Modular and Lean Architecture with Elasticity for Sparse Matrix Vector Multiplication on FPGAs

Abhishek Kumar Jain, Chirag Ravishankar, Hossein Omidian,
Sharan Kumar, Maithilee Kulkarni, Aashish Tripathi, Dinesh Gaitonde
AMD, San Jose, CA, United States
Email: abhishek.kumar.jain@amd.com

Abstract*—The use of domain-specific accelerators is becoming prominent for a variety of emerging domains such as graph analytics and HPC, where most of the computations revolve around Sparse Matrix-Vector (SpMV) Multiplication. Many of the existing SpMV accelerators do not scale well on FPGA fabric and exhibit significant performance and area overheads [1], [2], [3]. With the increased external memory bandwidths supported by FPGA platforms, SpMV accelerator design sizes are growing rapidly and exhibit timing closure challenges in physical implementation [4], [5]. To utilize all the High Bandwidth Memory (HBM) channels on the FPGA device, accelerator designers rely on the reuse and replication of the processing elements (PEs). As the number of PEs in a design grows, the achieved frequency of these large designs is often much lower than a single PE design [4], [5]. In this paper, we present a modular and lean architecture for the SpMV workload enabling elastic communication between building blocks. The proposed SpMV accelerator uses single-precision floating-point arithmetic (FP32) and achieves a frequency of 465 MHz for single-instance implementation. The lean nature of the design enables the scaling of the accelerator to sixteen instances, which utilizes all of the 32 HBM pseudo-channels available on the Alveo U280 FPGA platform. The accelerator design with sixteen SpMV instances, spanning multiple FPGA dies, can close timing at 310 MHz which is 80% higher than GraphLily [4] and 40% higher than HiSparse [5]. We demonstrate up to 50 GFLOPS performance on the Alveo U280 FPGA Platform which is 2.5× of GraphLily [4].

I. Introduction

Domain-specific Accelerators [6] and FPGA Overlays [7], [8] for machine learning (ML) are becoming mainstream because of high energy efficiency and performance. These perform dense linear algebra efficiently by minimizing data movement, exploiting high data reuse, regular memory access pattern and temporal and spatial locality [6], [7], [8], [9]. The use of domain-specific accelerators is becoming prominent for other domains as well (e.g., graph analytics and HPC) where most of the computations revolve around SpMV [10], [11].

FPGA-based SpMV Accelerators are emerging as a promising 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 the application [12], [4], [5]. SpMV accelerator design sizes are also growing rapidly with the increased external memory bandwidths supported by modern FPGA platforms, especially since the introduction of HBM-enabled FPGA devices [13]. The Alveo U280 platform provides 460 GBps HBM bandwidth. To effectively utilize this much bandwidth on FPGA devices, accelerator designers rely on the reuse and replication of the processing elements [4], [5].

Many of the existing SpMV accelerators do not scale well on HBM-enabled FPGAs and exhibit significant performance and area overheads [4], [14], [5], [3]. While scaling the existing SpMV accelerators on HBM-enabled FPGAs, we run out of resources before utilizing available bandwidth from all of the HBM pseudo-channels (PCs). According to [3], to utilize all of the 32 HBM PCs on the Alveo U280, Hitgraph [1] and ThunderGP [2] require approximately 6× more resources than are available on the Alveo U280. LUT requirements per PC are 16.9% and 21.3% for Hitgraph and ThunderGP, respectively. Clearly, we observe issues such as over-utilization and difficult timing closure as we scale the accelerator size. Some of the latest SpMV accelerators [4], [14], [5] rely on improved design choices to reduce resources requirement (LUT/PC) but find it difficult to close timing at HBM interface frequency (450 MHz on the Alveo U280). GraphLily [4], Sextans [14], and HiSparse[5] runs at 166, 197, and 237 MHz, respectively while targeting the Alveo U280 platform.

The achieved quality of results (QoR) of these large designs is often much lower because various high-level semantics of the design including the structure of the processing elements, how they are composed, the structure of the memory hierarchy, and the interconnect are abandoned in the implementation flows. One underlying reason for QoR loss is that back-end implementation tools compile the designs as one large entity.

In this paper, we present a streaming dataflow accelerator for SpMV around the concept of modular and lean building blocks. We design and integrate domain-specific atomic modules and interconnect to maximize compute while maintaining QoR and designer productivity. Our approach to composing the SpMV accelerator is inspired by the GraphOps dataflow library [15]. GraphOps relies on HLS to generate the building blocks while we use a hybrid approach of mixing HLS with RTL for introducing FPGA awareness in the implementation. For example, some of the blocks in a design can use hard FPGA primitives more efficiently when implemented in RTL than in HLS [16]. On the other hand, HLS tools allow the the functionality of a building block to be described at a higher level to reduce developer effort, enable design portability, and enable rapid design space exploration, thus improving productivity, verifiability, and flexibility. We rely on a mixed IP approach (HLS and RTL) to introduce modularity and customization opportunities in the design.

We finely control all aspects of the FPGA implementation flow to create area-efficient and high-performance physical implementations. We extract critical paths from post-implemented netlists as feedback to create leaner and more optimized building blocks at the HLS and RTL levels. Results show that the SpMV accelerator can operate close to the bandwidth limit of the hardware platform, the limiting constraint in sparse computations. The main contributions of this paper are as follows:

- Demonstrating that (a) the modular and lean design of the SpMV accelerator coupled with (b) thoughtful implementation can allow complete saturation of HBM channels. We can compose an efficient SpMV accelerator using modular building blocks and show the bandwidth utilization of up to 26.2 GB/s, 90% of the peak memory bandwidth of two HBM PCs (28.8 GB/s).
- Demonstrating the implementation of our unique SpMV design which uses all of the 32 HBM pseudo-channels (PC) on the Alveo U280 FPGA platform by carefully floorplanning 16 SpMV accelerator blocks. The design can close timing at 310 MHz which is 80% higher than GraphLily [4] (166 MHz) and 40% higher than HiSparse-PB [5] (218 MHz). We demonstrate up to 50 GFLOPS performance on the Alveo U280 FPGA Platform which is 2.5× of GraphLily [4].

The paper is organized as follows: Section II dives into the emerging HBM-enabled FPGA platforms, the limitations of the FPGA implementation flow in mapping the SpMV accelerator designs; Section III describes the streaming dataflow architecture for SpMV targeting FPGAs; In Section IV, we describe our implementation flow enhancements to control each phase of the FPGA CAD flow and create high-performance implementations. We present the results of implementing SpMV accelerator in section V and discuss the merits of using the proposed approach. Finally, a broad area of future work is discussed in Section VI.

II. Background

In this section, we start with a discussion on SpMV accelerators in general and highlight the accelerator design challenges. Next, we discuss emerging HBM-enabled FPGA platforms and the limitations of the FPGA implementation flow in mapping large-scale SpMV accelerators efficiently on the platform.

A. SpMV Accelerators on FPGA Platforms

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 [17], [18], graph analytics [19], and physics simulations [20] where sparse computation, especially SpMV, is a key component. Acceleration of SpMV is thus becoming increasingly important [21], [22]. 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 [23].

