A Domain-Specific Architecture for Accelerating Sparse Matrix Vector Multiplication on FPGAs

Abhishek Kumar Jain, Hossein Omidian, Henri Fraisse, Mansimran Benipal, Lisa Liu, Dinesh Gaitonde
Xilinx Inc., San Jose, CA, United States
Email: abhishek@xilinx.com

***Abstract*—FPGAs allow custom memory hierarchy and flexible data movement with highly fine-grained control. These capabilities are critical for building high performance and energy efficient domain-specific architectures (DSAs), especially for workloads with irregular memory access and data-dependent communication patterns. Sparse linear algebra operations, especially sparse matrix vector multiplication (SpMV), are examples of such workloads and are becoming important due to their use in numerous areas of science and engineering. Existing FPGA-based DSAs for SpMV do not allow customization through plug and play of the building blocks. For example, most of these DSAs require switching network/crossbar architecture as a building block for routing matrix data to banked vector memory blocks. In this paper, we first present an approach where a custom network is built using simple blocks arranged in a regular fashion to exploit low-level architecture details. Further, we make use of this network to replace expensive crossbars employed in GEMX SpMV engine and develop an end-to-end tool-flow around mixed IP approach (HLS/RTL). Due to the modularity of our design, our tool-flow allows us to insert an additional block in the design to guarantee zero-stall from the accumulation stage. On Alveo U200, we report performance numbers of up to 4.4 GFLOPS (92% peak bandwidth utilization) using our accelerator (attached with one DDR4).**

I. Introduction

Sparse Matrix Vector Multiplication (SpMV) refers to the multiplication of a sparse matrix A by a dense vector x to produce a result vector b. There are many application domains including sparse neural nets [1]–[3], graph analytics [4], physics simulations [5] where sparse computations, especially SpMV is a key component of the application. Acceleration of SpMV is thus becoming increasingly important [6]–[10]. Despite having significant parallelism, SpMV is challenging to optimize due to irregular memory access patterns and low memory-to-computation ratio. For real world sparse matrices, traditional processor architectures fail to effectively utilize the compute resources and exhibit poor energy efficiency [11].

Domain specific architectures (DSAs) for sparse linear algebra are emerging as a solution since these accelerators have the capability to boost performance and energy efficiency by customizing memory hierarchy, communication and compute logic to suit the needs of application [12]–[17]. FPGAs, which allow the accelerator to be modified post-deployment, are now commonly used for rapid prototyping and customization of DSAs [18], [19].

Existing FPGA based DSAs for SpMV have focused mostly on one or more of the following features: (a) handling arbitrary size matrix and vectors by proposing new blocking strategies [20], [21], (b) simplified data movement by proposing new encoding of non-zeros [22], (c) efficient caching and maximal reuse of data (especially input/output vectors) to avoid redundant memory accesses [22], (d) high frequency compute pipeline to process non-zeros of the matrix in a streaming fashion [21], (e) use of high-level synthesis (HLS) for hardware generation [20], [23], (f) efficient utilization of memory bandwidth [18], (g) avoiding unnecessary replication of data (mainly vector data) [20], [22], (h) minimizing stalls at memory interface and within compute pipeline by proposing special techniques [24].

Most of these DSAs do not provide a flow to allow customization through plug and play of the building blocks. Also, all building blocks within a DSA are described either as RTL sources or as HLS sources, but not a mix of both. Therefore, the design either provides high performance or flexibility, but not both at the same time. In this paper, we develop an end-to-end tool-flow around mixed IP approach (HLS/RTL) to introduce modularity and customization opportunities in the design. Our approach allows one to compose a DSA for SpMV using modular building blocks. Developers can customize the DSA by adding, updating or deleting blocks from the design in order to suit the needs of the application. Our approach of composing accelerator using modular building blocks is similar to GraphOps dataflow library [25]. Although GraphOps relies on HLS to generate the building blocks while we use a hybrid approach of mixing HLS with RTL for introducing the FPGA awareness in the implementation. For example, some of the building blocks in a design can use hard FPGA primitives more efficiently when implemented in RTL than in HLS [26].

In order to demonstrate the capability of our approach, we use our tool-flow to compose an efficient DSA for SpMV (targeting Xilinx Alveo boards) using a number of IP blocks (standard RTL based IPs; HLS generated IPs for vector caching, multiplications, reductions and accumulations; and a parameterizable 2D-mesh NoC overlay). Due to the modularity of our design, our tool-flow allowed us to insert an additional block in the design to guarantee zero-stall from the accumulation stage. We demonstrate that by focusing on minimizing the stalls and utilizing efficient switching networks, our SpMV DSA can provide competitive performance, close to peak GFLOPS and memory bandwidth utilization.

