A Hardware-Software Design Framework for SpMV Acceleration with Flexible Access Pattern Portfolio

Zhenyu Wu†, Maolin Wang‡*, Hayden Kwok-Hay So†
The University of Hong Kong†, The Hong Kong University of Science and Technology‡
{zhenyuwu, hso}@eee.hku.hk, maolinwang@ust.hk

***Abstract*—Sparse matrix-vector multiplications (SpMV) are notoriously challenging to accelerate due to their highly irregular data access pattern. Although a fully customized static accelerator design may be adequate for small problems that can fit entirely within an on-chip memory buffer, practical SpMV problems are large and have dynamic matrix structures that cannot easily be optimized at compile time. To address this need for trade-off between flexibility and performance, we present SPASM, a hardware-software framework that accelerates SpMV computation using a customizable portfolio of data access patterns as templates and a reconfigurable hardware to support their run-time execution. SPASM extracts local data access patterns of the input matrices and derives a set of template patterns to encode these inputs. Subsequently, a novel hardware computing structure is proposed to support vectorized computation and flexible switching between different template patterns for each tile computation. Furthermore, SPASM leverages the global compositions of input matrices to derive hardware configuration and workload schedules that improve load balancing among the parallel processing units. Importantly, although SPASM can optimize the pattern portfolio for a particular set of expected input matrices, the generated hardware can flexibly be used to accelerate SpMV of different input patterns albeit with reduced performance. Experimental results show that SPASM can achieve an average $2.81\times$ speedup compared to the state-of-the-art SpMV accelerator while keeping a relatively low customization cost.**

I. Introduction

Sparse matrix-vector multiplication (SpMV) is at the core of countless computational kernels across a wide spectrum of application domains from high-performance scientific computing [10], [23], to machine learning [14], [31] and graph analysis [15], [17]. A SpMV kernel executes the computation of Equation 1, where $\mathbf{A}$ is a sparse matrix, while$\vec{x}$ and $\vec{y}$ are dense vectors.

$$\vec{y} = \mathbf{A} \times \vec{x} + \vec{y} \tag{1}$$

While the dimensions of the input matrix may be large, the matrix $\mathbf{A}$ is sparse with limited number of non-zero (nz) values. As a result, accelerators commonly partition $\mathbf{A}$ in the row dimension for parallel computation as shown in Algorithm 1. Vector operations of the nz values for each partition can then be performed in parallel with shared access to the values of $\vec{x}$.

Algorithm 1 also highlights the key challenges associated with optimizing the SpMV kernel. First, SpMV involves accessing non-contiguous memory locations due to the sparse nature of the matrix $\mathbf{A}$, leading to highly irregular access pattern to the main memory. As shown in in line 5 of Algorithm 1, each multiply-and-accumulate operation requires a random access to both $\vec{x}$ and $\vec{y}$. This random access pattern is a major challenge for modern memory devices that are engineered for high-throughput contiguous block access. Special designs to the data layout in memory and sparse matrix encoding are often needed to enhance memory utilization for performance. Second, to fully exploit modern hardware capabilities, efficient vectorization and parallelization techniques are needed. Maximizing the utilization of vector processing units and leveraging multiple cores or threads are crucial to ensure high per-core performance. Finally, because of the varying sparsity patterns among the matrix partition, workload distribution across the parallel processing cores must be carefully balanced to optimal performance. As a result, a good schedule of the workloads of SpMV computation is important.

Algorithm 1 Parallel SpMV algorithm

```
1: Preprocess: A_partition = Partition(A, n_parallel)
2: Compute:
3: for n = 0 to n_parallel do in parallel
4:     for all nz in A_partition[n] do
5:         y[nz.y_idx]+= nz.val * x[nz.x_idx]
6:     end for
7: end for
```

To address these challenges, this work introduces SPASM, a Structured Pattern-Aware SpMV software-hardware framework to accelerate SpMV kernels. SPASM leverages both local patterns and global composition present within the sparse matrix, effectively overcoming the challenges discussed.

Exploring recurring block substructures, or local *patterns*, in sparse matrices is a common technique to regularize memory access patterns. [1], [10], [28] Our work extends previous software-only pattern-based sparse matrix representations with an automatic hardware design framework, allowing for a more flexible pattern portfolio. This flexibility provides more options to decompose irregular block substructures into regular ones.

The primary contribution of SPASM can be summarized as follows:

- **Flexible pattern portfolio support.** SPASM allows the customization of template patterns for matrices with

This work was supported by AI Chip Center for Emerging Smart Systems (ACCESS), sponsored by InnoHK funding, Hong Kong SAR. *Corresponding author: Maolin Wang.

different structured patterns with both software and hardware support. We propose a novel data format and algorithm that can adapt to diverse structured patterns. This adaptability enables SPASM to achieve significantly higher storage efficiency compared to existing sparse data formats.

- **The utilization of FPGA reconfigurability.** SPASM hardware is fully parameterized, allowing for customized designs for different global composition by allocating different bandwidths to sparse $\mathbf{A}$, $\vec{x}$ and $\vec{y}$. By leveraging this flexibility, SPASM can synthesize various hardware versions that are optimized for specific problem characteristics. This approach enables the generation of hardware implementations with different configurations that deliver optimal performance tailored to the specific problem at hand.

We implement the SPASM accelerator on FPGA and evaluate it on a wide range of sparse matrices with diverse densities and structured patterns. The results show that the SPASM data format shows the capability to exploit structured patterns and archives up to $2.40\times$ improvement in storage space cost compared to the COO format. Compared to the state-of-the-art SpMV accelerators, SPASM achieves average speedups of $6.74\times$ over HiSparse [7], $3.21\times$ over Serpens_a16 [25], and $2.81\times$ over Serpens_a24 [25].

II. Background & Motivation

Reducing the storage cost of sparse matrices is one of the key optimization methods of SpMV. Since the performance of SpMV is bounded by memory bandwidth, it will benefit from using a more compressed storage format for sparse matrices. Therefore, various sparse data formats are proposed to exploit structured patterns of sparse matrices. Pattern-aware data formats are designed to efficiently handle sparse matrices by leveraging their known structure. These formats are tailored to match specific patterns within the matrices as listed in Table I. Some research [9], [12], [18] use a composition of these formats to represent more complex patterns.

Existing coarse-grained, pattern-aware formats struggle to effectively manage the diverse pattern variations that occur in practice through composition. While storage formats with finer-grained patterns offer more flexible compositions, general-purpose architectures lack dedicated support for these formats. By jointly designing fine-grained pattern formats and associated hardware, there are opportunities for greater acceleration compared to popular formats running on general-purpose architectures.

A. Flexible composition of patterns with finer granularity

We use local patterns to refer to the fine-grained patterns that occur within small regions of the matrix. Examples of local patterns include row-wise, column-wise, diagonal [8], block [28], and DBB (Density Bound Block) [2], [16] patterns as demonstrated in Figure 1. In row-wise and column-wise patterns, the non-zero elements are concentrated along specific columns and rows. They are sometimes also called vector-wise patterns. Diagonal patterns are characterized by non-zero elements appearing specifically at the diagonal positions of submatrices within the sparse matrix. Block patterns refer to rectangular blocks of non-zero elements within the sparse matrix. DBB patterns are shown in the weight matrices in the machine learning domain. They are produced by imposing some pruning constraints. More local patterns can be generated through combinations of previous patterns.

TABLE I
AN OVERVIEW OF BASIC SPARSE DATA FORMATS.

Format	Pattern-aware?	Pattern	Padding?
COO	✗	None	✗
CSR & CSC	✗	None	✗
DIA	✓	Diagonal	✓
BSR	✓	Block	✓
ELL	✓	Diagonal, Banded	✓

These local patterns provide extra flexibility of representing diagonal, block diagonal, banded matrices, and more complex matrices arise in practice. We use global composition to refer to the composition of local patterns. Global patterns span across the entire matrix and exhibit properties that are consistent throughout the matrix. global composition may contain different types of local patterns as illustrated in Figure 1. global composition provide insights into the overall structure of the matrix and enable more extensive optimizations of workload scheduling across the entire computation.

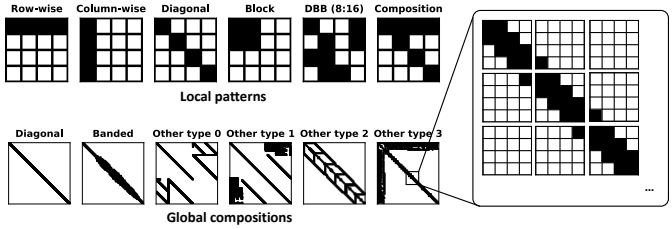


Fig. 1. Different types of local patterns and global composition.