Due to the dynamic nature of data flow, the peak performance of any SpMV accelerator depends primarily on the available memory bandwidth and the capability of the compute logic to effectively use it. SpMV accelerators need to perform two FLOPs (one multiplication and one addition) for each matrix value, resulting in a computation-to-communication ratio of 2 FLOPs per 'N+M' Bytes (assuming N Bytes for value and M Bytes for indexing). N is 4 Bytes for FP32 values and M is usually 4-8 Bytes depending on the encoding format. For example, in coordinate list (COO) encoded format where both row and column indexes are stored along with the non-zero values, assuming 2 Bytes each for row and column indexes, M results in 4 Bytes. Hence for every byte accelerator fetches from memory there is very little amount of compute. This is one of the reasons compute units become limited by memory bandwidth.

Emerging SpMV accelerators have started to make use of HBM-enabled FPGA platforms but these accelerators, unfortunately, do not scale well, resulting in under-utilization of the HBM bandwidth [4], [14], [5], [3]. In this paper, we show that modularity and composability can help in building better FPGA implementations. We provide a hardware designer with a set of composable building blocks, broad enough to target a wide array of sparse applications. Flexible and modular hardware blocks in our library can be used to construct accelerators for high-performance streaming sparse computations.

B. High Bandwidth Memory (HBM) Platforms

HBM is an emerging memory solution that offers high bandwidth by stacking multiple DRAM dies vertically. HBM provides multiple channels that can service memory requests concurrently. With the recent release of HBM-enabled FPGA platforms, developers can now exploit unprecedented external memory bandwidth. This allows more memory-bounded applications to benefit from FPGA acceleration. For example, the Alveo U280 has 32 HBM channels delivering 460 GBps bandwidth in total. The latest Versal HBM devices are capable of providing a further boost in bandwidth, 820 GBps, as HBM technology is improving at a very fast pace.

To utilize ever-increasing HBM bandwidth on FPGA devices and to make use of all the HBM channels on a device, accelerator designers rely on the reuse and replication of the processing elements [2], [24], [4], [5]. This results in very large designs that are mapped on the hardware platform where physical resources get heavily utilized, which leads to challenges in physical implementation.

C. FPGA Implementation Flow

The front-end of the implementation flow consists of high-level synthesis, which transforms the high-level design description into a netlist of physical gates. This netlist is then mapped to the target FPGA architecture with resources such as K-LUTs, DSPs, and Block RAM. The main objective of the front-end flow is to minimize the number of resources used (area) as well as the logical depth for each synchronous path (performance). The resulting netlist then goes through the back-end flow, which consists of placement and routing. The placement problem is to map the resources of the logical netlist

to resources on the physical FPGA while optimizing total wire length and critical path delays. This problem becomes challenging as designs get larger and more complex, coupled with constraints imposed by the FPGA architecture and device floorplans.

Commercial FPGA CAD tools provide support for incremental and out-of-context compilation, which allows the user to partition their design and implement each partition in isolation on a non-overlapping region of the FPGA floorplan. The key advantage of this approach is reduced compilation runtimes, predictable and repeatable results, as well as increased productivity. However, the algorithms do not have a full picture of the design, especially at interfaces between modules. Furthermore, the dedicated regions of the FPGA become unusable for parts of the netlist that are not part of the identified partition. These factors result in sub-optimality and loss of QoR.

Therefore, the typical flow for a user concerned about performance is to compile the entire design as one entity, which allows the tools to see the complete picture and globally optimize the design. We address the scalability challenges of this approach and attempt to mitigate them by carefully guiding the tools at various stages of the flow.

III. Streaming Dataflow Architecture For SpMV

Modern computing platforms, for example, the Alveo U280 FPGA and Tesla V100 GPU, support very high HBM bandwidth, 460 GBps and 900 GBps respectively. However, utilizing this much bandwidth efficiently is difficult for large-scale and highly sparse matrices due to very high random-access pattern and workload imbalance. The fixed memory hierarchy of GPU architectures can not avoid the penalty of cache misses while executing sparse workloads. On a cache miss, when a cache line is requested from main memory, a fraction of the cache line data is used and the rest is thrown away. This results in unnecessary data movement and power overheads. On the other hand, an FPGA accelerator with a customized memory hierarchy can avoid unnecessary data movement by efficiently generating main memory requests. In this section, we present the streaming dataflow architecture for the SpMV accelerator. Next, we describe the design of the SpMV accelerator around the modular building blocks and then we discuss the implementation on the FPGA fabric.

A. Accelerator Overview

Our SpMV accelerator uses gather-apply-scatter (GAS) model [1] as an execution strategy and coordinate list (COO) encoded adjacency matrix for storing sparse matrices in HBM banks. We design multi-ported multi-banked on-chip buffers (using memory blocks and packet-switched networks) to store input and output vectors on-chip. We exploit the benefits of low latency and parallel random access to on-chip buffers while streaming multiple non-zeros in parallel from HBM banks. The efficiency of the proposed SpMV accelerator can be determined by checking how efficiently it can process non-zeros coming from off-chip memory without introducing significant stalls. However, stalls are inevitable due to bank conflicts while accessing multi-ported multi-banked on-chip

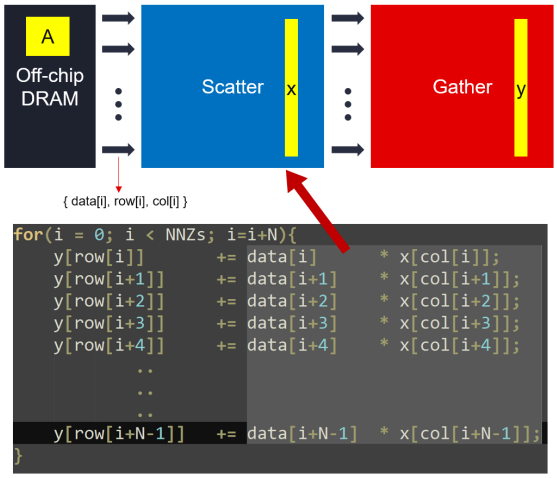


Fig. 1: Partially-unrolled SpMV Loop and corresponding Streaming Dataflow for Scatter-Gather SpMV Pipeline.

buffer. Sparse matrix non-zeros can be stored in memory by sorting them based on row/column indexes or by arranging them randomly or diagonally. In this paper, we also study the effect of the sparse matrix arrangement in memory on bank conflicts (see Section V-A).

Fig. 1 shows the SpMV loop (partially unrolled version) and corresponding hardware architecture. If a DRAM channel allows the injection of N non-zeros every cycle into the system, an architecture can be designed on top of FPGA fabric to operate on all of them in parallel. Most of the existing FPGA based SpMV accelerators follow a common theme of a streaming pipeline. One stage of the pipeline is generally referred to as scatter and the other as gather as shown in Fig. 1. The scatter stage uses a multi-port buffer for storing vector x. Every cycle, it requests a set of N non-zeros and uses column indexes to perform N reads in parallel, multiply it with corresponding data, and send the result forward to the gather stage. The gather stage also uses a multi-port buffer for storing y. It uses row indexes to perform N reads in parallel, performs addition with corresponding data, and finally writes the result back in y.

