DySpMM: From Fix to Dynamic for Sparse Matrix-Matrix Multiplication Accelerators

Hongyi Wang*
Tsinghua University
Beijing, China

Kai Zhong*
Tsinghua University
Beijing, China

Haoyu Zhang
Tsinghua University
Beijing, China

Shulin Zeng
Tsinghua University
Beijing, China

Zhenhua Zhu
Tsinghua University
Beijing, China

Xinhao Yang
Tsinghua University
Beijing, China

Shuang Wang
Tsinghua University
Beijing, China

Guohao Dai†
Shanghai Jiao Tong University
Shanghai, China

Huazhong Yang
Tsinghua University
Beijing, China

Yu Wang†
Tsinghua University
Beijing, China

ABSTRACT

Sparse Matrix-Matrix Multiplication (SpMM) is one of the key operators in many fields, showing *dynamic* features in terms of sparsity, element distribution, and data dependency. Previous studies have proposed FPGA-based SpMM accelerators with *fixed* configurations of on-chip dataflow, leaving three major challenges unsolved: 1) Partitioning matrices with the *fixed sub-matrix size* to fit limited on-chip buffer on FPGA leads to performance loss because the optimal sub-matrix size to minimize memory access varies with *dynamic* sparsity. 2) The *fixed row-wise allocation scheme* of sparse elements in streaming architecture leads to unbalanced workloads because of *dynamic* element distribution across sparse matrix rows. 3) Read-after-write (RAW) hazard caused by floating-point adder makes the elements in one row cannot be processed consecutively. Architectures with *fixed execution order* rely on time-consuming pre-processing to deal with *dynamic* data dependency. Motivated by the observation that *fixed* configurations lead to performance loss, we propose DySpMM by introducing the *dynamic* design methodology to SpMM architectures. The configurable data distributor is introduced to enable *dynamic sub-matrix size*, achieving up to 3.79× less memory access amount. The element-wise allocator is designed for *dynamic workload balance*, improving utilization up to 3.74×. The interleaved reorder unit is proposed to reorder the elements and *dynamically avoid RAW hazards* at runtime, avoiding time-consuming pre-processing. We implement DySpMM on U280 FPGA, and the evaluation shows that it achieves 1.42× geomean throughput compared with the state-of-the-art accelerator Sextans and 1.78× energy efficiency compared with V100S GPU.

ACM Reference Format:
Hongyi Wang, Kai Zhong, Haoyu Zhang, Shulin Zeng, Zhenhua Zhu, Xinhao Yang, Shuang Wang, Guohao Dai, Huazhong Yang, and Yu Wang. 2024. DySpMM: From Fix to Dynamic for Sparse Matrix-Matrix Multiplication Accelerators. In *61st ACM/IEEE Design Automation Conference (DAC '24), June 23–27, 2024, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3649329.3657362

1 INTRODUCTION

Sparse matrix has been widely used in many fields, such as social network [9] and circuit design [12]. As an important sparse linear algebra operator, Sparse Matrix-Matrix Multiplication (SpMM) is used in graph processing [1], compressed neural networks [13], and graph neural networks (GNN) [6]. SpMM shows ***dynamic*** features in terms of sparsity, element distribution, and data dependency. For example, the sparsity of the weight matrices in a pruned neural network is more than 1%, while the social network matrices in SNAP [3] can be less than 0.01%. Some prior studies [8] have worked on accelerating general SpMM tasks on GPUs, but the measured performance is generally lower than 10% of its theoretical peak performance while also consuming relatively high power at runtime. Thus, some FPGA-based accelerators [4, 7, 11] have been put forward recently for their high energy efficiency. They leverage streaming architecture to fully utilize the bandwidth of high bandwidth memory (HBM) and improve the performance. However, these designs use ***fixed*** configurations of on-chip dataflow for ***dynamic*** SpMM tasks, hindering further acceleration. Specifically, there are three challenges brought by ***fixed*** configurations.

First, **the fixed sub-matrix size ignores the varied features of different matrices, leading to redundant memory access**. To support arbitrary problem size, previous studies [4, 11] partition input matrices into sub-matrices with fixed sizes to fit on-chip buffer.

*Both authors contributed equally to this research.
†Corresponding authors.