B. local pattern frequency analysis

In this study, we focused on capturing the local patterns present within a 4-by-4 submatrix. This results in a total of $2^{4\times4}-1 = 65535$ possible local patterns, excluding the empty block. However, it is important to note that these patterns do not occur with equal frequency. Instead, certain patterns tend to dominate while others are rarely or never observed.

As depicted in Figure 2, which showcases the `cfd2` matrix, the top-8 most frequently occurring local patterns account for 48.21% of all observed patterns. This demonstrates that a small subset of patterns is responsible for a significant portion of the occurrences. For `Chebyshev4` matrix, it is also the similar case.

Figure 3 presents further evidence supporting the observation that most matrices are dominated by a small number of

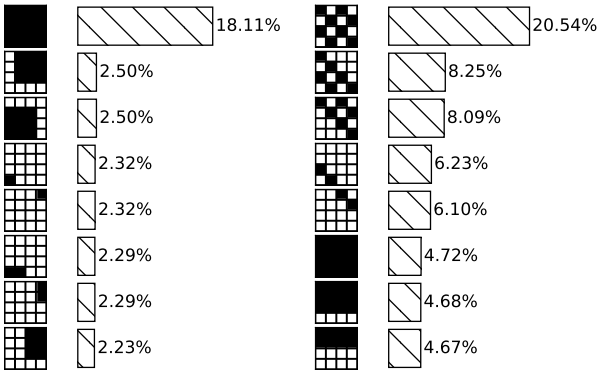


Fig. 2. Top-8 occurring local patterns and their frequency in `raefsky4` and `Chebyshev4` matrix. Dark grids represent non-zero elements.

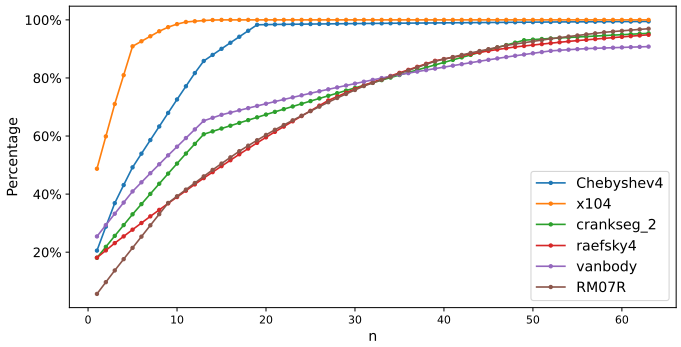


Fig. 3. CDF of top n occurring local patterns in different sparse matrices

patterns, even if these patterns may vary significantly across different matrices. This feature of local patterns enables the efficient customization and processing for each matrix by focusing only on the top-n occurring patterns, i.e. it's not necessary to process all occurring patterns but also a small portion of top-n occurring patterns. Furthermore, n could be varying when we let the top-n patterns counts up a certain portion of the total occurring patterns.

C. Hardware-friendly pattern portfolios

Harnessing the potential of local patterns directly is indeed a challenge, primarily due to the varying lengths of different patterns. These variations not only complicate pattern encoding but also increase the hardware complexity required for effective MAC operations. To address this challenge and make better use of local patterns, we propose the concept of *template patterns*.

Template patterns are defined as a series of local patterns with a fixed length. We can obtain all possible local patterns with some necessary paddings of zero. Paddings are needed when template patterns cover some locations that are originally empty and multiple template patterns overlap at certain locations.

Figure 4 shows the procedure of pattern decomposition. Consequently, the choice of template pattern can significantly impact the final number of paddings, particularly for matrices with distinct characteristics in their structured sparse patterns.

By utilizing template patterns, we can effectively handle local patterns in a more uniform manner. This approach mitigates the encoding complexities associated with varying pattern lengths and simplifies MAC operations by working with fixed-length templates. It provides a standardized representation and processing framework while enabling pattern-specific customization through the selection of appropriate template patterns.

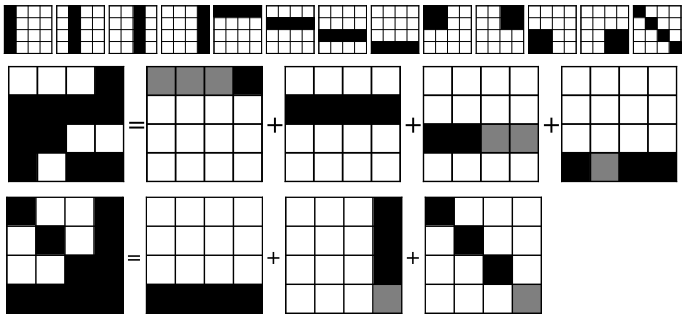


Fig. 4. Examples of the local pattern decomposition based on the given template patterns. Gray grids represent the paddings.

III. SPASM SPARSE DATA FORMAT

The SPASM data format utilizes a two-level tiling scheme to leverage both local and global composition present in sparse matrices. The SPASM data format is designed to be hardware-friendly, as the indices within the format can be directly used for memory and vector buffer accessing. This direct utilization of indices simplifies data access operations and enhances hardware efficiency in processing sparse matrices. Figure 5 demonstrates the encoding of a sparse matrix by proposed data format.

First, the local patterns are decomposed into combinations of template patterns. Then they are further tiled with the given tile size. The global composition are represented in the COO data format, where the `tileRowIdx` and `tileColIdx` serve as the column and row indices of non-empty tiles.

For the local patterns, a set of template patterns is selected based on the specific structure of the matrix. The position encoding of these local patterns consists of 5 fields. 13-bit `c_idx` and `r_idx` represent the column and row indices of the 4-by-4 submatrices within each tile. 1-bit `CE` indicates whether the current tile is the last one of the column. 1-bit `RE` indicates whether the current tile is the last one of the row. `RE` and `CE` also serve as controlling signals of vector buffers. The 4-bit `t_idx` stands for the template identifier. The position encoding of local patterns occupies 32 bits of storage, with each set of four values sharing the common position encoding.

With these position encoding fields, the proposed sparse data format can represent a total of 16 different template patterns. Additionally, the format supports a maximum tile size of $2^{13} \times 4 = 32768$.

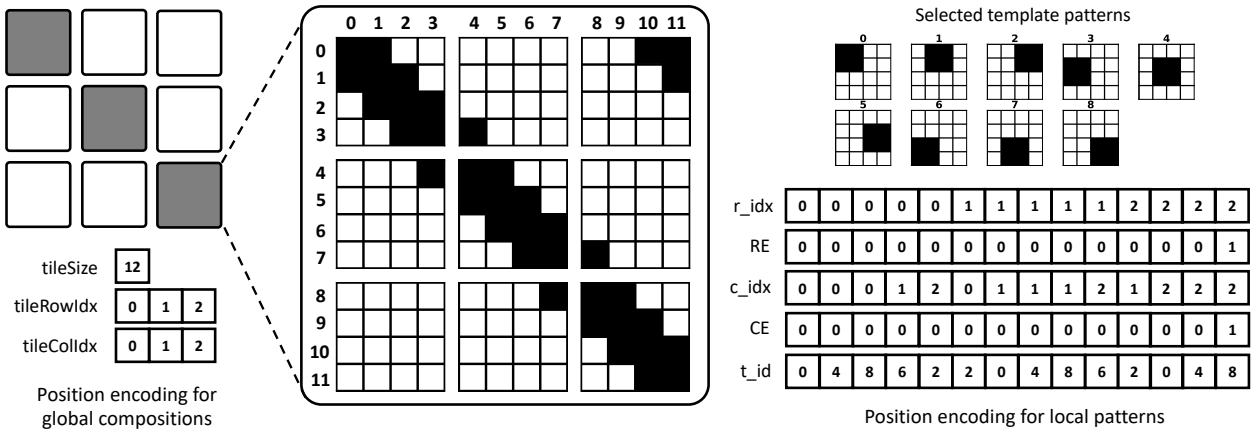


Fig. 5. An example of a block-diagonal sparse matrix encoded in the proposed SPASM sparse data format.

By combining the COO format for global composition and the position encoding for local patterns, the SPASM data format effectively captures and represents both local and global composition within sparse matrices. This allows for efficient storage and processing of structured sparse data while leveraging the benefits of template-based pattern encoding.