B. Accelerator Design

We design modular IP blocks (designed and verified in isolation) and stitch them together in Vivado IP integrator via latency-insensitive channels (AXI-streams with ready-valid handshake). To give an example, some of the building blocks in our SpMV accelerator (see Fig. 2) are: (a) 2×2 switches to construct network-on-chip (NoC): noc_0 and noc_1 (b) streaming Hazard Resolving Backpressure unit (HRB) units (hrb_0 to hrb_7) to handle carried dependencies. As shown in Fig. 2, the Load-store adaptor (LSA) is a module that connects the SpMV pipeline with HBM channels. Stream splitters (b_A0, b_A1, b_x) are the modules for splitting the wide stream into multiple narrow streams. Input banked vector buffer (BVB) modules (bvb_0 to bvb_7) are used for storing input vector banks and for the multiplication operation. The accumulator (ACC) modules (acc_0 to acc_7) are used for

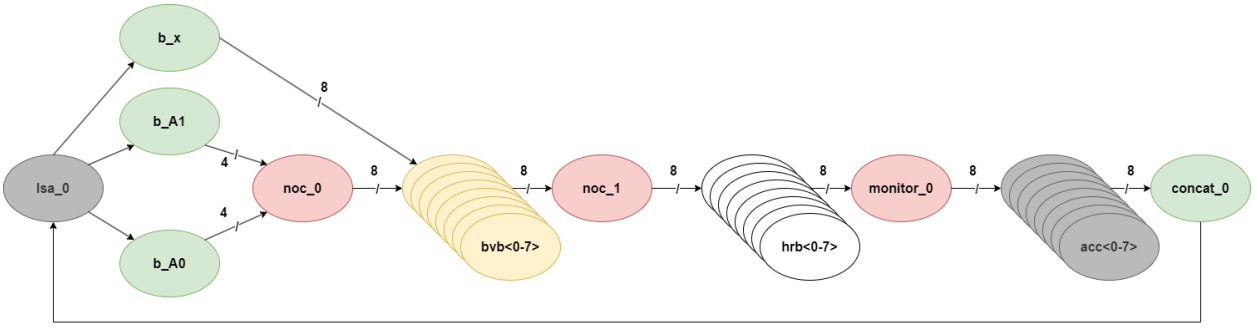


Fig. 2: Decomposing SpMV Pipeline into Modular Blocks communicating over Elastic Channels (Ready-Valid Handshake).

storing the output vector banks and for the accumulation operation. The concatenation module (concat_0) is used for packing results entries into a wide stream. The monitor (MON) module (monitor_0) is used for counting the non-zero packets flowing through the pipeline so that we can start draining the result as soon as the last non-zero gets processed. This modular accelerator design lets us extract very high performance out of the FPGA device (see Section IV-A). In other words, our method is similar to looking at the canvas and saying that we can implement this building block (compute or communication) optimally. Next, we discuss these blocks in detail.

1) Network-on-chip (NoC): 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 [25], [26] is one of the factors that limit the performance of SpMV accelerators. We present an approach where we replace the crossbar with a multi-stage switching network-on-chip (NoC) which is built using simple switches and achieves high-frequency implementation. By doing that, we avoid the switching network becoming the performance bottleneck. Fig. 3 shows the high-level diagram of our NoC built around 2×2 switches. It shows how we construct a multi-ported memory using the streaming switches and URAM banks, denoted as S and B in Fig. 3, respectively.

As non-zeros are provided at the input ports, the network routes the non-zeros as packets through multiple switch stages according to the column indices. The non-zero at any input port can have any column index, and the network routes it to the appropriate one of the output ports. In our design, 8 URAM banks (bank 0 through bank 7) are provided for storing the vector. The switching network can route the non-zero according to the 3 least significant bits of the 2 bytes that specify the column. In the 2-byte example, bits [15:3] are used as the read address (bits [2:0] are used in routing).

We choose to develop a buffered 2×2 switch architecture built around latency-insensitive dataflow units. These include elastic buffers (EBs) [27], 2-way split units, and 2-way merge units. Each 2×2 switch uses 2 split units, 2 merge units and 4 EBs. The EBs in our NoC architecture make use of a specific implementation, full bandwidth 2-slot EB [28], to avoid stalls and pipeline bubbles.

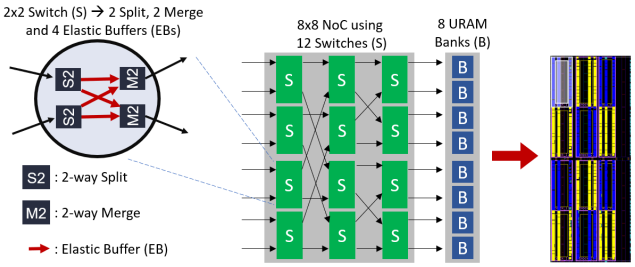


Fig. 3: Multi-stage Switch Network using Elastic Buffers.

2) Load Store Adaptor (LSA): Control of data-movement is critical for the flexibility of our design approach. To keep the design highly flexible and customizable, we resort to HLS for describing the load-store adaptor. Fig. 4 shows the composition of compute pipeline in which LSA is the bridge between HBM channels and the SpMV accelerator. The pipeline includes high-speed NoCs (block B and block D), Vector multipliers (MUL, block C) and accumulators (ACC, block F).

During execution, the LSA requests input vector elements from one of the HBM channels and fills the input vector buffer within block C. Then, it requests 8 non-zeros every cycle from both HBM channels (4 non-zeros from each 32 Byte channel) and feeds them to the NoC (block B) 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 by the non-zero value. The results of multiplication are then routed by the second NoC (block D) to their corresponding accumulators. After all of the accumulations are done, the LSA stores the result back to one of the HBM channels.

3) Banked Vector Buffer (BVB): 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 program allows 8K FP32 entries for each BVB based on which the kernel can handle up to 64K×64K matrices using 8 BVBs. HLS pragma is used to specify that the URAM should be used to hold vector entries. This allows URAM cascading for large BVB sizes.

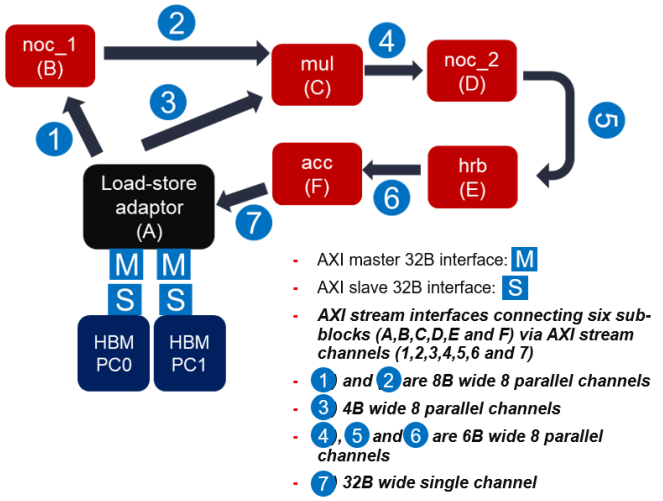
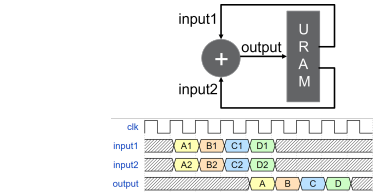


Fig. 4: System-level Diagram for Data Movement between HBM Channels and SpMV Building Blocks.

