Cuper: Customized Dataflow and Perceptual Decoding for Sparse Matrix-Vector Multiplication on HBM-Equipped FPGAs

Enxin Yi*, Yiru Duan*, Yinuo Bai*, Kang Zhao†, Zhou Jin*, Weifeng Liu*
*Super Scientific Software Laboratory, China University of Petroleum-Beijing, China
†Department of Integrated Circuits, Beijing University of Posts and Telecommunications, China
Email: *{enxin.yi, yiru.duan, bai}@student.cup.edu.cn, †zhaokang@bupt.edu.cn, *{jinzhou,weifeng.liu}@cup.edu.cn

***Abstract*—Sparse matrix-vector multiplication (SpMV) is pivotal in many scientific computing and engineering applications. Considering the memory-intensive nature and irregular data access patterns inherent in SpMV, its acceleration is typically bounded by the limited bandwidth. Multiple memory channels of the emerging high bandwidth memory (HBM) provide exceptional bandwidth, offering a great opportunity to boost the performance of SpMV. However, ensuring high bandwidth utilization with low memory access conflicts is still non-trivial. In this paper, we present Cuper, a high-performance SpMV accelerator on HBM-equipped FPGAs. Through customizing the dataflow to be HBM-compatible with the proposed sparse storage format, the bandwidth utilization can be sufficiently enhanced. Furthermore, a two-step reordering algorithm and perceptual decoder-centric hardware architecture are designed to greatly mitigate read-after-write (RAW) conflicts, enhance the vector reusability and on-chip memory utilization. The evaluation of 12 large matrices shows that Cuper's geomean throughput outperforms the four latest SpMV accelerators HiSparse, GraphLily, Sextans, and Serpens, by 3.28×, 1.99×, 1.75×, and 1.44×, respectively. Furthermore, the geomean bandwidth efficiency shows 3.28×, 2.20×, 2.82×, and 1.31× improvements, while the geomean energy efficiency has 3.59×, 2.08×, 2.21×, and 1.44× optimizations, respectively. Cuper also demonstrates 2.51× throughput and 7.97× energy efficiency of improvement over the K80 GPU on 2,757 SuiteSparse matrices.**

***Index Terms*—Sparse Matrix-Vector Multiplication, FPGAs, HBM, Accelerator**

I. Introduction

Sparse matrix-vector multiplication (SpMV) is pivotal in many scientific computing and engineering applications, and has a significant performance impact on graph computation, machine learning, information retrieval, and cloud computing, etc [1]. FPGAs are considered attractive platforms for accelerating SpMV. Compared to traditional CPUs and GPUs platforms, FPGAs can fully leverage the parallelism potential of SpMV by customizing dataflow and memory structure [2]–[4]. In addition, FPGAs typically have low power consumption.

However, there are several challenges to accelerate SpMV on FPGAs platforms. Firstly, due to the limited number of independent memory channels, conventional FPGAs with DDR memory system are poor at concurrent memory accesses and lead to a throughput mismatch between the memory structure and parallel processing engines (PEs). Secondly, randomly distributed non-zeros in sparse matrices may cause irregular memory access patterns, which restrict the available memory bandwidth as the DDR is better suited for continuous memory accesses [5].

High bandwidth memory (HBM) [6] features more memory channels and a greater memory bandwidth compared to traditional DDR memory, bringing great opportunity to accelerate SpMV. Some pioneering works have proposed sparse accelerators on HBM-equipped FPGAs and demonstrated their effectiveness in accelerating SpMV, i.e., HiSparse [6], GraphLily [7], Sextans [8], and Serpens [9], etc. HiSparse leverages customized sparse format and a scalable on-chip buffer design to improve bandwidth utilization. GraphLily is an FPGA overlay to support generalized graph applications. Sextans is an accelerator for sparse matrix-dense matrix multiplication (SpMM) that supports SpMV with redundant HBM channels. Serpens is a memory-centric general SpMV accelerator and achieves competitive performance to GPUs. However, fully benefiting high bandwidth of HBM to accelerate SpMV is still non-trivial. There are still three key challenges that need to be addressed: (1) Low utilization bandwidth of HBM with existing sparse storage formats. (2) The high latency introduced by read-after-write (RAW) conflicts lead to a limited compute occupancy. (3) Input vector and on-chip memory are underutilized.

