Matrix is a special variation of sparse matrix where $a(i,j) = a(j,i)$ for all i and j. The matrix can thus be divided diagonally into two components called upper and lower triangular parts, either of which can be determined from the other one. To avoid lots of meaningless operations that waste computation resources, various compression methods have been proposed.

The CSR format is shown in Fig.1(b), which is a commonly used representation for sparse matrices [13]. CSR uses three arrays to represent the matrix. *value*, *col_index*, and *row_index* store the non-zero elements, the column indices of non-zero elements, and the starting index of each row, respectively.

The other widely used storage format is the SSS format [14], a variation of the CSR, as shown in Fig.1(c). SSS stores the lower triangular part of the matrix and the main diagonal of the matrix in separate arrays in order to increase the data compression rate. Note that the SSS format pads the main diagonal with zeros if it is not fully occupied.

B. Motivation

Computation under SSS format often encounters too much zero padding when the main diagonal is not fully occupied,

* Corresponding author. This work was financially supported in part by National Key Research and Development Program of China under Grant 2021YFA1003602, in part by Shanghai Pujiang Program under Grant 22PJD003, and in part by the Natural Science Foundation for Distinguished Young Scholars of Jiangsu Province, China under Grant BK20200038.

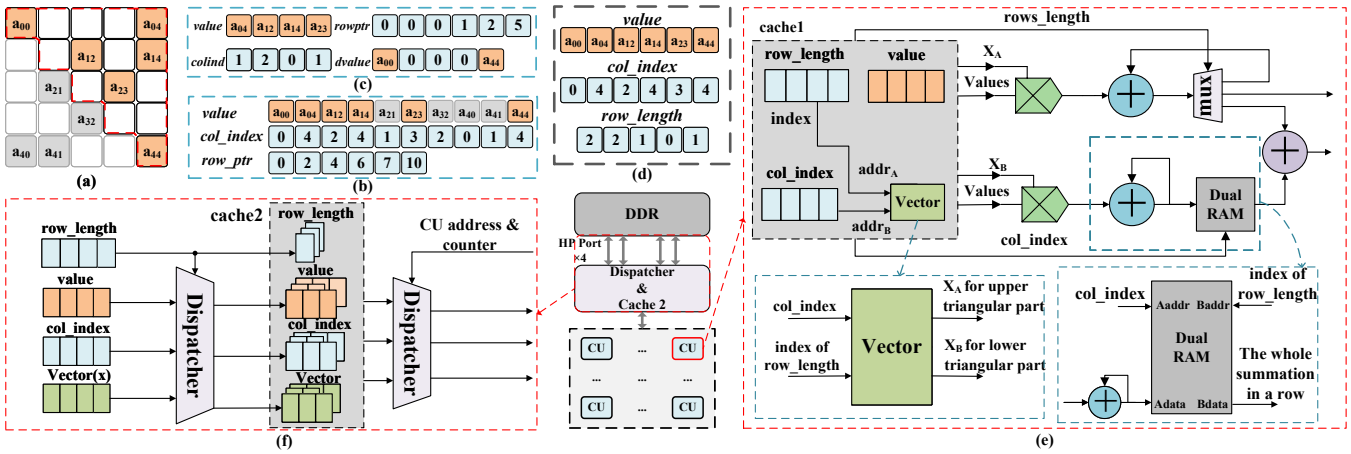

Fig. 1. Overview of the edge symmetric sparse-matrix multiplication (**eSSpMV**). (a) Symmetric sparse-matrix; (b) CSR format; (c) SSS format; (d) SCSR format; (e) The block diagram of the **eSSpMV**; (f) The block diagram of the dispatcher & cache2.

resulting in inefficiency. Moreover, multi-thread operation enlarges the inefficiency of repeating the main diagonal computation. Previous works implemented on multi-core systems or utilize the HBM technology, without considering the limitation of power consumption and computation resources, which means they cannot be deployed on resource-constrained edge platforms. Furthermore, both CSR and SSS are data-dependent to the access of matrix values, thus making them not suitable for fully-pipelined streaming FPGA designs. Also given the lack of a built-in hardware prefetcher, hiding the memory controller latency is crucial [15].