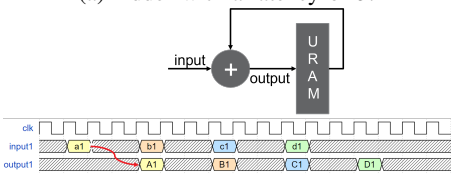
4) Hazard Resolving Backpressure unit (HRB): Once the non-zeros get routed to the input vector banks using the NoC and multiplied with the corresponding vector entries, the results of the multiplications have to be routed to the output vector banks using a second NoC for accumulation purpose. A major 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 accumulations on incoming data cannot be performed until the previous accumulation has been completed due to possible data dependency.

Fig. 5a shows a simple adder with a latency of N cycles (3 in this example). It means that 3 cycles are required for the adder to finish the add operation. Also, the adder is deeply-pipelined and inside the adder, there are 3 stages of pipeline which means the adder can accept a set of inputs every clock cycle and produce an output every clock cycle. Fig. 5b shows an accumulator with the same adder. Although the adder technically can accept inputs each cycle, due to the data dependency of the accumulator, we need to wait for the adder to finish the operation and then accept the next input (as it was shown in the waveform). Due to the data dependency in the accumulation, we are not fully using the adder's capabilities and the adder pipeline is technically not busy N-1 cycles out of N cycles (2 cycles out of 3 cycles in this example). Using resource sharing, we are able to assign those unused cycles to parallel accumulations if there are no data dependencies between separate accumulations. Fig. 5c shows the time-shared accumulator.

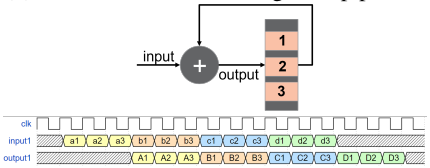
As discussed before, we accumulate based on the row index. It means that the inputs associated with the same row index should be added to each other. In other words, there are no data dependencies between inputs with different row indexes so it is possible to resource share the accumulator for rows with a different index. On the other hand, it is hazardous to push inputs with the same row index to the accumulator in consecutive clock cycles.



(a) Adder with a latency of 3.



(b) Accumulator with 3 stages of pipeline.



(c) Resource sharing an accumulator.

Fig. 5: Adder and accumulators with a latency of 3 cycles.

FPGA FP32 adders have certain latency L (4-8 clock cycles) to run at high frequencies and it could lead to data hazard where the value from the 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 the same indexes within a time window of L clock cycles.

To avoid these hazards, one possibility is to stall the incoming stream for L cycles after each accumulation (similar to Fig. 5b). This results in poor performance of the SpMV pipeline since the peak throughput now gets reduced by L times. To solve this problem, we design a special back-pressure system referred to as Hazard-resolving back-pressure (HRB) unit (block E in Fig. 4).

HRB keeps track of all the indexes in flight by using a shift register. It compares any incoming index with all of the shift register values and if there is no match, the index-value pair is safe to move forward for accumulation purposes. But if there is a match, the HRB would not accept any incoming index-value pair by simply asserting back-pressure. As soon as the conflicting index moves through the shift register, HRB would start accepting new index-value pairs. This way, HRB can reduce the effective initiation interval (II) of the compute pipeline from L to 1 as shown in Fig. 6.

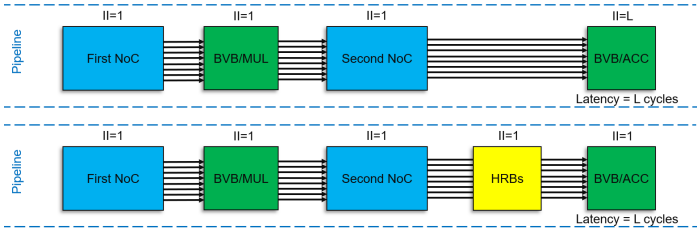


Fig. 6: Effect of HRB on II of SpMV accelerator pipeline.

5) Accumulator (ACC): Each of the 8 ACC blocks is coupled to receive the output product-row tuple from one of the 8 HRB blocks, and each ACC block accumulates totals of the products by row. The ACC blocks have respective memory banks for storing the totals as the totals are accumulated. The current totals in the memory banks are addressed by the row indices of the tuples. For example, ACC 0 has a memory bank that stores totals for rows 0, 8, 16, ...; ACC 1 has a memory bank that stores totals for rows 1, 9, 17, ...; etc.

We represent our SpMV accelerator as being composed of modular building blocks. Table I shows the resource utilization. Each 2×2 switch requires approximately 350 LUTs and 700 FFs. NoC uses 12 switches and hence a total of 4200 LUTs and 8400 FFs. Compared to the 2D-mesh NoC in [29] which uses 192 EB, 64 split, and 64 merge units, our NoC is much more area-efficient as it requires 4× fewer EBs (48), 2.6× fewer split and merge units (24).

Atoms	LUTs	FFs	DSPs	BRAMs	URAMs
NoC (2)	4200	8400			
BVB (8)	228	967	3		1
ACC (8)	530	1226	2	0.5	2
HRB (8)	336	379			
LSA (1)	4419	9572		12	
MON (1)	86	243			

TABLE I: Resource Utilization of Building Blocks.

To understand the resource saving with our new building blocks, we compare our SpMV accelerator design with our previous work [29]. The SpMV accelerator in [29] focuses more on designing the performance-oriented kernel (minimizing stalls in the pipeline and maximizing bandwidth utilization) at the expense of high resource requirements. The kernel uses an expensive network (2D-mesh) and heavy-duty adder trees, referred to as HRT units, for dealing with hazards. As the kernel is performance-oriented and resource hungry (LUT/PC = 3%), it does not scale well on the Alveo U280 HBM platform. In this paper, we design a new kernel by proposing a lean switch network and a lean hazard unit (HRB). Our new kernel results in 3× fewer LUTs and FFs, 5× fewer DSPs, and 2× fewer BRAMs. The lean nature of the kernel (LUT/PC = 1%) allows us to scale it better (up to 16 kernels).

IV. Implementation Flow Enhancements

In this section, we discuss mapping our SpMV accelerator on FPGA fabric and the various optimizations during implementation. Current implementation tools work in phases where each phase takes inputs/solution from the previous phase, processes it, and sends the solution to the next phase. In our approach, we carefully design and guide the tools by passing information between each of the phases of the flow.

A. Front-End Optimizations

We make the design of the accelerator aware of the implementation and mapping by taking an iterative approach through the current flows. Each of the building blocks is initially designed in HLS and taken through all the implementation steps of the flow. Then we analyze the post-routed implementation and trace back the critical paths and congestion bottlenecks back to the original HLS code at the atomic level. We iterate this process to tweak the high-level design and make it more amenable for back-end tools. Where appropriate, we also convert HLS into manually written verilog to further optimize the critical paths. We focused our optimizations on the HRB, Monitor, and ACC blocks as these were the identified bottlenecks in the fully integrated kernel implementation. Table II shows the resource utilization of these blocks before and after optimizations.

Atoms	Pre-Opts		Post-Opts		Improvement	
	LUTs	FFs	LUTs	FFs	LUTs	FFs
ACC (8)	530	1226	312	475	1.7×	2.6×
HRB (8)	336	379	79	322	4.2×	1.2×
MON (1)	86	243	46	166	1.9×	1.5×

TABLE II: Resource Utilization of Building Blocks Before and After Optimizations.