DAC '24, June 23–27, 2024, San Francisco, CA, USA

ACM ISBN 979-8-4007-0601-1/24/06
https://doi.org/10.1145/3649329.3657362

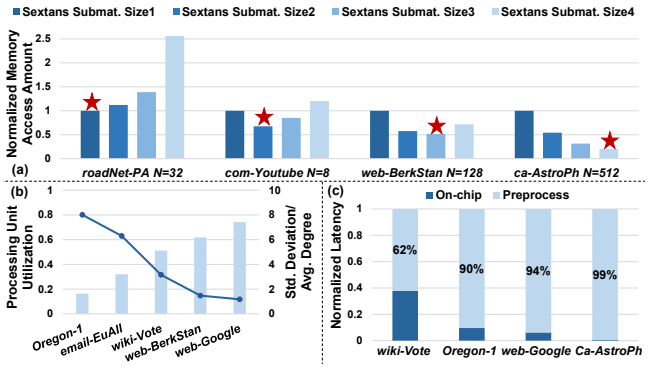

Figure 1: (a) Normalized memory access amount of Sextans [11] with different sub-matrix sizes. The optimal sub-matrix size varies with datasets. (b) The PE utilization rate of Sextans and the standard deviation over rows. Unbalanced matrices lead to low utilization. (c) The end-to-end latency of Sextans, whose pre-processing cost can be quite high.

However, we find that the optimal sub-matrix size to reduce memory access varies with matrix size and sparsity. As shown in Fig. 1(a), we list the memory access amount of state-of-the-art SpMM accelerator Sextans [11] with different sub-matrix size configurations. It shows that Sextans with fixed sub-matrix size may fetch 4.98× data from memory compared with the optimal configuration.

Second, **the fixed row-wise allocation scheme leads to an unbalanced workload, resulting in low PE utilization**. The existing streaming architectures bind sparse element stream, memory interface, and processing engine (PE) together. The sparse elements are fetched from parallel HBM channels and sent to PEs directly in a fixed row-wise order. Because sparse elements may be unbalanced distributed across rows, PEs that compute synchronously have to wait for the one with the heaviest workload, thus reducing the utilization and performance. As shown in Fig. 1(b), the imbalance degree of sparse matrices is measured by the standard deviation of non-zeros per row divided by its mean, the PE utilization of Sextans can be less than 20% on the highly unbalanced dataset.

Third, **the fixed execution order of PEs relies on time-consuming pre-processing to avoid read-after-write (RAW) hazard**. Fig. 1(c) shows the CPU pre-processing time can be 19× of the SpMM computation on FPGA. The pre-processing overhead can be omitted only when the same matrix is computed multiple times. However, many algorithms generate sparse matrices dynamically and process them only once, such as graph sampling in GNN [6]. It is necessary to reduce the pre-processing time of SpMM accelerators.

Motivated by the observation that ***fixed*** configurations lead to performance loss, we propose DySpMM with the ***dynamic*** design methodology to SpMM architectures. The contributions include:

1) The **configurable distributor** is designed to enable *dynamic sub-matrix size*. A lightweight theoretical model is proposed to predict the memory access amount and quickly select the optimal sub-matrix size configuration for each task, reducing the memory access by up to 3.79×.

2) The **element-wise allocator** is designed for *dynamic workload balance*. The merge tree that can add any number of adjacent input is introduced. The PE utilization is improved by up to 3.74×.

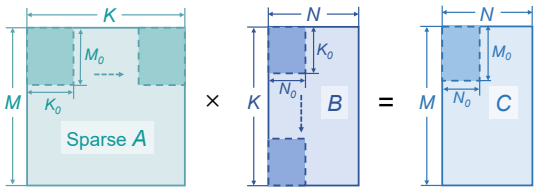

Figure 2: The matrix partitioning and computing sequence of SpMM. The iterating direction is shown by the arrow.

Algorithm 1: Determine sub-matrix size for each task

Input: sparsity S, matrix shape M, K, N, size of the basic operation N_b, size of result buffer $BufferSize$
Output: Optimal Sub-matrix Size M_0, N_0

$Data_{min} = INF$;
for n *in* $[1, 2, 4, 8]$ **do**
  $n_0 = n * N_b$; $m_0 = BufferSize/n_0$;
  $Data = 8SMK\left\lceil \frac{N}{n_0} \right\rceil + 4KN\left\lceil \frac{M}{m_0} \right\rceil + 8MN$;
  if $Data < Data_{min}$ **then**
    $Data_{min} = Data$; $N_0 = n_0$; $M_0 = m_0$;
return M_0, N_0;

3) The **interleaved reorder unit** is proposed for *dynamic hazard avoidance*. By reordering task queues in an interleaved way, it avoids pre-processing without increasing computation time.

We implement the dynamic design methodology of DySpMM on Xilinx U280 FPGA. Evaluation on 840 tasks with different matrix sizes and sparsity shows that DySpMM achieves **1.42**× geomean throughput compared with the state-of-the-art FPGA accelerator, and **1.78**× energy efficiency compared with V100S GPU.

