SERPENS: A High Bandwidth Memory Based Accelerator for General-Purpose Sparse Matrix-Vector Multiplication

Linghao Song, Yuze Chi, Licheng Guo, and Jason Cong
{linghaosong,chiyuze,lcguo,cong}@cs.ucla.edu
University of California, Los Angeles

ABSTRACT

Sparse matrix-vector multiplication (SpMV) multiplies a sparse matrix with a dense vector. SpMV plays a crucial role in many applications, from graph analytics to deep learning. The random memory accesses of the sparse matrix make accelerator design challenging. However, high bandwidth memory (HBM) based FPGAs are a good fit for designing accelerators for SpMV. In this paper, we present SERPENS, an HBM based accelerator for general-purpose SpMV, which features memory-centric processing engines and index coalescing to support the efficient processing of arbitrary SpMVs. From the evaluation of twelve large-size matrices, SERPENS is 1.91× and 1.76× better in terms of geomean throughput than the latest accelerators GraphLiLy and Sextans, respectively. We also evaluate 2,519 SuiteSparse matrices, and SERPENS achieves 2.10× higher throughput than a K80 GPU. For the energy/bandwidth efficiency, SERPENS is 1.71×/1.99×, 1.90×/2.69×, and 6.25×/4.06× better compared with GraphLily, Sextans, and K80, respectively. After scaling up to 24 HBM channels, SERPENS achieves up to 60.55 GFLOP/s (30,204 MTEPS) and up to 3.79× over GraphLily. The code is available at https://github.com/UCLA-VAST/Serpens.

1 INTRODUCTION

SpMV performs the computation of $\vec{y} = \alpha \cdot \mathbf{A} \times \vec{x} + \beta \cdot \vec{y}$ where $\vec{x}$ and $\vec{y}$ are two dense vectors, $\mathbf{A}$ is a sparse matrix, and α, β are two scalar constants. SpMV is the core computation routine in a wide range of applications, such as linear systems solvers [24] in scientific computing, the processing model [19] in graph analytics, and inference of sparse neural networks [16]. In the acceleration of dense algebra, the tensor size determines the data movement and thus researchers can use an analytic model to coordinate the computation to achieve very high performance [30]. However, it is difficult to accelerate SpMV because: (i) The vector $\vec{x}$ has only one element at each index that significantly prevents reuse in computation. Thus, it is hard to achieve a high computation throughout. (ii) The irregular distribution of non-zeros in the sparse matrix $\mathbf{A}$ leads to random memory accessing. The memory hierarchy faces high pressure from the random accessing. If we do not optimize the accelerator's memory, the performance will be even lower.

This work is supported in part by the NSF RTML Program (CCF-1937599), CDSC industrial partners (https://cdsc.ucla.edu/partners), and the Xilinx XACC Program.

DAC '22, July 10–14, 2022, San Francisco, CA, USA

ACM ISBN 978-1-4503-9142-9/22/07.
https://doi.org/10.1145/3489517.3530420

High bandwidth memory (HBM) [3] exposes more memory channels to users than conventional DDR memory. HBM-based FPGAs enable accelerator design and evaluation with high memory bandwidth. It is an opportunity to accelerate memory-intensive applications including SpMV with HBM FPGAs. GraphLily [18] and Sextans [27] are two of the latest accelerators leveraging HBM FPGAs for sparse workloads. GraphLily is an FPGA overlay to support graph applications which can be described by an SpMV BLAS processing model. Sextans is an accelerator for sparse matrix-matrix multiplication (SpMM). However, there are a few limitations in existing works for SpMV acceleration: (i) GraphLily overlay deploys extra hardware resource to support generalized operations, but many of the hardwarelized operations are idle in SpMV. So the FPGA hardware is not fully customized for SpMV acceleration. (ii) Sextans has to allocate memory channels to one sparse matrix and two dense matrices in SpMM processing. However, for SpMV acceleration, we can save some memory channels for dense matrices because the dense vector size is smaller than the dense matrix size. Thus, we can allocate more memory channels to speed up the processing of the sparse matrix. (iii) There lacks a modern FPGA-based accelerator for general-purpose SpMV as we treat FPGAs as a competitive candidate to GPUs for computing in data center.

We present SERPENS, a high bandwidth memory-based FPGA accelerator for general-purpose SpMV acceleration. Our features include: (i) SERPENS is an HBM FPGA accelerator for general-purpose[1] SpMV. SERPENS supports arbitrary spares matrices and achieves competitive performance to GPUs. (ii) The memory-centric processing engines (PEs) of SERPENS enable efficient streaming of sparse matrices. We partition the input dense vector into segments and accumulate output dense vector on chip to process in an output stationary [29] manner. Thus, we limit the random memory to on-chip BRAMs/URAMs to avoid the high latency of random off-chip memory accessing. The memory-centric PEs also make SERPENS a scalable architecture. (iii) SERPENS uses an index coalescing to fully utilize on-chip FPGA URAMs to support large-size problems. Similar to prior works [18, 27, 28], we preprocess the spares elements into accelerator-efficient storage. (iv) We conduct comprehensive evaluations. We evaluate on 12 million-level matrices, SERPENS surpasses GraphLiLy by 1.91× and Sextans by 1.76× in terms of geomean throughput. In terms of energy efficiency, SERPENS is 1.71× better than GraphLiLy and 1.90× better than Sextans. We evaluate on 2,519 SuiteSparse [10] matrices, SERPENS is 2.10× in terms of throughput, 6.25× in terms of and energy efficiency, and 4.06× in terms of bandwidth efficiency better than a K80 GPU. After scaling up to 24 HBM channels, SERPENS achieves up to 60.55 GFLOP/s (30,204 MTEPS) and up to 3.79× over GraphLily.