In this paper, we propose a high-performance SpMV accelerator Cuper on HBM-equipped FPGAs to resolve these challenges. Cuper is equipped with following techniques: (1) We utilize the proposed sparse storage format to customize HBM-compatible dataflow to support vectorized and streaming accesses to memory channels, thereby improving bandwidth utilization. (2) The two-step reordering algorithm combining conflict-aware row reordering and reuse-aware column reordering mitigates RAW conflicts and improves vector reusability. (3) In addition, we design a perceptual decoder that further enhances reusability and on-chip memory utilization by a flexible reuse register design and skipping redundant vector writes. Our main contributions can be summarized as follows:

- We customize dataflow by employing the proposed sparse storage format to improve bandwidth utilization of HBM.
- We introduce a two-step reordering algorithm for mitigating RAW conflicts and reusing vector.
- We design a perceptual decoder in order to further enhance vector reusability and on-chip memory utilization.

We implement Cuper on a Xilinx Alveo U280 FPGA. The

979-8-3503-4859-0/DATE24/© 2024 EDAA

evaluation shows that Cuper exhibits advantages in terms of throughput, bandwidth efficiency, and energy efficiency over HiSparse, GraphLily, Sextans, and Serpens, respectively. We also compare with an Nvidia Tesla K80 GPU on 2,757 SuiteSparse [10] matrices and demonstrate superior performance.

II. Background And Related Work

A. Sparse Matrix-Vector Multiplication

SpMV refers to the multiplication of a sparse matrix $\boldsymbol{A}$ with a dense vector x to obtain a dense vector y. Equation (1) is used to represent SpMV, whereby y_i is obtained from a dot product between $\boldsymbol{A}_{i,*}$ and vector x. Conventional storage methods often waste space due to the numerous zeros present in sparse matrices [11]. Thus, many works use compressed formats to store non-zero information. Three common sparse storage formats include coordinate (COO), compressed sparse row (CSR), and compressed sparse column (CSC).

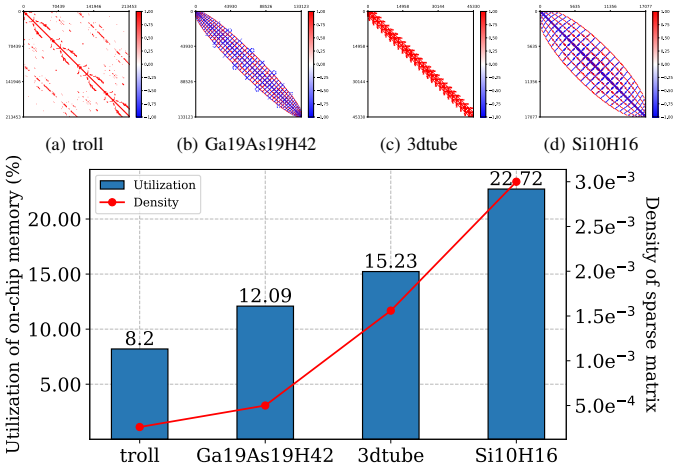
$$y_i = \sum_{j=0}^{\#Cols} \boldsymbol{A}_{i,j} \times x_j (\boldsymbol{A}_{i,j} \neq 0, 0 \leq i < \#Rows) \quad (1)$$

B. High Bandwidth Memory

HBM is an advanced memory solution for scientific computing and engineering applications. HBM achieves higher bandwidth than traditional DDR memory by vertically stacking multiple DRAM dies [6]. For example, the DDR4 memory in the Xilinx Alveo U250 provides four channels with a total bandwidth of 77 GB/s, while HBM based Xilinx Alveo U280 provides 32 channels and a total memory bandwidth of 460 GB/s. The advent of HBM presents a significant prospect for memory-intensive applications, like SpMV. Nevertheless, the hardware architecture needs to be carefully customized to fully exploit the potential of HBM for accelerating SpMV [12].

C. SpMV Accelerators on HBM-Equipped FPGAs

The hardware accelerator design for SpMV has garnered attention, especially with the introduction of HBM-equipped FPGAs, which expand the possibilities for optimization. HiSparse [6] employs a customized sparse matrix format and scalable on-chip buffers to improve bandwidth utilization, but the load-store forwarding inevitably introduces stall logic. Serpens [9] is a general SpMV accelerator based on HBM with memory-centric processing engines and index coalescing. However, due to the lack of consideration for blank structures in sparse matrices, Serpens writes redundant vector elements into on-chip memory. Furthermore, there are several works that leverage HBM FPGAs. Although they are not specifically customized for SpMV, they can support SpMV operations. GraphLily [7] designs the FPGA overlay to support a wide range of graph applications. However, certain general-purpose hardware allocations are idle during SpMV processing. Sextans [8] serves as an accelerator for generalized SpMM processing. For streaming, memory channels have to be allocated for three matrices. Therefore, utilizing Sextans directly for SpMV processing results in redundant bandwidth allocation.