Fig. 7 shows the post-implemented $fmax$ achieved for each of the building blocks before and after these optimizations. Each of the blocks is implemented out-of-context in isolation targeting 500MHz. The end result of these optimizations led to a fully integrated kernel implementation that met **465 MHz**. Compared to our previous work in [30], this work optimizes the building blocks within the kernel so that it can reach a high operating frequency. We observe a 37% improvement by pushing the $fmax$ from 340 MHz to 465 MHz. The fully integrated SpMV kernel uses 50K FFs (1.6%), 20K LUTs (2%), 40 DSP (0.44%), 16 BRAM (0.8%), 24 URAM (2.5%).

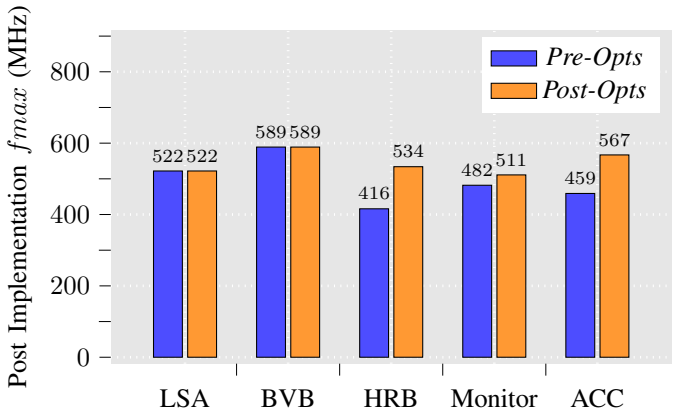


Fig. 7: Post Implementation $fmax$ of SpMV Building Blocks.

Since the communication between the building blocks are elastic, as we improve the critical paths within each of the building blocks, we can see significant benefits on a fully implemented kernel. Note that the kernel frequency is still lower than the worst atom frequency because the blocks are optimally placed in overlapping regions along with the HBM Memory Subsystem (HMSS) IP block (required as a bridge between kernels and HBM PCs), which leads to a more challenging placement problem and congested regions for the router.

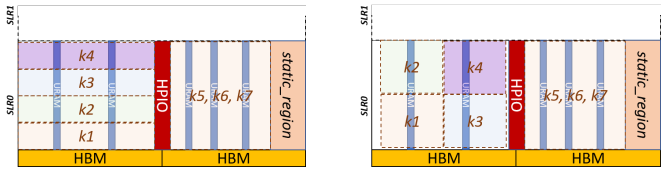
B. Passing High-Level Design Semantics

With the mixed-IP optimization approach described above, we are able to synthesize highly optimized building blocks. There are several high-level semantics that we want to consider during back-end implementation. Each kernel of the SpMV implementation is designed to saturate two HBM pseudo-channels (PCs) and can therefore be individually optimized. Since the HMSS IP is pipelined and elastic, it is not critical and therefore can be implemented after prioritizing the kernels. The information about replicable kernels and their boundaries is useful for back-end tools as each kernel can then be individually (and parallelly) optimized without considering the other kernels. There are several heuristics used to manage runtime and design scalability that come into effect when design sizes become large. With knowledge of this semantic, each kernel of 100K instances can be implemented without worrying about design size and scalability. We evaluate this by individually implementing each kernel in a non-overlapping region, managed with area constraints, then merging the solution and using it to implement a full 16-kernel solution. We evaluate this by creating 16 1-kernel designs each targeting a different HBM channel and constrained to a different non-overlapping region. Then the physical location of each instance is extracted from the post-implemented design. The same locations are then copied to create a merged 16-kernel solution. Since there is currently no interface to extract the routing information from a post-routed netlist, the larger 16-kernel design is routed again. With this approach, we see that each individual kernel is able to achieve an $fmax$ greater than 420MHz, which should then be theoretically achievable for the 16-kernel design.

C. Floorplanning

The device composition and floorplan of the platform are very important and need to be considered at the beginning of the design cycle. Having an idea of the resource requirements of each kernel and the connectivity requirements can impact the way we design them. For instance, the U280 device has a wide HPIO column bisecting the device into left and right halves. Any connection that requires crossing this column will incur a large delay penalty. This means that the kernels connecting to HBM channels must remain on their respective halves to minimize connections that cross this column. Preliminary implementations of the SpMV design without this consideration showed critical paths crossing these signals with delay penalties of 400-600ps. Since each kernel is independent, we are able to constrain all kernels communicating to an HBM at their respective halves. This minimized the HPIO crossing nets resulting in a much higher $fmax$.

To implement each kernel in a non-overlapping region, we need to carefully consider the resource utilization of the kernel and the capacity of the region where it is constrained. The U280 device has 5 columns of URAM (2 on the left half, and 3 on the right half), where each column contains 64 URAMs per SLR, while the remaining resources are ubiquitous in the device. Therefore, the area constraint for the region must include at least 24 available URAMs. Since each clock region contains 16 URAMs, we constrain the kernel to 2 clock



(a) Option 1 (b) Option 2

Fig. 8: Kernel Floorplanning Options.

regions. We explored several floorplanning configurations for the 16-kernel design. Fig. 8 illustrates two of the compelling options that were considered. Option one constrained each kernel to its own half, while spanning 4 clock regions wide by 1 clock region tall. This configuration is advantageous especially in a columnar architecture since a single kernel PnR solution can be exactly replicated at a vertical offset. All 8 kernels on the left half can be identical implementations of each other (similarly on the right half). On the right half of the HPIO, the capacity is reduced since the static region required for communication with the host PC requires FPGA resources, therefore we are unable to implement 4 kernels on 1 SLR on the right half. We can instead implement 3 kernels together, then replicate it to create 6 and 8 kernel implementations. Regardless of this capacity reduction, note that we cannot copy the left placements to the right since the column composition is different. It would indeed be beneficial for this use case to have a fabric composition that is regular and repeating such as those supported by more modern FPGA fabrics [31]. The second configuration option constrained the kernels to a 2×2 clock region area, where PnR in each column can be replicated vertically. Option 2 showed much better QoR because the 2×2 configuration created a tighter wirelength picture and a better balance between horizontal and vertical resources. Furthermore, horizontal connections incur larger (and more variable) delay penalties due to crossing the wider BRAM, URAM, and DSP columns. Vertical resources are faster and have a much tighter distribution of delays. We have found that critical paths tend to occur on paths that span horizontally more often than vertically. Fig. 9 shows the fully implemented 16-kernel SpMV design on the U280 platform using the option 2 configuration.

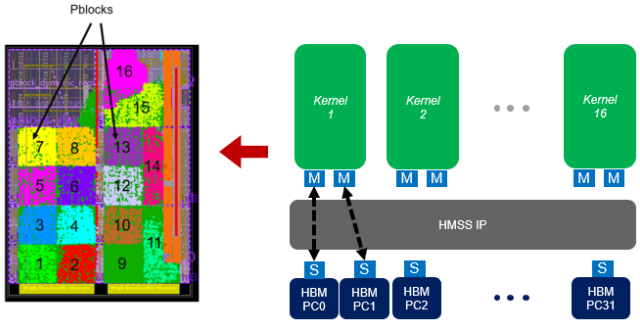


Fig. 9: Scaling the SpMV Accelerator Architecture with Multiple HBM Channels and Floorplanning at the kernel level.