The remainder of the paper is organized as follows: Section II provides an analysis of some of the existing SpMV DSAs from the research literature. Section III describes the proposed approach of composing SpMV DSAs and an example DSA built around modular IP blocks. Section IV provides a comparison of SpMV DSAs from the research literature with proposed DSA. Finally, Section V concludes the paper.

	Year	SpMV design	Platform (FPGA)	Off-chip Memory	BW (GB/s)	Arithmetic	Theoretical perf. (GFLOPS)	Frequency	Achieved perf. (GFLOPS) (% BW utilization)
1	2005	[27]	Nallatech BenDATA (Virtex-II)	6 ZBT SRAM	8	FP64	1.28	160 MHz	0.35 (40%)
2	2011	[28]	Convey HC-1 (Virtex-5)	16 DDR2-667	80	FP64	12.8	150 MHz	4 (30%)
3	2012	[29]	BEE3 (Virtex-5)	2 DDR2-400	6.4	FP64	1	100 MHz	0.2 (20%)
4	2013	[30] R^3	Convey HC-1 (Virtex-5)	16 DDR2-667	80	FP64	12.8	150 MHz	13.6 (compress.)
5	2014	[22]	DE5 (Stratix V)	2 DDR3-1333	21.3	FP32	4.8	150 MHz	2.4 (50%)
6	2016	[31]	Convey HC-2 (Virtex-5)	16 DDR2-667	80	FP64	12.8	150 MHz	6.5 (50%)
7	2016	[32] CASK	Maxeler Vectis (Virtex-6)	NA	40	FP64	6.4	100 MHz	∼sparsity
8	2018	[20] Xilinx GEMX	Alveo U200 (VU+)	1 DDR4-2400	20	FP32	4.8	250 MHz	1.9 (40%)
9	2019	[21] HitGraph	VU+ Board	4 DDR3-1600	60	FP32	10	200 MHz	6.4 (64%)

TABLE I: Existing FPGA-based SpMV Accelerators

II. Existing FPGA based SpMV Accelerators

Since SpMV is memory-bound and different hardware platforms have different memory bandwidth, peak performance alone is not sufficient to capture the efficiency. A more important metric than peak performance is the fraction of memory bandwidth utilized, which captures the overall efficiency of the architecture. Table I summarizes the key metrics such as the available bandwidth on the platform, the peak theoretical performance in GFLOPS limited by available memory bandwidth, the achieved best performance in GFLOPS, the % bandwidth utilization and the accelerator frequency in previous work targeting SpMV on FPGAs. We restrict our review to previous work which directly interfaces to memory and reports performance numbers on hardware for real matrices.

As shown in the Table I, platforms (BEE3, DE5 and NallaTech BenDATA) used in early work exhibit low performance due to limited available bandwidth [22], [27], [29]. For double-precision (FP64) CVBV SpMV with 4 Byte indexing overhead, BEE3 can only provide **6.4GB/s / (12B/2FLOPs)** = ≈1 GFLOPS. Although when processing sparse matrices using the accelerator, the best reported performance number is around 0.2 GFLOPS (20% peak bandwidth utilization). Many research groups have used Convey machines (HC-1 and HC-2) to implement SpMV because of high memory bandwidth (80 GB/s) available on the platform [28], [30], [31]. For double-precision (FP64) CSR SpMV with 4 Byte indexing overhead, Convey platform can provide a peak performance of **80GB/s / (12B/2FLOPs)** = ≈12.8 GFLOPS. Although when processing sparse matrices, the best reported performance number in [28] and [31] is around 4 GFLOPS (30% peak bandwidth utilization) and 6.5 GFLOPS (50% peak bandwidth utilization), respectively. In [30], the authors apply compression on the data to better utilize the available memory bandwidth and demonstrate an achievable performance of ≈13.6 GFLOPS.

Apart from poor utilization of bandwidth, another major concern was the limit on exploitable parallelism in early proposals. For example in [27], the authors focused mostly on exploiting the parallelism within a single row. Assuming 32 multipliers in the design, for all of the rows having less than 32 non-zeros, the remaining multipliers would be wasted due to zero padding. In [22], the authors developed a new encoding (CISR) and used the Terasic DE5 platform to implement an accelerator capable of processing non-zeros from multiple rows to maximize parallelism. Also, while many existing efforts relied on maintaining a separate copy of the entire vector *x* for every multiplier, the authors removed that need by proposing the concept of Banked Vector Buffer (BVB).