2 BACKGROUND OF SpMM

A general SpMM can be represented as Eq. 1. A is a sparse matrix, B is a dense matrix, and C^{IN} is a dense matrix for accumulation. α and β are scalar coefficients.

$$C^{OUT} = \alpha A \times B + \beta C^{IN} \tag{1}$$

The on-chip memory resource of SpMM accelerator is usually limited. Therefore, it is necessary to partition the large matrix into sub-matrices, as Fig. 2 shows. The shapes of the matrix A, B, and C are $M \times K$, $K \times N$, and $M \times N$ respectively, as the corresponding sub-matrix shapes are $M_0 \times K_0$, $K_0 \times N_0$, and $M_0 \times N_0$. To indicate clearly, we refer X_{ij} to the sub-matrix at i-th row and j-th column of all sub-matrices of X (A, B, or C). Eq. 2 shows the multiplication of input sub-matrices and accumulation of partial results to get C_{ij}^{INTER}. Then, Eq. 3 shows how to accumulate it with the corresponding part of C_{ij}^{IN} to get the result sub-matrix C_{ij}^{OUT}.

$$C_{ij}^{INTER} = \sum_{k=0}^{\lceil \frac{K}{K_0} \rceil - 1} A_{ik} \times B_{kj} \tag{2}$$

$$C_{ij}^{OUT} = \alpha C_{ij}^{INTER} + \beta C_{ij}^{IN} \tag{3}$$

3 MEMORY ACCESS MODEL

Although partitioning matrices can fully utilize the on-chip buffer, the fixed sub-matrix sizes can lead to redundant memory access. In this section, we establish a theoretical model for memory access and discuss how to choose specific sub-matrix sizes for different SpMM tasks. As mentioned before, an SpMM task is partitioned

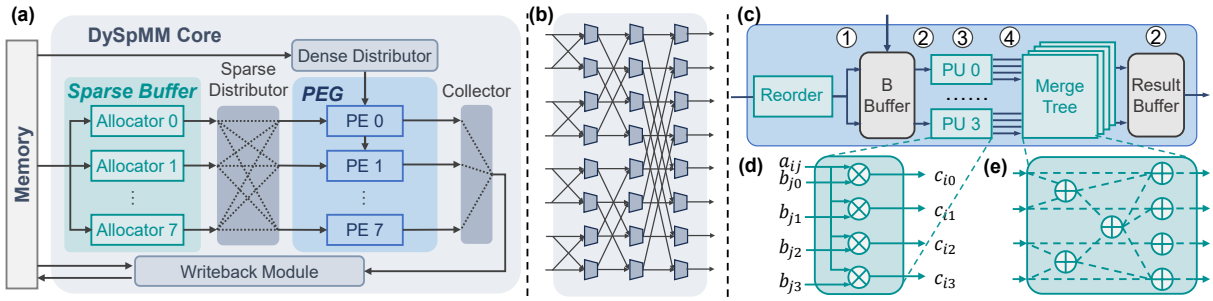

Figure 3: (a) Architecture of DySpMM Core. (b) The selector network in Sparse Distributor. (c) PE. (d) PU. (e) Merge Tree.

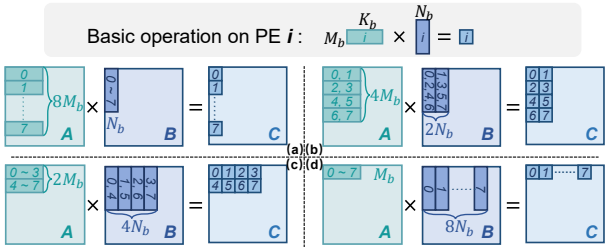

Figure 4: The basic operation of one PE and four mapping patterns of eight PEs. The numbers indicate which PE this data will be sent to.

to meet the buffer limitation, and the theoretical memory access amount can be expressed as Eq. 4.

$$Data = [(Read_{subA} + Read_{subB}) \times K_{ite} + Read_{subC} + Write_{subC}] \times M_{ite} \times N_{ite} \quad (4)$$

$Read_X$ refers to the memory access amount of matrix X (A, B, or C) and $Write_X$ refers to the output result data amount. Y_{ite} refers to the iterations on the Y (M, K, or N) dimension in each iteration. In this equation, both the memory access amount in each iteration and the iteration rounds depend on the sub-matrix size. As we intend to accumulate the partial result totally on-chip, the sub-matrix size is mainly limited by the result buffer size. To optimize total memory access, the problem can be formulated as Eq. 5.

$$\textbf{Min} \quad Data = 8SMK\left\lceil \frac{N}{N_0} \right\rceil + 4KN\left\lceil \frac{M}{M_0} \right\rceil + 8MN$$
$$\textbf{s.t.} \quad M_0 \times N_0 \le BufferSize \quad (5)$$

It can be seen that the optimal M_0 and N_0 vary with SpMM tasks, whose sparsity S and sizes M, N, and K can differ greatly. It is necessary to dynamically determine the sub-matrix size for each task. It is worth noting that there are more limitations to N_0 in the hardware implementation, as N_0 corresponds to the parallelism of the accelerator's vector unit. As shown in Sec. 4.2, for PEG has eight PEs, there are four options for N_0: N_b, $2N_b$, $4N_b$, and $8N_b$.