[1] For 'general-purpose' we mean that the accelerator (1) supports a general form SpMV $\vec{y} = \alpha \cdot \mathbf{A} \times \vec{x} + \beta \cdot \vec{y}$ and (2) can run an arbitrary SpMV without re-doing prototyping.

2 BACKGROUND AND MOTIVATION

2.1 High Bandwidth Memory

Conventional DDR memory provides limited memory bandwidth. For example, the DDR4 memory of a Xilinx Alveo U250 [1] provides four channels and a total bandwidth of 77 GB/s. Accelerators for computation-intensive applications such as deep learning accelerators [30] are able to achieve high performance with DDR memory. However, in SpMV and many related graph applications the data reuse is low and there are a large amount of random memory accesses. Such applications are memory-intensive and require the support of high memory bandwidth. HBM based accelerator Xilinx Alveo U280 [2] provides 32 channels and a total memory bandwidth of 460 GB/s, which is a good opportunity for the acceleration of memory-intensive applications. However, it is non-trivial to achieve efficient HBM channel interconnection [7, 8]. Accelerator architects need to customize their accelerators to fit the HBM channels to fully reap the bandwidth benefit.

2.2 SpMV Accelerators on HBM FPGAs

GraphLily [18] and Sextans [27] are two of the latest accelerators related to Serpens. They are both accelerators based on HBM FPGAs. Although they are not specialized for SpMV, they are able to support SpMV processing.

GraphLily [18] uses a BLAS-based processing model [19] which represents graph applications in a generalized SpMV to design an FPGA overlay as a general accelerator for graph processing. To run different graph applications, GraphLily configures the data type, the generalized binary multiplication, and the generalized reduction. For example, to support a floating-point SpMV, GraphLily sets the data type to float and maps the generalized binary multiplication to arithmetic multiplication and the reduction to arithmetic addition. SpMV never uses the other hardware instances of the generalized operations. Moreover, GraphLily deploys an arbiter vector unit to load data from off-chip memory and supply it to processing engines which have no bank conflicts. The arbiter vector unit is flexible in BFS and SSSP. However, for SpMV processing, we know the vector accessing sequence in advance. Thus, GraphLily does not fully customize the vector handling for SpMV. Nevertheless, the overlay makes GraphLily support a wide spectrum of graph applications including BFS, SSSP, and PageRank besides SpMV.

For SpMM acceleration, Sextans [27] balances the allocation of memory channels to one sparse and two dense matrices, because SpMM needs to stream on three large matrices of comparable sizes. Specifically, Sextans allocates 8 channels for the sparse matrix and 20 channels for the two dense matrices. Besides, to achieve a high computation throughput, Sextans shares a sparse elements with eight dense matrix elements. The sharing consumes FPGA logic resource and on-chip BRAMs. However, the dense vectors in SpMVs are quite smaller than the dense matrices in SpMM. To support an SpMV run, Sextans configures $N = 8$ to run an SpMM and retires the first column vector of the SpMM output as the result of SpMV. However, we can allocate less channels for the vector and more channels for the sparse matrix. Moreover, an SpMV accelerator can save the on-chip logic and memory resource previously used for the SpMM sharing.

2.3 Other Related Works

SpaceA [31] is a hybrid memory cube-based accelerator architecture for SpMV. Tensaurus [28] is an HMC based accelerator for sparse-dense linear algebra. GraphR [26] utilizes an SpMV processing model for graph acceleration. However, the three accelerators are evaluated by simulation rather than real execution. Fowers et al. [12] designed an FPGA accelerator for SpMV, but it is on DDR memory and the performance is poor. HitGraph [32] and ThuderGP [5] are FPGA accelerators for graph processing, but they utilize DDR memory and are no specialized for SpMV.

Data format/layout reorganization is a common technique for boosting systems performance. For example, CSR5 [22] and HiCOO [21] are data formats to accelerate SpMV and sparse tensor processing on muti-core CPUs and GPUs. TensorFlow [4] stores data in TFRecord format for fast processing. For accelerators, [12, 18, 27], and [28] all reorganize data format/layout to be accelerator friendly.

