SEXTANS: A Streaming Accelerator for General-Purpose Sparse-Matrix Dense-Matrix Multiplication

Linghao Song, Yuze Chi, Atefeh Sohrabizadeh, Young-kyu Choi†, Jason Lau, and Jason Cong
University of California, Los Angeles †Inha University
{linghaosong,chiyuze,atefehsz,lau,cong}@cs.ucla.edu,ykc@inha.ac.kr

ABSTRACT

Sparse-Matrix Dense-Matrix multiplication (SpMM) is the key operator for a wide range of applications including scientific computing, graph processing, and deep learning. Architecting accelerators for SpMM is faced with three challenges – (1) the random memory accessing and unbalanced load in processing because of random distribution of elements in sparse matrices, (2) inefficient data handling of the large matrices which can not be fit on-chip, and (3) a non-general-purpose accelerator design where one accelerator can only process a fixed-size problem.

In this paper, we present SEXTANS, an accelerator for general-purpose SpMM processing. SEXTANS accelerator features (1) fast random access using on-chip memory, (2) streaming access to off-chip large matrices, (3) PE-aware non-zero scheduling for balanced workload with an II=1 pipeline, and (4) hardware flexibility to enable prototyping the hardware once to support SpMMs of different size as a general-purpose accelerator. We leverage high bandwidth memory (HBM) for the efficient accessing of both sparse and dense matrices. In the evaluation, we present an FPGA prototype SEXTANS which is executable on a Xilinx U280 HBM FPGA board and a projected prototype SEXTANS-P with higher bandwidth competitive to V100 and more frequency optimization. We conduct a comprehensive evaluation on 1,400 SpMMs on a wide range of sparse matrices including 50 matrices from SNAP and 150 from SuiteSparse. We compare SEXTANS with NVIDIA K80 and V100 GPUs. SEXTANS achieves a 2.50x geomean speedup over K80 GPU and SEXTANS-P achieves a 1.14x geomean speedup over V100 GPU (4.94x over K80). The code is available at https://github.com/linghaosong/Sextans.

CCS CONCEPTS

• **Hardware → Hardware accelerators**; • **Computer systems organization → Reconfigurable computing**.

KEYWORDS

Accelerator, SpMM, Hardware Flexibility, High Bandwidth Memory.

ACM Reference Format:
Linghao Song, Yuze Chi, Atefeh Sohrabizadeh, Young-kyu Choi, Jason Lau, and Jason Cong. 2022. SEXTANS: A Streaming Accelerator for General-Purpose Sparse-Matrix Dense-Matrix Multiplication. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '22), February 27-March 1, 2022, Virtual Event, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3490422.3502357

FPGA '22, February 27-March 1, 2022, Virtual Event, CA, USA

ACM ISBN 978-1-4503-9149-8/22/02.
https://doi.org/10.1145/3490422.3502357

1 INTRODUCTION

Natural and scientific data are often sparse and large-scale which are stored as sparse matrices. The sparse matrices encode properties of nodes the connection between nodes. Sparse-matrix dense-matrix multiplication (SpMM) is a key computing routine in a wide rage of applications, such as social networks [58], chemical reactivity prediction [20], drug design [15], security analysis [84], sparse deep neural networks [42, 43, 79], and graph based machine learning [40, 53, 83]. SpMM performs the computation of $\mathbf{C} = \alpha\mathbf{A} \times \mathbf{B} + \beta\mathbf{C}$, where $\mathbf{A}$ is a sparse matrix, $\mathbf{B}$ and $\mathbf{C}$ are dense matrices, and α and β are two constant scalars. SpMM acceleration [34, 36, 41, 49, 78, 85, 86, 93] is attractive to researchers of computer systems and architectures.

Application-specific accelerators boost the performance of applications in many domains. However, the design of the SpMM accelerator faces many challenges.

• **Challenge 1 –** Workload imbalance makes SpMM difficult for parallelization. Row based parallelization [6, 74, 80] assigns the processing of one row as a task for a processing engine or a thread (block). However, because of the random distribution of non-zeros in each row, processing engines with early completion time will be idle. To overcome the workload imbalance, non-zero based parallelization [32, 55, 71] (or similarly edge-enteric processing in graph processing acceleration [16, 62, 70, 94]) is presented. However, non-zero based parallelization may incur the read-after-write dependency at the accelerator microarchitecture level, which leads to a larger initial interval (II) for the scheduling.

• **Challenge 2 –** Inefficient memory accessing is another challenge. Since the matrices of SpMM are large and can not be fit on chip, they are stored in off-chip memory. The processing of SpMM incurs random read accessing to matrix **A**, matrix **B**, and matrix **C**, and random write accessing to **C**. It is dramatically inefficient to issue the huge amount of random accesses to off-chip memory.

• **Challenge 3 –** How to design a general-purpose accelerator which does not need to be rerun the time-consuming flow of synthesis/place/route. While many accelerators have been designed for boosting computing performance and efficiency in many application domains such as deep learning [5, 11, 12, 23, 31, 35, 64–69, 77, 87, 88], dense linear algebra [23, 29, 30, 35, 77], graph processing [4, 17, 25, 26, 39, 70, 89, 91, 92, 95], genomic and bio analysis [8, 9, 13, 14, 33, 38, 51, 76, 81], data sorting [10, 52, 60, 63], most are designed for *one specific problem with fixed input and output size.* For FPGA accelerators even with improved tools such as [17, 77], a new design will still consume many hours or even a few days due to long synthesis and place/route time. Moreover, it is a nightmare

for end users who are not an accelerator expert to customize and rerun the flow to generate the accelerators for their applications.

In this paper we present Sextans a streaming accelerator for accelerating general-purpose sparse-matrix den-matrix multiplication. The contributions include:

• A hierarchical SpMM accelerator architecture. At the highest level, Sextans consists of (1) processing engine groups (PEGs), (2) modules to stream in/out matrices from/to off-chip HBM, and (3) modules to collect and perform element-wise multiplication and addition to obtain an updated **C**. PEGs and processing engines (PEs) are the key processing modules. A PEG consists of PEs and a PE consists of processing units.

• PE-aware non-zero scheduling for a balanced workload with an II=1 pipeline. The non-zero scheduling of Sextans is an out-of-order [75] scheduling. The key idea of the scheduling is that a non-zero is scheduled at the earliest cycle satisfying that the row index of the scheduled non-zero has no read-after-write (RAW) with the row index of non-zeros being processed in previous D (the distance of RAW dependency of a specific hardware platform) cycles. The scheduling leads to an II=1 pipeline. Similar to prior works [32, 71, 94], we incorporate the scheduling in the preprocessing of the spares elements.

• Multi-level memory optimizations with HBM for efficient accessing and streaming. We partition the three matrices of SpMM to fit the processing engines and on-chip memory (URAMs and BRAMs). For the off-chip memory, matrices are streamed in/out in batches with a window size so the off-chip memory accessing is always sequential. We partition the random memory read and write into a specific window, so random memory accessing is limited to on-chip fast memory.

• Hardware flexibility (HFlex) to support execution of different SpMMs directly by hardware as a general-purpose accelerator. We enable the HFlex feature with an iteration pointer list **Q** (similar to an instruction queue). We partition an arbitrary sparse matrix **A** into multiple **A** submatrices. We convert each **A** submatrix into a list of scheduled non-zeros. We store the scheduled non-zero lists of all **A** submatrices linearly in a memory space. We use an iteration pointer list **Q** to record the starting index of each scheduled non-zero list. In the processing, entries of **Q** serve as the loop iteration number and, as a result, we support the execution of an arbitrary SpMM without modification on the hardware. Sextans can be easily invoked by OpenCL runtime without handling the hardware and design details.

• Comprehensive evaluation. We present an FPGA prototype Sextans which is executable on a Xilinx U280 HBM FPGA board and a projected prototype Sextans-P with higher bandwidth competitive to V100 and more frequency optimization. We conduct comprehensive evaluation on 1,400 SpMMs on 200 sparse matrices. We compare Sextans with NVIDIA K80 and V100 GPUs. Sextans achieves a 2.50x geomean speedup over K80 GPU and Sextans-P achieves a 1.14x geomean speedup over V100 GPU (4.94x over K80).

2 BACKGROUND AND MOTIVATION

2.1 Sparse-Matrix Dense-Matrix Multiplication

SpMM performs the computation of $\mathbf{C} = \alpha\mathbf{A} \times \mathbf{B} + \beta\mathbf{C}$, where **A** is a sparse matrix, **B** and **C** are dense matrices, and α and β are two constant scalars. In algorithm modeling, the sparse matrix **A** represents a graph, the dense matrix **B** depicts the feature vectors of nodes, and the dense matrix **C** is involved if both old and new features are modeled. Because natural and social data structures are large and sparse SpMM is a very useful computing routine in many application domains.

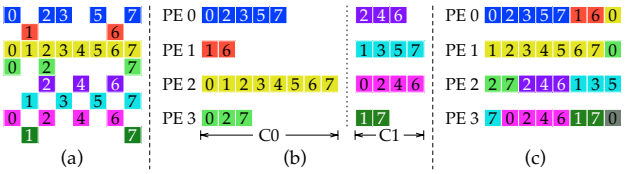


Figure 1: (a) a sparse matrix, (b) row based parallelization, and (c) non-zero based parallelization.

For example, in sparse deep neural networks [42, 43, 79], matrix **A** represents the pruned weight and matrix **B** represent feature maps, so the inference is performed by $\mathbf{C} = 1.0 \cdot \mathbf{A} \times \mathbf{B} + 0.0 \cdot \mathbf{C}$. In graph based machine learning [40, 53, 83], matrix **B** represent the node properties and matrix **A** represents the graph, so SpMM performs the graph propagation. Therefore, researchers of computer systems and architectures find SpMM acceleration [34, 36, 41, 49, 78, 85, 86, 93] attractive.

2.2 Sparse Matrix Multiplication Acceleration

SpMM is challenging for parallelization. Suppose we are using four processing engines (PEs) or threads to compute on a sparse matrix as shown in Figure 1 (a) where the non-zeros (of matrix **A**) in row are colored the same. The number on a square is the column index of each non-zero. An intuitive approach to paralleling SpMM on CPUs/GPUs is row based parallelization [6, 74, 80] as shown in Figure 1 (b). It takes 2 steps C0 and C1 to process the eight rows by the four PEs. However, there is PE workload imbalance. For example, PE2 takes 8 cycles to process the yellow row but other PEs finish their rows earlier and become idle in C0. The workload imbalance wastes computing resources. To overcome the workload imbalance, non-zero based parallelization [32, 55, 71] (or similarly edge-enteric processing in graph processing acceleration [62, 70, 94])is presented. As shown in Figure 1 (c), non-zeros of mutiple rows are packaged into a segment and the segments are assigned to PEs. The organization of non-zeros is implemented as explicit or implicit data formats [32, 55, 71] which also encode the scheduling of computing tasks. Because the segments are equal in length, the PE workload is balanced. However, those techniques can not be directly adopted in FPGA accelerators because pipeline and memory related issues occur, and details are discussed in Section 2.4.