For each SpMM task, we use a lightweight host program to select the optimal M_0, N_0. As shown in Alg. 1, the program will try four configurations of N_0 and M_0, and incorporate them into Eq. 5 to obtain the minimum $Data$. Finally, we choose N_0 and M_0 with the minimum $Data$ from the four cases as the configuration for hardware execution. The DySpMM hardware is capable of dynamically configuring the distributor to support any one of these four configurations, as introduced in Sec. 4.2. Although Eq.5 assumes uniform distribution of sparse matrices, it can reduce the memory access amount in most cases compared to fixed sub-matrix size while avoiding time-consuming fine-grained analysis of sparse matrix.

4 DYSPMM HARDWARE DESIGN

4.1 DySpMM Architecture Overview

As shown in Fig. 3(a), a DySpMM Core consists of Sparse Buffer, Processing Engine Group (PEG), Sparse Distributor, Dense Distributor, Collector, and Writeback Module. Two Distributors control the on-chip data path to determine whether each PE receives the same or different input data, to support four configurations of dynamic sub-matrix size. More details will be explained in Sec. 4.2. Sparse buffer is responsible for fetching sparse elements from off-chip memory and scheduling them in an element-wise balanced way in each Allocator, which is further illustrated in Sec. 4.3. PEG is the computing module, and each PE performs the basic matrix multiplication, whose details will be introduced in Sec.4.4. Writeback Module executes Eq.3 and store the result to memory.

4.2 Distributor and Dynamic Sub-Matrix Size

There are multiple PEs in PEG, working in parallel. Each PE performs a basic SpMM operation between sub-matrices with the shape of $M_b \times K_b$ and $K_b \times N_b$, and the whole Core has a sub-matrices shape of $M_0 \times K_0$ and $K_0 \times N_0$, as shown in Fig. 4. The number of PE, M_b, K_b, and N_b are all hardware parameters that can be explored with the FPGA resource limitations, as discussed in Sec. 4.5. Here, we use eight PEs for demonstration. A straightforward way for eight PEs to work in parallel is shown in Fig. 4(a). Each PE individually receives sparse elements of different basic sub-matrices A and keeps the same copy of dense sub-matrix B, which is exactly Sextans and HiSparse adopt. The sub-matrix size of the whole Core is fixed to $M_0 = 8M_b, N_0 = N_b$. The difference is that $N_b = 8$ in Sextans, while $N_b = 1$ in HiSparse, as it is designed for SpMV tasks.

Unlike Sextans and HiSparse, which have fixed sub-matrix size, DySpMM enables more mapping patterns of PEs through configurable distributors, thereby achieving dynamic sub-matrix size. As described in Sec. 3, dynamically adjusting the sub-matrix size can reduce memory access and improve throughput. There are four mapping patterns for eight PEs in total, as shown in Fig. 4. In each pattern, the basic sub-matrix i of A, B, and C are assigned to PE i. To support different mapping patterns, Sparse Distributor, Dense Distributor, and Collector are designed. **For sparse matrix A**, the elements are fetched and scheduled by Allocators. The details of Allocator will be discussed in Sec. 4.3. The Sparse Distributor controls the data paths between Allocators and PEs. **For dense matrix B**, it is fetched and sent to the buffers in different PEs through Dense Distributor. We improve the chain-based broadcast network[11], enabling to configure each PE to receive the same or different data from sub-matrix B. **For result matrix C**, Collector is responsible

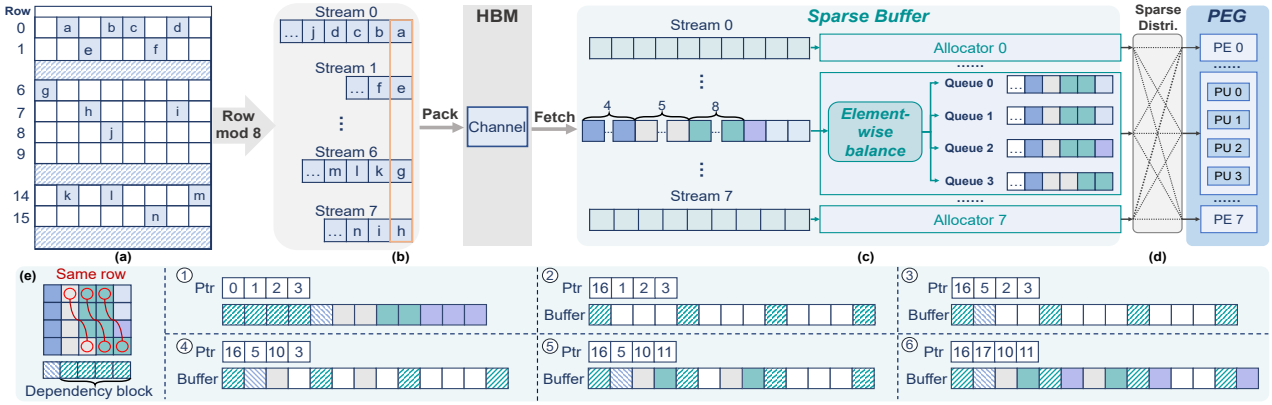

Figure 5: (a) Original sparse sub-matrix. (b) Packed and stored sparse elements in memory. (c) Allocator scheduling input streams for element-wise balance. (d) Workload distribution to PEs, each queue corresponding to a PU. (e) Reordering elements in dependency block to discontinuous positions, avoiding RAW hazard.

for fetching the result data from different PEs and sending them to the Writeback Module to execute Eq.3 and store in memory.