2.4 Motivation

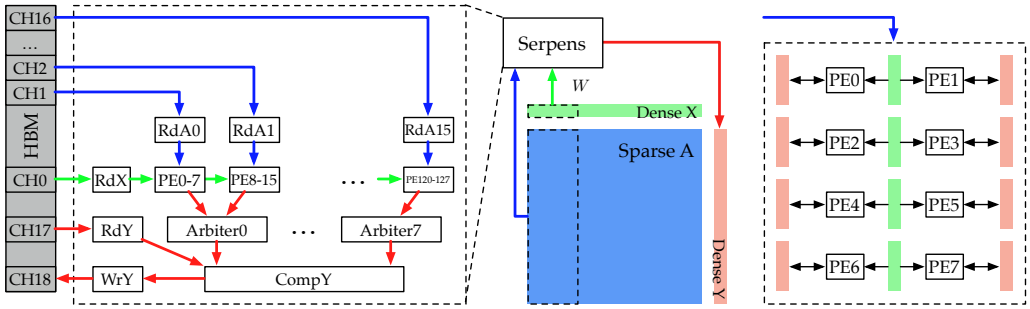
The lack of modern HBM-FPGA-based accelerators for SpMV motivates us to develop Serpens. Serpens utilizes massive HBM memory channels for high-throughput processing of sparse matrices. We customize the storing and sharing of dense vectors for the need of SpMV to fully utilize FPGA resource. We architect Serpens as a general-purpose accelerator to deliver competitive SpMV performance to GPUs for data-center computing.

3 SERPENS ACCELERATOR

3.1 Accelerator Architecture

3.1.1 HBM Channel Allocation. Figure1 (a) shows the overall architecture of the Serpens accelerator. For the off-chip memory accessing in SpMV, Serpens needs to (1) stream in the sparse $\mathbf{A}$ matrix, (2) stream in the dense $\vec{x}$ vector, and (3) stream in the dense $\vec{y}$ vector and write the result $\vec{x}$ vector. The dense vector size is much smaller than the sparse matrix size. For example, the matrix `hollywood` is 1.25 GB while the corresponding dense vector is 4 MB. Thus, we allocate one HBM channel for each dense vector, i.e., $\vec{x}$, input $\vec{y}$, and output $\vec{y}$, and sixteen HBM channels for the sparse $\mathbf{A}$ matrix. In total, Serpens occupies 19 HBM channels and the accumulative memory bandwidth is 273 GB/s.

3.1.2 Serpens Modules. We deploy a read (Rd) or write (Wr) module for each HBM channel. The Rd/Wr modules performs streaming memory accessing to off-chip HBM. The bitwidth of the Rd/Wr modules is 512. For the dense vector Rd/Wr modules, we coalesce 16 floating-point values into a 512-bit segment. For one sparse element, each of the row index, column index, and float attribute occupies 32 bits. Because we partition the vectors and matrices in SpMV processing (Sec.3.2), the indices are limited in a range at each iteration. Thus, we reduce the index bits and compress a row-column index pair into 32 bits to save memory bandwidth. We encode a sparse element with 64 bits. As a result, for the sparse matrix read module, we coalesce eight sparse elements into a 512-bit segment. Serpens deliver the spares elements from one HBM channel to 8 processing engines (PEs). One PE performs part of the matrix-vector multiplication $\mathbf{A} \times \vec{x}$. We use one arbiter to select computation results from 16 PEs and send the result to a CompY module. The CompY module



(a) Accelerator architecture. (b) Overall processing. (c) 8 PEs connected to one A channel.

Figure 1: SERPENS accelerator architecture and matrix-vector processing.

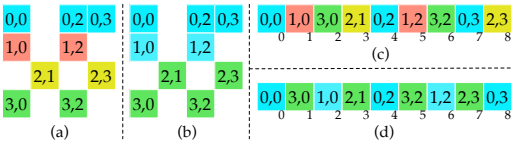


Figure 2: An example of "coloring" and reordering in Sextans [27]: (a), (c), and SERPENS: (b), (d). We assume the DSP latency is 2.

performs the element-wise (α, β) multiplications and additions to obtain final SpMV results.

3.2 SpMV Processing Order

Figure1 (b) shows the overall processing order in SERPENS. We partition the dense vector $\vec{x}$ into segments. The segment length $W = 8192$. In the processing, we stream in one $\vec{x}$ segment and store it in BRAMs. Then we stream in sparse $\mathbf{A}$ elements associated to the $\vec{x}$ segment. For the accumulation of $\mathbf{A} \times \vec{x}$, we use URAMs as accumulation buffers. After we finish the processing on one $\vec{x}$ segment, we iterate on next segment. With the partition and SpMV processing order, the benefits include: (i) SERPENS does not issue any random access to off-chip memory. All the off-chip memory accessing is sequential. Thus, SERPENS can fully utilize the off-chip memory bandwidth and amortise the latency of off-chip memory accessing. (ii) There are two kind of random accesses in SpMV: (1) irregularly reading elements from the dense vector $\vec{x}$ and (2) accumulating on vector $\vec{y}$. Now, SERPENS limits the two random accessing on chip. With the optimization of II=1 pipeline, we can achieve a high processing throughput. (iii) SERPENS will read/write any of the vectors and the sparse matrix only once without any duplicate reading/writing. Thus, SERPENS minimizes the off-chip communication.

3.3 Memory-Centric Processing Engines

HBM provides massive memory channels to users. However, the switching/crossing between HBM channels requires special handling to achieve high memory throughput [8]. SERPENS avoid one module to access cross multiple HBM channels by the memory-centric processing engines as shown in Figure1 (c). SERPENS distributes the sparse elements from one channel to 8 PEs. We implement the broadcasting of the dense vector segment from one channel to all PEs using a chain topology [9] to achieve a higher frequency [14]. There are four copies of the the dense vector $\vec{x}$ segment stored in BRAMs. Since each BRAM has two ports, we share one BRAM with two PEs to save half of the BRAM usage.