IV. SPASM FRAMEWORK

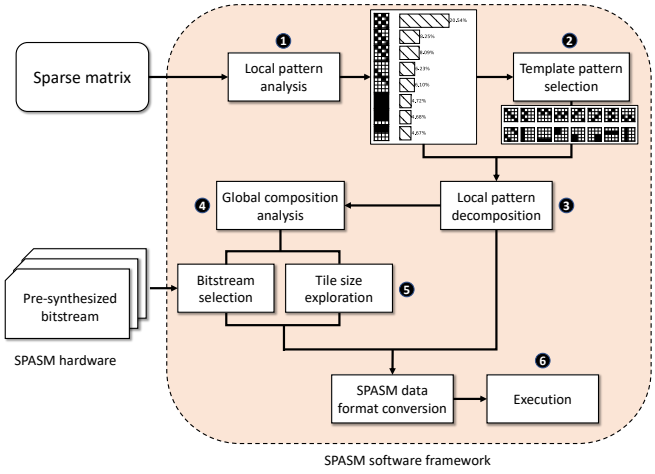


Fig. 6. The workflow of SPASM framework.

The overall workflow of the SPASM framework, as illustrated in Figure 6, consists of the following steps: ❶ local pattern analysis, ❷ template pattern selection, ❸ local pattern decomposition, ❹ global composition analysis, ❺ workload schedule exploration, and finally ❻ hardware execution.

In this section, we will provide detailed explanations for each step of the SPASM workflow.

A. Pattern decomposition

Finding the optimal way to decompose an arbitrary local pattern based on the given template patterns is the key of the SPASM framework. We use a brute force search algorithm to iterate over all possible ways to compose a pattern based on the given template patterns. A 16-bit `short int` presents the combination of all 16 template patterns. Then the one with the minimal number of paddings is selected as the optimal decomposition. The detailed code of the decomposition procedure is shown in Listing 1.

```
def find_best_decomp(pattern,templates):
    n=len(templates)
    best_num_paddings=16
    for i in range(2**n):
        remain=pattern
        num_padding=0
        overlap=0
        for t_id in range(n):
            if i&(1<<temp_id):
                padding=(~remain|overlap)&templates[t_id]
                overlap=overlap|templates[t_id]
                remain=remain&~templates[t_id]
                num_padding+=bin(padding).count('1')
        if num_padding<best_num_padding
            best_num_padding=num_padding
            best_decomp=i
    return best_decomp, best_num_paddings
```

Listing 1. local pattern decomposition procedure

B. Local pattern extraction

The procedure for extracting local patterns comprises the following steps:

❶ Local pattern analysis: In this initial stage, the sparse matrix is divided into 4-by-4 submatrices, and each submatrix is represented using a 16-bit long bitmask. The local pattern analysis aims to identify the occurrence and distribution of patterns within these submatrices. By analyzing the submatrices, a histogram of pattern distribution is generated and stored in a list of `(bitmask, frequency)` as shown in Algorithm 2.

❷ Template pattern selection: During the template pattern selection stage, a set of template patterns is generated to minimize the paddings associated with the observed patterns. These template patterns are selected to represent the local patterns effectively. The program optimizes the preprocessing phase by

Algorithm 2 ❶ local pattern analysis

```
function LP_ANALYSIS(mtx)
  ▷ Tile the matrix into 4-by-4 submatrices
    submtx = MTX_TILE(mtx, size=(4, 4))
    Initialize the key-value pair pfreq[]
    for all blk in submtx do
        blk_bitmask = BITMASK_GEN(blk)
        if blk_bitmask in pfreq.key then
            pfreq[blk_bitmask] += 1
        else
            pfreq.APPEND(blk_bitmask)
            pfreq[blk_bitmask] = 1
        end if
    end for
    return pfreq
end function
```

selectively decomposing the top-n patterns and evaluating the padding rate across different sets of template patterns. Since the top-n patterns hold significant importance and account for the majority of patterns present as discussed in the previous section, it becomes unnecessary to decompose all occurring patterns, enabling faster preprocessing. The template selection algorithm is shown in the Algorithm 3.

Algorithm 3 ❷ Template pattern selection

```
    Input: sparse matrix mtx, a set of template patterns temps
    Output: optimal template patterns best_temp of the sparse matrix
pfreq = LP_ANALYSIS(mtx)
SORT_BY_VALUE(pfreq)
  ▷ Select the top-n patterns that count up to a portion of total patterns
subset_pfreq = SELECT_TOP_N(pfreq, thresh)
  ▷ Find the best template patterns that minimize the number of paddings
best_paddings = inf
for all t in temps do
    num_paddings = 0
    for all p in subset_pfreq do
        _, paddings = FIND_BEST_DECOMP(p.key, t)
        num_paddings += p.val * paddings
    end for
    if num_paddings < best_paddings then
        best_paddings = num_paddings
        best_temp = t
    end if
end for
```

❸ Local pattern decomposition: Once the template patterns are selected, the program will decompose all occurring patterns by `find_best_decomp` routine.

C. Global composition extraction & workload schedule

To fully utilize the global composition of the sparse matrix, SPASM performs the following steps:

❹ Global composition analysis: In this step, the program will try to tile the local patterns and generate the position encoding for global compositions.

❺ Workload schedule exploration: The workload schedule exploration phase comprises two parts: bitstream selection and tile size exploration. SPASM hardware is parameterized and multiple versions of hardware are generated. Performance models are established for all pre-synthesized hardware. Then the program will try all possible combinations of hardware configurations and tile sizes. Given that different tile sizes yield distinct global compositions, any change in tile size triggers a return to the ❹ global composition analysis stage. This ensures that updated position encodings are generated to align with the revised tile size. The best combination of hardware configuration and tile size will be selected to maximize the performance.

The ❹ global composition analysis and ❺ workload schedule exploration are combined in one routine, which is shown in Algorithm 4.

Algorithm 4 ❹❺ Workload schedule

```
    Input: Decomposed pattern list plist, hardware configuration set hwconfig_set, tile size range tile_size_set
    Output: Selected hardware configuration best_hwconfig, selected tile size best_tile_size
best_cycles = inf
for all tile_size in tile_size_set do
  ▷ Generate the global composition list of given tile size
    GC_list = GC_GEN(tile_size)
    for all hw_config in hw_config_set do
  ▷ Estimate the performance on each hardware configuration
        cycles = PERF_MODEL(GC_list, hw_config, tile_size)
        if cycles < best_cycles then
            best_tile_size = tile_size
            best_hw_config = hw_config
            best_cycles = cycles
        end if
    end for
end for
```

D. Hardware architecture

1) VALU architecture: VALU (Vector Arithmetic Logic Unit) is purposefully designed to facilitate the multiplication of a template pattern and a vector. The architecture of VALU, as depicted in Figure 8, consists of 4 multipliers and 3 adders, enabling efficient computation and arithmetic operations. In the initial stage of VALU, the multipliers receive two inputs. One input is from the input vector buffer, while the other is selected by a 4-to-1 mux from the template pattern. This configuration allows for the targeted multiplication of specific elements from the template pattern with corresponding elements from the input vector. The adders in VALU acquire their operands through muxes, which are connected to different nodes as illustrated in the figure. By manipulating the selection signals of these muxes, the appropriate operands are chosen, facilitating arithmetic addition operations.

To construct the output vector, VALU utilizes 4 8-to-1 muxes. These muxes selectively pick results from eight possible nodes, combining them to generate the desired output vector. By controlling the selection signals of these muxes, VALU can perform arbitrary multiplications between a tem-

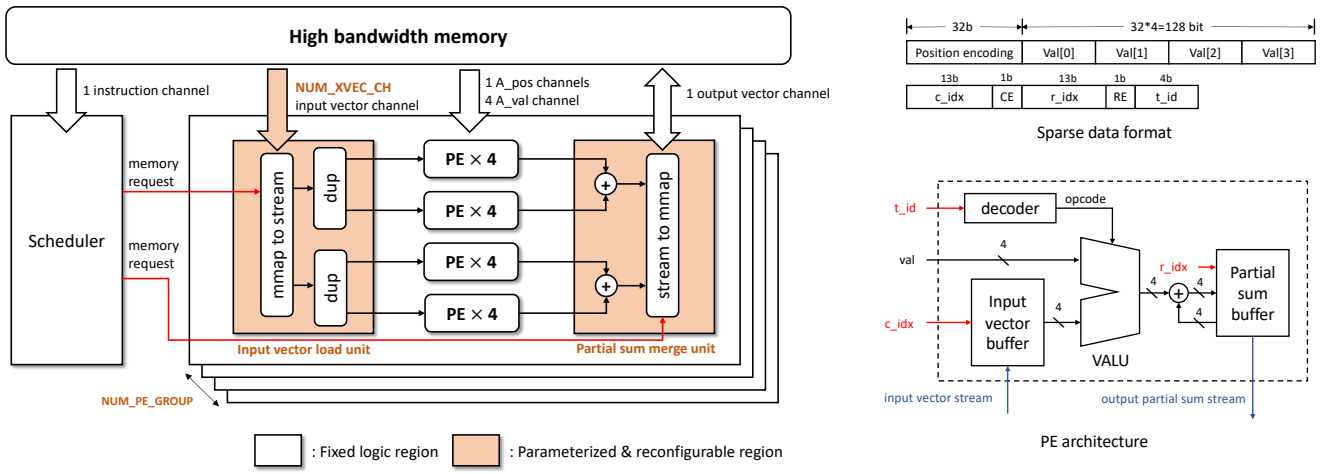


Fig. 7. The Overall architecture of SPASM and the proposed sparse data format.

plate pattern and a vector, ultimately forming a 30-bit long opcode.