For example, in the first pattern in Fig. 4(a), the Sparse Distributor directly will build connections between Allocator i and PE i so that eight PEs receive different basic sub-matrices A. The Dense Distributor will set all PEs to save the same basic sub-matrix B. The sub-matrix size of the whole Core is $M_0 = 8M_b, N_0 = N_b$. For the second pattern in Fig. 4(b), the Sparse Distributor is configured to broadcast the sparse data in Allocator 0 to PE 0, 1 and then the data in Allocator 1 to PE 0, 1. The other pairs also keep the same behavior. In this case, every two PEs receive sparse elements from the same basic sub-matrix A. The Dense Distributor will set PE 0, 2, 4, 6 to save the same basic sub-matrix B, and PE 1, 3, 5, 7 to save another basic sub-matrix of B. In this case, combining eight PEs as the whole Core, the sub-matrix size is $M_0 = 4M_b, N_0 = 2N_b$.

4.3 Dynamic Workload Balance

In order to read sparse matrix in parallel, we adopt the packed format similar to [4, 5]. Fig. 5(a) shows a sparse sub-matrix, whose elements are randomly distributed among rows. In Fig. 5(b), these rows are grouped as eight streams based on the results of row indices modulo eight. For parallel data access, elements across streams are packed and stored continuously in one memory channel.

Fig. 5(c) shows how Allocator schedules workload to achieve dynamic workload balance. Packed matrix will be read from memory and unpacked into streams, each of which is scheduled by an Allocator. There are four queues in an Allocator, which will be processed by the corresponding four Processing Units (PUs) in PE, as shown in Fig. 5(d). We design the Allocator to write sparse elements to four queues in turn so that the workload of queues can be well balanced. We call it **element-wise allocation** scheme, which is balanced and different from the unbalanced row-wise allocation scheme that allocates the elements of the same row to a single PU.

It is worth noting that DySpMM achieves dynamic balance of PU workload through the element-wise allocation scheme, but there is still an issue that needs to be addressed. For SpMM tasks, the multiplication results of sparse elements from the same row should be added to get the final output. After the element-wise allocation, elements from the same row will be sent to different PUs in parallel, which means that the results of any number of adjacent PUs may need to be added. As shown in Fig. 5(c), in the first cycle, the results of PU 0 and PU 1 should be added because their elements are from the same row, while PU 2 and PU 3 should not be added because their elements are from different rows. We introduce a merge tree module in PE to solve this issue, as shown in Fig. 3(e). A four-input merge tree consists of seven adders in three layers. The first two layers can merge the results of any adjacent PUs. The elements with no neighbors from the same row will bypass to the last layer adder. The last layer is used to accumulate the merge results with the intermediate results fetched from result buffers. The output of the merge tree will be written back to the result buffer.

4.4 PE Design and Dynamic Hazard Avoidance

PE components design. As shown in Fig. 3(c), each PE has a reorder unit, a B buffer, four processing units (PUs), a merge tree, and a result buffer. Different components work in a pipeline manner. The pipeline stages of PE are marked with numbers in Fig. 3(c). Stage 1 represents the reorder unit used to resolve RAW hazards. Every cycle there are four elements sent out, and each element a_{ij} consists of row index, column index, and value. In Stage 2, the column index j and row index i are used to fetch the B data and intermediate data from B buffer and result buffer, respectively. In Stage 3, each PU executes a scalar-vector multiplication between the value of a_{ij} and the B data with four parallel multipliers, as shown in Fig. 3(d). In Stage 4, the merge tree adds the results of adjacent PUs with elements from the same row, accumulates the results with intermediate data, and writes back to the result buffer.