For constructing BVB in [22], the vector *x* is partitioned into 32 banks of memory blocks and a 32×32 crossbar is used for routing incoming non-zeros to their corresponding vector banks based on the column indexes. More information about BVB is available in [22]. On a bank-conflict, requests are back-pressured into the crossbar's input queues, and eventually back to the channel. It was observed that these conflicts contribute to as much as 30% of the total stalls in the worst case. For single-precision (FP32) CISR SpMV with 4 Byte indexing overhead, DE5 platform can provide a peak performance of **21.3GB/s / (8B/2FLOPs)** = ≈4.8 GFLOPS. Although when processing sparse matrices using the accelerator, the best reported performance number is around 2.4 GFLOPS (50% peak bandwidth utilization).

The open source Xilinx GEMX SpMV engine [20] targets Alveo boards and is implemented as an HLS application. For FP32 arithmetic case in GEMX SpMV, all of the non-zeros values and vector elements are 4 Bytes while row/col indexes are 2 Bytes each. Hence, each non-zero is encoded with 8 Bytes. During execution, kernel requests input vector elements from DDR and fills the input BVB. Then, 8 non-zeros every cycle get streamed into the kernel from DDR (64 Byte interface). A crossbar (xBarCol) of size 8×8 is then used for routing the non-zeros to their corresponding vector banks. After reaching the correct bank, the column index from the non-zero is used to read the vector entry which gets multiplied with the non-zero value. The results of multiplication are then routed by second crossbar (xBarRow) to their corresponding accumulators. After all the accumulations are done, the kernel stores the results back to DDR.

Switching networks coupled with banked vector buffers can allow very high throughput irregular indexing by keeping the vector elements on-chip. But the complexity of generally used crossbar networks and their inefficient mapping on FPGA fabrics [20], [22] is one of the factor which limits the performance of SpMV accelerators. We present an approach where we replace the crossbar with a 2D-mesh NoC which is built using simple blocks arranged in a regular fashion that exploit low-level architecture details and achieve high frequency implementation. By doing that, we avoid the switching network to become the performance bottleneck. Although the concept of NoC overlaid on FPGAs is not new, we demonstrate in this paper that its applicability to SpMV and similar applications is of great relevance. Further, we make use of FPGA-friendly NoC to replace expensive crossbars employed in GEMX SpMV engine and develop an end-to-end tool-flow around mixed IP approach (HLS/RTL).

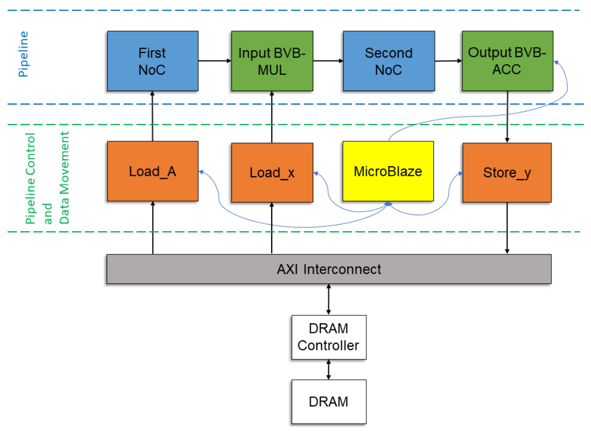

Fig. 1: SpMV Kernel attached with DRAM.

III. Proposed Approach of Composing SpMV DSA using Modular IP Blocks

Rapid customization through composition and modularity remains a major concern preventing the mainstream use of FPGAs for sparse computations. Our approach uses modular building blocks (HLS generated IPs and RTL based IPs with standard AXI interfaces) to compose a DSA for SpMV. Our design choices are guided to enable the following main features: efficient utilization of memory bandwidth by minimizing stalls at memory interface and within compute pipeline by proposing high performance routing networks and zero stall accumulation logic. In the following sections, we first explain the high-level architecture of the SpMV kernel, and then the customization for zero-stall accumulations.

A. High-level Architecture

The design of the SpMV kernel is very similar to GEMX SpMV engine except that we use modular IP blocks (designed/synthesized separately) and stitch them together in Vivado IP integrator via latency-insensitive channels (AXI-streams). The instantiation of IP blocks and stitching is automated using TCL scripts. Fig. 1 shows the high level block diagram of our approach where the kernel is attached to the external memory and the data movement is controlled by a soft-processor (Microblaze) and load-store units (AXI datamovers). The interfaces between AXI interconnect and load-store units are full AXI while the rest of them are AXI-stream channels. In the following sections, we first explain the IP blocks responsible for data movement and then the IP blocks used for the composition of the compute pipeline.