2.3 High Bandwidth Memory

High bandwidth memory (HBM) [2] is designed for applications which demands for high memory bandwidth. HBM provides many pseudo and/or physical channels for channel-level parallel accessing and thus delivers a high total bandwidth. For example, Xilinx U280 FPGA accelerator card [1] is equipped with an HBM which offers 32 pseudo channels. The bandwidth of each pseudo channel is 14.375 GB/s, for a total bandwidth of 460 GB/s. Because HBM is a new feature to FPGAs, existing studies of FPGA HBM mainly focus on tool development [17, 18, 37] and benchmarking [19, 50],

but very few applications. SpMM, a memory-intensive application which is distinguished from typical computation-intensive FPGA applications [5, 23, 31, 77, 87, 88], is a good fit for HBM. This work presents one of the first real application for HBM FPGA.

2.4 Motivation

Our work aims at achieving the following goals to addressing limitations in prior works.

• **A general-purpose and user-friendly SpMM accelerator.** Domain specific architectures [21, 22, 27, 45] have been designed for boosting computing performance and efficiency in many application domains such as deep learning [5, 11, 12, 23, 31, 35, 47, 64–69, 77, 87, 88], dense linear algebra [23, 29, 30, 35, 77], graph processing [4, 7, 17, 25, 26, 39, 48, 56, 70, 89, 91, 92, 95], genomic and bio analysis [8, 8, 9, 13, 14, 33, 38, 51, 76, 81], and data sorting [10, 52, 60, 63]. However, most accelerators are designed for one specific problem with fixed input and output size. To support a different problem configuration, the accelerator architecture is to be modified, which will consumes several weeks to months for architecture re-design and chip tape-out. For FPGA accelerators, even with improved tools such as [17, 77], a new design will still consume many hours even a few days due to long synthesis and place/route time. Another method is to design a kernel for a fixed-size problem and decompose a new problem to multiple kernels of the fixed-size problem and launch those kernels by the runtime. However, the runtime overhead is significant. For example, we can build a accelerator for a matrix multiplication with a fixed size of 4096×4096 with [77] and we map 50 SNAP matrices [54] where the row/column number ranges from 1,005 to 456,626 and the number of non-zeros ranges from 20,296 to 14,855,842. The average number of decomposed fixed-size kernels for the SNAP matrices is 1793. The OpenCL runtime overhead for launching one kernel is around 0.15ms. So the average runtime overhead for launching kernels is 269ms. As a comparison, the average execution time of SpMM on the SNAP matrices by an NVIDIA K80 GPU is 5.85ms. Moreover, it is a nightmare for end users who are not accelerator experts to customize and rerun the flow for their applications. As we discussed before, SpMM is an abstraction kernel for many applications. Sextans features the hardware flexibility (HFlex) to directly support execution of different SpMMs by hardware as a general-purpose and democratized SpMM accelerator. Users can deploy Sextans for different problems without rerunning the synthesis/place/route flow and they can easily invoke Sextans in their applications by OpenCL runtime without handling the hardware details.

• **PE-aware non-zero scheduling for a balanced workload.** Although non-zero based parallelization [32, 55, 62, 94] alleviates the load imbalance issue, directly applying non-zero based parallelization to accelerators incurs a dependency issue on the microarchitecture. For example, in Figure 1 (c) the blue elements held by PE 0 are accumulated to the same destination and the floating-point ADD usually takes 7 to 10 cycles (depending on specific FPGAs). Thus, a read-after-write (RAW) conflict occurs for blue elements 2, 3, 5, 7 and HLS tools will schedule a large initiation interval (II), leading to long computing latency. To overcome this RAW dependency issue, we present a PE-aware no-zero scheduling. The key idea is to apply out-of-order [75] scheduling to move back a conflict element and move forward a non-conflict element within a scheduling window to resolve the RAW dependency.

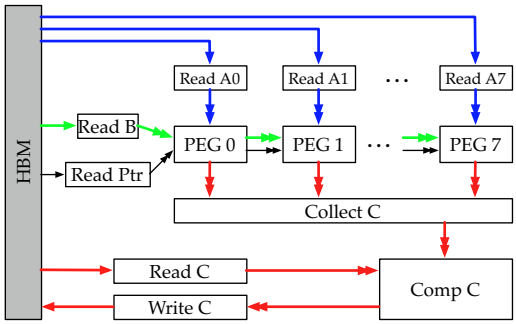


Figure 2: Sextans overall architecture.

• **Multi-level memory optimizations for efficient accessing and streaming.** SpMM face three memory related challenges: (**C1**) bank conflict, (**C2**) irregular memory accessing, and (**C3**) off-chip large matrix accessing.

C1 – A bank conflict happens when two or more processing units access the same bank. For example, in Figure 1 (c) PE 0 and PE 1 both need to read the element with column index 0 at Cycle 8. A bank conflict occurs if the the memory storing element 0 has only one port. To overcome the bank conflict, we duplicate the read-only matrix shard to the PEs.

C2 – The irregular column index shown as colored square numbers in Figure 1 (b) and (c) lead to irregular memory read requests, whereas the irregular row destination of PEs in Figure 1 (c) leads to irregular memory write requests. Although our accelerators are equipped with HBM which has higher memory bandwidth, the latency of accessing HBM is still high (up to 100 cycles) [18]. Inspired by the idea of caching random accessing on a higher memory hierarchy in graph processing [70, 94], we partition the random memory read and write into a specific window, so random memory accessing is limited to on-chip fast memory. We store read-only dense in BRAMs to limit random read on chip. We use a scratchpad memory (FPGA URAMs) to limit random accumulation (read and write) on chip. We also achieve an II=1 scheduling that will further hide on-chip accessing latency.

C3 – The three matrices **A**, **B**, and **C** in SpMMs are so large that we can not store them on chip. For example, one evaluated matrix `bundle_adj` consumes 3.2 GB memory footprint but the total on-chip memory (SRAM) of a Xilinx U280 FPGA is 41 MB. We partition the three matrices according to the processing window size and store the partitioned matrix shards in HBM. We do not issue individual element accessing to HBM but only read or write a matrix shard. This allows HBM to be streamed for efficient accessing.

3 SEXTANS ARCHITECTURE

3.1 Overall Architecture

3.1.1 Overall Architecture. Figure 2 illustrates the overall architecture of Sextans. An arrow demotes the data transfer direction and a double-arrow indicates a FIFO connection. Data transfers of **A**, **B**, and **C** are colored in blue, green, and red respectively. We deploy 8 processing engine groups (PEG 0 – 7) to compute $\mathbf{C}_{AB} = \mathbf{A} \times \mathbf{B}$. Each PEG contains 8 PEs. To stream in and supply the disjoint partitioned matrix $\mathbf{A}_{pj}$ (defined in Equation 4) from HBM to each PEG,

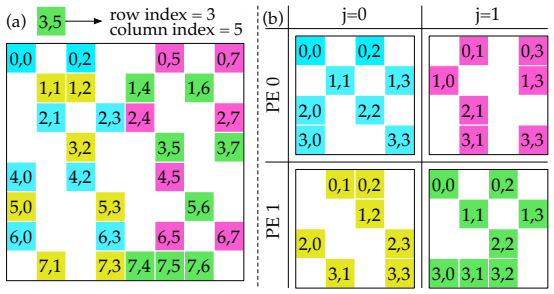


Figure 3: (a) A sparse matrix example and (b) its partitioning.

we disseminate 8 Read A modules. We deploy a Read B module to stream in a window of matrix $\mathbf{B}_{ji}$ from HBM and broadcast $\mathbf{B}_{ji}$ to the 8 PEGs. Each PEG also serves as a relaying node to form a chain-based broadcasting network. We do not use a one-to-all broadcasting network because a one-to-all broadcasting leads to low frequency [24] and route failure. We deploy a Read Ptr module to deliver pointers $\mathbf{Q}$ of out-of-order scheduled non-zeros to PEGs. A chain-based broadcasting network delivers the pointers to PEGs. A Collect C module collects the disjoint $\mathbf{C}_{\alpha AB}|_p$. A Comp C module performs the element-wise computation of $\mathbf{C}_{out} = \mathbf{C}_{\alpha AB} + \beta \cdot \mathbf{C}_{in}$ where $\mathbf{C}_{AB}$ is supplied by the Collect C model, $\mathbf{C}_{in}$ is supplied by a Read C module which streams in $\mathbf{C}_{in}$ from HBM, and $\mathbf{C}_{out}$ is sent to a Write C module to be streamed out to HBM. We assign 1 HBM channel to pointers $\mathbf{Q}$, 4 HBM channels to matrix $\mathbf{B}$, 8 HBM channels to matrix $\mathbf{A}$, 8 HBM channels to matrix $\mathbf{C}_{in}$, and 8 HBM channels to matrix $\mathbf{C}_{out}$.

3.1.2 Overall Processing Logic. We discuss the overall logic of how we partition the spares and dense matrices and how we process SpMM. We separate the computation of $\mathbf{C} = \alpha\mathbf{A} \times \mathbf{B} + \beta\mathbf{C}$ into three phases,

$$\begin{cases} \mathbf{C}_{AB} = \mathbf{A} \times \mathbf{B} \\ \mathbf{C}_{\alpha AB} = \alpha \cdot \mathbf{C}_{AB} \\ \mathbf{C}_{out} = \mathbf{C}_{\alpha AB} + \beta \cdot \mathbf{C}_{in} \end{cases} . \tag{1}$$

$\mathbf{C}_{in}$, $\mathbf{C}_{AB}(\mathbf{C}_{\alpha AB})$ and $\mathbf{C}_{out}$ are the input $\mathbf{C}$ matrix, the intermediate multiplication $\mathbf{C}$ matrix, and the output $\mathbf{C}$ matrix respectively. The computation of $\mathbf{C}_{AB} = \mathbf{A} \times \mathbf{B}$ is the most challenging phase in the SpMM acceleration. With the intermediate multiplication $\mathbf{C}_{AB}$ we perform the element-wise multiplication with α to obtain $\mathbf{C}_{\alpha AB}$. Then we stream in $\mathbf{C}_{in}$ from off-chip memory, execute element-wise multiplication/addition for $\mathbf{C}_{out} = \mathbf{C}_{\alpha AB} + \beta \cdot \mathbf{C}_{in}$, and stream out $\mathbf{C}_{out}$ to off-chip memory.