Therefore, for potential applications on edge platforms with limited resources, especially on eFPGAs, we aim to deploy SSpMV with high performance and low power consumption. Meanwhile, an analysis model is proposed to complete the exploration of design space efficiently and automatically.

III. Edge SSpMV

A. SCSR format

The key to computing SSpMV efficiently is optimizing memory access and parallelism. For memory access, it is necessary to use an appropriate matrix format. We optimize and propose the SCSR format by combining SSS with an existing CSR method [16]. It stores matrix information using three arrays, *value*, *col_index*, and *row_length*, as shown in Fig. 1(d). The *value* array utilizes the symmetry features to further save storage place, by storing only non-zero elements of the upper part and the main diagonal without filling zeros. *col_index* and *row_length* store the column indices of the non-zero elements and the number of non-zero elements in each row, respectively. In addition, *row_length* helps quickly determine whether or not the current row is computed. Although the compression rate of SCSR is lower than SSS, the improved format can be well applied to hardware parallel pipeline computing.

B. Proposed eSSpMV

eSSpMV Architecture: We make full use of the parallelism of FPGA, and simultaneously complete multiple vector multiplication operations with the same data input. Our computation units are denoted as CU, of which the specific design is also shown in Fig. 1(e). In terms of structure, the main modules of an initial CU include multipliers, adders, MUXs, and Dual BRAMs. Our design utilizes four 128-bit HP ports to transfer data between DDR and **eSSpMV**. Due to the symmetry, multiplications of the upper and lower part can be done in parallel, with only one of them being stored. Specifically, the multiplier for the upper triangular is the vector value indexed by the *col_index*, while those for lower part are indexed by the *row_length*. Then, for the upper triangular part, the MUX determines whether to continue the accumulation or output to the next stage based on the value of *row_length*. For the lower triangular part, its column address can be retrieved from the index of *row_length* of the upper part. Correspondingly, the row address of the lower part is derived from the *col_index*. Thus, values that have the same column address are computed, accumulated, and then updated in BRAM immediately. Eventually, all the values of the same column are computed and thus to obtain the result for one row of the lower part. The corresponding lower part is certainly completed by the time the upper part reaches the last adder, which also shows the advantages of storing only the upper part. Each of these modules has an enable port and a gated clock to ensure that modules work only when necessary, in order to further save power consumption. Our design can avoid conflicts of the main diagonal elements because we use *row_length* to determine whether the current row operation is finished.

Parallel Design of eSSpMV: The proposed CU can be reconfigured to process vectors in mixed-grained parallelism, where fine-grained means the number of DSPs within one CU, and coarse-grained implies the number of CUs.

The essence of fine-grained parallelism is to multiply the number of DSP in a CU, which is similar to the unrolling operation in High-Level Synthesis. It is achieved given the premise that the operations of the values in *value* and *vector* are separated, just as stated in the previous introduction of a single **eSSpMV**. Therefore, after the fine-grained parallelism is processed, the number of cache access channels is precisely

Algorithm 1: Pipeline based parallelization

```
Initialize CU[n] & Row[m]
for i ⩽ n do
    CU[i] = Row[i];
j = n
while j < length(Row) do
    for i ⩽ n do
        CU[i] = CU[i] - 1;
        if CU[i] == 0 then
            CU[i] = Row[j];
            j = j + 1;
```

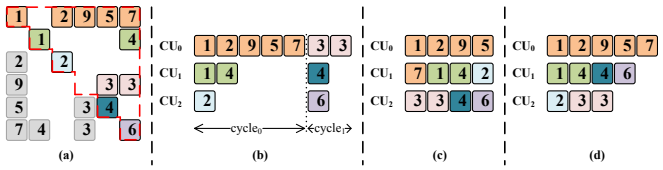

Fig. 2. Different parallelization strategy. (a) A symmetric sparse matrix; (b) Row-based parallelization; (c) Ideal parallelization; (d) Pipeline based parallelization.

the multiple of the DSPs. Considering the large variation in the number of elements between rows, the number of DSPs in one CU is particularly considered to reduce the DSP efficiency.