1) Load-store units and control processor: Control of data-movement is critical for the flexibility of our design approach. In order to keep the design highly flexible and customizable, we resort to programmable building blocks including a soft-processor (Microblaze) and load-store units (AXI datamovers). Microblaze acts as a control processor which manages the overall data movement by sending commands to load-store units and to the compute pipeline. It uses stream channels for sending information (Address and Size) to load-store units through a set of commands. It then waits for the completion and moves to the next one. Here is an overview of the control flow:

- $Load\_x(Addr\_x, Size\_x)$: Bring vector x from external memory and load it into input BVB
- $Load\_A(Addr\_A, Size\_A)$: Bring matrix A from external memory and stream it through pipeline. During this phase, matrix non-zeros get routed through first NoC, non-zero values get multiplied with their corresponding vector entries, multiplication results get routed through second NoC and reaches output BVB for accumulations.
- $Store\_y(Addr\_y, Size\_y)$: Microblaze sends store command (including address and size) to the store unit which then waits for y to arrive at the input queue.

Finally, Microblaze sends a token to the output BVB to initialize the drain process. Once the store unit starts receiving y at the input queue, it sends y into external memory.

2) Compute Pipeline: Fig. 1 shows the composition of compute pipeline which includes high-speed NoCs and BVB blocks. We developed a customized RTL IP for the NoC and use HLS-generated IPs for the BVB-MUL and BVB-ACC blocks. Fig. 2 shows the high level diagram of our 2D-mesh NoC including the architecture of each switch. For routing, we resort to a simple XY algorithm where each non-zero is routed horizontally first and then vertically. Within each switch, the column index of the non-zero is checked and if it does not belong to the corresponding bank, it is routed further horizontally otherwise, it is moved downwards. The Split unit (S) within each switch performs the above mentioned operation and the merge unit (M) arbitrates between the data coming from the vertical direction and the data coming from the split unit.

Instead of using a buffer-less NoC architecture and letting the packets deflect [33], we choose to develop a buffered NoC architecture built around latency-insensitive dataflow units. These include elastic buffers (EBs) [34], [35], 2-way split units and 2-way merge units. The EBs in our NoC architecture make use of a specific implementation, full bandwidth 2-slot EB [36], to avoid stalls and pipeline bubbles.

Since an arbitrary number of EBs can be placed within a switch, we choose to minimize the EB overhead (number of EBs in a switch and their depth) which affects the resource requirements of the NoC. We observe that placing only three EBs in a switch (one at the input of S, one between S and M, and the last one just at the output of M) is sufficient

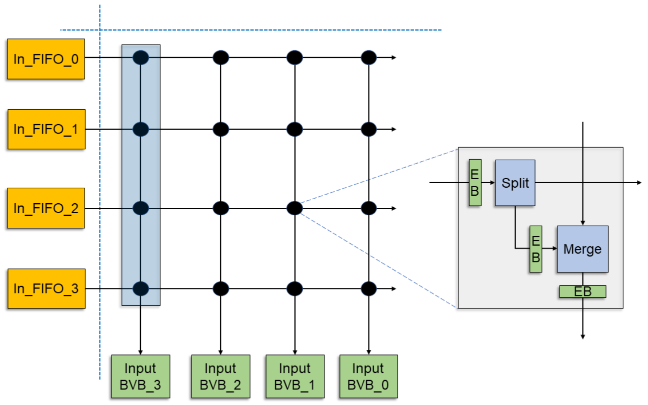

Fig. 2: 2D-mesh NoC.

to minimize stalls. Also, we determine the suitable depth of EBs by modelling the NoC in SystemC and then performing simulations for 250 matrices under different settings. We found that the EB depth in the range of $2 - 4$ is sufficient enough to minimize the stalls for most of the matrices.

Microarchitectural features of the NoC include the size of the NoC (N), width of datapath (*DW* bits), number of EBs per switch (*NUM_EB*) and depth of each EB (*EB_depth*). These features can be sized up and down based on the availability of the resources. The FF requirements can be calculated from the equation below:

$$FF\_required = N^2 * DW * NUM\_EB * EB\_depth \quad (1)$$

The RTL of the switch (three 64b EBs per switch where EB_Depth = 4) is able to meet timing constraint of 1.35 ns (740 MHz) while only consuming around 50 CLBs (300 LUTs and 800 FFs). Due to the regularity of NoC architecture, scaling does not affect the implementation frequency. We explore the efficiency gap between the proposed NoC and the crossbar network in GEMX SpMV Engine by scaling the size. Fig. 3 shows the drop in frequency for the crossbar network while the frequency of our proposed NoC remains immune to the scale. We also observe that both of the networks consume an entire SLR on Alveo U200 when scaled to a size of 32×32.