Table 1: The design parameters of SERPENS accelerator.

Architecture			
HBM Channels	PEs/Channel	BRAM18Ks/PE	URAMs/PE
$H_A = 16/24$	8	128	$U = 3$

Bit-Width			
Memory Bus	Data	Index	Instruction
512	32(float)	32(row+col.)	32

There is no bank conflict when the 8 PEs fetch dense elements from different BRAM addresses. We make URAM address for each PE disjoint to avoid the URAM bank conflict caused by the accessing from multiple PEs.

The memory-centric PEs also enable rapid scalability. Users can easily customize SERPENS according to their need on various memory channels and bandwidths. We show the performance of SERPENS when we scale up channel allocation for the sparse matrix from 16 to 24 in Sec. 4.4.

3.4 Index Coalescing & Non-Zero Reordering

Index coalescing is a micro-architecture level optimization we apply to improve URAM utilization. The minimum bit width of a URAM configuration is 72. It is a waste to store a 32-bit float (FP32) value to one URAM address (entry). Thus, we coalesce two values whose destination row indices are consecutive into one common URAM address. In the reordering of non-zeros (sparse elements), besides read-after-write (RAW) conflict [27], we also need to handle the URAM access conflicts caused by coalesced indices.

Figure2 compares the non-zero reordering in Sextans [27] and in SERPENS. We assume the latency T of DSP accumulation on a float is 2 cycles. We view the recording in Sextans [27] as a process of two steps – coloring and reordering. The coloring step colors elements in the same row with the same color shown in Figure 2 (a). In the recording step, we ensure none of the same color elements are in any of the T-cycle windows (Figure 2 (c)). After we applied the index coalescing, we store two elements of consecutive row indices to the same URAM address. To avoid URAM address conflicts, we only need to color elements of two consecutive rows with the same color shown in Figure 2 (b) and then apply the same reordering rule for SERPENS. Figure 2 (d) illustrates the reordered non-zeros taking both RAW and index coalescing conflicts into consideration. We summarize the design parameters of SERPENS accelerator in Table 1.

3.5 Resource & Performance Analysis

We estimate SERPENS BRAM/URAM consumption and cycle count when achieving an II=1 pipeline.

3.5.1 BRAM Consumption. When streaming in the dense vector and storing the vector to BRAMs, one 512-bit block contains 16 FP32 values. The bitwidth of a BRAM18K is 18, so we need 2 BRAM18Ks to store one FP32. For the whole 512 bits, we require 32 BRAM18Ks. Since one BRAM18K has two ports, we reduce the BRAM18K number to 16. When streaming in sparse elements, for each memory channel, we dispatch 8 sparse elements to 8 PEs per cycle. Thus we need $16 \times 8 = 128$ BRAM18Ks. Because we share one BRAM with two PEs in Serpens, the actual number of BRAM18Ks per channel is 64. We assume there are H_A HBM memory channels allocated to the sparse matrix **A**. In total, we require $64 \cdot H_A$ BRAM18Ks. One BRAM contains two BRAM18Ks on Xilinx FPGAs. Thus, the number of BRAMs is:

$$\#\text{BRAMs} = 32 \cdot H_A. \tag{1}$$

3.5.2 URAM Consumption. Assuming that we assign U URAMs to each PE. There are $8 \cdot H_A$ PEs in total, because URAMs are disjoint for different PEs, the total number of URAMs is:

$$\#\text{URAMs} = 8 \cdot H_A \cdot U. \tag{2}$$

Assuming the depth of a URAM configured by a wdith of 72 bits is D, with the index coalescing in Serpens, the on-chip accumulation row depth is:

$$\#\text{Row Depth} = 16 \cdot H_A \cdot U \cdot D. \tag{3}$$

3.5.3 Cycle Count. We assume the row number, column number, and number of non-zeros (sparse elements) of the sparse matrix is M, K, and NNZ, respectively.

Because we allocate one memory channel to the dense vector $\vec{x}$, the cycle count of streaming in vector $\vec{x}$ is $K/16$. Serpens performs the streaming in the dense input vector $\vec{y}$ and the streaming the out dense output vector $\vec{y}$ in parallel, thus the cycle count on the two dense $\vec{y}$ vectors is $M/16$.

In the computation, at each cycle, one PE processes 8 sparse elements. Because there are $8 \cdot H_A$ PEs in total, the cycle count for processing the sparse elements is $\text{NNZ}/(8 \cdot H_A)$.

We add up the streaming and computation cycle counts to obtain the overall cycle count:

$$\#\text{Cycle} = (M + K)/16 + \text{NNZ}/(8 \cdot H_A). \tag{4}$$

4 EVALUATION

4.1 Evaluation Setup

4.1.1 The Evaluated Accelerators. We evaluate the SpMV routine on Serpens and two FPGA-related accelerators – Sextans [27], GraphLily [18], and an Nvidia Tesla K80 GPU. Table 2 lists the frequency, memory bandwidth and power of the four evaluated accelerators.