Coarse-grained parallelism is to duplicate a single CU design into multiple CUs. During the process, two challenges are involved: memory access and the imbalance across rows. For the memory access problem, new advanced FPGAs (*e.g.*, Xilinx Zynq MPSoC) support multiple memory interfaces that enable parallel memory access mechanisms for multiple computational hardware modules [17]. Hence, we divide the rows in a sparse matrix into different regions. As shown in Fig.1(f), each CU has a private *cache1*, and all the CUs share a public *cache2*. This design guarantees a fully-pipelined streaming after the coarse-grained parallelism. The *cache2* is similar to a two-dimensional first-in-first-out (FIFO) with multiple ports, and its largest size would be the number of CUs and the largest dimension for the matrix. When a CU is about to finish the computation, the shared *cache2* is informed and then assigns its top-row data to the CU's private *cache1* and request for a new row of data to be updated at its bottom. In this way, the data access of **eSSpMV** is significantly reduced. The notification from CU to *cache2* is achieved as the *row_length* data in the SCSR can be used as the flag for the computation.

To address the imbalance across rows, our design is shown in Fig.2. Considering the symmetrical sparse matrix in Fig.2(a), where the colored squares represent non-zero elements. With 3 CUs or threads for computing, previous methods shown in Fig.2(b) [18, 19] can hardly avoid the imbalance across rows, resulting in a waste of resources and power consumption. The ideal solution [20, 21] to balance the rows is shown in Fig.2(c). However, it requires analysis and packaging of the whole data and consumes significant computing resources. Therefore, we adopt a pipeline-based method as indicated in Algorithm 1. The key to this method is a subset sum problem according to [22]. As shown in the example of Fig.2(d), n rows of data are assigned to n CUs. After a CU completes its task, it is immediately filled by the $n+1$ row of data for the computation. In this process, the non-zero elements in each row gradually decrease since the matrix is upper-triangular, for which the workload between different CUs is balanced, to a certain extent. It thus significantly alleviates the problem, although cannot entirely solve the imbalance for extreme cases where the number of non-zero elements varies greatly. Moreover, it does not require the pre-processing of the whole matrix at all or a separate design of the dispatch module. An evaluation of the design performance is presented in the next Section.

TABLE I
COMPARISON WITH OTHER SSPMV DESIGNS OF RESOURCE UTILIZATION

	[23] (2020)	**eSSpMV**
FPGA	ZYNQ XCZU9EG	ZYNQ XCZU7EV
Frequency (MHz)	200	250
Data precision (bit)	FP32	FP32
Max. matrix size	50000	50000
LUT	5755	3494
FF	4385	5621
BRAM	N/A	61.5
DSP	1512	1024
GFLOPS	2.5	3.2

IV. EXPERIMENTS

A. Experimental Setup

We deploy the **eSSpMV** with 32-bit data width on Xilinx Zynq XCZU7EV FPGA in the customized boards. For basic comparison, we choose the SOTA design (only one is proposed for edge device) [23], with its target board (Zynq XCZU9EG) in a fast-stream mode that supports sparse symmetric matrix. We also compare our design with typical edge devices, including ARM A53 CPU on Raspberry pi 3B, ARM A72 CPU on RK3399PRO, and NVIDIA GPUs on Jetson Xavier NX. In order to perform parallel processing on the ARM CPU platform, we use the Armadillo library [24] and Tengine [25] to deploy parallel processing of sparse symmetric matrix. On the GPU platform, the CUDA with cuSPARSE library is used to realize parallel processing [26]. The power consumption is measured by PN2000 electricity usage monitor [27, 28]. For evaluation, we use 10 SSpMs with various matrix dimension sizes and different numbers of non-zero elements, which are all from the University of Florida sparse matrix collection [29]. Their dimensions and the numbers of non-zero elements range from 48 to 41,731 and 400 to 1,317,655, respectively.