RAW hazard. Because the floating-point adder on FPGA is pipelined and has a delay of several cycles to ensure the frequency, it may cause RAW hazard when accumulating data. For example, in Fig. 5(c) and (e), there are sparse elements of the same row in adjacent cycle 0 and cycle 1 (in green), which means that their results should be accumulated with the same intermediate result. However, because of the multi-cycle latency of adders, the addition result of the first cycle has not been calculated yet when the accumulation needs to continue in the second cycle, thus RAW hazard will occur. To solve the RAW hazard, DySpMM processes the original queues with the reorder unit. As shown in Fig. 5(e), we call cycle 0, 1, 2, 3 within a dependency block, and cycle 4 is in another block. There are RAW hazards in the block since the adjacent cycles in

the block have sparse elements of the same row. And there is no hazard between blocks. The key point of the reorder unit is to rearrange elements in the same block to discontinuous positions in the queue, and rearrange elements from different blocks to continuous positions. In Fig. 5(e), 1) shows elements in the the original order. There are four (equal to the adder latency) registers as pointers to keep track of the possible positions to rearrange elements. 2) shows the output after four cycles. The reorder unit rearranges elements of the first dependency block to discontinuous positions 0, 4, 8, 12 of the output buffer. The first pointer changes to 16 as the next possible position. In 3), 4), and 5), the second, third, and fourth dependency blocks are reordered. In 6), the fifth dependency block (in purple) is rearranged to positions 5, 9, 13, since 5 is the smallest pointer. And it changes to 17 after reordering the fifth block.

4.5 Scalable Parameters and Implementation

There are six scalable parameters of DySpMM: n_{core}, n_{pe}, n_{pu}, N_b, d_B, and d_C. n_{core} represents the number of DySpMM Cores. Each Core utilizes independent memory interfaces for reading matrix A and writing matrix C, and shares the same memory interface to get matrix B in a broadcast way. n_{pe} is the number of PEs in each Core, which determines how many PE mapping patterns there are, and therefore limits the dynamic sub-matrix size to several possible configurations. As mentioned in Sec. 4.2, there are four mapping patterns for eight PEs. n_{pu} is the number of PUs in each PE, which also represents how many PUs to dynamically balance the workload. N_b is the number of multipliers in each PU. It is also the size and computation parallelism on the N dimension of the basic SpMM task mapped to each PE, as described in Sec. 4.2. It also represents the port width of the B buffer and result buffer. d_B and d_C represent the depth of B buffers and result buffers in each PE. Although Fig. 3 demonstrates eight PEs in each PEG, four PUs in each PE, and four multipliers in each PU, they can be adjusted. For specific FPGA platform, memory bandwidth, number of DSPs, and BRAM (and URAM) size are considered to guide the implementation. Take U280 FPGA as an example, and for the convenience of analysis, we assume that these parameters are powers of two.

Considering memory bandwidth, since each Allocator (equal to the number of PE) can schedule a 64-bit element per cycle, the bandwidth requirement to fetch A of each Core is $(8{\times}n_{pe})byte/cycle$. In common, an HBM channel can be connected with a 512-bit data path to reach $64byte/cycle$ bandwidth [14]. So n_{pe} should not be less than eight so that each Core can fully utilize at least one HBM channel. Because the Sparse Distributor sending data to PEs has the logical overhead $O(n_{pe} \cdot log(n_{pe}))$, we find that $n_{pe} = 16$ brings large LUT overhead. So $n_{pe} = 8$ in our implementation. There are 32 HBM channels on U280, we utilize 28 of them with $n_{core} = 8$. Each Core utilizes three channels for reading A, reading C, and writing result, and sharing four channels for reading B.

Considering DSP, the floating-point adder and multiplier are implemented by two DSPs. The number of multipliers in each Core is $n_{pe}{\times}n_{pu}{\times}N_b$, and the number of adders is $n_{pe}{\times}(2{\times}n_{pu}{-}1){\times}N_b$. There is also a Writeback Module using DSP, which performs Eq. 3, including two multiplications and one addition. Because the HBM channel for reading C supplies 512 bits (16 elements) per cycle, there are 32 multipliers and 16 adders in one Writeback Module. So the total utilization of DSP is $2 \times n_{core} \times (n_{pe} \times (3 \times n_{pu} - 1) \times N_b + 32 + 16)$. Because we have decided that $n_{core} = 8, n_{pe} = 8$, there are some possible n_{pe} and N_b configurations to maximize the parallelism under the limitation of U280's 9024 DSPs. We try Alg. 1 with different N_b as discussed in Sec. 3, and finally choose $n_{pe} = N_b = 4$ since it has the lowest average memory access.

Table 1: Resource utilization of DySpMM and baselines.

Resource	LUT	FF	DSP	BRAM	URAM
U280 Available	1304K	2607K	9024	2016	960
DySpMM	73%	53%	71%	74%	80%
Sextans	29%	26%	36%	76%	80%
HiSparse	47%	23%	8%	7%	53%

Table 2: Hardware Parameters of Four Accelerators.

