

SCALER: A Stream-Aware Accelerator with Hierarchical Memory for Sparse LU Factorization on HBM FPGAs

Xin Xu¹, Zhiying Zhu¹, Zishu Li², Dan Niu³, Cheng Zhuo¹, Zhou Jin^{1*}

¹College of Integrated Circuits, Zhejiang University, China

²College of Electronic Information and Optical Engineering, Nankai University, China

³School of Automation, Southeast University, China

*Corresponding author: z.jin@zju.edu.cn

Abstract—Sparse LU factorization plays a pivotal role in many scientific and engineering applications. However, its inherent high sparsity and random non-zero distribution lead to irregular data dependencies and memory access patterns, leaving efficient acceleration on FPGAs largely unexplored. Recently, high concurrency of High Bandwidth Memory (HBM) has provided new opportunities for accelerating sparse LU factorization. Nonetheless, achieving high bandwidth utilization remains challenging given random dependencies and complex computation patterns.

In this paper, we present SCALER, a high-performance sparse LU factorization accelerator on HBM FPGAs. SCALER employs a sparse storage format with vectorized packing for data coalescing, customizing HBM-compatible data streams to boost bandwidth utilization. A two-tier hierarchical memory module enhances access efficiency and data reuse by optimizing memory management and reducing redundant transfers. Furthermore, a multi-stage pipelined data prefetching mechanism hides latency, leveraging the overlap of HBM access stages to improve off-chip memory communication efficiency. Finally, a stream-aware synchronization strategy transforms irregular dependencies into hierarchical streaming access, efficiently maximizing parallelism. Evaluation on 11 matrices demonstrates SCALER’s geometric mean (geomean) throughput, energy efficiency and bandwidth efficiency surpass cuDSS solver on NVIDIA Tesla V100 GPU by 1.79×, 4.20× and 5.12×, respectively. It also outperforms the cuDSS solver on NVIDIA RTX 4090 GPU by 1.44×, 3.05× and 4.12× for the same metrics.

Index Terms—Sparse LU factorization, FPGAs, HBM, Accelerator, Hierarchical memory, Stream-aware synchronization

I. INTRODUCTION

Solving the sparse linear systems $Ax = b$ is a fundamental computational task in scientific and engineering applications [1], [2], such as circuit simulation [3]–[7], finite element analysis [8], [9] and power system modeling [10], [11], etc. In practice, direct solvers based on sparse LU factorization are preferred over iterative solvers [12]–[16], as they provide reliable accuracy and avoid the challenges of selecting and tuning preconditioners [17]. Since sparse linear systems typically require repeated solutions, the sparse LU factorization stage has emerged as a bottleneck in computational time [7], making its acceleration a critical topic in high-performance computing [18]. Therefore, developing dedicated accelerators for sparse LU factorization is highly significant.

FPGAs provide an attractive platform for hardware accel-

eration through customized parallelism and dataflow designs [19], [20]. Compared to CPUs and GPUs, FPGAs not only offer lower latency and power consumption, but also could provide higher resource utilization for sparse matrix workloads [21]. Furthermore, sparse matrix computations are inherently memory-intensive. HBM [22] provides higher bandwidth and more independent memory channels than traditional DDR memory, making HBM FPGAs effective for sparse LU factorization’s memory access challenges.

Significant progress has been made in accelerating sparse kernels on HBM FPGAs such as sparse matrix-vector multiplication (SpMV) [23]–[25], sparse matrix-matrix multiplication (SpMM) [26]–[28] and sparse triangular solve (SpTRSV) [29]. However, efforts to accelerate sparse LU factorization remain limited, primarily due to its higher hardware implementation complexity. Specifically, accelerating sparse LU factorization on HBM FPGAs faces these challenges: (1) Existing sparse storage formats hinder efficient vectorized streaming HBM access, causing low bandwidth utilization. (2) Limited on-chip memory and frequent L/U factor accesses lead to poor data reuse and high latency. (3) Inefficient data prefetching often leads to unnecessary high-latency HBM accesses, degrading off-chip communication efficiency. (4) Complex and irregular data dependencies hinder parallel scheduling.

In light of these challenges, we present SCALER, a high-performance sparse LU factorization accelerator on HBM FPGAs. Firstly, we design an HBM-compatible sparse storage format that encodes matrix and dependency information into flattened arrays, enabling vectorized packing and improving bandwidth utilization. Secondly, we construct a two-tier hierarchical memory module to optimize sparse L/U factors access and reduce redundant off-chip data transfers, promoting on-chip data reuse and memory access efficiency. Additionally, we design a multi-stage pipelined prefetching strategy that selectively fetches essential matrix data and overlaps HBM access stages, hiding latency and enhancing off-chip communication efficiency. Finally, we propose a stream-aware synchronization scheduling strategy, transforming irregular dependencies into hierarchical streaming accesses, enabling efficient dependency management and maximizing parallelism.