We describe Serpens accelerator in Xilinx high level synthesis (HLS) C++ and prototype with Vitis 2020.2. For Sextans, we utilize the open-sourced code and prototype with Vitis 2020.2. For GraphLily, we obtain the open-sourced bitstream (.xclbin). We run the three FPGA accelerators on a Xilinx Alveo U280 FPGA board then measure the execution time by Xilinx Run Time and the power consumption by `xbutil`. To perform SpMV on GPU, we use CuSPARSE [23] routine `csrmv` with CUDA 10.1. We measure the GPU execution time by `cudaEventElapsedTime` and power consumption by `nvidia-smi`. We amortize the execution time by 100 runs.

Table 2: The specification of the evaluated accelerators.

	Sextans [27]	**GraphLily** [18]	**Serpens**	**Tesla K80**
Frequency	197 MHz	166 MHz	223 MHz	562 MHz
Bandwidth	&417 GB/s	&285 GB/s	&273 GB/s	#480 GB/s
Power	52 W	43 W	48 W	130 W

& Utilized bandwidth, # maximum bandwidth.

Table 3: The specification of evaluated matrices.

Twelve Large Matrices/Graphs			
ID	**Matrix**	**#Vertices**	**#Edges**
G1	`googleplus` [20]	108 K	13.7 M
G2	`crankseg_2` [10]	63.8 K	14.1 M
G3	`Si41Ge41H72` [10]	186 K	15.0 M
G4	`TSOPF_RS_b2383` [10]	38.1 K	16.2 M
G5	`ML_Laplace` [10]	377 K	27.6 M
G6	`mouse_gene` [10]	45.1 K	29.0 M
G7	`soc_pokec` [20]	1.63 M	30.6 M
G8	`coPapersCiteseer` [10]	434 K	21.1 M
G9	`PFlow_742` [10]	743 K	37.1 M
G10	`ogbl_ppa` [17]	576 K	42.5 M
G11	`hollywood` [20]	1.07 M	113 M
G12	`ogbn_products` [17]	2.45 M	124 M

SuiteSparse [10] Matrices			
Number of Matrices	2,519	**NNZ**	1,000 – 89,306,020
Row/column	24 – 2,999,349	**Density**	8.75E-7 – 1

GraphLily employs 19 HBM channels and 1 DDR4 channel, translating to 285 GB/s memory bandwidth. Sextans uses 29 HBM channels and the bandwidth is 417 GB/s. Serpens-A16 uses 19 HBM channels for a bandwidth of 273 GB/s.

4.1.2 The Evaluated Matrices. For the comparison of Serpens with Sextans and GraphLily, we evaluate them on 12 large matrices/graphs which are selected from SNAP [20], OGB [17], and SuiteSparse [10]. The number of vertices (rows) ranges from 45K to 2.45M and the number of edges (non-zeros) can be as high as 124M. For the comparison of Serpens with K80, we evaluate on 2,519 sparse matrices whose number of non-zeros (NNZ) is greater than 1,000 and less than 100M from SuiteSparse. The geomean density of the evaluated SuiteSparse matrices is 1.4E-3. Table 3 shows the specifications of the evaluated matrices. We evaluate single floating-point SpMV. We compare the execution time (ms), throughput in million traversed edges per second (MTEPS), bandwidth efficiency defined as (throughput)/(memory bandwidth), and energy efficiency defined as (throughput)/(power consumption) of the three FPGA accelerators. We set N=8 (the minimal supported N) for Sextans [27] to obtain SpMV results. We run GraphLily [18] on SpMV mode.

4.2 Comparison with Related Accelerators

Table 4 shows the execution time, throughput, bandwidth and energy efficiency of the three FPGA accelerators. Sextans [27] is an HBM-FPGA SpMM accelerator. An SpMV accelerator is able to switch to process SpMM and vice versa in functionality. However, their designs are different and customized for the performance of tow different kernels – SpMV/SpMM. We use Table 5 to compare Serpens and Sextans [27]. We use the matrix `TSOPF_RS_b2383_c1` from SuiteSparse [10] to illustrate the difference. The SpMM(N=16) latencies of Serpens (running 16 SpMVs) and Sextans are 8.56 ms and 2.87 ms, respectively, and the SpMV latencies of Serpens and Sextans are 0.535 ms and 1.44 ms, respectively. We got lower performance if we use an SpMM accelerator for perform SpMV and vice versa. The different customization in the accelerators lead to their performance expertise. For memory channel allocation for

Table 4: Performance of Sextans [27], GraphLily [18], and SERPENS on twelve large matrices/graphs. The improvement is the ratio of a performance metric of SERPENS compared to that of GraphLily.