D. CAD Flow Directives

Commercial implementation tools expose several algorithmic directives to the user that we can use to customize aspects of the flow. Each directive guides the heuristic and algorithmic choices in a certain way to produce different implementation results. It is non-trivial to identify the combination of directives that is ideal for a certain netlist due to their complex behaviors and interaction. Recent work has explored using machine learning techniques to identify the directives leading to optimal QoR for a given netlist [32]. In this work, we run the tools in 'ALL' implement strategies mode, which simultaneously launches various combinations of directives. We then choose the directive combination based on the achieved results on the 16-kernel SpMV implementation. "ExtraNetDelay" during placer, "AggressiveExplore" during Physical Optimization and "NoTimingRelaxation" mode in Router showed the better result achieving an $fmax$ of 310MHz on the 16-kernel design. All subsequent experiments were performed with this combination of directives.

V. Evaluation and Benchmarking

We implemented and benchmarked the SpMV accelerator on the Alveo U280 using Vivado and Vivado HLS 2021.1. The Alveo U280 supports 8 GB of HBM, with 32 AXI channels supporting an aggregate peak bandwidth of 460 GB/s.

A. SpMV Accelerator Evaluation

We initially construct a design with a single kernel to observe how much bandwidth can be utilized from two HBM pseudo-channels (peak = 28.8 GBps). Compared to other SPMV accelerator designs which are able to achieve a performance of around 100-250 Mhz [33], [34], [35], [25], our implementation achieves 465Mhz. We evaluate the performance of the kernel using sparse matrices from the University of Florida Sparse Matrix collection. We use a set of matrices from [33]. Similar to [33], we report GFLOPS numbers and % Peak Bandwidth Used and compare our results with the results of other SpMV accelerators, as shown in Fig. 10 and Fig. 11. For the benchmark set of matrices, our SpMV accelerator shows the bandwidth utilization ranging between $38-74\%$ while the bandwidth utilization is $10-18\%$ for BEE3 [33], $13-29\%$ for HC-1 [34], $10-75\%$ for CASK [35], and $28-40\%$ for GEMX SpMV engine [25].

The improved BW utilization is mainly due to minimizing the stalls and bank conflicts in the design. For example, the load-store adaptor efficiently supplies read requests to memory without getting many stalls from the pipeline. This is because the HRB within the pipeline guarantees very few stalls from the accumulation stage. The switching network also exhibits minimal stalls as the back-pressure due to bank conflicts get absorbed in distributed elastic buffers. Also, since we use COO-encoded matrix and store 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 the scatter stage and if we sort the matrix by row, the bank conflicts would be at the gather stage. The coordinate list (COO) encoded format allowed us to store the matrix in random order resulting in minimal

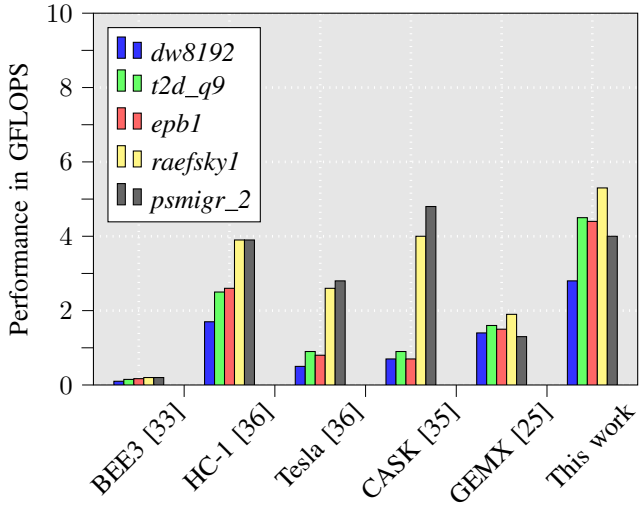


Fig. 10: Performance (in GFLOPS) comparison of different accelerators for a set of matrices.

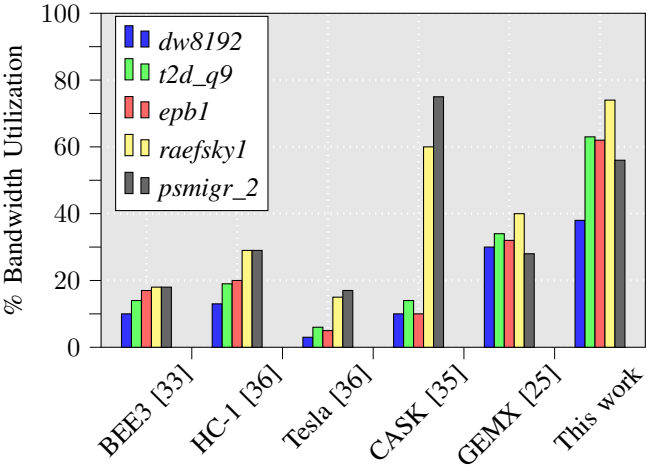


Fig. 11: % Peak Bandwidth Utilization of different accelerators for a set of matrices.

bank conflicts. We randomly reorder the matrix non-zero offline on the host machine. We use *random.shuffle()* function from Python. It uses Fisher-Yates shuffle as the underlying algorithm [37].

Apart from the benchmark set in [33], we have used several other matrices (from the University of Florida Sparse Matrix collection) to observe the effect of random, row-major and column-major traversal. Fig. 12 shows the results in terms of efficiency numbers. 100% efficiency would mean the performance of 8 non-zero processed every clock cycle. Random traversal is showing good results, up-to 90% memory bandwidth utilization (26.2 GB/s out of 28.8 GB/s), followed by column-major and finally row-major. The poor performance in the case of row-major is expected because HRB would create heavy back pressure when the same row indexes are coming back-to-back. Fig. 12 proves that the pipeline is getting stalled heavily in the case of row-major.

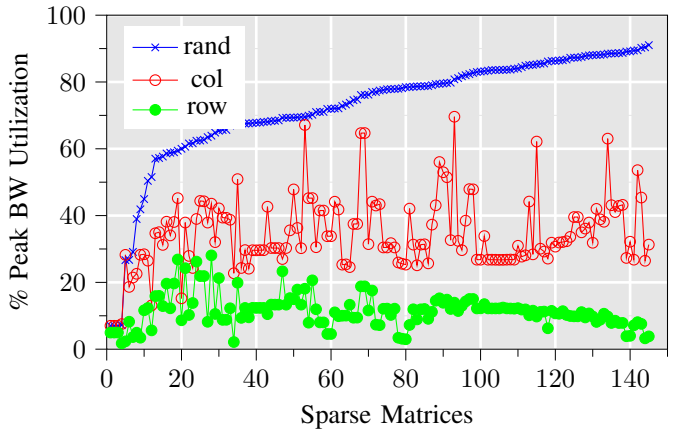


Fig. 12: Efficiency results for different traversals.

We now discuss the performance scaling of our accelerator with multiple HBM channels. We take the benchmark set in [33], perform partitioning of the matrix in N partitions, and use N kernels in parallel to process those. N matrix partitions are the horizontal chunks of the original matrix. Fig. 13 shows the GFLOPS scaling as we scale the number of kernels on the device.