(e) The trend of vector memory utilization with matrix density

Fig. 1: (a), (b), (c), and (d) demonstrate the spatial structure of four sparse matrices, (e) is input vector on-chip memory utilization of the matrices in Serpens.

III. Motivation

Designing a high-performance general SpMV accelerator utilizing the high bandwidth capabilities of HBM-equipped FPGAs presents several challenges, including:

- **Existing sparse storage formats pose challenges in fully exploiting the high bandwidth potential of HBM.** The pointer array in CSR/CSC prevents fully streaming accesses to non-zeros, the accelerator has to first access the pointer array before reading non-zeros. Also, the continuous storage of non-zeros prevents row/column vectorized.
- **Inherent RAW conflicts lead to low compute occupancy.** The accumulation phase of SpMV unavoidably causes RAW conflicts. The typical pipeline stall solution strategy leads to high latency and poor throughput.
- **Lack of efficient utilization of the input vector and on-chip memory**. The blank structures in sparse matrices, arising from irregularly distributed short and empty rows, lead to low vector reusability and redundant on-chip writes. As shown in Fig. 1(e), in Serpens [9], matrices with varying densities and spatial structures can impact the memory utilization of the input vector.

IV. Preprocessing And Dataflow

A. Proposed Sparse Storage Format

HBM's stacking layers introduce high latency. To benefit from the high bandwidth of HBM, streaming accesses must be ensured to amortize latency penalties. Indirect accesses of traditional sparse storage formats can prevent streaming accesses, resulting in poor bandwidth utilization. Therefore, we propose a sparse storage format to support streaming accesses to HBM channels and vectorized, as shown in Fig. 2(a). To alleviate the impact of blank structures in sparse matrices on the utilization of on-chip memory, we utilize sparse slices as the basic storage unit. The fixed partition size of slice structure

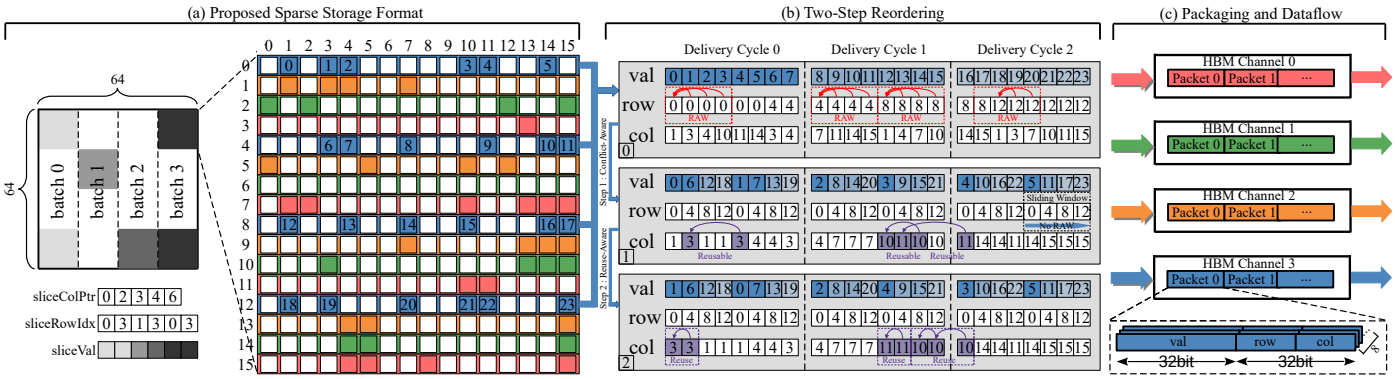


Fig. 2: An example matrix A of size 64-by-64 stored in six sparse slices of size 16-by-16. The slice structure contains three arrays `sliceColPtr`, `sliceRowIdx`, and `sliceVal` representing slice offsets, slice row indices, and non-zero information in sparse slice. The non-zeros in COO within a sparse slice are cyclically grouped by rows. A sliding window is set for each group, which is filled sequentially with non-conflicting non-zeros, based on which non-zeros with the same column index are aggregated. The reordered non-zeros are then packed into packets, forming dataflow distributed across the four HBM channels.

restricts the load between PEs to maintain a relatively balanced state. SpMV executed in column-major order is considered to have higher input vector reusability. Therefore, sparse slices are stored by CSC. Additionally, in order to reduce extra control overhead, we utilize COO to store non-zero information (`sliceVal`) in each sparse slice. Synchronized parsing of COO indices and values ensures full streaming accesses to each HBM channel. At the same time, the compact storage of COO supports vectorized delivery.