B. FPGA Deployment Results

We complete the **eSSpMV** deployment in 128 CUs while the number of multipliers (2× DSP48E2) per CU is 4. A comparison of the resource consumption between SOTA and our deployment is shown in Table I, in which our work exhibits a computing resource reduction of 39.3% and 32.3% for LUT

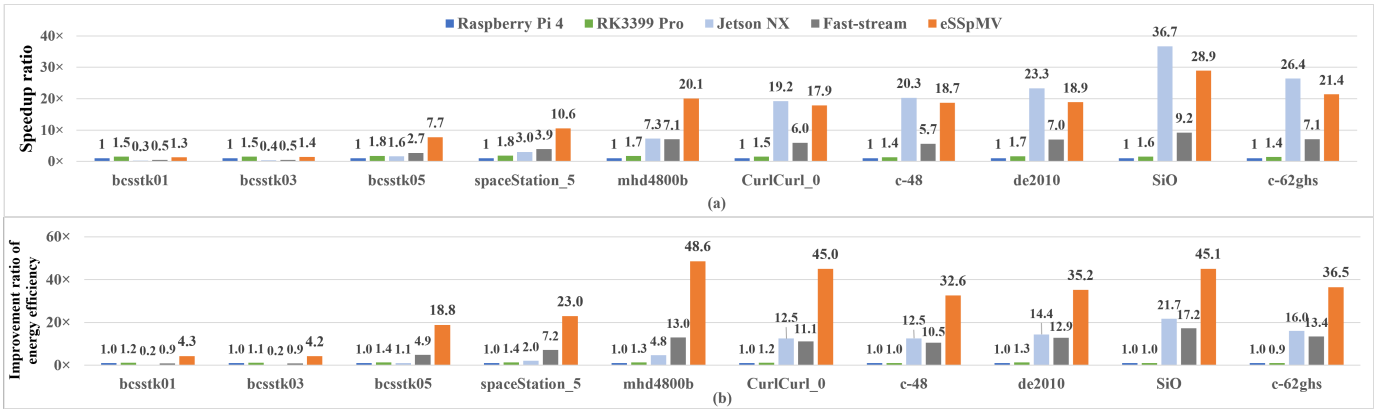

Fig. 3. Comparison with edge platforms. (a) Speedup ratio comparison; (b) Improvement ratio of energy efficiency comparison.

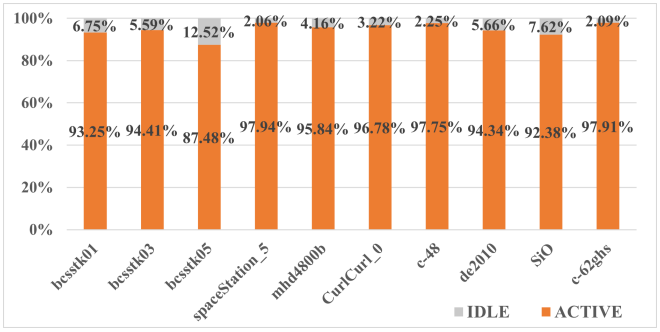

Fig. 4. The DSP efficiency across 10 datasets.

and DSP, respectively. The relatively low resource consumption leaves sufficient design space for complete operational systems (*e.g.*, SVM or Quasi-Newton method), which meets **eSSpMV** design purpose as a basic computation unit.

C. DSP efficiency

The problem of row imbalance exists in these architectures. Hence, it is necessary to discuss the efficiency of DSP to examine the dispatching mechanism of Algorithm 1. The DSP efficiency is calculated as follows:

$$DSP\_EFF = \frac{OP\_DSP\_Cycle\_num}{TEST\_Cycle\_num \times DSP_{num}}, \quad (1)$$

where $OP\_DSP\_Cycle\_num$ means the cycle of DSP operating none zero data. The DSP efficiency of **eSSpMV** in 10 datasets is shown in Fig.4. **eSSpMV** achieves excellent results in the range from 92.38% to 97.94% on almost all datasets, only except the worst case on the dataset of bcsstk05 due to two reasons: 1. bcsstk05 has a great number of non-zero elements, which definitely leads to idle computation cycles; 2. There is a significant variation in the number of non-zero elements across rows. However, we believe that mechanism of our design is still generally effective for resource-limited edge devices. It is also worth mentioning that this mechanism also saves resources as it only depends on the data of *row_length* and does not require analyzing the distribution of non-zero elements in the matrix beforehand.