$\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ are large matrices that do not fit on chip, thus we need to partition the three matrices and reform the computation of $\mathbf{C}_{AB} = \mathbf{A} \times \mathbf{B}$. The dimension of $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ are $M \times K$, $K \times N$ and $M \times N$ respectively. First, we partition $\mathbf{B}$ rows. We partition each row into segments with a length of N_0. So $\mathbf{B}$ becomes N/N_0 submatrices $\mathbf{B}_i$ with the dimension of $\mathbf{B}_i$ set to $K \times N_0$. The multiplication of $\mathbf{C}_{AB} = \mathbf{A} \times \mathbf{B}$ changes to:

$$\begin{cases} \mathbf{C}_{AB} = \{\mathbf{C}_{AB_i} | i \in \{0, 1, ..., N/N_0 - 1\}\} \\ \mathbf{C}_{AB_i} = \mathbf{A} \times \mathbf{B}_i \end{cases} . \tag{2}$$

Next we partition $\mathbf{B}_i$ columns. Each column converts to K/K_0 column segments with a length of K_0. Because the $\mathbf{A}$ row is associated the $\mathbf{B}_i$ column, we divide the $\mathbf{A}$ row into row segments. These segments have a length of K_0 (also referred to as window size in the following content). Thus, $\mathbf{B}_i$ is partitioned into K/K_0 submatrices

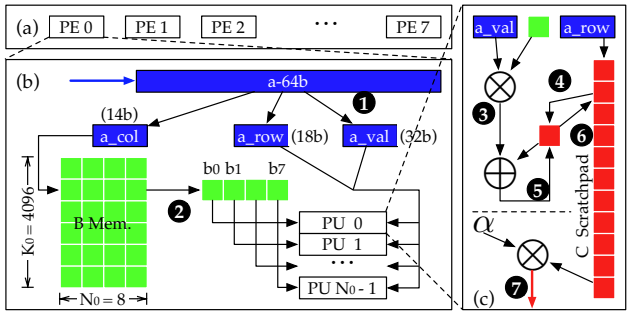


Figure 4: The architecture of (a) a PEG, (b) a PE, and (c) a PU.

$\mathbf{B}_{ji}$ and $\mathbf{A}$ is partitioned into K/K_0 submatrices $\mathbf{A}_j$. The computation of $\mathbf{C}_{AB_i} = \mathbf{A} \times \mathbf{B}_i$ becomes:

$$\begin{cases} \mathbf{C}_{AB_i} = \sum_j^{K/K_0} \mathbf{C}_{A_jB_{ji}} \\ \mathbf{C}_{A_jB_{ji}} = \mathbf{A}_j \times \mathbf{B}_{ji} \end{cases} . \tag{3}$$

$\mathbf{A}_j$ is a sparse matrix and the uneven distribution of non-zeros leads to the workload imbalance issues as discussed earlier. In order to achieve an approximately uniform probability distribution of the non-zeros we split non-zeros of $\mathbf{A}_j$ into P bins. The bin p holds non-zeros whose row index row satisfies $(row \bmod P) == p$. So $\mathbf{A}_j$ transforms into P submatrices $\mathbf{A}_{pj}$ and we change the computation of $\mathbf{C}_{A_jB_{ji}} = \mathbf{A}_j \times \mathbf{B}_{ji}$ to:

$$\begin{cases} \mathbf{C}_{A_jB_{ji}} = \{\mathbf{C}_{A_{pj}B_{ji}} | p \in \{0, 1, ..., P-1\}\} \\ \mathbf{C}_{A_{pj}B_{ji}} = \mathbf{A}_{pj} \times \mathbf{B}_{ji} \end{cases} . \tag{4}$$

The partitioning of the three matrices determines the coarse-grained scheduling for $\mathbf{C}_{AB} = \mathbf{A} \times \mathbf{B}$. Equation 2 and Equation 3 are processed sequentially and Equation 4 is performed in parallel. In Sextans architecture, we pass parameters N and K to the accelerator. Then we calculate the iteration number N/N_0 of Equation 2 and the iteration number K/K_0 of Equation 3. Two outer loops schedules Equation 2 and Equation 3. We set up a total number of P parallel processing engines (PEs) to perform Equation 4. We assign the submatrix multiplication $\mathbf{C}_{A_{pj}B_{ji}} = \mathbf{A}_{pj} \times \mathbf{B}_{ji}$ to the p-th PE. Because the iterator p (introduced by Equation 4) is only associated with the p-th PE, $\mathbf{A}_{pj}$ and $\mathbf{C}_{A_{pj}B_{ji}}$ of each PE are disjoint. However, the iterator i (introduced by Equation 2) and j (introduced by Equation 3) are exposed to every PE, so every PE accesses $\mathbf{B}_{ji}$.

We show an example of sparse matrix and its partitioning in Figure 3, assuming we have 2 PEs, and window size is 4. In iteration $j = 0$, we assign the elements (colored in blue and yellow) with column index 0 - 3 to the two PEs. In iteration $j = 1$, we assign the elements (colored in magenta and green) with column index 4 - 7 to the two PEs. For PE 0, we designate the elements with row index 0, 2, 4, 6 of Figure 3 (a). For PE 1, we designate the elements with row index 1, 3, 5, 7 of Figure 3 (a). The original row index is scheduled to be interleaved (mod P) to achieve statistically even distribution of non-zeros. Note that both row index and column index are compressed. For example, we convert the green element (3,5) to green (1,1) in iteration $j = 1$ for PE 1.

3.2 Processing Engine

Figure 4 shows the architecture of a PEG, a PE, and a processing unit (PU). A PEG contains 8 PEs and a PE is the key module in

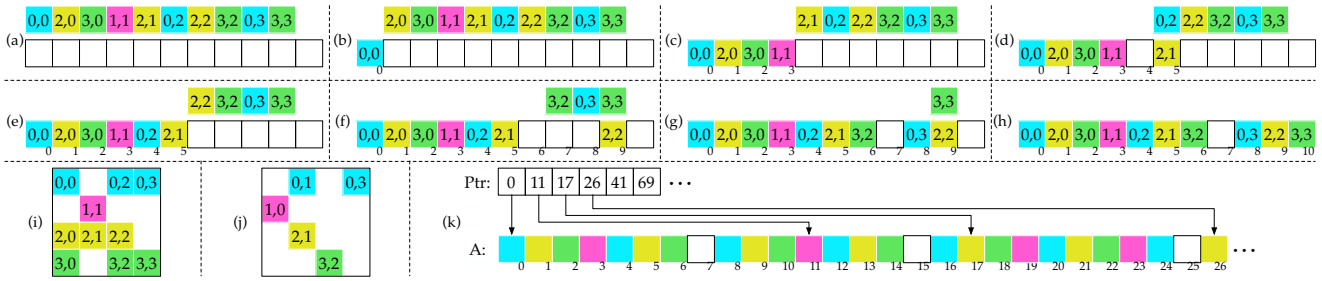


Figure 5: (a – h) The non-zero scheduling for the example (i). (i, j) The non-zero graphs for two example submatrix A_{pj}. (k) Supporting HFlex with a pointer array which records the starting index of scheduled non-zeros of A_{pj} submatrices.

Sextans architecture. A PE contains $N_0 = 8$ PUs. A PU performs the computation related to one non-zero scalar of matrix $\mathbf{A}_{pj}$.

To better illustrate the architecture of PE and PU, we change the submatrix multiplication $\mathbf{C}_{A_{pj}B_{ji}} = \mathbf{A}_{pj} \times \mathbf{B}_{ji}$ of p-th processing engine into sparse scalar format to:

$$\begin{cases} c_{kq} \mathrel{+}= a_{kl} \cdot b_{lq} \\ \forall a_{kl} \in \vec{u}_l,\ a_{kl} \neq 0,\ \forall \vec{u}_l \in \mathbf{A}_{pj},\ \vec{u}_l \neq \vec{0} \\ c_{kq} \in \mathbf{C}_{A_{pj}B_{ji}},\ b_{lq} \in \mathbf{B}_{ji} \end{cases}, \tag{5}$$

where c_{kq}, a_{kl}, and b_{lq} is a scalar(non-zero) of $\mathbf{C}_{A_{pj}B_{ji}}$, $\mathbf{A}_{pj}$, and $\mathbf{B}_{ji}$ respectively. $\vec{u}_l$ is a column vector of $\mathbf{A}_{pj}$. We iterate on the non-zeros a_{kl} of the column vector. Thus the processing is an outer-product like manner [59]. However, a RAW dependency conflict may happen when a non-zero a_{kl} from the next column vector has the same row index as a non-zero which is being processed. Thus, a RAW dependency conflict happens and the HLS can not achieve an II=1 scheduling. We will present a PE-aware scheduling to resolve this issue in Section 3.3. The iterator q on the second dimension of $\mathbf{C}_{A_{pj}B_{ji}}$ does not rely on any dependency, so we can schedule the processing on q in parallel. The length of the second dimension of $\mathbf{C}_{A_{pj}B_{ji}}$ is $N_0 = 8$.

A PE consumes one non-zero of $\mathbf{A}_{pj}$ and performs the multiplication and accumulation for N_0 parallel elements of $\mathbf{B}_{ji}$ and $\mathbf{C}_{A_{pj}B_{ji}}$ in an II=1 pipeline as shown in Figure 4 (b). One non-zero originally consumes 96 bits where each of row index, column index, and floating-point value consumes 32b its. Because the sparse matrix is partitioned, we can compress the row and column index to save memory footprint. We encode the row index, column index, and value of the non-zero in a 64-bit element `a-64b`. The first step ❶ is to decode `a-64b` to a 14-bit column index `a_col`, a 18-bit row index `a_row`, and a 32-bit floating-point value `a_val`. `a_col` is indexed to on-chip B memory and `a_row` is indexed to on-chip C scratchpad memory. The window size (depth) of B memory is $K_0 = 4096$ which consumes 12-bit. The depth of C scratchpad memory is 12,288[1] which consumes 14-bit. So 14/18 bits are sufficient for `a_col`/`a_row`. In the next step ❷, `a_col` is used to retrieve $N_0 = 8$ b elements b_0 to b_7 from the B memory. b_i is sent to the i-th PU. The 8 PUs share `a_row` and `a_val` from step ❶. Inside a PU (Figure 4 (c)), `a_val` is multiplied with a b element b_i ❸. `a_row` is the index of the accumulation C element c_{kq} and is sent to C scratchpad memory to fetch c_{kq} ❹. ❺ performs the accumulation of $c_{kq} \mathrel{+}= a_{kl} \cdot b_{lq}$. ❻ stores the updated c_{kq} back to C scratchpad memory. After the computation of $\mathbf{C}_{A_{pj}B_{ji}} = \mathbf{A}_{pj} \times \mathbf{B}_{ji}$ for one window is done, we iterate and process the next window (Equation 3). After the computation of Equation 3 is done, we perform the element-wise computation of $\mathbf{C}_{\alpha A_{pj}B_{ji}} = \alpha \cdot \mathbf{C}_{A_{pj}B_{ji}}$. In the step ❼, the C elements are streamed out from the C scratchpad memory and multiplied with a constant α element by element. The multiplied results $\mathbf{C}_{\alpha A_{pj}B_{ji}}$ are sent to the Collect C module. Note that the HLS schedules some steps of ❶ - ❼ to be processed concurrently.