	Frequency	Bandwidth	Power	GeoMean Perf.
V100S	1597 MHz	1134 GB/s	231 W	132.17 GFLOPS
Sextans	189 MHz	402 GB/s	52 W	44.08 GFLOPS
HiSparse	237 MHz	259 GB/s	45 W	1.96 GFLOPS
DySpMM	180 MHz	402 GB/s	61 W	62.58 GFLOPS
DySpMM$_s$	500 MHz	900 GB/s	173 W	231.36 GFLOPS

Considering BRAM, the B buffer is implemented with BRAM. A dual port BRAM macro is 36 bits wide and 1024 deep, and can be used as two single port buffers with one element width and 512 depth. So d_B should not be less than 512, and the number of BRAM used by B buffer is $n_{core} \times n_{pe} \times n_{pu} \times N_b \times (d_B/512)/2$. There are 2016 BRAMs, so $d_B = 512$ or $d_B = 1024$. We choose $d_B = 1024$ for more on-chip reuse of B, and thus $K_b = 1024$.

Considering URAM, the result buffer is implemented with URAM. A dual port URAM macro is 72 bits wide and 4096 deep, and can be used as two single port buffers with two elements width and 2048 depth. So d_C should not be less than 2048, and the number of URAM used by the result buffer is $n_{core} \times n_{pe} \times (N_b/2) \times (d_C/2048)/2$. There are 960 URAMs, so $d_C = 24576$ for largest buffer size to reuse data, and thus $M_b = 24576$.

5 EVALUATION

5.1 Evaluation Setup

Hardware Implementation. We implement DySpMM in HLS with TAPA[2] library and Vitis 2021.1, targeting Xilinx U280 FPGA equipped with 32-channel HBM. DySpMM runs at 180 MHz, and the power consumption provided by Vivado is 61 W. The latency is measured by the timing function in the host program.

FPGA baseline. State-of-the-art FPGA SpMM / SpMV accelerator Sextans [11] and HiSparse [4] are FPGA baselines, which are also implemented on U280 FPGA. The resource utilization is reported in Tab. 1. Compared to Sextans and HiSparse, DSP resources are more fully utilized in DySpMM, while DySpMM consumes the same bandwidth and on-chip buffers.

GPU baseline. We use Nvidia V100S with cuSparse [10] library as the GPU baseline. GPU power is measured by *nvidia-smi*. It should be noted that V100S has much higher frequency, larger bandwidth, and more computing resources compared to U280.

Datasets. We randomly select 120 sparse matrices from SuiteSparse [3] covering a wide range of sparsity and sizes. The number of rows, non-zeros, and sparsity ranges from $1.02{\times}10^2$ to $9.16{\times}10^5$, $1.53{\times}10^2$ to $3.68{\times}10^7$, and $5.97{\times}10^{-6}$ to $2.03{\times}10^{-1}$, respectively. For dense matrix, we also choose N=8, 16, 32, 64, 128, 256, and 512, which results in a total of 840 SpMM tasks.

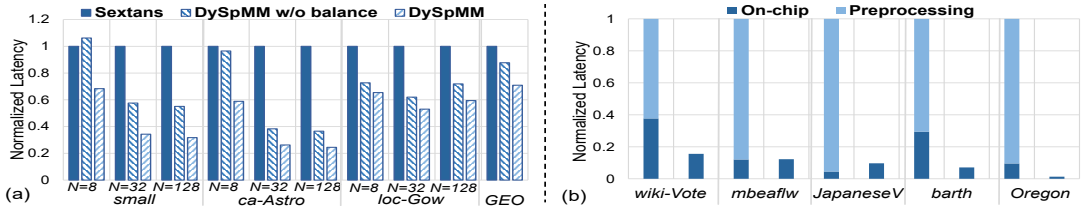

Figure 6: The ablation experiments for (a) dynamic sub-matrix size and dynamic workload balance on hardware performance, where each sparse matrix is evaluated by N=8, 32, and 128, and (b) the on-chip RAW hazard avoidance on latency ($N = 512$).

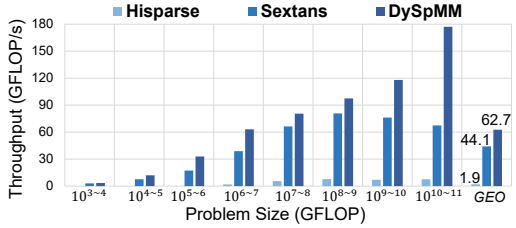

Figure 7: Throughput in different problem size ranges.

5.2 Performance Results

Compared with Existing FPGA Accelerators. Fig. 7 illustrates the geomean throughput across various problem size ranges. The problem size is defined as the number of floating-point operations, i.e., $2 \times nnz \times N + M \times N$. The geomean speed-up of DySpMM to Sextans and HiSparse are **1.42**× and **31.93**×, respectively. HiSparse shows significantly lower throughput, since its SpMV-oriented design lacks computing parallelism on dimension N. Compared with Sextans, DySpMM achieves significant speedup, which becomes more evident as the problem size increases. This is partially attributed to DySpMM's ability to better leverage the computational resources under the same memory and bandwidth constraints.