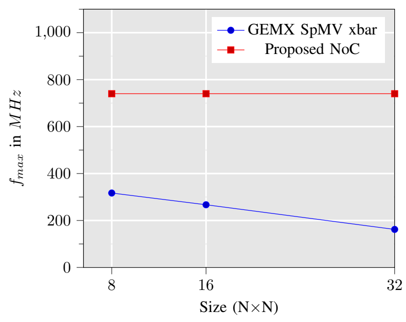

Fig. 3: Comparison of F_{max} between GEMX SpMV xbar and Proposed NoC.

For input BVBs and multiplications (input BVB-MUL), output BVBs and accumulations (output BVB-ACC), we choose to describe the hardware in C/C++ and used Vivado HLS to synthesize the IP blocks. The simplicity of HLS code allows customization opportunities. For example, the size of BVB can be selected based on the matrix dimensions. HLS pragma is used to specify that the URAM should be used to hold vector entries. This allows URAM cascading for large BVB sizes.

B. Customization of pipeline for Zero-stall accumulations

Once the non-zeros get routed to the input vector banks using the first NoC and multiplied with the corresponding vector entries, the results of the multiplications has to be routed to the output vector banks using a second NoC for accumulation purpose. A major challenge in achieving high performance comes from the need to accumulate values that are delivered in consecutive clock cycles into a deeply-pipelined FP32 adder. This is because subsequent additions on incoming data can not be performed until the previous addition has been completed.

Since FP32 adders have certain latency L (4-8 clock cycles) to run at high frequencies, it could lead to data hazard where the value from incoming row index is read before the result of previous accumulation is written back to that row index. It happens when incoming data packets have same indexes within a time window of L clock cycles.

In order to avoid these hazards, one possibility is to stall the incoming stream for L cycles after each accumulation. This results in poor performance of SpMV pipeline since the peak throughput now gets reduced by L times. In order to solve this problem and enable zero-stall accumulations, we design special reduction trees, referred to as Hazard-resolving Reduction Tree (HRT). HRT looks at a window of L cycles and add all of the multiplied results belonging to same indexes. We implemented these HRTs in C++ and used Vivado HLS to synthesize them as IP blocks. Our design requires one HRT at the input of each accumulator so we take the design from Fig. 1 and insert the IP blocks just between the second NoC and the output BVB-ACC.

IV. Experiments and Results

We implemented and benchmarked the proposed SpMV DSA on Xilinx Alveo boards (U200/U250) using Vivado and Vivado HLS 2019.1 version. Alveo cards support up to 64 GB of DRAM, with four DDR4-2400 supporting an aggregate peak off-chip bandwidth of 77 GB/s. The FPGA communicates with the host system using PCIe Gen 3×16. We assume the input is a COO-encoded sparse matrix, with single precision floating point values and 16-bit indexes (column and row indexes). Although it is not a limitation, for our experiments, we restrict the settings of example DSA to 16-bit indexes. Hence in order to apply SpMV on a large matrix (size of more than 64K×64K), one can either change the DSA settings (index sizes and vector buffer sizes) at design time or a tiling strategy can be used.

Our tool flow allows one to choose the target Alveo board (U200/U250), the number of kernels to map and the scale of the kernel. For experiment purpose, we use a design with 1 kernel on Alveo U200 and observe how much bandwidth can be utilized from just a single DDR4 (peak = 19.2 GB/s). Since one DDR allow to bring 8 non-zeros every clock cycle, we specify scale as 8 and the tool then uses 8×8 NoCs and rest of the blocks accordingly (8 input/output BVBs and 8 HRTs). The design is able to meet given timing constraints of 300 MHz while other existing SpMV designs from Table I run at lower frequencies (100-250 MHz). For our design on U200, SpMV kernel uses 165K FFs (7%), 60K LUTs (5%), 200 DSP (3%), 32 BRAM (1.5%), 24 URAM (2.5%).

We evaluate the performance of our proposed DSA using sparse matrices from the University of Florida Sparse Matrix collection. In our experiments, the COO-encoded matrix is first loaded into FPGA's DRAM by the host processor via PCIe interface. We assume that SpMV is executed iteratively on FPGA where matrix A gets reused across iterations. This is common in many use-cases of SpMV such as Pagerank [37] and HPCG [38].