[1]We use a Xilinx U280 FPGA. The URAM size is 4096×72 bit. With $N_0 = 8$, we allocate $12288/4096 \times 8/2 = 12$ URAMs for each PE. 768 (80%) URAMs are consumed.

3.3 Non-zero Scheduling

The non-zero scheduling is platform-specific and PE-aware because the scheduling is based on the distance D of RAW dependency of a specific hardware platform and the processing status of a PE. Instead of in-order scheduling which is unable to fully utilize the pipeline, the non-zero scheduling of Sextans uses an out-of-order [75] scheduling. The key idea of the scheduling is that we schedule a non-zero at the earliest cycle so that the row index of the scheduled non-zero has no RAW with the row index of non-zeros being processed in previous D cycles. The scheduling leads to an II=1 pipeline. Similar to prior works [32, 71, 94], we incorporate the scheduling in the preprocessing of the spares elements. We provide the non-zero scheduling as a host C++ wrapper for users.

We show an example of non-zero scheduling in Figure 5 (a) - (h) for a sparse matrix Figure 5 (i). We assume the RAW dependency distance D is 4 in this example. The non-zeros which has the same row index are colored the same. The non-zeros of Figure 5 (i) is listed in column-major order in Figure 5 (a). In Figure 5 (b), the first non-zero, blue (0,0) – (row index, column index) is scheduled to Cycle 0. For the next three non-zeros, we can safely schedule them to a following cycle because there is no RAW conflict in Figure 5 (c). In Figure 5 (d), yellow (2,1) conflicts with yellow (2,0) at Cycle 1 because $4 - 1 < D = 4$, and it is scheduled to the earliest Cycle 5. The blank(bubble) Cycle 4 is filled by blue (0,2) in Figure 5 (e). Next, yellow (2,2) is scheduled to Cycle $5 + 4 = 9$ in Figure 5 (f). In Figure 5 (g) green (3,2) and blue (0,3) is scheduled to Cycle 6 and Cycle 8 respectively. The final non-zero green (3,3) is scheduled to Cycle 10. Although the scheduling may contain bubbles such as Cycle 7 in Figure 5 (h), bubbles are aggressively eliminated. As a comparison, column-major in-order scheduling consumes 15 cycles and row-major in-order scheduling consumes 28 for the example of Figure 5 (i).

3.4 Hardware Flexibility for Arbitrary SpMMs

Algorithm 1 Sextans HFlex SpMM.

Require:
(1) matrix **A**, **B**, and **C**, (2) pointer list **Q**, (3) constant α and β, and (4) matrix hyperparameter M, K, and N.

Ensure:
$\mathbf{C} = \alpha\mathbf{A} \times \mathbf{B} + \beta\mathbf{C}$.

1: **for** $(0 \le i < N/N_0)$ **do**
2: $\quad \mathbf{C}_{AB_i} \leftarrow \mathbf{0}$
3: $\quad$ **for** $(0 \le j < K/K_0)$ **do**
4: $\quad\quad$ Read $\mathbf{B}_{ji}$
5: $\quad\quad$ **for all** $(0 \le p < P)$ **do in parallel**
6: $\quad\quad\quad$ **for** $(\mathbf{Q}_j \le r < \mathbf{Q}_{j+1})$ **do**
7: $\quad\quad\quad\quad$ **for all** $(0 \le q < N_0)$ **do in parallel**
8: $\quad\quad\quad\quad\quad c_{kq} \leftarrow c_{kq} + a_{kl} \cdot b_{lq}$
9: $\quad\quad\quad\quad$ **end for**
10: $\quad\quad\quad$ **end for**
11: $\quad\quad$ **end for**
12: $\quad$ **end for**
13: $\quad \mathbf{C}_i \leftarrow \alpha\mathbf{A} \times \mathbf{B}_i + \beta\mathbf{C}_i$
14: **end for**

Sextans features the hardware flexibility (HFlex) to directly support execution of different SpMMs by hardware as a general-purpose SpMM accelerator. We enable the HFlex feature with a pointer list **Q** (similar to an instruction queue). An arbitrary sparse matrix **A** is partitioned into multiple submatrices $\mathbf{A}_{pj}$. Each $\mathbf{A}_{pj}$ is converted into a list of scheduled non-zeros. The scheduled non-zero lists of all submatrices $\mathbf{A}_{pj}$ are stored linearly in a memory space. We use a pointer list **Q** to record the starting index of each scheduled non-zero list. For example, we place the scheduled non-zero list of Figure 5 (i) in the space of 0 - 10 as show in Figure 5 (l). The scheduled non-zero list of a following **A** submatrix displayed in Figure 5 (k) is placed in the space of 11 - 16. So we set $\mathbf{Q}_1 = 11$ and $\mathbf{Q}_2 = 17$. The first entry of **Q** is always set to 0. The number of entries in **Q** is $K/K_0 + 1$, because K_0 is the length for partitioning the whole sparse matrix **A** and K_0 is also the window size for the out-of-order non-zero scheduling.

We present Sextans HFlex SpMM processing in Algorithm 1. The inputs to the algorithm are (1) matrix **A** (with non-zeros scheduled), matrix **B**, matrix **C**, (3) pointer array **Q**, (3) two constants α, β, and (4) three hyperparameters M, K, N which describe the shape/dimension of the SpMM. Line 5 - 11 performs the core computation of SpMM. We deploy 8 PEGs where each PEG contains 8 PEs. So the parallel factor for Line 5 is $P = 64$. Line 6 - 10 is a PE region where a PE uses the pointer list **Q** to determine the loop number for a specific sparse submatrix $\mathbf{A}_{pj}$. Inside a PE, we unroll the scalar computation for a factor of $N_0 = 8$ to share one sparse **A** scalar with 8 dense **B** scalars. With Sextans HFlex SpMM processing method, the parameters passed to the hardware accelerator are fixed. Specifically, memory pointers of **A**, **B**, **C**, and **Q**, and constant scalars M, K, N, α and β are passed to the accelerator. For a different SpMM, the data of matrix **A**, **B** and **C** only affects the contents stored in the memory space specified by the memory pointers. We pass the memory pointers and constant scalars according to a specific SpMM to the accelerator without changing the accelerator. Thus, Sextans supports the HFlex feature.

Table 1: Incremental and accumulative speedups with the increase of optimizations applied on `crystm03`.

	Baseline	OoO Scheduling	8 PUs	64 PEs
Incr.	1×	9.97×	7.97×	45.3×
Accum.	1×	9.97×	79.6×	3608×

3.5 Discussion on Other Architectural Issues

We discuss five architectural issues in this section.

• (1) **Streaming in B matrix.** SpMM actually issues random read to **B** which is stored in off-chip HBM but we need to alleviate the random read to off-chip memory. Matrix **B** is partitioned into windows (with a window size K_0). In one logical cycle, the random accessing only happens on a **B** window. So we stream in a **B** window (Line 4 of Algorithm 1) before invoking the 8 PEGs. Thus, the read accessing to HBM is sequentially batched.

• (2) **Initialization of C matrix.** Similar to the situation of accessing **B**, we can not afford the cost for random accessing **C** in off-chip memory. We keep an on-chip scratchpad memory to accumulate **C**. As a result, **C** must be initiated to be 0 (Line 2 of Algorithm 1). Each Sextans PE performs initiates a disjoint set of **C** in parallel.

• (3) **Irregular accessing on chip.** One input a-64b 4 will issue one random read access (indexed by a_col) to **B** and one random read and write access (indexed by a_row) to **C**. The two random accesses happen on on-chip memory. Although the latency for a specific access or computation is larger than 1 and the latency for processing one **A** element is 15 cycles on a Xilinx U280 FPGA, with Sextans non-zero scheduling we achieve an II=1 pipeline.

• (4) **Synchronization.** We do not place explicit synchronization barriers for the PEs. Instead, we use FIFO to form a loose synchronization which is implicitly implemented in the broadcasting where **B** elements are sent from one Read B module (producer) to PEs. The FIFO depth is 8. So at most 8 ahead/delay cycles of asynchronization are tolerated between PEs. If the asynchronization cycles are larger than 8, there is one PE (PEx) which has consumed all elements of its FIFO and another PE (PEy) whose FIFO is full. At the producer side, the sending is stalled. PEx keeps idle because the connected FIFO is empty. After PEy consumes at least one element in the connected FIFO, the processing resumes.

• (5) **Speedup breakdown.** To understand the speedup breakdown, in Table 1 we use Matrix `crystm03` as an example to show the incremental and accumulative speedups with the increase of the optimizations applied. For the baseline, we cache dense matrix blocks, scream in sparse matrix in row order (CSR), and there is no sharing. The 8 PUs relate to computation (sharing) optimization, the 64 PEs relate to memory optimization, and the OoO scheduling relates to both computation and memory optimizations.

3.6 Performance and On-chip Memory Resource Analysis

3.6.1 Performance Analysis. We use Algorithm 1 for performance analysis. The dimension of three matrices **A**, **B**, and **C** are $M \times K$, $K \times N$, and $M \times N$ respectively. The number of non-zeros in sparse matrix **A** is NNZ. For the initialization of **C** (Line 2), P PEs perform it in parallel, so the cycle count is:

$$t_{\text{initC}} = K/P. \tag{6}$$

Table 2: The specification of SpMM evaluation.