Compared to the GPU Baseline. The geomean throughput of different designs is shown in Tab. 2. The GPU beats all FPGA-based accelerators because of its higher memory bandwidth and computational resources. For fairer comparison, we build a C++ simulator to model the performance of DySpMM with 500MHz frequency, 900GB/s bandwidth, and 2× DSP. The scaled DySpMM$_s$ achieves 1.75× performance improvement compared to V100S.

5.3 Energy Efficiency Results

We further evaluate energy efficiency. The geomean energy efficiency of DySpMM, Sextans, HiSparse, and V100S are 1.02×10^9 FLOP/J, 8.48×10^8 FLOP/J, 4.35×10^7 FLOP/J, and 5.72×10^8 FLOP/J. DySpMM achieves **1.20**×, **23.45**×, and **1.78**× energy efficiency compared to Sextans, HiSparse, and V100S, respectively. DySpMM$_s$ has 2.34× energy efficiency improvement compared to V100S.

5.4 Ablation Study

Dynamic sub-matrix size. Fig. 6(a) shows the normalized latency of FPGA-based accelerators on different sparse matrices with different Ns. DySpMM tends to achieve better performance improvement on larger N for the same sparse matrix. The reason lies in smaller N constrains the search space of sub-matrix size, making it unable to explore parallelism on dimension N. The results show that DySpMM with only dynamic sub-matrix size (i.e., w/o balance) can achieve 1.14× geomean throughput compared with Sextans.

Dynamic workload balance. For the proposed element-wise workload balance, the performance improvement is relatively stable between tasks with the same sparse matrix, but depends heavily on the unbalance degree of the sparse matrix. The peak speedup of our element-wise scheduling increases with the degree of unbalance, achieving a maximum speedup of 3.74× and a geomean speedup of 1.24× based on dynamic sub-matrix in our testbench.

Dynamic RAW hazard avoidance. Since Sextans rely on pre-processing to resolve RAW hazards, we evaluate its pre-processing latency on Intel Xeon Silver 4208 CPU with multi-thread C++ program. In Fig.6(b), we compare the end-to-end latency. The result shows that pre-processing accounts for a non-negligible proportion of end-to-end latency, while DySpMM can avoid this overhead. The evaluation on the whole benchmark shows the geomean end-to-end speed-up considering the pre-processing is 14.13×.

6 CONCLUSION

This work analyzes shortcomings of the *fixed* design of SpMM accelerator and proposes DySpMM with *dynamic design methodologies*. The configurable distributor can enable dynamic sub-matrix size to reduce memory access. The element-wise allocator can dynamically balance the workload to improve utilization. The interleaved reorder unit can dynamically avoid RAW hazard to reduce pre-processing. DySpMM achieves **1.42**× geomean throughput compared to SOTA accelerator and **1.78** × energy efficiency compared with GPU.

ACKNOWLEDGMENTS

This work was supported by National Natural Science Foundation of China (No. 62325405, 62104128, U19B2019, U21B2031, 61832007), and Beijing National Research Center for Information Science and Technology (BNRist).

REFERENCES

[1] Buluç et al. 2011. The Combinatorial BLAS: Design, Implementation, and Applications. *Int. J. High Perform. Comput. Appl.* (2011).
[2] Yuze Chi et al. 2021. Extending High-Level Synthesis for Task-Parallel Programs. In *FCCM*. 204–213.
[3] Timothy A. Davis et al. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.*, Article 1 (dec 2011), 25 pages.
[4] Yixiao Du et al. 2022. High-Performance Sparse Linear Algebra on HBM-Equipped FPGAs Using HLS: A Case Study on SpMV. (2022).
[5] Jeremy Fowers et al. 2014. A high memory bandwidth fpga accelerator for sparse matrix-vector multiplication. In *FCCM*.
[6] William L. Hamilton et al. 2017. Inductive Representation Learning on Large Graphs. In *NIPS*.
[7] Yuwei Hu et al. 2021. GraphLily: Accelerating Graph Linear Algebra on HBM-Equipped FPGAs. In *ICCAD*.
[8] Guyue Huang et al. 2020. Ge-spmm: General-purpose sparse matrix-matrix multiplication on gpus for graph neural networks. In *SC*.
[9] Maxim Naumov et al. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. *CoRR* (2019).
[10] Nvidia. 2022. https://docs.nvidia.com/cuda/cusparse/index.html.
[11] Linhao Song et al. 2021. Sextans: A Streaming Accelerator for General-Purpose Sparse-Matrix Dense-Matrix Multiplication. *FPGA* (2021).
[12] Hanrui Wang et al. 2020. GCN-RL Circuit Designer: Transferable Transistor Sizing with Graph Neural Networks and Reinforcement Learning. *CoRR* (2020).
[13] Hanrui Wang et al. 2021. SpAtten: Efficient Sparse Attention Architecture with Cascade Token and Head Pruning. *HPCA* (2021), 97–110.
[14] Zeke Wang et al. 2020. Shuhai: Benchmarking high bandwidth memory on fpgas. In *FCCM*. IEEE.