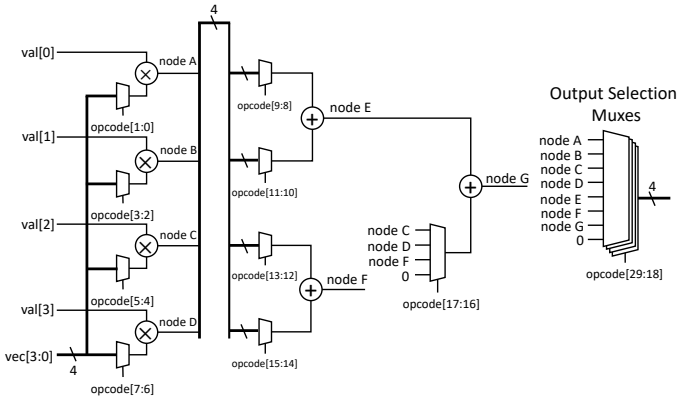


Fig. 8. The detailed architecture of VALU.

2) PE architecture: A SPASM PE consists of several components: an input buffer for vector $\vec{x}$, a partial sum buffer for vector $\vec{y}$, a VALU opcode decoder, and the VALU itself. The PE exhibits a parallelism of 4, enabling it to process four scalar-scalar multiplications simultaneously. The input vector buffer is organized in a double-buffered fashion. As a result, while the PE performs computations for the current tile, it simultaneously fetches the data for the next tile. This overlapping of data retrieval and computation enhances overall efficiency. One of the operands required by the VALU is directly obtained from the value stream of the sparse matrix $\mathbf{A}$, while the other operand is the corresponding packed $\vec{x}$ vector indexed by the `c_idx`. The VALU opcode decoder is implemented using a look-up table. At the initialization stage, the look-up table is loaded with the opcode of problem-specific patterns. It decodes the appropriate opcode for the VALU based on the `t_id`, ensuring the VALU performs the desired operation accurately. By changing the content of the look-up table, the SPASM PE is able to support the processing of flexible pattern portfolios.

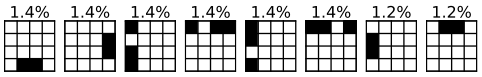
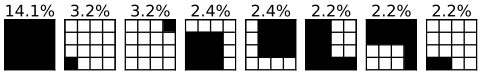
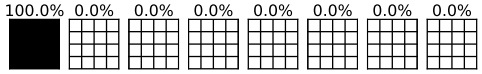
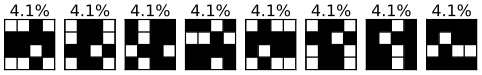
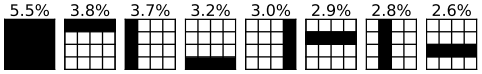
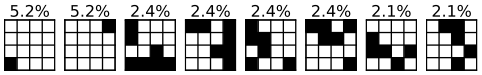
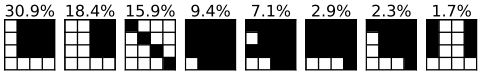
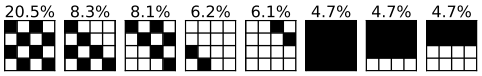
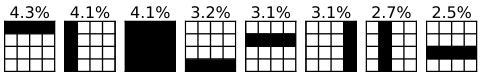
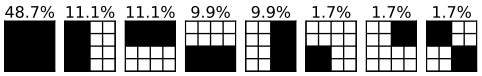
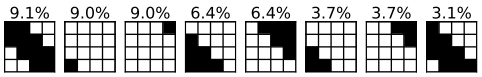
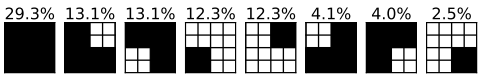
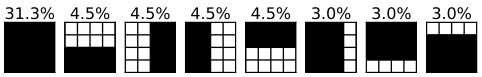
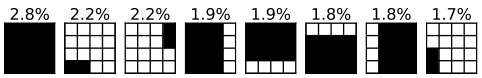
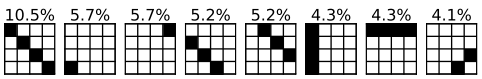
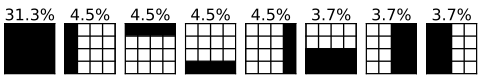
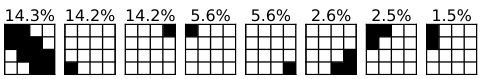
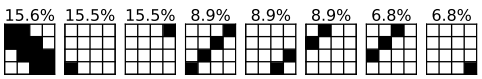
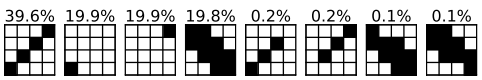
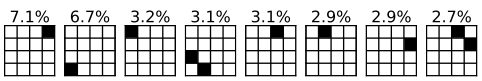
The result produced by the VALU is accumulated in the partial sum buffer at the location indexed by `r_idx`. This accumulation process allows for the proper storage of intermediate and final results. The control signals `CE` and `RE` are responsible for determining whether to switch to the next tile and directly control the input vector buffer and partial sum buffer.

3) Overall architecture: As the overall architecture shown in Figure 7, the SPASM accelerator is composed of multiple PE groups, each containing 16 PEs, allowing for a maximum of 64 parallelism. Within each PE group, every 4 PEs are assigned to a single High Bandwidth Memory (HBM) channel to fully utilize the available memory bandwidth. This allocation scheme ensures efficient access to the values of the sparse matrix $\mathbf{A}$.

Additionally, all 16 PEs within a PE group share a single HBM channel for loading the position encoding of matrix $\mathbf{A}$. The number of HBM channels allocated for loading the vector $\vec{x}$ and the number of PE groups can be parameterized, allowing for flexibility in configuring the SPASM accelerator based on specific requirements. This parameterization enables customization according to available memory bandwidth and desired computational capabilities.

One HBM channel is dedicated to loading and updating the vector $\vec{y}$. The input vector load unit and the partial sum merge unit are reconfigurable components of the SPASM accelerator. This reconfigurability means that different hardware versions can be synthesized to accommodate various global sparse patterns. By adapting these units, the accelerator can be tailored to specific problem characteristics, enhancing its overall efficiency and performance. The SPASM hardware accelerator will cost 1 + NUM_PE_GROUP $\times$ (NUM_XVEC_CH + 6) HBM channels.

TABLE II
SELECTED SPARSE MATRIX WORKLOADS SORTED BY DENSITY. **GC** STANDS FOR THE GLOBAL COMPOSITIONS.

Name	nnz	density	Application domain	Local patterns	GC
mycielskian14	3.70e+06	2.45e-02	Graph problem		
ex11	1.10e+06	3.97e-03	CFD		
raefsky3	1.49e+06	3.31e-03	CFD		
mip1	1.04e+07	2.35e-03	optimization problem		
rim	1.01e+06	1.99e-03	CFD		
3dtube	3.24e+06	1.58e-03	CFD		
bbmat	1.77e+06	1.18e-03	CFD		
Chebyshev4	5.38e+06	1.16e-03	structural problem		
Goodwin_054	1.03e+06	9.75e-04	CFD		
x104	1.02e+07	8.66e-04	structural problem		
cfd2	3.09e+06	2.03e-04	CFD		
ML_Laplace	2.77e+07	1.95e-04	structural problem		
af_0_k101	1.76e+07	6.92e-05	structural problem		
PFlow_742	3.71e+07	6.73e-05	2D/3D problem		
c-73	1.28e+06	4.46e-05	optimization problem		
af_shell10	5.27e+07	2.32e-05	structural problem		
tmt_sym	5.08e+06	9.62e-06	electromagnetics problem		
tmt_unsym	4.58e+06	5.44e-06	electromagnetics problem		
t2em	4.59e+06	5.40e-06	electromagnetics problem		
stormG2_1000	3.46e+06	4.76e-06	optimization problem		