		G1	G2	G3	G4	G5	G6	G7	G8	G9	G10	G11	G12	GMN
Execution Time: ms	**Sextans**	3.06	1.38	1.64	1.36	2.73	2.72	–	3.58	–	–	–	–	2.20
	GraphLily	1.73	1.47	1.85	1.57	2.96	2.80	7.04	3.63	4.52	4.59	12.4	18.6	3.74
	SERPENS-A16	1.87	0.930	0.853	0.730	1.37	1.37	4.52	2.09	2.05	2.04	6.20	6.32	1.96
Throughput: GFLOP/s	**Sextans**	9.01	20.60	18.55	23.81	20.47	21.33	-	18.14	-	-	-	-	18.15
	GraphLily	15.96	19.36	16.44	20.64	18.87	20.69	9.17	17.90	16.75	18.74	18.36	13.60	16.86
	SERPENS-A16	14.71	30.56	35.62	44.39	40.75	42.26	14.29	31.06	37.01	42.26	36.70	39.90	32.21
Throughput: MTEPS	**Sextans**	4,470	10,255	9,162	11,878	10,099	10,651	–	8,951	–	–	–	–	9,005
	GraphLily	7,920	9,639	8,117	10,296	9,305	10,331	4,352	8,828	8,212	9,243	9,094	6,668	8,310
	SERPENS-A16	7,300	15,214	17,594	22,144	20,099	21,098	6,782	15,324	18,142	20,847	18,176	19,565	15,876
	Improvement	0.922×	1.58×	2.17×	2.15×	2.16×	2.04×	1.56×	1.74×	2.21×	2.26×	2.00×	2.93×	1.91×
Bandwidth Efficiency: MTEPS/(GB/s)	**Sextans**	10.7	24.6	22.0	28.5	24.2	25.5	–	21.5	–	–	–	–	21.6
	GraphLily	27.8	33.8	28.5	36.1	32.7	36.2	15.3	31.0	28.8	32.4	31.9	23.4	29.2
	SERPENS-A16	26.7	55.7	64.4	81.1	73.6	77.3	24.8	56.1	66.5	76.4	66.6	71.7	58.2
	Improvement	0.962×	1.65×	2.26×	2.25×	2.25×	2.13×	1.63×	1.81×	2.31×	2.35×	2.09×	3.06×	1.99×
Energy Efficiency: MTEPS/W	**Sextans**	86.0	197	176	228	194	205	–	172	–	–	–	–	173
	GraphLily	184	224	189	239	216	240	101	205	191	215	211	155	193
	SERPENS-A16	152	317	367	461	419	440	141	319	378	434	379	408	331
	Improvement	0.826×	1.41×	1.94×	1.93×	1.94×	1.83×	1.40×	1.56×	1.98×	2.02×	1.79×	2.63×	1.71×

Table 5: Comparisons of SERPENS, Sextans [27], and GraphLily [18].

	Kernel	#Ch. - Sparse A	#Ch. - Dense B/C(X/Y)	#Ch. - Instr.
SERPENS	SpMV	16/24	1/1	1
Sextans	SpMM	8	4/8	1
GraphLily	Graph	16	1/1	-
	OoO NZ	**Sharing Sparse A**	**Index Coalescing**	**Perf - SpMV/SpMM**
SERPENS	Yes	No	Yes	High/Low
Sextans	Yes	Yes	No	Low/High
GraphLily	No	No	No	-/-

Table 6: Resource utilization of Sextans, GraphLily, and SERPENS-A16 on a Xilinx U280 FPGA board.

	LUT	FF	DSP	BRAM	URAM
Sextans	331K(29%)	594K(25%)	3233(36%)	1238(68%)	768(80%)
GraphLily	390K(35%)	493K(21%)	723(8%)	417(24%)	512(53%)
SERPENS	173K(15%)	327K(14%)	720(8%)	655(36%)	384(40%)

the matrices, since the vector size is significantly smaller than the sparse matrix size, thus SERPENS needs to allocate one channel for a vector and allocate memory channels for the sparse matrix. So SERPENS performs better than Sextans for SpMV. However, the dense element sharing helps Sextans perform better than SERPENS for SpMM. GraphLily is an FPGA overlay which is able to support a few graph kernels that can be executed in a BLAS processing model. GraphLily supports generalized multiplication and generalized reduction. For example, GraphLily can configure a generalized multiplication as one of (1) algebraic multiplication, (2) algebraic addition, (3) logic AND, and (4) zero output. In the processing of SpMV where GraphLily configures a generalized multiplication as an algebraic multiplication, the hardware resource for the other two operations is idle. Thus, GraphLily lacks deeper specialization for SpMV and the SpMV execution time of GraphLily is larger than the execution time of SERPENS.

4.2.1 Execution Time. Sextans is not able to support Matrix G7 and G9 – G12 directly on the hardware. For the other matrices, the execution time of SERPENS is less than the execution time of Sextans. For the comparison with GraphLily, SERPENS is slightly slower (1.87 ms v.s. 1.73 ms) on G1 but faster than GraphLily on the other 11 matrices.

4.2.2 Throughput. We use (NNZ)/(execution time) to calculate the throughput (MTEPS). The throughput directly corresponds to the execution time. A shorter execution time leads to a higher throughput. The maximum throughput achieved by GraphLily is 10,331 MTEPS while the maximum throughput achieved by SERPENS is 22,144 MTEPS. For the geometric throughput, GraphLily and SERPENS achieve a geomean throughput of 8,310 MTEPS and 15,876 MTEPS respectively, leading to a 1.91× throughput improvement of SERPENS over GraphLily.