D. Comparison with Edge Platforms

We compare the speedup ratio shown in Fig. 3(a) with ARM A53 [30] as the baseline. **eSSpMV** performs on average 2.9× faster than fast-stream (unroll set to 4). This is because fast-stream runs at a lower frequency than **eSSpMV** and has a deeper pipeline depth. Also, there are lots of empty data in the actual operation, which means that unroll=4 is not a complete parallel operation. In addition, the fast-stream design is based on CSR format. It fills 0 in specific positions to ensure the operation is correct, which makes several operations redundant. Compared with CPU, **eSSpMV** shows 9.3× speedup on average. For GPU comparison, **eSSpMV** gets an advantage under matrix size below 5, 000. It is also very close to GPU by exploiting parallelism. GPUs benefit from utilizing a large number of hardware threads and coalesced memory access, therefore they gain advantages in huge matrix processing.

We compare the energy efficiency of our design on different platforms in Fig.3(b). The improvement ratio of energy efficiency is defined as follows:

$$IMP\_ENG\_EFF = \frac{PLP_{target}}{PLP_{A53}}, \quad (2)$$

where the PLP means the value of useful power-latency-product [31], denoted as below:

$$PLP = Power\,consumption(mW) \times Latency(ms). \quad (3)$$

Following the above definition, **eSSpVM** shows 23.7×, 13.1× higher energy efficiency compared with edge CPU and edge GPU implementation, respectively. Compared with fast-stream, **eSSpMV** the improvement ratio is 3.0× instead.

V. CONCLUSION

In this paper, we present **eSSpMV**, an eFPGA-based hardware accelerator for symmetrical sparse matrix-vector multiplication. With an improved SCSR format and parallelism features, **eSSpMV** resolves the challenge of inefficient random memory accessing while acquiring up to 97.94% DSP efficiency. **eSSpMV** achieves up to 2.9× speedup than the SOTA with lower power consumption, averagely 9.3× faster than edge CPU devices, and 13.1× more energy-efficient than edge GPU devices. For future work, there is a vast design space in the configuration of the number of DSPs in a single CU and the number of CUs in **eSSpMV**. An analytical model for automatically exploring **eSSpMV** deployment strategies under complicated constraints will be designed.

REFERENCES

[1] J. Dass, Y. Narawane, R. N. Mahapatra, and V. Sarin, "Distributed training of support vector machine on a multiple-FPGA system," *IEEE Transactions on Computers*, vol. 69, no. 7, pp. 1015–1026, 2020.

[2] Z. Qu, L. Deng, B. Wang, H. Chen, J. Lin, L. Liang, G. Li, Z. Zhang, and Y. Xie, "Hardware-enabled efficient data processing with tensor-train decomposition," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 2, pp. 372–385, 2022.

[3] J. Liu and Q. Liu, "Resource reduction of BFGS quasi-newton implementation on FPGA using fixed-point matrix updating," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, 2018, pp. 301–3015.

[4] C. Alappat, A. Basermann, A. R. Bishop, H. Fehske, G. Hager, O. Schenk, J. Thies, and G. Wellein, "A recursive algebraic coloring technique for hardware-efficient symmetric sparse matrix-vector multiplication," *ACM Transactions on Parallel Computing*, vol. 7, no. 3, pp. 19:1–19:37, 2020.

[5] K. Ahmad, H. Sundar, and M. Hall, "Data-driven mixed precision sparse matrix vector multiplication for gpus," *ACM Transactions on Architecture and Code Optimization*, vol. 16, no. 4, pp. 51:1–51:24, 2019.

[6] K. Guo, L. Sui, J. Qiu, J. Yu, J. Wang, S. Yao, S. Han, Y. Wang, and H. Yang, "Angel-eye: A complete design flow for mapping CNN onto embedded FPGA," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 1, pp. 35–47, 2018.

[7] M. Pligouroudis, R. A. G. Nuno, and T. Kazmierski, "Modified compressed sparse row format for accelerated FPGA-based sparse matrix multiplication," in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2020, pp. 1–5.

[8] T. Gkountouvas, V. Karakasis, K. Kourtis, G. Goumas, and N. Koziris, "Improving the performance of the symmetric sparse matrix-vector multiplication in multicore," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS)*, 2013, pp. 273–283.