In order to provide accurate measurements, we instantiate AXI timer IP within the design to measure accurate cycle counts. We start the timer just before loading vector x and stop it just after storing the result vector y. We report GFLOPS

				BEE3 [29]	HC-1 [28]	Tesla S1070 [28]	CASK [32]	GEMX SpMV [20]	This work
DDR BW (GB/s)				6.4 GB/s	80 GB/s	100 GB/s	40 GB/s	20 GB/s	20 GB/s
BW limited GFLOPS				1 GFLOPS	12.8 GFLOPS	16 GFLOPS	6.4 GFLOPS	4.8 GFLOPS	4.8 GFLOPS
Matrix	R×C	NNZ	Sparsity	GFLOPS / % Peak Bandwidth Used					
dw8192	8192×8192	41746	99.94%	0.10 / 10%	1.7 / 13%	0.5 / 3%	0.7 / 10%	1.4 / 30%	**1.9** / **40%**
t2d_q9	9801×9801	87025	99.91%	0.15 / 14%	2.5 / 19%	0.9 / 6%	0.9 / 14%	1.6 / 34%	**4.0** / **85%**
epb1	14734×14734	95053	99.96%	0.17 / 17%	2.6 / 20%	0.8 / 5%	0.7 / 10%	1.5 / 32%	**2.8** / **60%**
raefsky1	3242×3242	294276	97.20%	0.20 / 18%	3.9 / 29%	2.6 / 15%	**4.0** / 60%	1.9 / 40%	3.8 / **80%**
psmigr_2	3140×3140	540022	94.52%	0.20 / 18%	3.9 / 29%	2.8 / 17%	**4.8** / 75%	1.3 / 28%	3.8 / **80%**

TABLE II: Quantitative comparison of different SpMV accelerators with ours.

numbers and % Peak Bandwidth Used and compare our results with the results of other SpMV accelerators. It is difficult to perform an accurate comparison since DRAM bandwidth, number of FPGAs, type of FPGAs and arithmetic precision differ. Because SpMV is memory-bound, a more important metric than peak performance alone is the fraction of memory bandwidth utilized, which captures the overall efficiency of the architecture. Table II shows quantitative comparison of our SpMV accelerator (1 kernel attached with 1 DDR4) with others with a focus on memory bandwidth utilization. As shown in Table II, bandwidth utlization using the proposed approach is consistently better than the others. For the benchmark set of matrices, proposed accelerator shows the bandwidth utilization ranging between $40-85\%$ while the bandwidth utilization is $10-18\%$ for BEE3 [29], $13-29\%$ for HC-1 [27], $10-75\%$ for CASK [32] and $28-40\%$ for GEMX SpMV engine [20].

Improved utilization is mainly due to minimizing the stalls in the design. For example, Load_A datamover is efficiently supplying read requests to memory without getting many stalls from pipeline. The reason is that HRT within the pipeline guarantees zero-stall from accumulation stage and 2D-mesh NoC exhibits minimal stalls because back-pressure due to bank conflicts are getting absorbed in distributed EBs. Also, since we are using COO-encoded matrix and storing non-zeros in random order, there are relatively fewer bank conflicts compared to row-major/column-major traversal. If we sort the matrix by column, the bank conflicts would be at input-BVB and if we sort the matrix by row, the bank conflicts would be at output-BVB. COO-encoded format allowed us to store the matrix in random order resulting in minimal bank conflicts.

Since CASK framework generates matrix specific architecture and allows aggressive replication of input vector (to allow parallel indexing with no-stalls) for small dimension matrices, it shows high performance and utilization for *raefsky1* and *psmigr_2* compared to other matrices. *raefsky1* and *psmigr_2*, both are relatively dense and have less memory requirements to hold input and output vector on-chip. Our SpMV kernel uses a generic approach where a single architecture is used for all the matrices in the benchmark set. Our kernel settings allows us to hold up to 64K vector entries at a time. For larger matrices (with more than 64K rows / columns), the kernel settings can be changed at design time. For example, to handle a matrix of size 1M×1M, we can use an index size of 20 bits. Vivado HLS would then synthesize the BVB accordingly to have more URAMs (cascaded) for every multiplier.

Apart from the benchmark set, we have used several other matrices from the University of Florida Sparse Matrix collection. We observe close to 90% memory bandwidth utilization for most of these matrices. While running a well known problem of FEM Cantilever (*cant* matrix, 62451×62451, 2034917 non-zeros, 99.94% sparsity) on SpMV DSA (1 kernel attached with 1 DDR), we observe a performance of 4.4 GFLOPS which corresponds to 92% peak bandwidth utilization. We also used Alveo U250 where 4 kernels are attached with 4 DDR. By partitioning the matrix in 4 equal partitions and using 4 kernels to process them, we observe quadruple performance (17.6 GFLOPS). We run the same matrix on HBM-enabled P6000 GPU using cuSPARSE (CSR) library and observe a performance of 28.4 GFLOPS which corresponds to 40% peak bandwidth utilization. It shows that our proposed SpMV DSA (4 kernel - 4 DDR on U250) performs within about 60% of GPU performance for a given matrix, even though it has 5.6× lower memory bandwidth. Alveo U250 has 77 GB/s and P6000 GPU has 433 GB/s memory bandwidth. We expect the performance of our approach to scale linearly when instantiating multiple SpMV kernels on HBM-enabled FPGA platforms (Alveo U280/U50). We leave the validation of performance scaling on HBM-enabled FPGA platforms as future work.