4.2.3 Bandwidth Efficiency. On the same graph/matrix, the bandwidth efficiency is determined by the execution time and the accelerator's memory bandwidth. GraphLily's bandwidth is higher than SERPENS' bandwidth (285 GB/s v.s. 273 GB/s). With a faster execution time, SERPENS achieves a geomean bandwidth efficiency of 58.2 MTEPS / (GB/s), 1.99× compared with GraphLily's bandwidth efficiency. The highest bandwidth efficiency achieved by SERPENS is 81.1 MTEPS / (GB/s) on G4.

4.2.4 Energy Efficiency. Some hardware resource of GraphLiLy overlay may be idle when performing one specific graph kernel, so the power consumption of GraphLiLy is lower than that of SERPENS (43 W v.s. 48 W). However, SERPENS is 1.91× faster than GraphLiLy, leading to a 1.71× energy efficiency improvement.

4.2.5 Resource Utilization. Table 6 lists the FPGA resource utilization of the three accelerators on the same U280 board. Sextans requires the highest resource utilization because Sextans needs to compute on the dense **B** matrix which is larger than the $\vec{x}$ vector in SpMV. In contrast to GraphLily, SERPENS consumes less LUT, FF, DSP, and URAM. Because SERPENS is customized specifically for SpMV, it does not need the extra FPGA resource that GraphLily requires for its generalized operations. However, SERPENS consumes more BRAMs than GraphLily, because SERPENS explicitly deploys more BRAMs to access on-chip memory in parallel.

4.2.6 Other SpMV Accelerators. We compare SERPENS with two other real-execution SpMV accelerators [25] and SparseP [13] in Table 7. [25] is based on an FPGA and SparseP [13] is based on a real PIM system. SERPENS-A24 has the highest peak performance and SERPENS-A16 performs better and has lower memory bandwidth than both [25] and [13].

4.3 Comparison with K80 GPU

We compare SERPENS-A16 with K80 GPU on a wide range of 2,519 sparse matrices from SuiteSparse [10] to demonstrate the performance of SERPENS as a general-purpose accelerator in a data center.

Table 7: Comparison with other SpMV accelerators.

	Bandwidth	Peak Performance
Serpens-A16	273 GB/s	44.2 GFLOP/s
Serpens-A24	388 GB/s	60.4 GFLOP/s
[11]	258 GB/s	25.0 GFLOP/s
[25]	357 GB/s	34.0 GFLOP/s
SparseP [13]	1770 GB/s	4.66 GFLOP/s

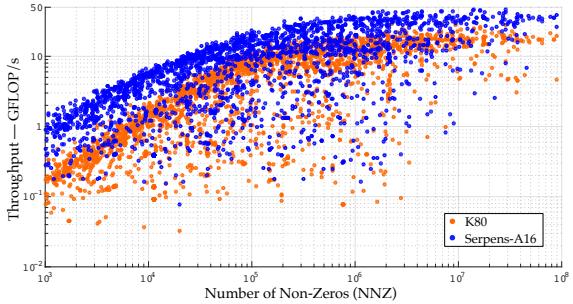


Figure 3: SpMV throughput (in GFLOP/s) of K80 and Serpens plotted with increasing NNZ.

Table 8: The SpMV throughput (GFLOP/s) of the 24 HBM channel version Serpens and improvement over GraphLily.

	G1	G2	G3	G4	G5	G6
Serpens-A24	15.33	36.05	45.07	60.55	52.30	57.96
Improvement	0.960×	1.86×	2.74×	2.93×	2.77×	2.80×
	G7	G8	G9	G10	G11	G12
Serpens-A24	18.34	36.47	46.86	56.11	45.08	51.56
Improvement	2.00×	2.04×	2.80×	3.00×	2.46×	3.79×

K80 is a more powerful accelerator than Serpens in terms of frequency and bandwidth as shown in Table 2.

We plot the SpMV throughputs of K80 and Serpens in Figure 3. Serpens achieves higher throughput on almost all matrices than K80. The maximum throughputs of K80 and Serpens are 46.43 GFLOP/s (14,521 MTEPS) and 29.12 GFLOP/s (23,158 MTEPS) respectively. The geomean throughput of Serpens compared to K80 is 2.31×. For the geomean bandwidth efficiency, K80 achieves 2.10 MTEPS /(GB/s). With a geomean bandwidth efficiency of 8.52 MTEPS/(GB/s), Serpens outperforms K80 by 4.06×. For the geomean energy efficiency, K80 achieves 7.75 MTEPS/W. Serpens has a geomean energy efficiency of 48.4 MTEPS/W (6.25× better).

4.4 Scalability

We scale up HBM channel allocation from 16 to 24 to further boost performance. Vanilla Vitis failed place and route because of the congestion caused by heavy HBM channel usage. With the aid of TAPA [6] and Autobridge [15], we successfully place and route the 24 HBM channel version, resulting in 270 MHz frequency. We compare Serpens-A24 with GraphLily [18] in Table 8. Serpens-A24 achieves up to 60.55 GFLOP/s (30,204 MTEPS) and a throughput of up to 3.79× improvement over GraphLily.