Number of SpMMs	1,400
Number of Matrices	200
Row/column	5 – 513,351
NNZ	10 – 37,464,962
Density	5.97E-6 – 4.00E-1
N	$N = 8, 16, 32, 64, 128, 256, 512.$

At Line 4, a window of **B** is streamed in. We partitioned the BRAM which stores **B** with a factor of $F_B = 4$. The BRAM has two ports, so we can store $2 \cdot F_B$ elements in one cycle. Thus the cycle count for streaming in **B** is:

$$t_{\text{streamB}} = K_0/(2 \cdot F_B). \tag{7}$$

In the PE region Line 6 - 10, the average non-zeros for each $\mathbf{A}_{pj}$ is $\text{NNZ}/(P \times (K/K_0))$. So the cycle count of Line 7 - 9 is:

$$t_{\text{PE}} = (\text{NNZ} \times K_0)/(P \times K). \tag{8}$$

We process the element-wise computation of Line 13 with a parallel factor of $F_c \times N_0$, $F_c = 16$. So the cycle count is:

$$t_{\text{compC}} = M/F_C. \tag{9}$$

The total cycle count is:

$$\begin{aligned} t &= (t_{\text{initC}} + (K/K_0) \times (t_{\text{streamB}} + t_{\text{PE}}) + t_{\text{compC}}) \times (N/N_0) \\ &= \left(\frac{K}{2 \cdot F_B} + \frac{\text{NNZ}}{P} + \frac{M}{F_C}\right) \times \frac{N}{N_0}. \end{aligned} \tag{10}$$

3.6.2 On-chip Memory Resource Analysis. Storing **B** windows consumes BRAMs. One BRAM block is 1024×18 bits. A window of $K_0 = 4096$ FP32 values requires $4096/1024 \times 2 = 8$ BRAM blocks. The partition factor is hidden because $F_B = 4 < 8$. With $N_0 = 8$ PUs for processing $N_0 = 8$ elements of **B** in parallel, we assign $8 \times N_0$ BRAM blocks for each PE. Since a BRAM block has two ports, we share one BRAM block between 2 PEs. So the total number of BRAM blocks used is $8 \times N_0 \times P/2 = 2048$.

We use URAMs as much as possible. A URAM block size is 4096×72 bits. One URAM entry can store 2 FP32 values. With $N_0 = 8$ PUs, we need 4 URAM blocks. We set a URAM depth of 12288 for each PE. So a total number of $12288/4096 \times 8/2\times = 12 \times 64 = 768$ URAM blocks are consumed, which accounts for 80% of available URAMs on a Xilinx U280 FPGA.

4 EVALUATION

4.1 Evaluation Setup

We evaluate on 1,400 SpMMs with 200 sparse matrices and 7 N values ranging from 8 to 512. Table 2 illustrates the properties of the sparse matrices. Of the 200 sparse matrices, we select 50 from SNAP [54] and 150 from SuiteSparse [28]. We exclude matrices which are out-of-memory for a 5 GB memory budget. The row/column number of the evaluated matrices spans 5 to 513,351. The number of non-zeros (NNZ) ranges from 10 to 37,464,962. The density of the sparse matrices ranges from 5.97E-6 to 4.00E-1. The evaluated sparse matrices include 73.5% of all non-temporal matrices in SNAP and Sextans supports 93.6% of matrices in SuiteSparse. We evaluate single floating-point (FP32) SpMM.

We evaluate on four platforms – an NVIDIA Tesla K80 GPU, an NVIDIA V100 GPU, an FPGA prototype Sextans and a projected prototype Sextans-P with higher bandwidth competitive to V100 and more frequency optimization. The performance of K80, V100 and Sextans is measured by runtime and the performance of Sextans-P is simulated. We list the specifications of the four evaluation platforms in Table 3. For Sextans accelerator, we first prototype the accelerator on a Xilinx U280 HBM FPGA. The FPGA prototype provides both a verification of the Sextans architecture and a reference performance model for simulation. Moreover, Sextans is a highly compatible working prototype which can be deployed in data center. For a fair comparison, we select two GPUs. K80 has a memory bandwidth of 480 GB/s which is comparable to the memory bandwidth of U280, i.e. 460 GB/s, since memory accessing and bandwidth are critical for sparse and graph workloads [4, 39, 86]. K80 is a more powerful platform than Sextans because the frequency of K80 (562 MHz) is much higher than the frequency of Sextans (189 MHz) and the memory bandwidth of K80 is also slightly higher. We measure the power consumption of FPGA by Xilinx Board Utility `xbutil` and the power of GPUs by `nvidai-smi`. We also use a high-end GPU V100 and configure the memory bandwidth, Sextans-P according to that of V100 for fair comparison. We set the frequency of Sextans-P to achieve 350 MHz with the help of Autobridge [37] which is ongoing (most designs achieved 350 MHz with Autobridge). According to $P = C \cdot V^2 \cdot f$, we project the measured power of Sextans by frequency increase to 96 W as the power of Sextans-P. Table 3 also lists the on-chip memory size which is sensitive to memory-bound applications and the peak SpMM throughputs of the four platforms.

Table 3: Process technology size, frequency, memory bandwidth, on-chip memory, power, and achieved peak SpMM throughput of the four platforms.

	Tech.	Freq.	Bdw.	On-chip Mem.	Power	Peak Th.
Tesla K80	28 nm	562 MHz	480 GB/s	24.5MB	130 W	127.8 GFLOP/s
Sextans	16 nm	189 MHz	460 GB/s	22.7MB	52 W	181.1 GFLOP/s
Tesla V100	12 nm	1.297 GHz	900 GB/s	33.5MB	287 W	688.0 GFLOP/s
Sextans-P	16 nm	350 MHz	900 GB/s	24.5MB	96 W	343.6 GFLOP/s

Table 4: Resource utilization of Sextans prototype on a Xilinx U280 FPGA board.

	Used	Available	Utilization (%)
BRAM	3086	4032	76
DSP48	3316	9024	36
FF	690,255	2,607,360	26
LUT	379,649	1,303,680	29
URAM	768	960	80

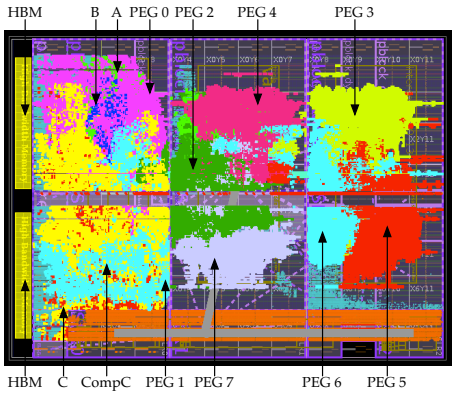


Figure 6: Layout of Sextans prototype on a U280 FPGA.

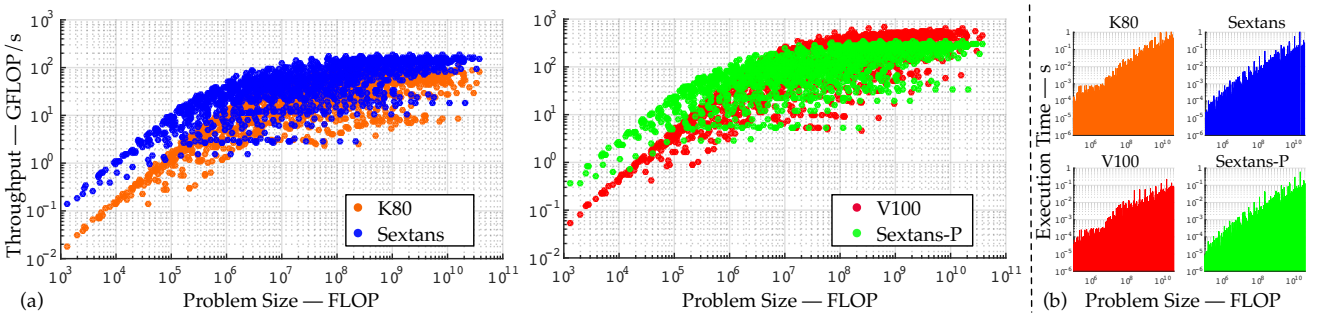


Figure 7: (a) Throughput (in GFLOP/s) and (b) execution time (in seconds) plotted with increasing problem size (in FLOP). The peak throughputs of K80, Sextans, V100, and Sextans-P are 127.8 GFLOP/s, 181.1 GFLOP/s, 688.0 GFLOP/s, and 343.6 GFLOP/s respectively.

For the two GPU platforms, we use CuSPARSE [57] routine `csrmm` for the execution of floating-point SpMM. The CUDA version is 10.2. We measure the GPU execution time with CUDA runtime API `cudaEventElapsedTime` [3]. For the FPGA prototype, we use Xilinx high level synthesis (HLS) tool Vitis 2019.2. We list the resource utilization of Sextans in Table 4. The utilization rates of block RAM (BRAM) and ultra RAM (URAM) are higher than other resources because SpMM is memory intensive. We show the layout of the Sextans accelerator in Figure 6. We only highlight the main components of the accelerator including eight processing engine groups (denoted by PEG 0 – 7 in Figure 6), memory reading units for sparse matrix **A** and dense matrix **B** (denoted by A, B in Figure 6), memory reading and writing units for dense matrix **C** (denoted by C in Figure 6), and the compute units for partial **C** (denoted by CompC in Figure 6). We launch the FPGA accelerator with OpenCL [73] runtime and measure the FPGA execution time. We build an in-house simulator to simulate the performance of Sextans-P after the prototyping on FPGA. The simulator is based on the emulation C++ code of Sextans. Since Sextans is a streaming accelerator, we model the computing time and memory accessing time and record the larger one as the processing time at each stage. The frequency and memory bandwidth in the simulation is configured the same as V100 that we list in Table 3.

4.2 Results

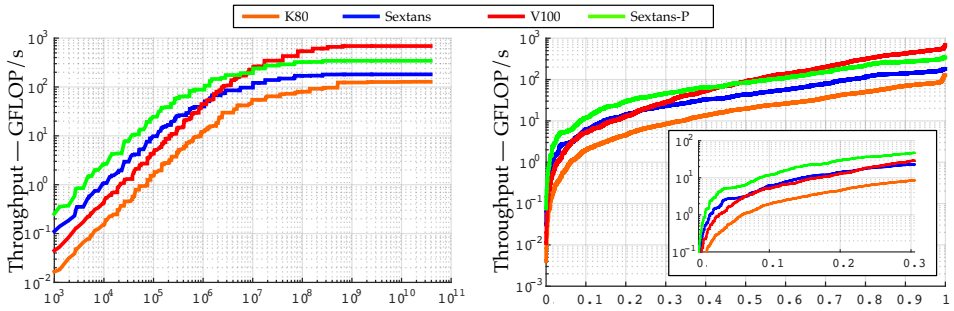
4.2.1 Overall Performance. We plot the throughput (in GFLOP/s) of 2,000 SpMMs of 200 sparse matrices with 7 N configurations (N = 8, 16, 32, ..., 512) on the four platforms in Figure 7 (a) and the execution time in second in Figure 7 (b) with the increase of the problem size in FLOP. The problem size is defined as the number of floating-point operations for executing one SpMM $\mathbf{C} = \alpha \cdot \mathbf{A} \times \mathbf{B} + \beta \mathbf{C}$, and the problem size is proportional to N. The throughput is calculated as p/t where p is the problem size and t is the execution time. Overall, the peak throughputs of K80, Sextans, V100, and Sextans-P are 127.8 GFLOP/s, 181.1 GFLOP/s, 688.0 GFLOP/s, and 343.6 GFLOP/s respectively. The geomean speedups of the four platforms normalized to K80 are 1.00×, 2.50×, 4.32×, and 4.94× respectively. The geomean speedup of Sextans-P to V100 is 1.14×.