V. Conclusions and Future Work

We have proposed an approach to compose a DSA for SpMV using modular building blocks where developers can customize the DSA by adding/updating or deleting blocks from the design in order to suit the need of application and/or design constraints. In addition, we have demonstrated the practicality of this approach by composing a DSA for SpMV using a number of IP blocks. Further, we have showed that this DSA has competitive performances by achieving high frequency and delivering close to peak GFLOPS and memory bandwidth utilization when implemented on Xilinx Alveo boards (U200/U250). On Alveo U200, we report performance numbers of up to 4.4 GFLOPS (92% peak bandwidth utilization) using proposed SpMV kernel (attached with one DDR4) for a set of matrices. In the future, we plan to extend our work to High Bandwidth Memory (HBM) platforms such as U280/U50 Alveo boards where the available bandwidth is 6× higher than on U200/U250 boards. Also, we plan to run the SpMV pipeline at more than 300 MHz by separating clock domains. These extensions would allow us to process around 128 non-zeros every cycle at approximately 450 MHz, resulting in a peak performance of 57.6 billion non-zeros/second (115 GFLOPS). We also plan to evaluate the power consumption to show that FPGAs can allow deployment of energy efficient and fully customizable DSAs for sparse computations.

References

[1] B. Liu, M. Wang, H. Foroosh, M. Tappen, and M. Pensky, "Sparse convolutional neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 806–814.

[2] S. Narang, E. Elsen, G. Diamos, and S. Sengupta, "Exploring sparsity in recurrent neural networks," *arXiv preprint arXiv:1704.05119*, 2017.

[3] A. K. Mishra, E. Nurvitadhi, G. Venkatesh, J. Pearce, and D. Marr, "Fine-grained accelerators for sparse machine learning workloads," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2017, pp. 635–640.

[4] D. Buono, J. A. Gunnels, X. Que, F. Checconi, F. Petrini, T.-C. Tuan, and C. Long, "Optimizing sparse linear algebra for large-scale graph analytics," *Computer*, vol. 48, no. 8, pp. 26–34, 2015.

[5] X. Álvarez, A. Gorobets, and F. X. Trias, "Strategies for the heterogeneous execution of large-scale simulations on hybrid supercomputers," in *Proceedings of the European Conference on Computational Fluid Dynamics*, 2018.

[6] P. Grigoraş, P. Burovskiy, W. Luk, and S. Sherwin, "Optimising sparse matrix vector multiplication for large scale fem problems on fpga," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2016, pp. 1–9.

[7] S. Williams, N. Bell, J. W. Choi, M. Garland, L. Oliker, and R. Vuduc, "Sparse matrix-vector multiplication on multicore and accelerators," *Scientific Computing with Multicore and Accelerators*, pp. 83–109, 2010.

[8] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarathy, and P. Sadayappan, "Fast sparse matrix-vector multiplication on gpus for graph applications," in *Proceedings of the international conference for high performance computing, networking, storage and analysis*. IEEE Press, 2014, pp. 781–792.

[9] D. Buono, F. Artico, F. Checconi, J. W. Choi, X. Que, and L. Schneidenbach, "Data analytics with nvlink: An spmv case study," in *Proceedings of the Computing Frontiers Conference*. ACM, 2017, pp. 89–96.

[10] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris, "Understanding the performance of sparse matrix-vector multiplication," in *Proceedings of the Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE, 2008, pp. 283–292.

[11] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk, "Energy efficient architecture for graph analytics accelerators," in *Proceedings of the ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2016, pp. 166–177.

[12] U. Gupta, B. Reagen, L. Pentecost, M. Donato, T. Tambe, A. M. Rush, G.-Y. Wei, and D. Brooks, "Masr: A modular accelerator for sparse rnns," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2019, pp. 1–14.

[13] K. Hegde, H. Asghari-Moghaddam, M. Pellauer, N. Crago, A. Jaleel, E. Solomonik, J. Emer, and C. W. Fletcher, "Extensor: An accelerator for sparse tensor algebra," in *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*. ACM, 2019, pp. 319–333.