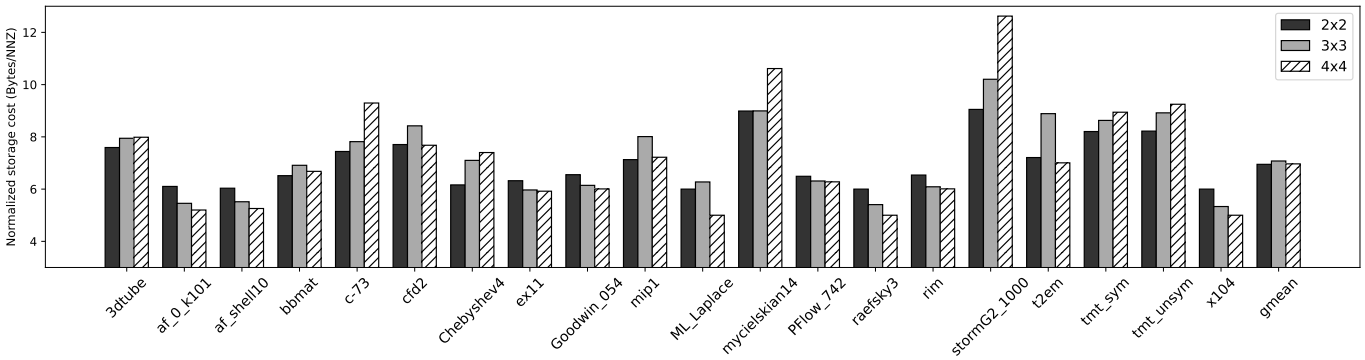


Fig. 9. Storage costs under different choices of tile sizes.

V. Experiment results & Evaluation

A. Evaluation setup

1) Workload: We use 20 sparse matrices for benchmarking purposes. For fair comparisons. The selection of sparse matrices from the SuiteSparse collection [6]. The workload matrices are presented in Table II. The chosen workloads exhibit a range of densities, from 4.76×10^{-6} to 2.45×10^{-2}, ensuring a comprehensive evaluation of the SPASM framework's performance across different data sparsity levels.

Furthermore, the selected matrices possess different local and global compositions to ensure fair benchmarking. This variation in structured patterns allows for a thorough assessment of the SPASM framework's effectiveness in handling different types of structured patterns commonly encountered in real-world applications.

2) Baseline: For the FPGA baselines, the HiSparse and Serpens accelerators, which are state-of-the-art FPGA accelerators for general Sparse Matrix-Vector (SpMV) kernels, are utilized. These accelerators are publicly accessible and well-regarded in the field. Specifically, Serpens has two versions: Serpens_a16 and Serpens_a24, which make use of 16 and 24 High Bandwidth Memory (HBM) channels for loading the sparse matrix respectively. HiSparse and Serpens serve as comparison points to evaluate the performance of the SPASM framework.

In addition to the FPGA baselines, the GPU baseline is measured using the cuSPARSE library [20] on the NVIDIA RTX 3090 graphics card. The SpMV throughput achieved on this GPU is used as a reference for comparison. To provide an overview of the different hardware platforms used for the baselines, Table III presents a comparison of their key characteristics.

TABLE III
The specification of different baseline hardware platforms.

	Frequency	Bandwidth	Peak perf.
HiSparse	237 MHz	273 GB/s	60.7 GFLOP/s
Serpens_a16	282 MHz	288 GB/s	72.2 GFLOP/s
Serpens_a24	276 MHz	403 GB/s	106 GFLOP/s
RTX 3090	1560 MHz	935.8 GB/s	35.58 TFLOP/s

TABLE IV
The detailed characteristics of SPASM hardware accelerators with different configurations.

	Frequency	Bandwidth	Peak perf.
SPASM_4_1	252 MHz	417 GB/s	129 GFLOP/s
SPASM_3_4	265 MHz	446 GB/s	102 GFLOP/s
SPASM_3_2	251 MHz	360 GB/s	96.4 GFLOP/s

3) Bitstream generation: In Section IV-D, it was mentioned that the SPASM hardware accelerators can be configured by adjusting the parameters NUM_XVEC_CH and NUM_PE_GROUP. By modifying these parameters, different hardware configurations can be generated. The SPASM hardware accelerators are implemented using Vitis HLS on the Xilinx Alveo U280 platform. The U280 features an 8 GB HBM capacity, delivering a total bandwidth of 460 GB/s. Additionally, it includes approximately 34 MB of on-chip RAM. To enhance the placement and routing of the hardware accelerators, TAPA [5] and Autobridge [13] are utilized. For evaluation purposes, three versions of the hardware accelerators are developed, each denoted as SPASM_{NUM_PE_GROUP}_{NUM_XVEC_CH}. These versions correspond to different combinations of NUM_PE_GROUP and NUM_XVEC_CH parameters and are customized for adapting different global compositions. The detailed characteristics of these versions are presented in Table IV.

B. Local pattern size selection

The choice of local pattern sizes can significantly impact both the number of zero paddings and the architectural design of VALU. A larger local pattern size enables multiple template pattern elements to share a single position encoding, thereby diminishing the position encoding overhead per element.

In our proposed data format, after zero-padding, $pattern\_size$ elements occupy a total of $(pattern\_size+1)\times 4$ bytes. Therefore, considering the padding rate, the average storage cost for a non-zero element can be calculated as $(pattern\_size+1)/(pattern\_size\times(1-padding\_rate))\times 4$

bytes. It is evident that a larger pattern size can maintain storage efficiency if the padding rate remains constant.

Furthermore, a larger pattern size facilitates the vectorization of on-chip buffer accesses. However, opting for a larger tile size introduces a wider range of patterns, which in turn increases the complexity of hardware design and pattern encoding. To address this tradeoff, we evaluated the storage cost under 2-by-2, 3-by-3, and 4-by-4 local pattern sizes, as depicted in Figure 9.

The results indicate that both the 2-by-2 and 4-by-4 choices are marginally more efficient than the 3-by-3 option. We have opted for the 4-by-4 size to maximize parallelism. Local pattern sizes exceeding 4-by-4 are not considered due to the proliferation of patterns and hardware complexity.

C. Template pattern selection

There are a total of 1820 possible template patterns of 4-by-4 submatrix. However, only 16 template patterns are chosen to be encoded with a `t_id` length of 4. The selection of these template patterns is crucial as it can significantly impact the padding rate when decomposing local patterns. Therefore, to quantify the impact of template pattern selection, we evaluate the storage cost of decomposed patterns.

Finding the optimal template patterns that minimize the padding rate for a specific sparse matrix is an NP-hard problem. However, certain frequently occurring structures, such as row vectors, column vectors, diagonal vectors, and blocks, can be observed intuitively in many matrices. Therefore, the chosen template patterns should be designed to exploit these types of structures effectively.

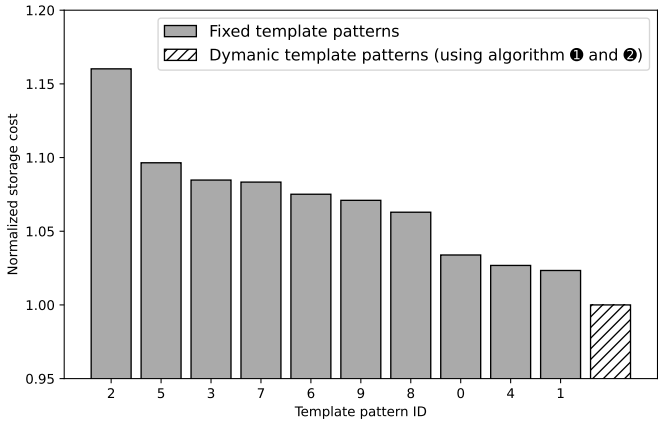
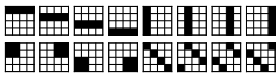
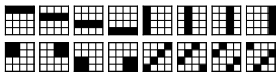
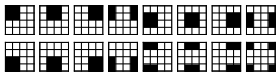
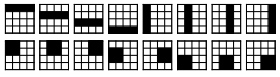
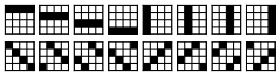
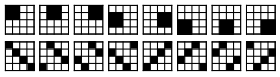
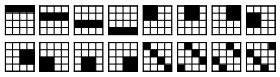
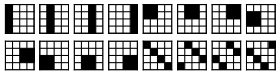
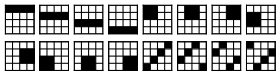
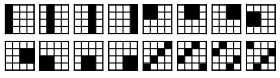


Fig. 10. Storage costs of different template pattern selection. Dynamic template patterns enable the selection of template patterns for specific workload matrices.