[9] 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.

[10] A. K. Jain, S. Kumar, A. Tripathi, and D. Gaitonde, "Sparse deep neural network acceleration on HBM-enabled FPGA platform," in *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, 2021, pp. 1–7.

[11] L. Song, Y. Chi, A. Sohrabizadeh, Y.-k. Choi, J. Lau, and J. Cong, "Sextans: A streaming accelerator for general-purpose sparse-matrix dense-matrix multiplication," in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2022, pp. 65–77.

[12] A. Elafrou, G. Goumas, and N. Koziris, "Conflict-free symmetric sparse matrix-vector multiplication on multicore architectures," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2019, pp. 1–15.

[13] U. W. Pooch and A. Nieder, "A survey of indexing techniques for sparse matrices," *ACM Computing Surveys*, vol. 5, no. 2, pp. 109–133, 1973.

[14] E. Chow and Y. Saad, "ILUS: An incomplete LU preconditioner in sparse skyline format," *International Journal for Numerical Methods in Fluids*, vol. 25, no. 7, pp. 739–748, 1997.

[15] A. Parravicini, L. G. Cellamare, M. Siracusa, and M. D. Santambrogio, "Scaling up HBM efficiency of top-K spmv for approximate embedding similarity on fpgas," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, 2021, pp. 799–804.

[16] J. Sun, G. Peterson, and O. Storaasli, "Mapping sparse matrix-vector multiplication on fpgas," in *Reconfigurable Systems Summer Institute - RSSI 2007*, 2007, pp. 1–10.

[17] M. Hosseinabady and J. Nunez-Yanez, "Sparse matrix-dense matrix multiplication on heterogeneous CPU+FPGA embedded system," in *Proceedings of the 11th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures / 9th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms*, 2020, pp. 1–6.

[18] B.-Y. Su and K. Keutzer, "clspmv: A cross-platform opencl spmv framework on gpus," in *Proceedings of the 26th ACM international conference on Supercomputing (ICS)*, 2012, pp. 353–364.

[19] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing (SC)*, 2007, pp. 1–12.

[20] 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 (FCCM)*, 2014, pp. 36–43.

[21] W. Liu and B. Vinter, "CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication," in *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS)*, 2015, pp. 339–350.

[22] G. Shobaki and K. Wilken, "Optimal superblock scheduling using enumeration," in *37th International Symposium on Microarchitecture (MICRO)*, 2004, pp. 283–293.

[23] M. Hosseinabady and J. L. Nunez-Yanez, "A streaming dataflow engine for sparse matrix-vector multiplication using high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 6, pp. 1272–1285, 2020.

[24] C. Sanderson and R. Curtin, "Armadillo: a template-based C++ library for linear algebra," *The Journal of Open Source Software*, vol. 1, no. 2, p. 26, 2016.

[25] H. Lan, J. Meng, C. Hundt, B. Schmidt, M. Deng, X. Wang, W. Liu, Y. Qiao, and S. Feng, "Feathercnn: Fast inference computation with tensorgemm on ARM architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 3, pp. 580–594, 2020.

[26] M. Naumov, "Incomplete-LU and cholesky preconditioned iterative methods using CUSPARSE and CUBLAS."

[27] Y. Yu, T. Zhao, M. Wang, K. Wang, and L. He, "Uni-opu: An fpga-based uniform accelerator for convolutional and transposed convolutional networks," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 7, pp. 1545–1556, 2020.

[28] Y. Yu, C. Wu, T. Zhao, K. Wang, and L. He, "Opu: An fpga-based overlay processor for convolutional neural networks," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 1, pp. 35–47, 2020.

[29] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software*, vol. 38, no. 1, pp. 1:1–1:25, 2011.

[30] Y. Yu, T. Zhao, K. Wang, and L. He, "Light-OPU: An FPGA-based overlay processor for lightweight convolutional neural networks," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2020, pp. 122–132.

[31] Y. Hu, Y. Zhu, H. Chen, R. Graham, and C.-K. Cheng, "Communication latency aware low power noc synthesis," in *Proceedings of the 43rd annual Design Automation Conference (DAC)*, 2006, pp. 574–579.