From Figure 7 (a) we see the overall trend that with the increase of the problem size the throughput of the four platforms increases and after a problem size threshold is reached, the throughput gets saturated at the peak throughput of the four platforms. We compare Sextans with K80 in one subfigure and Sextans-P with V100 in another because the two platfroms in the same subfigure are comparable. We see the throughput dots of Sextans at the top of dots of K80. The dots of Sextans-P are higher than the dots of V100 for problem size $< 10^7$ FLOP. Although Sextans-P has the same memory bandwidth as V100, the frequency of V100 is much higher than that of Sextans-P (1297 MHz v.s. 350 MHz). So the saturated throughput of V100 is higher than that of Sextans-P.

For a problem size less than 10^6 FLOP, we see the throughput increases with the increase of problem size for the four platforms, because there are setup and ending processing overheads for the four platforms. For example, in Sextans architectures, on-chip memory for partial **C** is initialized before the main processing loop and **C** is written back to off-chip memory after the main processing loop. GPUs also need similar setup processing and writing back from on-chip buffer to device memory. However, with the increase of the problem size the overhead is amortized and it is better to parallelize a larger size problem. So we see the performance (GFLOP/s) increases. Notice that for problem size less than 10^6 FLOP, Sextans performs better than both K80 (computing power comparable with Sextans) and V100 (computing power much higher than Sextans). That is because CUDA runtime launches GPU SpMM kernels. The CUDA runtime has a small overhead which is not observable on large-size problems but on problems less than 10^6 FLOP, the overhead degrades the GPU performance. However, FPGA accelerators can fuse two or more kernels into one so the runtime overhead between kernels can be eliminated.

From Figure 7 (b) we see the execution time decreases successively from K80, Sextans, V100 to Sextans-P. For each specific platform, the execution time increases with the increase of the problem size. We can also see spikes in the execution time plots and dots deviating from the throughput trend. The trend of throughput and execution time can be mainly determined by the problem size for a specific platform, but for a specific sparse matrix, the non-zero distribution pattern also determines the execution time. So for matrices with a close problem size, the various non-zero distributions lead to distinguished execution times, which are reflected as spikes in Figure 7 (b) and deviating dots in Figure 7 (a).

4.2.2 Peak and CDF Performance. To better understand the performance of K80, Sextans, V100, and Sextans-P, we show the peak throughput with the increase of the problem size in Figure



(a) Peak throughput. (b) CDF throughput.

Figure 8: (a) Peak throughput with the increase of problem size and (b) CDF throughput of the four evaluated platforms.

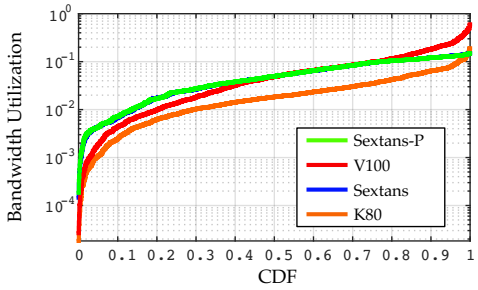


Figure 9: Memory bandwidth utilization.

8 (a) and the cumulative distribution function (CDF) throughput in Figure 8 (b). The peak throughput is defined as the maximum throughput of all problems whose size is smaller than a specific problem size (X-axis). The peak plot helps us to understand the peak performance on problems less than a specific size and the CDF plot helps to capture every performance point.

We compare the problem size for a specific platform to reach the peak throughput. Sextans and Sextans-P is the most efficient because Sextans and Sextans-P reaches its peak throughput at a smallest problem size around 8×10^7 FLOP. For the two GPU platforms they reach their peak throughput around $\times 10^9$ FLOP. In Figure 8 (b) we see Sextans-P has the highest throughput compared to the other platforms for CDF < 0.5. The larger gap indicates higher efficiency of one platform compared with another. We observe that for a problem size less than 10^6 FLOP and CDF less than 0.1, the throughputs of Sextans is higher than both K80 (computing power comparable with Sextans) and V100 (computing power much higher than Sextans). As we discussed before, that is because the small CUDA runtime is amplified on small-size problems.

4.2.3 Memory Bandwidth Utilization. We compare the memory bandwidth utilization of the four evaluated platforms in Figure 9. The memory bandwidth utilization is defined as $(4 \times (\text{NNZ} + N \times (2 \times M + K)))/\text{Bdw}$ where NNZ is the number of non-zeros of a sparse matrix, M is the number of rows of sparse matrix **A** and also the number of rows of dense matrix **C**, N is the number of columns of dense matrix **B** and **C**, and Bdw is the maximum available memory bandwidth as listed in Table 3. A factor 4 is multiplied because a floating-point value occupies 4 bytes. A factor 2 is multiplied to M because matrix **C** is read and written once each. Notice that the memory bandwidth utilization is not a memory bandwidth occupation rate which is calculated by (Occupied Bandwidth)/(Maximum Available Bandwidth). We do not use memory bandwidth occupation rate because an inefficient design where memory bandwidth is fully occupied but nothing is done can achieve a 100% memory bandwidth occupation rate. A higher memory bandwidth utilization is better but one cannot achieve a 100% memory bandwidth utilization for two reasons. First, it is impossible to read/write all matrices once which requires the on-chip memory to be as large as the off-chip memory. Second, for sparse matrix we only count NNZ here but the index also occupies memory space. For example, besides the 4 bytes for the value of a non-zero, coordinate list (COO) format uses another 4 bytes for row index and 4 bytes for column index. In compressed sparse row (CSR) format, another 4 bytes are needed by the index of each non-zero while the row index is compressed and the compressed row also occupies extra memory.

The geomean bandwidth utilization of the four platforms are 1.47%, 3.85%, 3.39%, and 3.88% respectively. Memory bandwidth utilization is relatively low for sparse workloads [4, 39, 86] especially for small size sparse matrices. SpMM is memory bound and the memory bandwidth utilization of Sextans-P is 1.15× of V100. Sextans-P and V100 work at the same memory bandwidth (900 GB/s). The memory bandwidth utilization translates to the 1.14× geaomean speedup of Sextans-P compared with V100. Sextans achieves a 2.62× bandwidth utilization compared to K80. The frequency and memory bandwidth of K80 is higher than that of Sextans (562 MHz / 480 GB/s v.s. 189 MHz / 460 GB/s). So Sextans achieves a 1.68× geomean speedup compared to K80. Although the memory bandwidth utilization of Sextans is 1.14× compared to the memory bandwidth utilization of V100, the frequency and memory bandwidth of Sextans is much lower than that of V100 (189 MHz / 460 GB/s v.s. 1297 MHz / 900 GB/s). So the geomean speedup of Sextans is lower than that of V100.

The maximum memory bandwidth utilization of K80, Sextans, V100, and Sextans-P are 19.00%, 14.92%, 59.96%, and 14.96% respectively. V100 achieves the highest memory bandwidth utilization and is significantly better than the other three platforms according to Figure 9. The maximum memory bandwidth utilization of FPGAs is lower than that of GPUs because of HLS tool limitation. The Xilinx HLS tool requires users to handle the connection of memory pointers to M AXI bundles. For SpMM we need to handle memory pointers of matrix **A**, **B**, **C**, and pointer list **Q**. The Xilinx U280 platform allows a maximum number of 32 M AXIs. There are 32 pseudo HBM channels. One memory pointer can only be mapped to one M

Table 5: Comparison with related sparse and dense accelerators.

	Kernels	Mat. NNZ	Prob. Size	Throughput	FPGA	Simulation	Real Exe.	HFlex
T2S-Tensor [72]	Dense MM, MV, etc[1]	2×10^3	-	738 GFLOP/s	**Yes**	No	**Yes**	No
AutoSA [77]	Dense MM, etc[1]	4×10^6	7×10^9	950 GFLOP/s	**Yes**	No	**Yes**	No
Tensaurus [71]	SpMV, SpMM, etc[2]	4.2×10^6	-	512 GFLOP/s[3]	No	**Yes**	No	No
[32]	SpMV	5×10^6	$< 1 \times 10^7$	3.9 GFLOP/s	**Yes**	No	**Yes**	No
Spaghetti [46]	SpGEMM	1.6×10^7	-	27 GFLOP/s	**Yes**	No	**Yes**	No
ExTensor [44]	SpMM, SpGEMM, etc[2]	6×10^6	-	64 GFLOP/s	No	**Yes**	No	No
SIGMA [61]	SpGEMM	-	-	-	No	**Yes**	No	No
SpArch [90]	SpGEMM	1.65×10^7	-	10.4 GFLOP/s	No	**Yes**	No	No
OuterSPACE [59]	SpGEMM	1.65×10^7	-	2.9 GFLOP/s	No	**Yes**	No	No
SpaceA [82]	SpMV	1.4×10^7	1.43×10^7	-	No	**Yes**	No	No
Sextans	SpMM	3.7×10^7	3×10^{10}	181.1 GFLOP/s	**Yes**	No	**Yes**	**Yes**
Sextans-P	SpMM	3.7×10^7	3×10^{10}	343.6 GFLOP/s	No	**Yes**	No	**Yes**

[1] Other dense tensor kernels such as TTMc, MTTKRP are also supported. [2] Other dense tensor kernels such as sparse TTM, sparse TTV are also supported.
[3] 512 GFLOP/s is achieved on dense multiplication, and the throughput of sparse multiplication is lower.

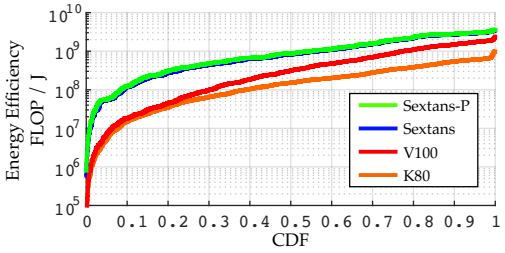


Figure 10: Energy efficiency of the four evaluated platforms.

AXI. Thus, the number of parallel HBM channels is significantly limited. As a result, the memory bandwidth utilization is low.

4.2.4 Energy Efficiency. We compare the energy efficiency of the four evaluated platforms in Figure 10. The energy efficiency is defined as $p/(t \cdot \text{Power})$ where p is the problem size, t is the execution time, and Power is the power consumption of a specific platform (listed in Table 3). The energy efficiency of Sextans and Sextans-P is similar. The geomean energy efficiency of K80, Sextans, V100, and Sextans-P are 1.06×10^8 FLOP/J, 6.63×10^8 FLOP/J, 2.07×10^8 FLOP/J, and 7.10×10^8 FLOP/J respectively. Sextans is 6.25× better than K80 and 3.20× better than V100. Normalized to K80, the energy efficiency improvements of Sextans, V100, and Sextans-P are 6.25×, 1.95×, and 6.70×, respectively. The maximum energy efficiency of the four platforms are 9.83×10^8 FLOP/J, 3.48×10^9 FLOP/J, 2.40×10^9 FLOP/J, and 3.60×10^9 FLOP/J respectively.