A combination of row vectors, column vectors, diagonal vectors, anti-diagonal vectors, and blocks is utilized to construct candidate template patterns that cover frequently occurring structures. These candidate template patterns are depicted in Table V. We perform ❶ local pattern analysis and ❷ template pattern selection on the workload matrices using algorithms in Algorithm 2 and 3.

TABLE V
TEMPLATE PATTERNS AND THEIR DESCRIPTION. **RW**: ROW-WISE, **CW**: COLUMN-WISE, **BW**: BLOCK-WISE.

ID	Template patterns	Description
0		4 **RW** patterns, 4 **CW** patterns, 4 **BW** patterns, 4 diagonal patterns
1		4 **RW** patterns, 4 **CW** patterns, 4 **BW** patterns, 4 anti-diagonal patterns
2		16 **BW** patterns with different sampling window placement
3		4 **RW** patterns, 4 **CW** patterns, 8 **BW** patterns
4		4 **RW** patterns, 4 **CW** patterns, 4 diagonal patterns, 4 anti-diagonal patterns
5		8 **BW** patterns, 4 diagonal patterns, 4 anti-diagonal patterns
6		4 **RW** patterns, 8 **BW** patterns, 4 diagonal patterns
7		4 **CW** patterns, 8 **BW** patterns, 4 diagonal patterns
8		4 **RW** patterns, 8 **BW** patterns, 4 anti-diagonal patterns
9		4 **CW** patterns, 8 **BW** patterns, 4 anti-diagonal patterns

We try to decompose all workload matrices on given fixed template patterns as well as dynamic template patterns. The results of storage cost are shown in Figure 10. As we can see, the final storage cost is related to the selection of template patterns. There are no one-fit-all template patterns and the best solution is to select the proper template patterns for each specific workload.

D. Storage cost comparison of different data formats

We compared the storage cost of our data format to several different kinds of other data formats: COO data format, CSR data format, BSR data format, and data formats used in Serpens [25] and HiSparse [7]. We assume the indices in COO, CSR, and BSR data format are 32-bit `int`, and the values are stored in the `float` type. Data formats adopted in HiSparse and Serpens used 2-level tiling schemes, which is similar to the SPASM data format. We ignore the cost of the first-level position encoding of tiles for convenience, which only occupies negligible storage space compared to the second-level nonzero position encoding of nonzeros. For the BSR data format, we set the block size to be 2-by-2.

Figure 11 provides a visual representation of the improvement achieved by different data formats across the workload

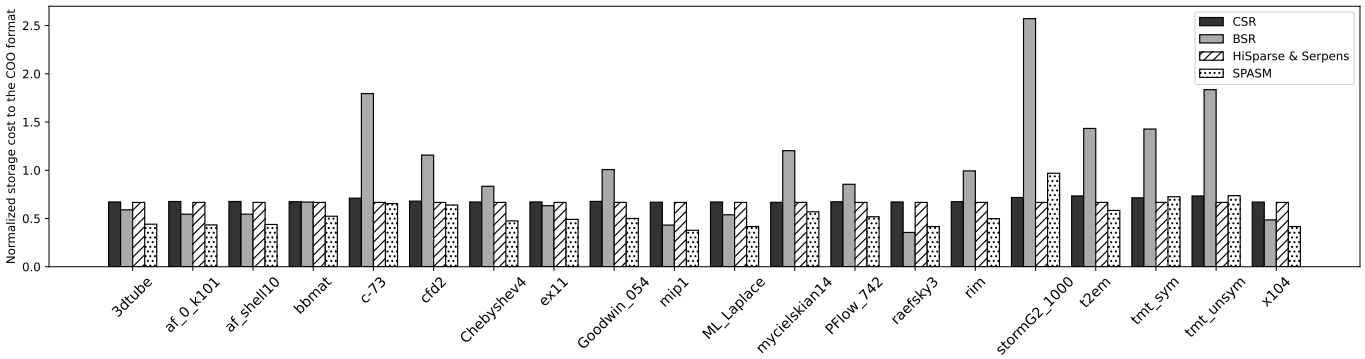


Fig. 11. Storage cost comparison between SPASM data format and other data formats. The results are normalized to the COO cost.

matrices. The minimum, maximum, and average (geometric mean) improvements are summarized in Table VI.

TABLE VI
OVERALL STORAGE SPACE IMPROVEMENT OF DIFFERENT DATA FORMATS. THE DATA IS NORMALIZED TO THE COO FORMAT.

Data format	Min.	Max.	Average
COO	1.00×	1.00×	1.00×
CSR	1.36×	1.49×	1.46×
BSR	0.39×	2.81×	1.16×
HiSparse & Serpens	1.50×	1.50×	1.50×
SPASM	0.98×	2.40×	1.79×

Based on the results, it is evident that the proposed SPASM data format outperforms other formats in terms of average storage improvement. The advantage of SPASM lies in its ability to handle various types of structured patterns efficiently. Unlike BSR, which primarily suits block patterns, SPASM is designed to accommodate a broader range of structured patterns. This flexibility allows SPASM to achieve better overall performance, particularly in terms of minimal and average storage improvements.

E. SpMV execution results

1) Throughput: We run the SpMV kernel on different baseline hardware and the throughput (GFLOP/s) is measured as $(2 \times nnz + num\_of\_rows)/exe\_time$. The comparison with previous FPGA accelerators is shown in Figure 12. SPASM achieves up to $14.40\times$, $23.27\times$, and $23.27\times$ speedup with respect to HiSparse, Serpens_a16 and Serpens_a24. The geometric mean of speedup is $6.74\times$, $3.21\times$ and $2.81\times$.

Compared with the GPU baseline, SPASM has a geometric mean of $0.75\times$ throughput with a maximum improvement of $2.51\times$.

SPASM exhibits high efficiency in hardware resource utilization. As shown in Figure 13, the percentage of peak computing and bandwidth utilized of SPASM is much higher compared to the other baselines, which benefit from both the architectural design and customized processing for specific workloads.

2) Bandwidth efficiency: Bandwidth efficiency is measured by throughput per unit bandwidth, in (GFLOP/s)/(GB/s). Since the SPASM will select different hardware versions for execution. Bandwidth efficiency is calculated respectively. Compared to FPGA baselines, SPASM archives up to $8.94\times$, $16.05\times$, and $22.47\times$ improvement. The geometric mean of bandwidth efficiency improvement over HiSparse, Serpens_a16, and Serpens_a24 are $4.18\times$, $2.21\times$, and $2.71\times$. For the GPU baseline, SPASM achieves up to $5.64\times$ of bandwidth efficiency, with a geometric mean of $1.68\times$.

TABLE VII
POWER CONSUMPTION AND ENERGY EFFICIENCY OF DIFFERENT HARDWARE PLATFORMS.

	Power	Energy efficiency
RTX 3090	333 W	0.23 (GFLOP/s)/W
HiSparse	45 W	0.37 (GFLOP/s)/W
Serpens	48 W	0.97 (GFLOP/s)/W
SPASM	58 W	1.24 (GFLOP/s)/W

3) Power & Energy efficiency: Energy efficiency is calculated as throughput per unit power, in (GFLOP/s)/W. The comparison of average power consumption and energy efficiency is shown in Table VII. The FPGA power is measured by Xilinx Board Utility `xbutil` and the GPU power is measured by NVIDIA System Management Interface `nvidia-smi`.Compared to the GPU baseline and HiSparse, SPASM achieves $5.39\times$ and $3.35\times$ energy efficiency improvement. Serpens_a16 and Serpens_a24 have close bandwidth efficiency, and SPASM achieves $1.28\times$ improvement compared to them.

4) Preprocessing cost: Table VIII lists the preprocessing time costs of several workload matrices. The preprocessing program is executed on Intel Xeon CPU E5-2650 with a single core. The customization time of the matrices is within several minutes, which can be further amortized if matrices are reused multiple times. As discussed in [26], this situation happens frequently in the scientific computing domain where the iterative SpMV computations are performed.