B. Two-Step Reordering Algorithm

To mitigate RAW conflicts in the SpMV accumulation phase and enhance input vector reusability, we propose a two-step reordering algorithm.

Step 1. Conflict-Aware Row Reordering: To fully utilize the 512-bit off-chip memory bandwidth of FPGAs (in the case of Xilinx FPGAs), we employ an index aggregation strategy. Due to the fixed size of the sparse slices, we can aggregate the row and column indices into 32-bit, enabling us to encode a single non-zero in 64-bit (i.e., 32-bit index and 32-bit value). With this strategy, eight non-zeros can be delivered to the chip per cycle. Thus, as illustrated in Fig. 2(b-0), it takes three cycles to deliver the 24 non-zeros (in blue). However, delivery in this order introduces a large number of RAW conflicts. We assume a 4-cycle latency (depending on the specific FPGAs) for the digital signal processing (DSP) to process a floating-point accumulation. As shown in cycle 0 of Fig. 2(b-0), the first four non-zeros all accumulate to the same address '0'. Due to DSP latency, the accumulation of the 2nd, 3rd, and 4th non-zeros introduces RAW conflicts with the 1st non-zero, leading to increased latency. To effectively mitigate RAW, we propose a conflict-aware row reordering algorithm. It involves filling a sliding window of width four with consecutively selected conflict-free row addresses from the initial sequence. If there is no more matching address, the window is temporarily made idle. In Fig. 2(b-1), following this reordering strategy that any four consecutive accumulations have no RAW conflicts.

Step 2. Reuse-Aware Column Reordering: We further analyze the reordered sequence and find that there are reusable vector elements in it, as shown by the purple in Fig. 2(b-1). Therefore, we propose a reuse-aware column reordering algorithm. With the priority of step 1 reordering algorithm, we gather non-zeros with the same column index greatest possible to improve data locality, and the sequence after reordering is shown in Fig. 2(b-2). Continuous reuse of vector elements is achieved by setting flexible reuse registers in the hardware architecture (Section V-C).

C. Dataflow Processing

Due to on-chip resource limitations, we employ a batched approach to perform SpMV. Fig. 2(a) illustrates this with the sparse matrix divided into four batches, processing one column (128 columns in this work) of sparse slices per batch. The next step is packaging, benefiting from the index aggregation strategy, we can pack eight non-zeros into a packet for vectorized delivery. These packets are then formed into dataflow and stored in the HBM channels. The last step is dataflow allocation, contiguous row dataflow allocation leads to channel conflicts among PEs, resulting in high access latency. Therefore, we cyclically allocate rows of dataflow to each channel, as shown in Fig. 2(c).

V. Hardware Architecture

A. Overall Architecture

The overall architecture of Cuper is illustrated in Fig. 3. The preprocessed dataflow from the host platform stream to the FPGAs through HBM channels. The dedicated computational cores array utilizes a matrix loader and crossbar switch to load sparse matrix $\boldsymbol{A}$ dataflow. Within each core, the perceptual decoder parses the index information and loads the input vector via the controller and vector fetcher. Then, the accumulator merges the product of the multiplication for the corresponding row address generated by the PE group. Upon completion of all batches processing, the partial sums are sorted in parallel