4.3 Comparison with Related Accelerators

We compare Sextans and Sextans-P with accelerators for similar sparse workloads in Table 5. We also include the accelerators for dense workloads as a comparison to see the performance gap between sparse and dense accelerators. The related accelerators include FPGA implementation and ASIC simulation works and the supported workloads include dense matrix matrix/vector multiplication (MM, MV), SpMM, sparse matrix vector multiplication (SpMV) and sparse matrix - sparse matrix multiplication (SpGEMM). Some accelerators also target for high order tensor operation such as metricized tensor times Khatri Rao product (MTTKRP) and tensor times matrix chain (TTMc). Compared with the related works, Sextans supports the largest sparse problem size (largest matrix NNZ and problem size) and achieves the highest throughput. Sextans is the only real executable accelerator for supporting SpMM. Sextans is the only accelerator which features the hardware flexibility (HFLex) which supports arbitrary problems directly by the hardware. For existing accelerators, to support a different problem configuration, the accelerator architecture must be modified. For ASIC accelerators, an architecture re-design and chip tape-out can take anywhere from weeks to months. For FPGA accelerators even with an automatic design flow such as AutoSA [77] where the architecture re-design time is saved, a new design will still need many hours or even a few days due to long synthesis and place/route time. Although an accelerator for a fixed-size problem can be built and a new problem can be decomposed as multiple kernels for the fixed-size accelerator, the runtime overhead for launching multiple kernel is high (0.15ms× launching times). The HFlex feature enables Sextans to support a new problem without re-running the time-consuming flows including synthesis/place/route for both FPGA and ASIC and a fabrication for ASIC.

5 CONCLUSION

SpMM is the key operator for a wide range of applications. We present Sextans, an accelerator for general-purpose SpMM processing. We propose (1) HFlex processing to enable prototyping the hardware accelerator once to support all SpMMs as a general-purpose accelerator, (2) PE-aware non-zero scheduling for balance workloads with an II=1 pipeline, and (3) on-chip and off-chip memory optimization to resolve the challenge of inefficient random memory accessing and off-chip accessing of large matrices. We present an FPGA prototype Sextans which is executable on a Xilinx U280 HBM FPGA board and a projected prototype Sextans-P with higher bandwidth competitive to V100 and more frequency optimization. We conduct comprehensive evaluation on 1,400 SpMMs on 200 matrices to compare Sextans with NVIDIA K80 and V100 GPUs. Sextans achieves a 2.50x geomean speedup over K80 GPU and Sextans-P achieves a 1.14x geomean speedup over V100 GPU (4.94x over K80).

ACKNOWLEDGMENT

We thank the anonymous reviewers and our labmates Weikang Qiao, Zhe Chen, and Licheng Guo, for their valuable feedbacks. This work is supported in part by NSF RTML Program (CCF-1937599), CDSC industrial partners [2] and Xilinx XACC Program.

[2] https://cdsc.ucla.edu/partners

REFERENCES

[1] [n.d.]. Alveo U280 Data Center Accelerator Card Data Sheet. https://www.xilinx.com/content/dam/xilinx/support/documentation/data_sheets/ds963-u280.pdf.

[2] [n.d.]. HIGH BANDWIDTH MEMORY (HBM) DRAM. https://www.jedec.org/standards-documents/docs/jesd235a.

[3] [n.d.]. How to Implement Performance Metrics in CUDA C/C++. https://developer.nvidia.com/blog/how-implement-performance-metrics-cuda-cc/.

[4] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 105–117.

[5] Aman Arora, Samidh Mehta, Vaughn Betz, and Lizy K. John. 2021. Tensor Slices to the Rescue: Supercharging ML Acceleration on FPGAs. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 23–33.

[6] Nathan Bell and Michael Garland. 2009. Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. 1–11.

[7] Maciej Besta, Raghavendra Kanakagiri, Grzegorz Kwasniewski, Rachata Ausavarungnirun, Jakub Beránek, Konstantinos Kanellopoulos, Kacper Janda, Zur Vonarburg-Shmaria, Lukas Gianinazzi, Ioana Stefan, Juan Gómez Luna, Jakub Golinowski, Marcin Copik, Lukas Kapp-Schwoerer, Salvatore Di Girolamo, Nils Blach, Marek Konieczny, Onur Mutlu, and Torsten Hoefler. 2021. SISA: Set-Centric Instruction Set Architecture for Graph Mining on Processing-in-Memory Systems. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 282–297.

[8] Damla Senol Cali, Gurpreet S. Kalsi, Zülal Bingöl, Can Firtina, Lavanya Subramanian, Jeremie S. Kim, Rachata Ausavarungnirun, Mohammed Alser, Juan Gomez-Luna, Amirali Boroumand, et al. 2020. GenASM: A High-Performance, Low-Power Approximate String Matching Acceleration Framework for Genome Sequence Analysis. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 951–966.

[9] Fan Chen, Linghao Song, Yiran Chen, et al. 2020. PARC: A Processing-in-CAM Architecture for Genomic Long Read Pairwise Alignment using ReRAM. In *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 175–180.

[10] Han Chen, Sergey Madaminov, Michael Ferdman, and Peter Milder. 2019. Sorting Large Data Sets with FPGA-Accelerated Samplesort. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 326–326.

[11] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. 2014. DaDianNao: A Machine-Learning Supercomputer. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 609–622.

[12] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. 2016. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE Journal of Solid-State Circuits* 52, 1 (2016), 127–138.

[13] Zhe Chen, Garrett J. Blair, Hugh T. Blair, and Jason Cong. 2020. BLINK: Bit-Sparse LSTM Inference Kernel Enabling Efficient Calcium Trace Extraction for Neurofeedback Devices. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*. 217–222.

[14] Zhe Chen, Hugh T. Blair, and Jason Cong. 2019. LANMC: LSTM-Assisted Non-Rigid Motion Correction on FPGA for Calcium Image Stabilization. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 104–109.

[15] Artem Cherkasov, Eugene N. Muratov, Denis Fourches, Alexandre Varnek, Igor I. Baskin, Mark Cronin, John Dearden, Paola Gramatica, Yvonne C. Martin, Roberto Todeschini, et al. 2014. QSAR Modeling: Where Have You Been? Where Are You Going To? *Journal of Medicinal Chemistry* 57, 12 (2014), 4977–5010.

[16] Yuze Chi, Guohao Dai, Yu Wang, Guangyu Sun, Guoliang Li, and Huazhong Yang. 2016. NXgraph: An Efficient Graph Processing System on a Single Machine. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 409–420.

[17] Yuze Chi, Licheng Guo, Jason Lau, Young-kyu Choi, Jie Wang, and Jason Cong. 2021. Extending High-Level Synthesis for Task-Parallel Programs. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 204–213.

[18] Young-kyu Choi, Yuze Chi, Weikang Qiao, Nikola Samardzic, and Jason Cong. 2021. HBM Connect: High-Performance HLS Interconnect for FPGA HBM. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 116–126.

[19] Young-kyu Choi, Yuze Chi, Jie Wang, Licheng Guo, and Jason Cong. 2020. When HLS Meets FPGA HBM: Benchmarking and Bandwidth Optimization. *arXiv preprint arXiv:2010.06075* (2020).

[20] Connor W. Coley, Wengong Jin, Luke Rogers, Timothy F. Jamison, Tommi S. Jaakkola, William H. Green, Regina Barzilay, and Klavs F. Jensen. 2019. A Graph-Convolutional Neural Network Model for the Prediction of Chemical Reactivity. *Chemical Science* 10, 2 (2019), 370–377.

[21] Jason Cong, Zhenman Fang, Muhuan Huang, Peng Wei, Di Wu, and Cody Hao Yu. 2018. Customizable Computing—From Single Chip to Datacenters. *Proc. IEEE* 107, 1 (2018), 185–203.

[22] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, Karthik Gururaj, and Glenn Reinman. 2014. Accelerator-Rich Architectures: Opportunities and Progresses. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[23] Jason Cong and Jie Wang. 2018. PolySA: Polyhedral-Based Systolic Array Auto-Compilation. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.

[24] Jason Cong, Peng Wei, Cody Hao Yu, and Peipei Zhou. 2018. Latte: Locality Aware Transformation for High-Level Synthesis. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 125–128.

[25] Guohao Dai, Yuze Chi, Yu Wang, and Huazhong Yang. 2016. FPGP: Graph Processing Framework on FPGA A Case Study of Breadth-First Search. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 105–110.

[26] Guohao Dai, Tianhao Huang, Yuze Chi, Jishen Zhao, Guangyu Sun, Yongpan Liu, Yu Wang, Yuan Xie, and Huazhong Yang. 2018. GraphH: A Processing-in-Memory Architecture for Large-Scale Graph Processing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 4 (2018), 640–653.

[27] William J. Dally, Yatish Turakhia, and Song Han. 2020. Domain-Specific Hardware Accelerators. *Commun. ACM* 63, 7 (2020), 48–57.

[28] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1–25.

[29] Johannes de Fine Licht, Grzegorz Kwasniewski, and Torsten Hoefler. 2020. Flexible Communication Avoiding Matrix Multiplication on FPGA with High-Level Synthesis. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 244–254.

[30] Tiziano De Matteis, Johannes de Fine Licht, and Torsten Hoefler. 2020. FBLAS: Streaming Linear Algebra on FPGA. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–13.

[31] Caiwen Ding, Shuo Wang, Ning Liu, Kaidi Xu, Yanzhi Wang, and Yun Liang. 2019. REQ-YOLO: A Resource-Aware, Efficient Quantization Framework for Object Detection on FPGAs. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 33–42.

[32] Jeremy Fowers, Kalin Ovtcharov, Karin Strauss, Eric S. Chung, and Greg Stitt. 2014. A High Memory Bandwidth FPGA Accelerator for Sparse Matrix-Vector Multiplication. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 36–43.

[33] Daichi Fujiki, Arun Subramaniyan, Tianjun Zhang, Yu Zeng, Reetuparna Das, David Blaauw, and Satish Narayanasamy. 2018. GenAx: A Genome Sequencing Accelerator. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 69–82.

[34] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. 2020. Sparse GPU Kernels for Deep Learning. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–14.

[35] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao, et al. 2021. Gemmini: Enabling Systematic Deep-Learning Architecture. Evaluation via Full-Stack Integration. In *Proceedings of the 58th Annual Design Automation Conference (DAC)*.