For example, `Chebyshev4` execution on SPASM is 6.29 ms faster than that on Serpens_a24 and it costs 1872 ms for

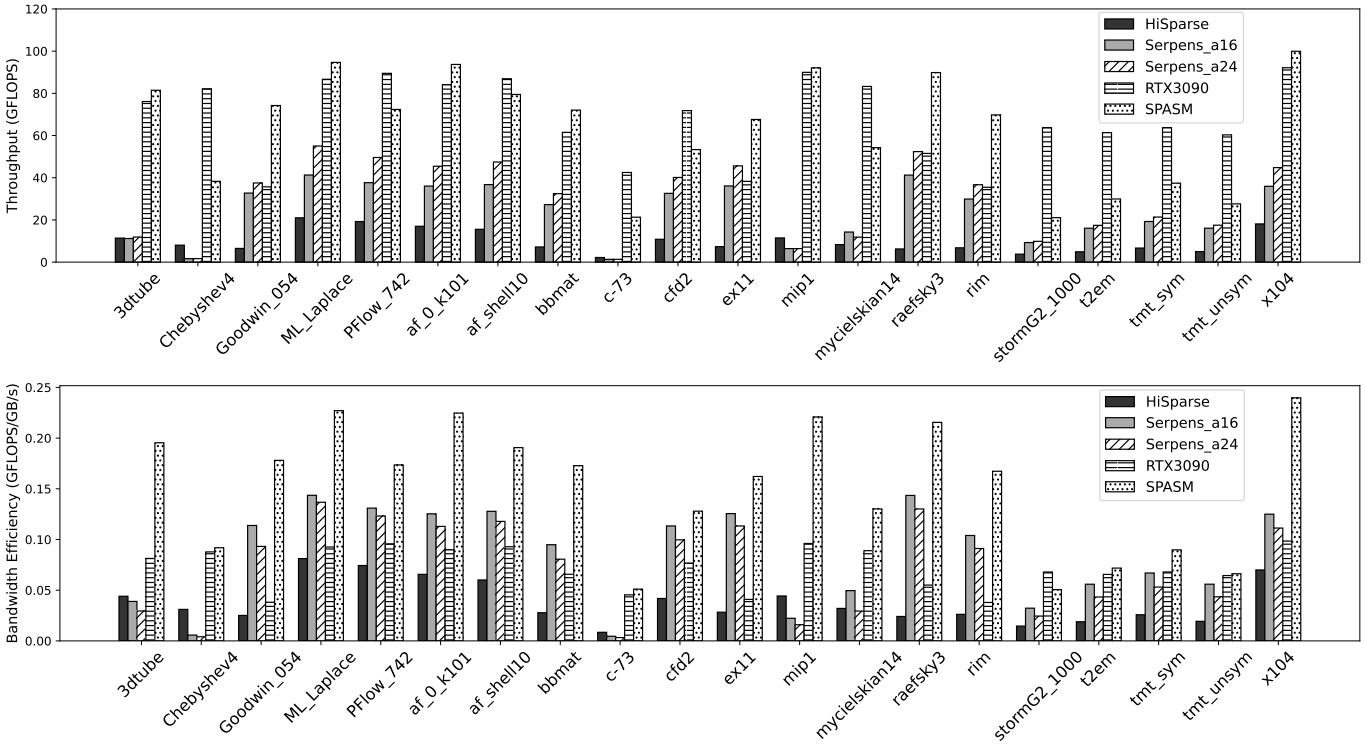


Fig. 12. Throughput and bandwidth efficiency compared to other platforms.

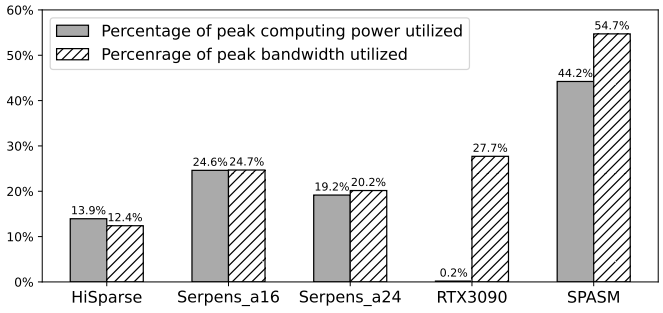


Fig. 13. Percentage of the bandwidth and computing power utilized on different platforms.

preprocessing. The preprocessing time of Serpens_a24 was not reported and we assume it doesn't take any time. Therefore, at least $(1872ms/6.29ms) \approx 298$ iterations are required to save the overall processing time. The preprocessing time could be long, but it can be amortized by thousands or millions of iterations in scientific computing situations where the same matrix is used. In the optimization domain, evaluating an optimal trading strategy in finance is another example where preprocessing costs are negligible [21]. This process involves solving tens of thousands of quadratic programming problems, all sharing the same sparsity pattern in their matrices. Each quadratic program demands at least thousands of SpMV iterations.

TABLE VIII
PREPROCESSING AND EXECUTION TIME OF SELECTED WORKLOADS.

Name	❶	❷	❸	❹❺	exe.
ML_Laplace	3258 ms	190 ms	1723 ms	2095 ms	0.59 ms
PFlow_742	4408 ms	5138 ms	2366 ms	3583 ms	1.05 ms
raefsky3	153 ms	24 ms	82 ms	99 ms	0.033 ms
Chebyshev4	732 ms	358 ms	361 ms	421 ms	0.33 ms

F. Ablation study

We will provide a comprehensive analysis of performance gain. We will evaluate the performance gained from:

- ❷ Template pattern selection.
- ❺ Workload schedule exploration.

The baseline configuration uses the SPASM_4_1 version with the fixed tile size of 1024 and fixed template pattern 0. The result of the ablation study is shown in Figure 14.

The ❺ workload schedule exploration, including the bitstream selection and tile size exploration, improves the overall performance by $1.13\times$ on average. The ❷ template pattern selection further enhances the performance by $1.04\times$. The improvement can vary significantly according to the different local and global compositions. Dynamic scheduling works well with the matrices with matrices that may have imbalanced workload distribution like `mip1`, achieving $1.82\times$ improvement, while dynamic pattern selection fits the matrices with different patterns from the fixed template patterns like `c-73`,

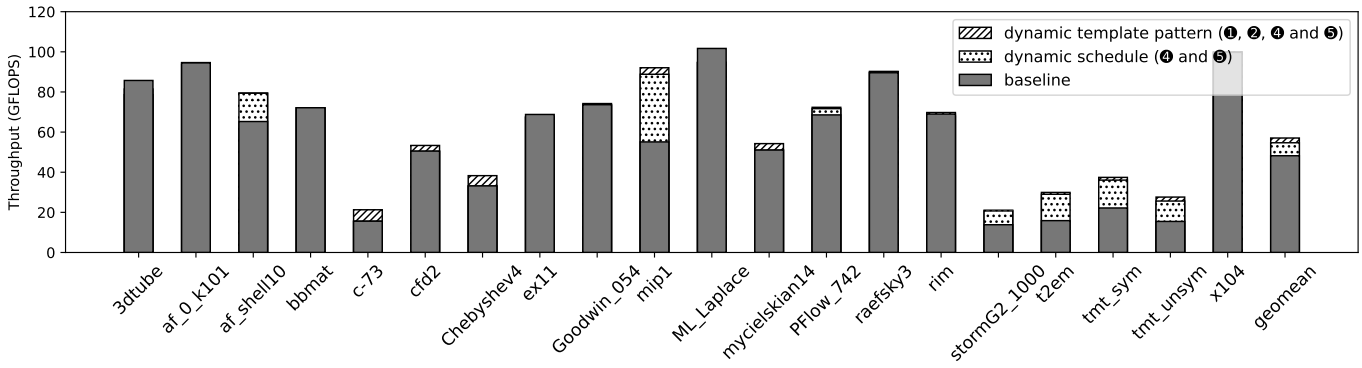


Fig. 14. Performance improvement due to different modules.

which dominates by the anti-diagonal patterns, achieving $1.36\times$ speedup.

VI. Related works

A. Structured pattern utilization

Another relevant work in the domain of pattern-based SpMV optimization targeting CPU is presented in [1]. It introduces PBR (Pattern-based Representation) to reduce index overhead without the need for padding. However, the patterns present in PBR have varying lengths, giving rise to new challenges in memory alignment and hardware efficiency. [8] is another CPU optimization work focusing on utilization of diagonal patterns. In the context of GPU optimization, [28] utilizes block-wise structured patterns for SpMV optimization. [27] adopts a hierarchical data structure to capture the local patterns. Additionally, TileSpMV [18] exploits various structured patterns within each tile by proposing a hybrid data format for GPU computing. The NVIDIA tensor core architecture enables efficient processing of 2:4 DBB patterns on GPUs [19]. In the machine learning domain, some research works focus on structural pruning [3], [4], [11], [16], [30], [32] where different variations of DBB patterns are produced during the training stage by imposing specific pruning constraints. However, these works are limited to the domain of machine learning and are unable to handle arbitrary structured patterns.

B. SpMV accelerator design

Moving on to SpMV accelerator designs, there are hardware accelerators designed for general SpMV acceleration. Examples include [22], Graphily [15], HiSparse [7] and Serpens [25], which are FPGA accelerators. HiSpMV [24] is another FPGA accelerator specifically focusing on imbalanced matrices. SpaceA [33] is an ASIC design that adopts a near-memory processing architecture to provide sufficient memory bandwidth for SpMV computation. However, these hardware accelerators are not optimized for structured patterns as they target general SpMV computation.