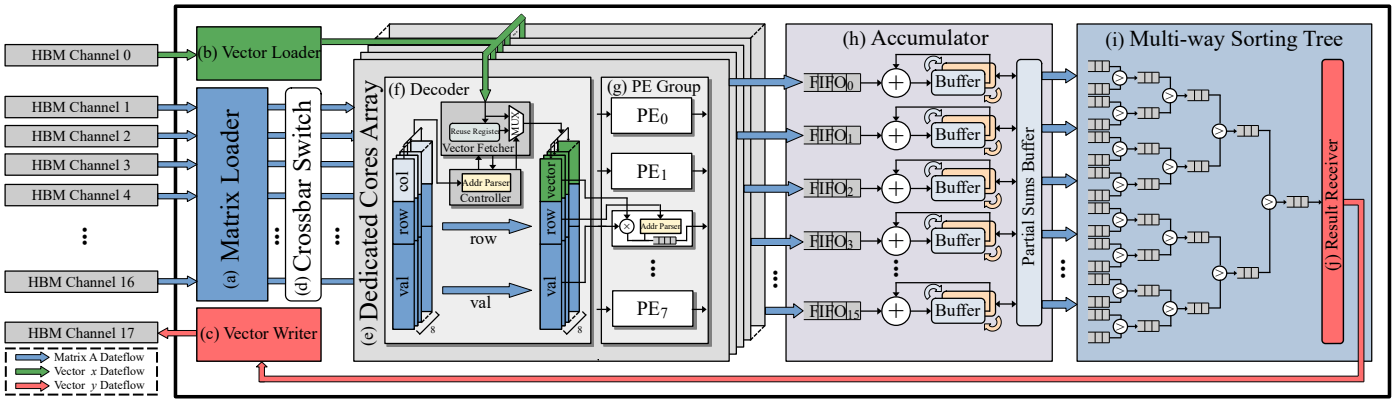


Fig. 3: Overall Architecture of Cuper.

by the multi-way sorting tree. After that, the result vector is delivered to the vector writer through the result receiver. Finally, the vector writer streams the final result to the host platform through HBM channels, concluding the SpMV computation.

B. Channel Allocation

Before performing SpMV, the sparse matrix $\boldsymbol{A}$ and the vector x in off-chip memory are streamed to on-chip memory. It is worth noting that we do not recommend allocating HBM channels equally, as the matrix size is typically much larger than the vector size. Thus, we allocate 16 HBM channels for reading sparse matrix $\boldsymbol{A}$, one channel for reading vector x, and one channel for writing back vector y. This allocation strategy is designed to optimize bandwidth overhead. Cuper uses 18 HBM channels for a bandwidth of 258 GB/s.

The loader and the writer perform streaming accessing. The matrix loader (Fig. 3(a)) supports highly concurrent dataflow reads and is interconnected to an array of dedicated computational cores via a crossbar switch, this ensures load balancing of each core. The vector loader (Fig. 3(b)) and the vector writer (Fig. 3(c)) have a 512-bit width configuration and vectorize 16 FP32 values in each cycle.

C. Dedicated Computational Cores Array

HBM provides massive memory channels. However, the global accesses between channels introduce high latency. Therefore, we configure 16 highly parallel cores (Fig. 3(e)) in dedicated computational cores array to facilitate independent concurrent access and optimize bandwidth utilization. These cores handle the SpMV multiplication, each containing a perceptual decoder (Fig. 3(f)) and a PE group (Fig. 3(g)).

1) Perceptual Decoder: The perceptual decoder consists of a controller and a vector fetcher. Initially, incoming packets are parsed into eight elements. Then each element is further separated into three parts: column index, row index, and value. The vector fetcher reads the input vector elements continuously based on the column addresses parsed by the controller. To implement the vector reuse strategy of the reordering algorithm, we set flexible reuse registers in the vector fetcher to save the last read vector value and column address. The controller verifies whether the column address of the vector to be acquired exists in the reuse register. If the match successes, the controller schedules the multiplexer (MUX) to directly export the vector element from the reuse register in order to shorten the read path. If the match failures, the vector fetcher reads vector from the vector loader and saves them to the block RAM (BRAM) for computation, while the contents of the reuse registers are replaced. After that, the decoder will pack the vector element, row index, and value and send them to the PE group. Because the proposed sparse storage format is based on sparse slices, the perceptual decoder can skip the blank structures to reduce redundant vector writes for higher on-chip memory utilization.

2) PE Group: The proposed sparse storage format ensures efficient synchronization parsing of address and value parsing, enabling the design of a pipelined PE for high processing throughput. Each packet contains eight groups of non-zeros. Therefore, eight PEs are configured in a PE group and each PE performs the multiplication operation of the matrix value with the vector value. As there is no data dependency during the multiplication phase of SpMV, all PEs can compute in parallel. The row index is then parsed as an address and pushed into the FIFO along with the multiplication result, which is subsequently streamed to the accumulator by the core.

D. Accumulator

As shown in Fig. 3(h), we design an accumulator for the fast merging of partial sums. The accumulator obtains the multiplication results from the dedicated cores array. Due to the speed mismatch between data delivery and floating-point accumulations, we set up a FIFO for every adder to stockpile awaiting inputs and convey new inputs from the FIFO upon the adder completing the present computation. The accumulated results are promptly stored in the partial sums buffer, composed of ultra RAM (URAM), to support quick random accesses. Moreover, the batched approach of SpMV introduces memory switching latency between batches, for which we set up ping-pong buffers to cover this latency overhead. Once the current batch concludes, the partial sums stored in the ping buffer (in blue) are ready to be written to the partial sums buffer, and the pong buffer (in orange) is switched to calculate the partial sums of the next batch. By adopting the above switching principle, the latency in memory switching between batches is concealed.