5 CONCLUSION

We present Serpens, an HBM based accelerator for SpMV acceleration. Serpens is a general-purpose design which supports an arbitrary SpMV. We design memory-centric processing engines in Serpens for full utilization of memory bandwidth and the scalability with memory channels. We improve URAM utilization for vector storage by index coalescing, and the index coalescing is integrated with non-zero recording. In the evaluation, we compare Serpens with two related FPGA accelerators Sextans [27] and GraphLily [18]. Serpens outperforms the latest accelerators GraphLiLy and Sextans by 1.91× and 1.76×, respectively, in terms of geomean throughput. For the comparison of Serpens with K80 GPU on SuiteSparse [10], Serpens achieves 2.10× higher throughput. We scale up Serpens to support 24 HBM channels for the spares matrix. After scaling up to 24 HBM channels, Serpens achieves a throughput of up to 60.55 GFLOP/s (30,204 MTEPS) and up to 3.79× over GraphLily.

REFERENCES

[1] [n.d.]. Alveo U200 and U250 Data Center Accelerator Cards Data Sheet. https://www.xilinx.com/products/boards-and-kits/alveo/u250.html.
[2] [n.d.]. Alveo U280 Data Center Accelerator Card Data Sheet. https://www.xilinx.com/products/boards-and-kits/alveo/u280.html.
[3] [n.d.]. HIGH BANDWIDTH MEMORY (HBM) DRAM. https://www.jedec.org/standards-documents/docs/jesd235a.
[4] Martín Abadi et al. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*.
[5] Xinyu Chen et al. 2021. ThunderGP: HLS-based Graph Processing Framework on FPGAs. In *FPGA*.
[6] Yuze Chi et al. 2021. Extending High-Level Synthesis for Task-Parallel Programs. In *FCCM*.
[7] Young-kyu Choi et al. 2020. When HLS Meets FPGA HBM: Benchmarking and Bandwidth Optimization. *arXiv preprint* (2020).
[8] Young-kyu Choi et al. 2021. HBM Connect: High-Performance HLS Interconnect for FPGA HBM. In *FPGA*.
[9] Jason Cong et al. 2018. Latte: Locality Aware Transformation for High-Level Synthesis. In *FCCM*.
[10] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM TOMS* (2011).
[11] Yixiao Du et al. 2022. High-Performance Sparse Linear Algebra on HBM-Equipped FPGAs Using HLS: A Case Study on SpMV. In *FPGA*.
[12] Jeremy Fowers et al. 2014. A High Memory Bandwidth FPGA Accelerator for Sparse Matrix-Vector Multiplication. In *FCCM*.
[13] Christina Giannoula et al. 2022. SparseP: Towards Efficient Sparse Matrix Vector Multiplication on Real Processing-In-Memory Systems. In *SIGMETRICS*.
[14] Licheng Guo et al. 2020. Analysis and Optimization of the Implicit Broadcasts in FPGA HLS to Improve Maximum Frequency. In *DAC*.
[15] Licheng Guo et al. 2021. AutoBridge: Coupling Coarse-Grained Floorplanning and Pipelining for High-Frequency HLS Design on Multi-Die FPGAs. In *FPGA*.
[16] Song Han et al. 2015. Learning both Weights and Connections for Efficient Neural Networks. In *NIPS*.
[17] Weihua Hu et al. 2020. Open Graph Benchmark: Datasets for Machine Learning on Graphs. *arXiv preprint* (2020).
[18] Yuwei Hu et al. 2021. GraphLily: Accelerating Graph Linear Algebra on HBM-Equipped FPGAs. In *ICCAD*.
[19] Jeremy Kepner et al. 2016. Mathematical Foundations of the GraphBLAS. In *HPEC*.
[20] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.
[21] Jiajia Li et al. 2018. HiCOO: Hierarchical Storage of Sparse Tensors. In *SC*.
[22] Weifeng Liu and Brian Vinter. 2015. CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication. In *ICS*.
[23] Maxim Naumov et al. 2010. CUSPARSE Library. In *GPU Tech. Conf.*
[24] Yousef Saad. 2003. *Iterative Methods for Sparse Linear Systems*. SIAM.
[25] Fazle Sadi et al. 2019. Efficient SpMV Operation for Large and Highly Sparse Matrices using Scalable Multi-way Merge Parallelization. In *MICRO*.
[26] Linghao Song et al. 2018. GraphR: Accelerating Graph Processing Using ReRAM. In *HPCA*.
[27] Linghao Song et al. 2022. Sextans: A Streaming Accelerator for General-Purpose Sparse-Matrix Dense-Matrix Multiplication. In *FPGA*.
[28] Nitish Srivastava et al. 2020. Tensaurus: A Versatile Accelerator for Mixed Sparse-Dense Tensor Computations. In *HPCA*.
[29] Vivienne Sze et al. 2017. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proc. IEEE* (2017).
[30] Jie Wang et al. 2021. AutoSA: A Polyhedral Compiler for High-Performance Systolic Arrays on FPGA. In *FPGA*.
[31] Xinfeng Xie et al. 2021. SpaceA: Sparse Matrix Vector Multiplication on Processing-in-Memory Accelerator. In *HPCA*.
[32] Shijie Zhou et al. 2019. HitGraph: High-throughput Graph Processing Framework on FPGA. *IEEE TPDS* (2019).