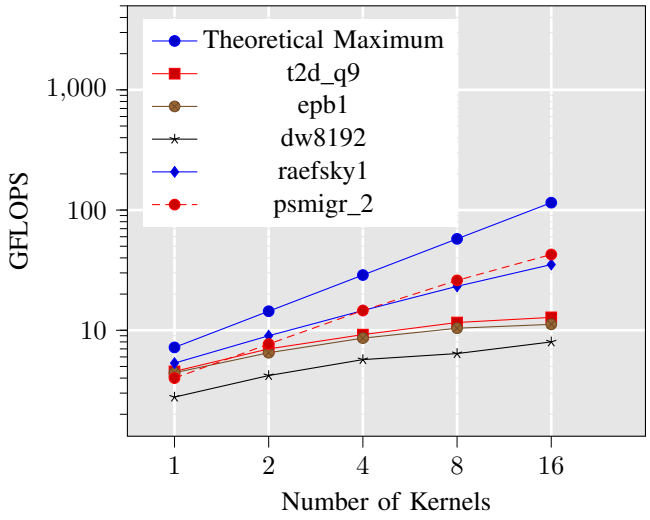


Fig. 13: Scaling the GFLOPS with Multiple HBM Channels and Kernels.

B. Scaling the Design

As the number of kernels in the design increases, Fig. 14 shows that the $fmax$ can decrease by upto 35%. In the pre-optimized version of the design, we start from 340 MHz for 1 kernel, while a 16-kernel implementation can achieve a max of 227 MHz, a 33% drop in performance. Similarly, while the 1 kernel design improves significantly due to the atomic changes discussed above, at 16 kernels, this improvement diminishes. With floorplanning, replication, and CAD strategies described in section IV, we can reduce this performance degradation and effectively scale the design. The 16-kernel performance of 310 MHz (which translates to a theoretical peak performance of 80 GFLOPs) is achieved when all of these strategies are combined. It is interesting that at lower kernel counts, K1 and K2, we see better performance without any implementation strategies. This is because the heuristics of the algorithms can better manage the smaller design sizes and find a more optimal solution. As the design size grows, guidance from our implementation strategies becomes essential. At K4 and above, we see the 'Post-Opts+Impl.Strategies' flow outperforming the 'Post-Opt' flow. There is an 8-12% improvement with implementation strategies for larger designs (K8 and K16).

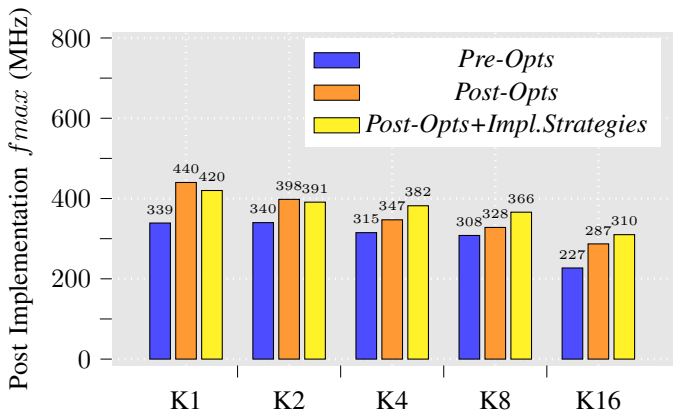


Fig. 14: Post Implementation $fmax$ for SpMV Design with increasing Kernels.

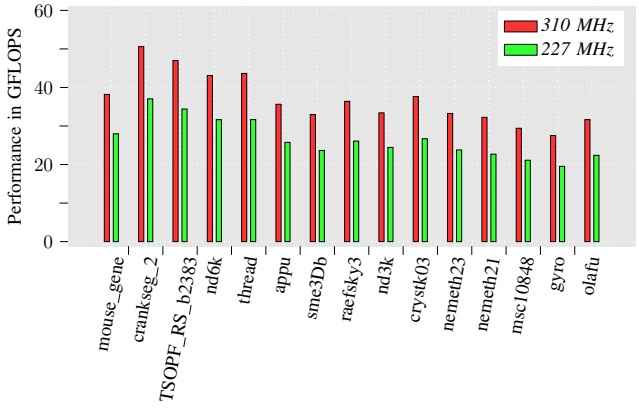


Fig. 15: Achieved Performance Results on the Alveo U280 for a set of sparse matrices.

C. Benchmarking Results

Fig. 15 shows the performance of the baseline 16-kernel design (227 MHz) and optimized 16-kernel design (310 MHz) for a set of matrices from the University of Florida Sparse Matrix collection. We observe a performance of up-to 50.6 GFLOPS using 310 MHz 16-kernel design. In theory, it should have been possible for the implementation tools to maintain the highest achieved frequency (465Mhz) irrespective of the number of kernels we instantiate. If this was achieved in practice, a performance of 76 GFLOPS (for $crangseg\_2$

sparse matrix) could have been achieved using our K16 design running at 465 MHz. However, we observe a 50% drop in $fmax$ when we scale the design size and it translates directly into SpMV performance (38 GFLOPS). As we apply our techniques of recovering QoR, we observe a boost of 36% in $fmax$. Hence, our highest reported performance so far is 50.6 GFLOPS (25,300 MTEPS) with 310 MHz K16 implementation. It will be a focus of this work in the future to manage implementation at an even finer level to prevent this loss of performance.

Fig. 16 shows the comparison of our implementation with state-of-the-art FPGA-based SpMV accelerators. The Alveo U280 is the target FPGA platform for all three accelerators, GraphLily [4], Sextans [14], and our 16-kernel design. Our design is able to close timing at 310 MHz which is 80% higher than GraphLily (166 MHz) and 57% higher than Sextans (197 MHz). For $crankseg\_2$ matrix, our design outperforms both GraphLily and Sextans by 2.5×, by delivering a performance of 50.6 GFLOPS on Alveo U280. Performance gain in our case comes from the following aspects: (a) higher operating frequency (b) using all 32 HBM PCs (c) fewer stalls because of random traversal.

GraphLily uses Fixed-point arithmetic while Sextans uses FP32. Our 16-kernel design uses FP32 arithmetic and consumes the following resources on U280: 500K LUTs (38%), 1178K FFs (45%), 644 DSPs (7%), 460 BRAMs (23%), and 384 URAMs (40%). Table III compares the % resource utilization of our SpMV accelerator with others. Please note that our design is the one that uses all 32 HBM PCs while other designs find it difficult to scale to all PCs.

	LUTs	FFs	DSPs	BRAMs	URAMs
GraphLily [4]	35	21	8	24	53
Sextans [14]	29	26	36	76	80
This Work	38	45	7	23	40

TABLE III: Comparing % Resource Utilization of different SpMV Accelerators on the Alveo U280 Platform.

Next, in Fig. 17, we show the comparison of our implementation with CPU, GPU, and FPGA-based SpMV accelerators for $mouse\_gene$ sparse matrix. Our SpMV design on the Alveo U280 is able to outperform CPU by 3× and GPU by 1.3×. Fig. 17 also shows that we consistently outperform many different implementations. For example, the latest implementation of ThunderGP [2] on the HBM-enabled FPGA platform (Alveo U280) provides 2.3× performance improvement over ThunderGP implemented on a non-HBM platform (Alveo U250). Our design is able to outperform ThunderGP-HBM implementation by 2×.