E. Multi-way Sorting Tree

To ensure the ordering of the result vector, we design a parallel multi-way sorting tree, as shown in Fig. 3(i). The multi-way sorting tree consists of FIFOs and comparators. The comparators direct the vector elements with smaller addresses in the sub-node buffer to the parent-node buffer. Upon completion of the sorting process, the result receiver (Fig. 3(j)) delivers the result vector to the vector writer, which streams the final result to off-chip memory through an HBM channel.

VI. Evaluation

A. Experiment Setup

1) FPGAs Baselines: We evaluate Cuper and the four state-of-the-art SpMV accelerators leveraging HBM FPGAs - HiSparse [6], GraphLily [7], Sextans [8], and Serpens [9]. Table I lists the frequency, number of HBM channels, memory bandwidth, and power specification for the evaluated accelerators.

TABLE I: The specification of the evaluated accelerators.

Accelerator	Frequency	#Channels	Bandwidth	Power
HiSparse [6]	237 MHz	18 HBM	258 GB/s	45 W
GraphLily [7]	166 MHz	19 HBM	285 GB/s	43 W
Sextans [8]	197 MHz	29 HBM	417 GB/s	52 W
Serpens [9]	223 MHz	19 HBM	273 GB/s	48 W
Nvidia Tesla K80 GPU	562 MHz	-	480 GB/s	130 W
Cuper (this work)	205 MHz	18 HBM	258 GB/s	41 W

We develop Cuper using Vivado High-Level Synthesis (HLS) C++ and implement with Vitis Toolchain 2021.2. The resource utilization of Cuper is reported in Table II. For Hisparse, GraphLily, Sextans, and Serpens, we obtain the provided open-source bitstream (.xclbin). We run Cuper and the four SpMV accelerators on a Xilinx Alveo U280 FPGA.

TABLE II: Resource utilization of Cuper on a Xilinx Alveo U280 FPGA.

LUT	FF	DSP	BRAM	URAM
307K (26.4%)	314K (13.5%)	920 (10.8%)	1024 (29.2%)	512 (53.3%)

2) GPUs Baselines: We also compare Cuper with an Nvidia Tesla K80 GPU, and the parameters of K80 GPU are detailed in Table I. To evaluate SpMV on K80 GPU, we use the official kernel `csrmv` in cuSPARSE with CUDA 9.0.

3) Datasets: To compare Cuper with the state-of-the-art FPGAs accelerators, we select 12 large-size matrices from the SuiteSparse Matrix Collection [10] for evaluation, which come from different fields such as 2D/3D problem, structural problem, and graph problem, etc. Table III shows the detailed information of the evaluated matrices, where **#slices** indicates the number of sparse slices. For the comparison of Cuper with K80 GPU, we select 2,757 matrices from SuiteSparse. The number of rows ranges from 1 to 1.5M and the number of non-zeros can be as high as 113M.

4) Metrics: (1) **Throughput**, measured as the number of billion floating-point operations per second (GFlops). (2) **Bandwidth efficiency**, measured by throughput per unit of memory bandwidth (MFlops/(GB/s)). (3) **Energy efficiency**, measured by throughput per watt (MFlops/W). In this experiment, we evaluate FP32 SpMV. We use `Xilinx Runtime (XRT)` to measure the average execution time for 50 runs of accelerators on U280 FPGA and `xbutil` to measure power consumption. We use `cudaEventElapsedTime` to measure the average execution time of SpMV on K80 GPU and `nvidia-smi` to measure power consumption.

TABLE III: The information of the 12 evaluated matrices.

Matrix	Size	#non-zeros	Density	#slices	Field
sit100	10K	61K	5.7E-4	1K	2D/3D
olafu	16K	1M	3.8E-3	1K	Structural
Si10H16	17K	875K	3.0E-3	9K	Theoretical Chemistry
finance256	37K	298K	2.1E-4	3K	Optimization
3dtube	45K	3M	1.5E-3	13K	CFD
crankseg_2	63K	14M	3.4E-3	61K	Structural
Si34H36	97K	5M	5.4E-4	72K	Theoretical Chemistry
mycielskian17	98K	100M	1.0E-2	282K	Undirected Graph
Ga19As19H42	133K	8M	5.0E-4	118K	Theoretical Chemistry
troll	213K	11M	2.6E-4	45K	Structural
web-BerkStan	685K	7M	1.6E-5	180K	Directed Graph
webbase-1M	1M	3M	3.1E-6	196K	Weighted Directed Graph