[36] Tong Geng, Ang Li, Runbin Shi, Chunshu Wu, Tianqi Wang, Yanfei Li, Pouya Haghi, Antonino Tumeo, Shuai Che, Steve Reinhardt, et al. 2020. AWB-GCN: A Graph Convolutional Network Accelerator with Runtime Workload Rebalancing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 922–936.

[37] Licheng Guo, Yuze Chi, Jie Wang, Jason Lau, Weikang Qiao, Ecenur Ustun, Zhiru Zhang, and Jason Cong. 2021. AutoBridge: Coupling Coarse-Grained Floorplanning and Pipelining for High-Frequency HLS Design on Multi-Die FPGAs. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 81–92.

[38] Licheng Guo, Jason Lau, Zhenyuan Ruan, Peng Wei, and Jason Cong. 2019. Hardware Acceleration of Long Read Pairwise Overlapping in Genome Sequencing: A Race Between FPGA and GPU. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 127–135.

[39] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. 2016. Graphicionado: A High-Performance and Energy-Efficient Accelerator for Graph Analytics. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–13.

[40] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*. 1025–1035.

[41] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *2016 ACM/IEEE 43rd Annual International Symposium on*

Computer Architecture (ISCA). IEEE Computer Society, 243–254.

[42] Song Han, Huizi Mao, and William J. Dally. 2015. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *arXiv preprint arXiv:1510.00149* (2015).

[43] Song Han, Jeff Pool, John Tran, and William J. Dally. 2015. Learning both Weights and Connections for Efficient Neural Networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems-Volume 1*. 1135–1143.

[44] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W. Fletcher. 2019. ExTensor: An Accelerator for Sparse Tensor Algebra. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 319–333.

[45] John L. Hennessy and David A. Patterson. 2019. A New Golden Age for Computer Architecture. *Commun. ACM* 62, 2 (2019), 48–60.

[46] Reza Hojabr, Ali Sedaghati, Amirali Sharifian, Ahmad Khonsari, and Arrvindh Shriraman. 2021. SPAGHETTI: Streaming Accelerators for Highly Sparse GEMM on FPGAs. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 84–96.

[47] Kuan-Chieh Hsu and Hung-Wei Tseng. 2021. Accelerating Applications Using Edge Tensor Processing Units. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[48] Yuwei Hu, Yixiao Du, Ecenur Ustun, and Zhiru Zhang. 2021. GraphLily: Accelerating Graph Linear Algebra on HBM-Equipped FPGAs. In *2021 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–9.

[49] Guyue Huang, Guohao Dai, Yu Wang, and Huazhong Yang. 2020. GE-SpMM: General-Purpose Sparse Matrix-Matrix Multiplication on GPUs for Graph Neural Networks. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12.

[50] Hongjing Huang, Zeke Wang, Jie Zhang, Zhenhao He, Chao Wu, Jun Xiao, and Gustavo Alonso. 2021. Shuhai: A Tool for Benchmarking High Bandwidth Memory on FPGAs. *IEEE Trans. Comput.* (2021).

[51] Wenqin Huangfu, Xueqi Li, Shuangchen Li, Xing Hu, Peng Gu, and Yuan Xie. 2019. MEDAL: Scalable DIMM based Near Data Processing Accelerator for DNA Seeding Algorithm. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 587–599.

[52] Sang-Woo Jun, Shuotao Xu, et al. 2017. Terabyte Sort on FPGA-Accelerated Flash Storage. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 17–24.

[53] Thomas N. Kipf and Max Welling. 2016. Semi-Supervised Classification with Graph Convolutional Networks. *arXiv preprint arXiv:1609.02907* (2016).

[54] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[55] Weifeng Liu and Brian Vinter. 2015. CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. 339–350.

[56] Kiran Kumar Matam, Gunjae Koo, Haipeng Zha, Hung-Wei Tseng, and Murali Annavaram. 2019. GraphSSD: Graph Semantics Aware SSD. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 116–128.

[57] Maxim Naumov, L. Chien, Philippe Vandermersch, and Ujval Kapasi. 2010. CUSPARSE Library. In *GPU Technology Conference*.

[58] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. *arXiv preprint arXiv:1906.00091* (2019).

[59] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. 2018. OuterSPACE: An Outer Product Based Sparse Matrix Multiplication Accelerator. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 724–736.

[60] Weikang Qiao, Jihun Oh, Licheng Guo, Mau-Chung Frank Chang, and Jason Cong. 2021. FANS: FPGA-Accelerated Near-Storage Sorting. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 106–114.

[61] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. 2020. SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 58–70.

[62] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: Edge-centric Graph Processing using Streaming Partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 472–488.

[63] Nikola Samardzic, Weikang Qiao, Vaibhav Aggarwal, Mau-Chung Frank Chang, and Jason Cong. 2020. Bonsai: High-Performance Adaptive Merge Tree Sorting. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 282–294.

[64] Yakun Sophia Shao, Jason Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, et al. 2019. Simba: Scaling Deep-Learning Inference with. Multi-Chip-Module-Based Architecture. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 14–27.

[65] Yongming Shen, Michael Ferdman, and Peter Milder. 2017. Escher: A CNN Accelerator with Flexible Buffering to Minimize Off-Chip Transfer. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 93–100.

[66] Yongming Shen, Michael Ferdman, and Peter Milder. 2017. Maximizing CNN Accelerator Efficiency Through Resource Partitioning. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 535–547.

[67] Linghao Song, Fan Chen, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. 2020. AccPar: Tensor Partitioning for Heterogeneous Deep Learning Accelerators. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 342–355.

[68] Linghao Song, Jiachen Mao, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. 2019. HyPar: Towards Hybrid Parallelism for Deep Learning Accelerator Array. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 56–68.

[69] Linghao Song, Xuehai Qian, Hai Li, and Yiran Chen. 2017. PipeLayer: A Pipelined ReRAM-Based Accelerator for Deep Learning. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 541–552.

[70] Linghao Song, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. 2018. GraphR: Accelerating Graph Processing Using ReRAM. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 531–543.

[71] Nitish Srivastava, Hanchen Jin, Shaden Smith, Hongbo Rong, David Albonesi, and Zhiru Zhang. 2020. Tensaurus: A Versatile Accelerator for Mixed Sparse-Dense Tensor Computations. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 689–702.

[72] Nitish Srivastava, Hongbo Rong, Prithayan Barua, Guanyu Feng, Huanqi Cao, Zhiru Zhang, David Albonesi, Vivek Sarkar, Wenguang Chen, Paul Petersen, et al. 2019. T2S-Tensor: Productively Generating High-Performance Spatial Hardware for Dense Tensor Computations. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 181–189.

[73] John E. Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science & Engineering* 12, 3 (2010), 66.

[74] Bor-Yiing Su and Kurt Keutzer. 2012. clSpMV: A Cross-Platform OpenCL SpMV Framework on GPUs. In *Proceedings of the 26th ACM international conference on Supercomputing*. 353–364.

[75] Robert M. Tomasulo. 1967. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development* 11, 1 (1967), 25–33.

[76] Yatish Turakhia, Gill Bejerano, and William J. Dally. 2018. Darwin: A Genomics Co-processor Provides up to 15,000 X Acceleration on Long Read Assembly. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. 199–213.

[77] Jie Wang, Licheng Guo, and Jason Cong. 2021. AutoSA: A Polyhedral Compiler for High-Performance Systolic Arrays on FPGA. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 93–104.

[78] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. 2021. GNNAdvisor: An Adaptive and Efficient Runtime System for {GNN} Acceleration on GPUs. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*. 515–531.

[79] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2016. Learning Structured Sparsity in Deep Neural Networks. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*. 2082–2090.

[80] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. 2007. Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms. In *SC'07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*. IEEE, 1–12.

[81] Lisa Wu, David Bruns-Smith, Frank A. Nothaft, Qijing Huang, Sagar Karandikar, Johnny Le, Andrew Lin, Howard Mao, Brendan Sweeney, Krste Asanović, et al. 2019. FPGA Accelerated INDEL Realignment in the Cloud. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 277–290.

[82] Xinfeng Xie, Zheng Liang, Peng Gu, Abanti Basak, Lei Deng, Ling Liang, Xing Hu, and Yuan Xie. 2021. SpaceA: Sparse Matrix Vector Multiplication on Processing-in-Memory Accelerator. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 570–583.

[83] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2018. How Powerful are Graph Neural Networks?. In *International Conference on Learning Representations*.

[84] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 590–604.

[85] Mingyu Yan, Lei Deng, Xing Hu, Ling Liang, Yujing Feng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. 2020. HyGCN: A GCN Accelerator with Hybrid Architecture. In *2020 IEEE International Symposium on High Performance*

Computer Architecture (HPCA). IEEE, 15–29.

[86] Carl Yang, Aydın Buluç, and John D. Owens. 2018. Design Principles for Sparse Matrix Multiplication on the GPU. In *European Conference on Parallel Processing*. Springer, 672–687.

[87] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 161–170.

[88] Jialiang Zhang and Jing Li. 2017. Improving the Performance of OpenCL-based FPGA Accelerator for Convolutional Neural Network. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 25–34.

[89] Mingxing Zhang, Youwei Zhuo, Chao Wang, Mingyu Gao, Yongwei Wu, Kang Chen, Christos Kozyrakis, and Xuehai Qian. 2018. GraphP: Reducing Communication for PIM-Based Graph Processing with Efficient Data Partition. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 544–557.

[90] Zhekai Zhang, Hanrui Wang, Song Han, and William J. Dally. 2020. SpArch: Efficient Architecture for Sparse Matrix Multiplication. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 261–274.

[91] Shijie Zhou, Charalampos Chelmis, and Viktor K. Prasanna. 2016. High-Throughput and Energy-Efficient Graph Processing on FPGA. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 103–110.

[92] Shijie Zhou, Rajgopal Kannan, Viktor K. Prasanna, Guna Seetharaman, and Qing Wu. 2019. HitGraph: High-throughput Graph Processing Framework on FPGA. *IEEE Transactions on Parallel and Distributed Systems* 30, 10 (2019), 2249–2264.

[93] Maohua Zhu, Tao Zhang, Zhenyu Gu, and Yuan Xie. 2019. Sparse Tensor Core: Algorithm and Hardware Co-Design for Vector-wise Sparse Neural Networks on Modern GPUs. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 359–371.

[94] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In *2015 {USENIX} Annual Technical Conference ({USENIX} {ATC} 15)*. 375–386.

[95] Youwei Zhuo, Chao Wang, Mingxing Zhang, Rui Wang, Dimin Niu, Yanzhi Wang, and Xuehai Qian. 2019. GraphQ: Scalable PIM-Based Graph Processing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 712–725.