[14] K. Kanellopoulos, N. Vijaykumar, C. Giannoula, R. Azizi, S. Koppula, N. M. Ghiasi, T. Shahroodi, J. G. Luna, and O. Mutlu, "Smash: Co-designing software compression and hardware-accelerated indexing for efficient sparse matrix operations," in *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2019, pp. 600–614.

[15] W. S. Song, V. Gleyzer, A. Lomakin, and J. Kepner, "Novel graph processor architecture, prototype system, and results," in *Proceedings of the IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2016, pp. 1–7.

[16] R. Dorrance and D. Markovic, "A 190gflops/w dsp for energy-efficient sparse-blas in embedded iot," in *Proceedings of the IEEE Symposium on VLSI Circuits (VLSI-Circuits)*. IEEE, 2016, pp. 1–2.

[17] R. Dorrance, F. Ren, and D. Marković, "A scalable sparse matrix-vector multiplication kernel for energy-efficient sparse-blas on fpgas," in *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*. ACM, 2014, pp. 161–170.

[18] Y. Umuroglu, "Accelerating sparse linear algebra and deep neural networks on reconfigurable platforms," 2018.

[19] J. Fowers, K. Ovtcharov, M. K. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi *et al.*, "Inside project brainwave's cloud-scale, real-time ai processor," *IEEE Micro*, vol. 39, no. 3, pp. 20–28, 2019.

[20] Xilinx gemx. [Online]. Available: https://github.com/Xilinx/gemx

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

[22] J. Fowers, K. Ovtcharov, K. Strauss, E. S. Chung, and G. Stitt, "A high memory bandwidth fpga accelerator for sparse matrix-vector multiplication," in *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*. IEEE, 2014, pp. 36–43.

[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*, 2019.

[24] S. Sun, M. Monga, P. H. Jones, and J. Zambreno, "An i/o bandwidth-sensitive sparse matrix-vector multiplication engine on fpgas," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 59, no. 1, pp. 113–123, 2011.

[25] T. Oguntebi and K. Olukotun, "Graphops: A dataflow library for graph analytics acceleration," in *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*. ACM, 2016, pp. 111–117.

[26] A. K. Jain, S. A. Fahmy, and D. L. Maskell, "Efficient overlay architecture based on dsp blocks," in *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*. IEEE, 2015, pp. 25–28.

[27] L. Zhuo and V. K. Prasanna, "Sparse matrix-vector multiplication on fpgas," in *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*. ACM, 2005, pp. 63–74.

[28] K. K. Nagar and J. D. Bakos, "A sparse matrix personality for the convey hc-1," in *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*. IEEE, 2011, pp. 1–8.

[29] S. Kestur, J. D. Davis, and E. S. Chung, "Towards a universal fpga matrix-vector multiplication architecture," in *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*. IEEE, 2012, pp. 9–16.

[30] K. Townsend and J. Zambreno, "Reduce, reuse, recycle (r 3): A design methodology for sparse matrix vector multiplication on reconfigurable platforms," in *Proceedings of the International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*. IEEE, 2013, pp. 185–191.

[31] S. Li, Y. Wang, W. Wen, Y. Wang, Y. Chen, and H. Li, "A data locality-aware design framework for reconfigurable sparse matrix-vector multiplication kernel," in *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2016, pp. 1–6.

[32] P. Grigoras, P. Burovskiy, and W. Luk, "Cask: Open-source custom architectures for sparse kernels," in *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*. ACM, 2016, pp. 179–184.

[33] N. Kapre and J. Gray, "Hoplite: A deflection-routed directional torus noc for fpgas," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 10, no. 2, p. 14, 2017.

[34] G. Michelogiannakis and W. J. Dally, "Elastic buffer flow control for on-chip networks," *IEEE Transactions on computers*, vol. 62, no. 2, pp. 295–309, 2011.

[35] I. Seitanidis, A. Psarras, G. Dimitrakopoulos, and C. Nicopoulos, "Elastistore: An elastic buffer architecture for network-on-chip routers," in *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*. IEEE, 2014, pp. 1–6.

[36] G. Dimitrakopoulos, A. Psarras, and I. Seitanidis, "Link-level flow control and buffering," in *Microarchitecture of Network-on-Chip Routers*. Springer, 2015, pp. 11–35.

[37] F. Sadi, J. Sweeney, S. McMillan, T. M. Low, J. C. Hoe, L. Pileggi, and F. Franchetti, "Pagerank acceleration for large graphs with scalable hardware and two-step spmv," in *Proceedings of the IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 2018, pp. 1–7.

[38] J. Dongarra, M. A. Heroux, and P. Luszczek, "High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems," *The International Journal of High Performance Computing Applications*, vol. 30, no. 1, pp. 3–10, 2016.