There are also accelerators that specifically exploit structured patterns. For instance, [10] is an FPGA accelerator designed and optimized for block-diagonal matrices typically encountered in finite element problems. Furthermore, RSQP [29] is an FPGA-based optimization solver that can adapt the hardware design based on the features of sparse matrices. Nevertheless, it should be noted that these accelerators have limited applicability, working only for a specific set of problems and are not generally applicable.

VII. Conclusion

We have developed SPASM, a hardware-software framework aimed at accelerating SpMV computations by harnessing the structured patterns found in sparse matrices. SPASM utilizes template patterns, enabling the representation of various types of local patterns. SPASM shows its ability to describe a broader range of structured patterns compared to other existing methods. Additionally, SPASM incorporates the extraction of global composition to enhance scheduling and load balancing. By combining both local and global sparse patterns, the SPASM framework can effectively select customized template patterns and hardware configurations for specific matrices. These features collectively contribute to SPASM's ability to outperform other existing solutions for accelerating SpMV computations.

References

[1] M. Belgin, G. Back, and C. J. Ribbens, "Pattern-based sparse matrix representation for memory-efficient smvm kernels," in *Proceedings of the 23rd International Conference on Supercomputing*, ser. ICS '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 100–109. [Online]. Available: https://doi.org/10.1145/1542275.1542294

[2] S. Cao, C. Zhang, Z. Yao, W. Xiao, L. Nie, D. Zhan, Y. Liu, M. Wu, and L. Zhang, "Efficient and effective sparse lstm on fpga with bank-balanced sparsity," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 63–72. [Online]. Available: https://doi.org/10.1145/3289602.3293898

[3] R. L. Castro, A. Ivanov, D. Andrade, T. Ben-Nun, B. B. Fraguela, and T. Hoefler, "Venom: A vectorized n:m format for unleashing the power of sparse tensor cores," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3581784.3607087

[4] Z. Chen, Z. Qu, L. Liu, Y. Ding, and Y. Xie, "Efficient tensor core-based gpu kernels for structured sparsity under reduced precision," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3458817.3476182

[5] Y. Chi, L. Guo, J. Lau, Y.-k. Choi, J. Wang, and J. Cong, "Extending high-level synthesis for task-parallel programs," in *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2021, pp. 204–213.

[6] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, dec 2011. [Online]. Available: https://doi.org/10.1145/2049662.2049663

[7] Y. Du, Y. Hu, Z. Zhou, and Z. Zhang, "High-performance sparse linear algebra on hbm-equipped fpgas using hls: A case study on spmv," in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 54–64. [Online]. Available: https://doi.org/10.1145/3490422.3502368

[8] T. Fukaya, K. Ishida, A. Miura, T. Iwashita, and H. Nakashima, "Accelerating the spmv kernel on standard cpus by exploiting the partially diagonal structures," 2021.

[9] J. Gao, W. Ji, Z. Tan, Y. Wang, and F. Shi, "Taichi: A hybrid compression format for binary sparse matrix-vector multiplication on gpu," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 12, pp. 3732–3745, 2022.

[10] P. Grigoraş, P. Burovskiy, W. Luk, and S. Sherwin, "Optimising sparse matrix vector multiplication for large scale fem problems on fpga," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, 2016, pp. 1–9.

[11] Y. Guan, C. Yu, Y. Zhou, J. Leng, C. Li, and M. Guo, "Fractal: Joint multi-level sparse pattern tuning of accuracy and performance for dnn pruning," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 416–430. [Online]. Available: https://doi.org/10.1145/3620666.3651351

[12] D. Guo, W. Gropp, and L. N. Olson, "A hybrid format for better performance of sparse matrix-vector multiplication on a gpu," *The International Journal of High Performance Computing Applications*, vol. 30, no. 1, pp. 103–120, 2016. [Online]. Available: https://doi.org/10.1177/1094342015593156

[13] L. Guo, Y. Chi, J. Wang, J. Lau, W. Qiao, E. Ustun, Z. Zhang, and J. Cong, "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*, ser. FPGA '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 81–92. [Online]. Available: https://doi.org/10.1145/3431920.3439289

[14] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'15. Cambridge, MA, USA: MIT Press, 2015, p. 1135–1143.

[15] Y. Hu, Y. Du, E. Ustun, and Z. Zhang, "Graphlily: Accelerating graph linear algebra on hbm-equipped fpgas," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2021, pp. 1–9.

[16] Z.-G. Liu, P. N. Whatmough, Y. Zhu, and M. Mattina, "S2ta: Exploiting structured sparsity for energy-efficient mobile cnn acceleration," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022, pp. 573–586.

[17] R. R. McCune, T. Weninger, and G. Madey, "Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing," *ACM Comput. Surv.*, vol. 48, no. 2, oct 2015. [Online]. Available: https://doi.org/10.1145/2818185

[18] Y. Niu, Z. Lu, M. Dong, Z. Jin, W. Liu, and G. Tan, "Tilespmv: A tiled algorithm for sparse matrix-vector multiplication on gpus," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021, pp. 68–78.

[19] NVIDIA, 2020. [Online]. Available: https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

[20] NVIDIA, "cuSPARSE," https://docs.nvidia.com/cuda/cusparse/index.html, 2024.

[21] P. Nystrup, S. Boyd, E. Lindström, and H. Madsen, "Multi-period portfolio selection with drawdown control," *Annals of Operations Research*, vol. 282, no. 1-2, pp. 245–271, 2019.

[22] J. Oliver, C. Álvarez, T. Cervero, X. Martorell, J. D. Davis, and E. Ayguadé, "Accelerating spmv on fpgas through block-row compress: A task-based approach," in *2023 33rd International Conference on Field-Programmable Logic and Applications (FPL)*, 2023, pp. 151–158.

[23] S. L. Olivier, N. D. Ellingwood, J. W. Berry, and D. M. Dunlavy, "Performance portability of an spmv kernel across scientific computing and data science applications," in *2021 IEEE High Performance Extreme Computing Conference, HPEC 2021, Waltham, MA, USA, September 20-24, 2021*. IEEE, 2021, pp. 1–8. [Online]. Available: https://doi.org/10.1109/HPEC49654.2021.9622869

[24] M. B. Rajashekar, X. Tian, and Z. Fang, "Hispmv: Hybrid row distribution and vector buffering for imbalanced spmv acceleration on fpgas," in *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 154–164. [Online]. Available: https://doi.org/10.1145/3626202.3637557

[25] L. Song, Y. Chi, L. Guo, and J. Cong, "Serpens: a high bandwidth memory based accelerator for general-purpose sparse matrix-vector multiplication," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, ser. DAC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 211–216. [Online]. Available: https://doi.org/10.1145/3489517.3530420

[26] J. D. Trotter, S. Ekmekçibaşı, J. Langguth, T. Torun, E. Düzakın, A. Ilic, and D. Unat, "Bringing order to sparsity: A sparse matrix reordering study on multicore cpus," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3581784.3607046

[27] R. M. Veras and F. Franchetti, "A scale-free structure for real world networks," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, 2017, pp. 1–7.

[28] R. W. Vuduc and H.-J. Moon, "Fast sparse matrix-vector multiplication by exploiting variable block structure," in *High Performance Computing and Communications*, L. T. Yang, O. F. Rana, B. Di Martino, and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 807–816.

[29] M. Wang, I. McInerney, B. Stellato, S. Boyd, and H. K.-H. So, "Rsqp: Problem-specific architectural customization for accelerated convex quadratic optimization," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3579371.3589108

[30] Y. Wang, C. Zhang, Z. Xie, C. Guo, Y. Liu, and J. Leng, "Dual-side sparse tensor core," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 1083–1095.

[31] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," in *Proceedings of the 30th International Conference on Neural Information Processing Systems*, ser. NIPS'16. Red Hook, NY, USA: Curran Associates Inc., 2016, p. 2082–2090.

[32] Y. N. Wu, P.-A. Tsai, S. Muralidharan, A. Parashar, V. Sze, and J. Emer, "Highlight: Efficient and flexible dnn acceleration with hierarchical structured sparsity," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 1106–1120. [Online]. Available: https://doi.org/10.1145/3613424.3623786

[33] X. Xie, Z. Liang, P. Gu, A. Basak, L. Deng, L. Liang, X. Hu, and Y. Xie, "Spacea: Sparse matrix vector multiplication on processing-in-memory accelerator," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 570–583.