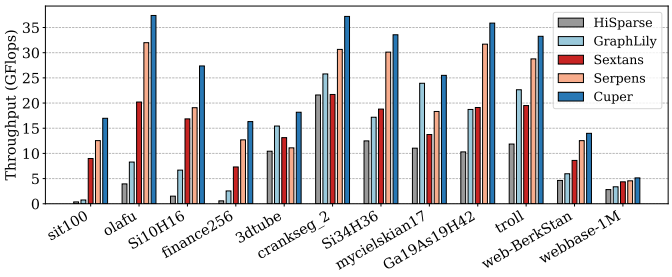


Fig. 4: Throughput comparison of the four SpMV accelerators with Cuper on the 12 evaluation matrices.

B. Comparison with FPGAs Accelerators

1) Throughput: We use throughput to normalize the performance of different accelerators. Throughput is calculated with (#non-zeros)/(execution time). GraphLily and Sextans lack deeper SpMV customization, resulting in longer execution time and lower throughput. In the case of HiSparse and Serpens, inevitable stall logic and redundant vectors loaded into on-chip memory consume part of the execution time. In Fig. 4, our performance evaluation on 12 large matrices reveals that Cuper's geomean throughput is 3.28×, 1.99×, 1.75×, and 1.44× higher compared with HiSparse, GraphLily, Sextans, and Serpens, respectively.

2) Bandwidth Efficiency: Compared with FPGAs accelerators, Cuper utilizes fewer HBM channels, resulting in a relatively lower peak bandwidth. However, with customized dataflow and a dedicated computational cores array, Cuper achieves higher bandwidth efficiency. Table IV shows that the geomean bandwidth efficiency of Cuper is improved over HiSparse, GraphLily, Sextans, and Serpens by 3.28×, 2.20×, 2.82×, and 1.31×, respectively. Cuper attains a remarkable bandwidth efficiency of 144.93 MFlops/(GB/s) on `olafu`.

TABLE IV: Bandwidth efficiency and energy efficiency of HiSparse [6], GraphLily [7], Sextan [8], Serpens [9], and Cuper on the 12 evaluated matrices. The improvement is the performance enhancement of Cuper over Serpens.

Matrix	Bandwidth efficiency (MFlops/(GB/s))						Energy efficiency (MFlops/W)					
	HiSparse	GraphLily	Sextans	Serpens	Cuper	Improvement	HiSparse	GraphLily	Sextans	Serpens	Cuper	Improvement
sit100	1.43	2.63	21.56	45.93	65.79	1.43×	8.21	17.45	172.90	261.22	414.00	1.58×
olafu	15.29	29.06	48.45	117.17	144.93	1.24×	87.69	192.67	388.55	666.43	912.04	1.37×
Si10H16	5.86	23.45	40.42	69.87	106.07	1.52×	33.62	155.43	324.16	397.44	667.52	1.68×
finance256	2.24	8.94	17.53	46.48	63.24	1.36×	12.87	59.27	140.59	264.38	397.99	1.51×
3dtube	40.45	54.16	31.48	40.67	70.45	1.73×	231.94	358.99	252.51	231.34	443.32	1.92×
crankseg_2	83.73	90.46	52.05	112.29	144.20	1.28×	480.10	599.57	417.42	638.65	907.45	1.42×
Si34H36	48.40	60.29	45.09	110.30	130.07	1.18×	277.51	399.59	361.59	627.37	818.54	1.30×
mycielskian17	42.79	83.95	32.96	67.13	98.84	1.47×	245.34	556.45	264.39	381.83	621.98	1.63×
Ga19As19H42	39.92	65.72	45.84	116.10	139.06	1.20×	228.90	435.58	367.66	660.34	875.08	1.33×
troll	45.96	79.44	46.76	105.38	128.89	1.22×	263.55	526.53	375.03	599.39	811.12	1.35×
web-BerkStan	17.95	20.90	20.64	45.91	54.21	1.18×	102.96	138.55	165.57	261.13	341.15	1.31×
webbase-1M	10.90	11.83	10.44	16.69	19.91	1.19×	62.52	78.42	83.78	94.95	125.33	1.32×

3) Energy Efficiency: Our reordering algorithm and perceptual decoder improve vector reusability, mitigating redundant on-chip memory reads and writes, Cuper has lower power consumption. As shown in Table IV, Cuper demonstrates 3.59×, 2.08×, 2.21×, and 1.44× improvements in geomean energy efficiency compared with the four accelerators, respectively. Cuper achieves an improvement over Serpens of up to 1.92×.