VI. Conclusions and Future Work

As memory bandwidth on modern FPGA platforms increases, SpMV accelerator design should scale to utilize the available bandwidth. The proposed approach demonstrates that with the lean implementation of the kernels, scaling of the design is possible, and with the modular implementation, SpMV accelerator can close timing at high frequency. We

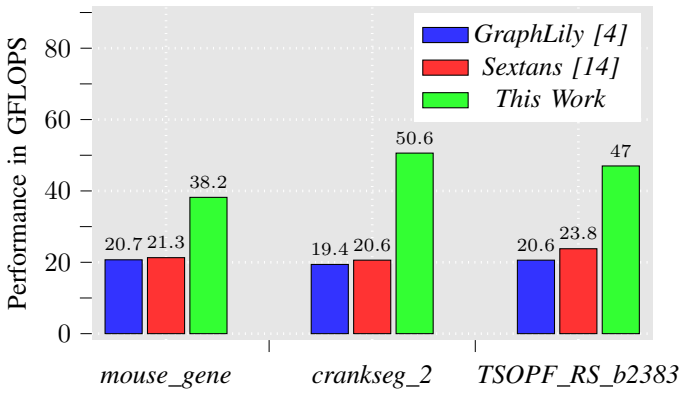


Fig. 16: Comparing Performance with GraphLily and Sextens.

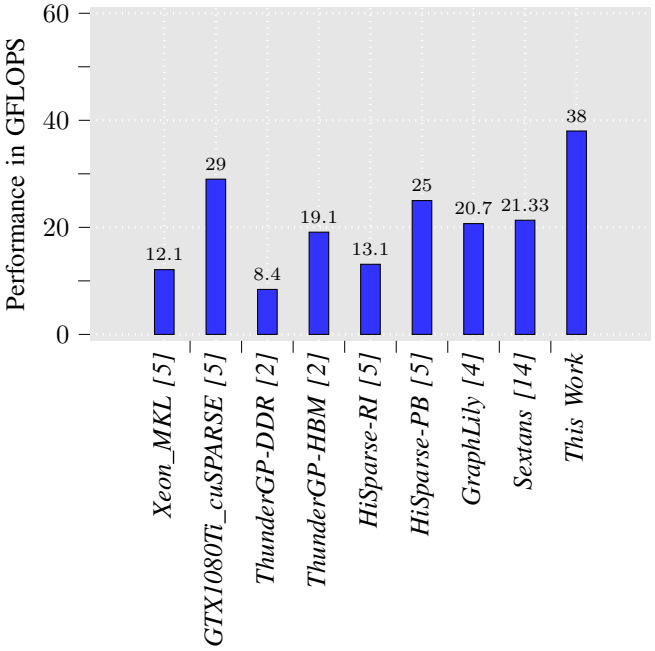


Fig. 17: Comparing the Performance of different SpMV implementations for *mouse_gene* matrix.

demonstrated that single instance implementation of the accelerator, running at 465 MHz, can deliver close to peak bandwidth utilization when connected to two HBM pseudo-channels. Further, we have scaled the design to utilize all of the 32 HBM pseudo-channels on the Alveo U280 FPGA platform by floorplanning 16 SpMV blocks. The scaled version of the design is able to close timing at 310 MHz which is 80% higher than GraphLily (166 MHz) and 40% higher than HiSparse-PB (218 MHz). We are able to show up-to 50 GFLOPS performance on the Alveo U280 FPGA Platform which is 2.5× of GraphLily performance. As future work, we plan to support a library of compute and communication atoms designed for Sparse Linear Algebra and show the benefits of this approach on a variety of applications. We plan to enhance the replication strategy by supporting this in implementation tools using open-sourced tool chains such as RapidWright [38].

REFERENCES

[1] 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 (TPDS)*, vol. 30, no. 10, pp. 2249–2264, 2019.

[2] X. Chen, F. Cheng, H. Tan, Y. Chen, B. He, W.-F. Wong, and D. Chen, "ThunderGP: resource-efficient graph processing framework on FPGAs with hls," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2022.

[3] X. Chen, Y. Chen, F. Cheng, H. Tan, B. He, and W.-F. Wong, "Regraph: Scaling graph processing on HBM-enabled FPGAs with heterogeneous pipelines," in *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 1342–1358.

[4] Y. Hu, Y. Du, E. Ustun, and Z. Zhang, "GraphLily: accelerating graph linear algebra on HBM-equipped FPGAs," in *Proceedings of the IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 2021, pp. 1–9.

[5] Y. Du, Y. Hu, Z. Zhou, and Z. Zhang, "High-performance sparse linear algebra on HBM-Equipped FPGAs using hls: A case study on SpMV," in *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2022, pp. 54–64.

[6] N. P. Jouppi, C. Young, N. Patil, and D. Patterson, "A domain-specific architecture for deep neural networks," *Communications of the ACM*, vol. 61, no. 9, pp. 50–59, 2018.

[7] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman *et al.*, "Serving dnns in real time at datacenter scale with project brainwave," *IEEE Micro*, vol. 38, no. 2, pp. 8–20, 2018.

[8] P. D'Alberto, V. Wu, A. Ng, R. Nimaiyar, E. Delaye, and A. Sirasao, "xDNN: inference for deep convolutional neural networks," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 15, no. 2, pp. 1–29, 2022.

[9] S. Neuendorffer, A. K. Khodamoradi, K. Denolf, A. K. Jain, and S. Bayliss, "The evolution of domain-specific computing for deep learning," *IEEE Circuits and Systems Magazine*, vol. 21, no. 2, pp. 75–96, 2021.

[10] 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.

[11] 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.

[12] 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.

[13] P. Holzinger, D. Reiser, T. Hahn, and M. Reichenbach, "Fast HBM access with FPGAs: Analysis, architectures, and applications," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2021, pp. 152–159.

[14] 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 International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2022, pp. 65–77.

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] L. Song, Y. Chi, L. Guo, and J. Cong, "Serpens: a high bandwidth memory based accelerator for general-purpose sparse matrix-vector multiplication," in *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, 2022, pp. 211–216.

[25] X. GEMX. (2018). [Online]. Available: https://github.com/Xilinx/gemx

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

[27] 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.

[28] 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.

[29] A. K. Jain, H. Omidian, H. Fraisse, M. Benipal, L. Liu, and D. Gaitonde, "A domain-specific architecture for accelerating sparse matrix vector multiplication on FPGAs," in *Proceedings of the International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2020, pp. 127–132.

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

[31] B. Gaide, D. Gaitonde, C. Ravishankar, and T. Bauer, "Xilinx adaptive compute acceleration platform: Versal architecture," in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2019, pp. 84–93.

[32] N. Kapre, B. Chandrashekaran, H. Ng, and K. Teo, "Driving timing convergence of FPGA designs through machine learning and cloud computing," in *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2015, pp. 119–126.

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

[34] 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.

[35] 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.

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

[37] F. P. Juniawan, H. A. Pradana, D. Y. Sylfania *et al.*, "Performance comparison of linear congruent method and fisher-yates shuffle for data randomization," in *Journal of Physics: Conference Series*, vol. 1196, no. 1. IOP Publishing, 2019, p. 012035.

[38] C. Lavin and A. Kaviani, "Rapidwright: Enabling custom crafted implementations for fpgas," in *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2018, pp. 133–140.