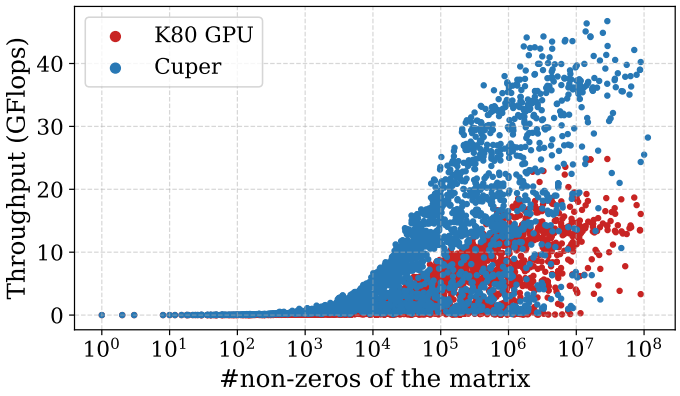


Fig. 5: SpMV throughput comparison between K80 and Cuper GPU on 2,757 evaluated matrices.

C. Comparison with K80 GPU

We evaluate an Nvidia Tesla K80 GPU and Cuper on 2,757 SuiteSparse matrices with varying dimensions. As shown in Table I, K80 GPU is more advanced in frequency and memory bandwidth than Cuper. The SpMV throughputs of K80 GPU and Cuper is shown in Fig. 5. Cuper demonstrates superior performance on almost all matrices. The maximum throughputs of K80 GPU and Cuper are 24.81 GFlops and 46.74 GFlops. The geomean throughput of Cuper compared with K80 GPU is 2.51×. Furthermore, the geomean energy efficiency of Cuper is 7.97× higher than K80 GPU.

VII. Conclusion

In this paper, we have proposed Cuper, a high-performance SpMV accelerator on HBM-equipped FPGAs. We customized dataflow for full utilization of HBM. To effectively mitigate RAW conflicts and improve vector reusability, we designed a two-step reordering algorithm. The perceptual decoder-centric hardware architecture further improved the reusability and on-chip memory utilization. The evaluation showed that Cuper demonstrates significant enhancements in throughput, bandwidth efficiency, and energy efficiency compared with the four state-of-the-art SpMV accelerators and K80 GPU.

VIII. Acknowledgment

Zhou Jin and Weifeng Liu are the corresponding authors of this paper. This work was supported by the National Natural Science Foundation of China (Grant No. U23A20301, 62204265) and the State Key Laboratory of Computer Architecture (ICT, CAS) (Grant No. CARCHA202115). We are also very grateful to AMD under the Heterogeneous Accelerated Compute Clusters (HACC) program.

References

[1] J. Park, W. Yi, D. Ahn, J. Kung, and J.-J. Kim, "Balancing computation loads and optimizing input vector loading in lstm accelerators," *TCAD*, 2019.

[2] K. Lu, Z. Li, L. Liu, J. Wang, S. Yin, and S. Wei, "Redesk: A reconfigurable dataflow engine for sparse kernels on heterogeneous platforms," in *ICCAD*, 2019.

[3] S. Li, D. Liu, and W. Liu, "Optimized data reuse via reordering for sparse matrix-vector multiplication on fpgas," in *ICCAD*, 2021.

[4] B. Liu and D. Liu, "Towards high-bandwidth-utilization spmv on fpgas via partial vector duplication," in *ASP-DAC*, 2023.

[5] C. Su, H. Liang, W. Zhang, K. Zhao, B. Ai, W. Shen, and Z. Wang, "Graph sampling with fast random walker on hbm-enabled fpga accelerators," in *FPL*, 2021.

[6] 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 *FPGA*, 2022.

[7] Y. Hu, Y. Du, E. Ustun, and Z. Zhang, "Graphlily: Accelerating graph linear algebra on hbm-equipped fpgas," in *ICCAD*, 2021.

[8] 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 *FPGA*, 2022.

[9] L. Song, Y. Chi, L. Guo, and J. Cong, "Serpens: A high bandwidth memory based accelerator for general-purpose sparse matrix-vector multiplication," in *DAC*, 2022.

[10] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *TOMS*, 2011.

[11] Y. Niu, Z. Lu, M. Dong, Z. Jin, W. Liu, and G. Tan, "Tilespmv: A tiled algorithm for sparse matrix-vector multiplication on gpus," in *IPDPS*, 2021.

[12] Y.-k. Choi, Y. Chi, W. Qiao, N. Samardzic, and J. Cong, "Hbm connect: High-performance hls interconnect for fpga hbm," in *FPGA*, 2021.