Our main contributions are summarized as follows:

- To the best of our knowledge, this is the first sparse LU factorization accelerator on HBM FPGAs.
- We customize an HBM-compatible dataflow with the proposed sparse storage format and vectorized packing scheme to improve bandwidth utilization.
- We construct a two-tier hierarchical memory module to enhance on-chip data reuse and memory access efficiency.
- We explore a multi-stage pipelined prefetcher with latency hiding to boost off-chip communication efficiency.
- We introduce a stream-aware synchronization scheduling strategy to manage dependencies and maximize parallelism.

Evaluation shows that SCALER delivers geomean gains of $1.79\times$, $4.20\times$ and $5.12\times$ over cuDSS solver on NVIDIA V100 GPU in terms of throughput, energy and bandwidth efficiency, respectively. Compared with cuDSS solver on NVIDIA RTX 4090 GPU, SCALER maintains superior performance with respective geomean improvements of $1.44\times$, $3.05\times$ and $4.12\times$ across the same three metrics.

II. BACKGROUND

A. Sparse LU Factorization

LU factorization decomposes the matrix A of a system of linear equations $Ax = b$ into the product of a lower triangular matrix L and an upper triangular matrix U , transforming the original problem $Ax = b$ into two efficient triangular solves, $Ly = b$ and $Ux = y$. The factors L and U are computed iteratively by the following formulation:

$$U_{ij} = A_{ij} - \sum_{k=0}^{i-1} L_{ik}U_{kj}, \quad 0 \leq i \leq j < N \quad (1)$$

$$L_{ij} = \frac{1}{U_{jj}} \left(A_{ij} - \sum_{k=0}^{j-1} L_{ik}U_{kj} \right), \quad 0 \leq j < i < N \quad (2)$$

For sparse matrices, LU factorization generally proceeds in three stages: preprocessing, symbolic factorization and numerical factorization. Preprocessing reorders the matrix to reduce fill-ins, while symbolic factorization determines the non-zero pattern of L and U , enabling efficient memory allocation [30]. Numerical factorization computes the numerical values of the factors. In many scenarios, such as circuit simulation, where the non-zero pattern of matrix A is invariant across Newton-Raphson [31] and transient iterations despite changing element values. Therefore, preprocessing and symbolic factorization can be performed once, while numerical factorization is performed many times, making it the most critical and time-consuming stage in sparse LU factorization [32].

A widely used approach for the numerical factorization stage is the GP left-looking algorithm [33]. In this algorithm, the current column of L and U factors is obtained from the contributions of all previously factorized columns on the left, through a series of multiply-accumulate (MAC) and division (DIV) operations. Moreover, it allows independent columns to be processed concurrently, enabling column-level parallelism and further improving performance.

B. Sparse Matrix Accelerators on HBM FPGAs

HBM is an advanced memory technology optimized for parallel data transfers, providing much higher bandwidth than DDR memory through vertically stacked DRAM structures [23]. HBM-based FPGAs integrate the high-bandwidth transfer capability of HBM with the programmability of FPGAs, enabling flexible dataflow scheduling and efficient channel utilization to accelerate sparse matrix computations [34].

Numerous efforts have explored accelerating sparse matrix computations on HBM FPGAs. For SpMV acceleration, HiSparse [23] designs a sparse matrix format and on-chip buffers, with an HLS-based dynamic conflict resolution to maximize HBM bandwidth utilization. Serpens [24] presents a general SpMV accelerator leveraging memory-centric processing engines and index coalescing to address irregular memory access patterns. For SpMM, Sextans [26] achieves high-performance through Processing Element (PE) aware nonzero scheduling and multi-level memory optimization on HBM. SDMA [27] proposes a SpMM accelerator for GNNs using equal-value partitioning, vertex clustering and adaptive dataflow. SpTRSV is more challenging due to data dependencies. LevelST [29] tackles this by linearizing dependency graphs, partitioning the matrix into diagonal and off-diagonal blocks and leveraging ring structure for resource sharing.

Compared to the above-mentioned workloads, sparse LU factorization involves complex computation patterns and irregular data dependencies, which significantly complicates memory access and parallelization strategies, leaving research on sparse LU acceleration on HBM FPGAs largely unexplored.

III. MOTIVATION

Designing a high-performance sparse LU factorization accelerator that fully exploits the high-bandwidth and parallel capability of HBM FPGAs faces several challenges:

- **Existing sparse storage formats misalign with HBM access patterns.** In CSR/CSC, pointer arrays restrict streaming access to nonzeros. In addition, the irregular distribution of nonzeros complicates data merging and further reduces HBM bandwidth utilization.
- **Limited on-chip memory resources and insufficient data reuse.** On-chip memory resources of FPGAs are scarce and improper allocation strategies may evict reusable data prematurely. Such inefficiency lowers memory utilization and induces redundant HBM accesses.
- **High HBM access latency and off-chip communication efficiency.** Despite its massive bandwidth, HBM suffers from high access latency which limits performance. Prefetching is largely ineffective under irregular accesses, causing unnecessary off-chip memory transactions.
- **Complex data dependencies cause difficulties in parallel scheduling.** In sparse LU factorization, the dependencies between columns are complex and irregular, making parallel scheduling extremely difficult on FPGAs.

IV. DATAFLOW PREPARATION

Sparse LU factorization features irregular nonzero distributions and inter-column dependencies. Without dedicated stor-

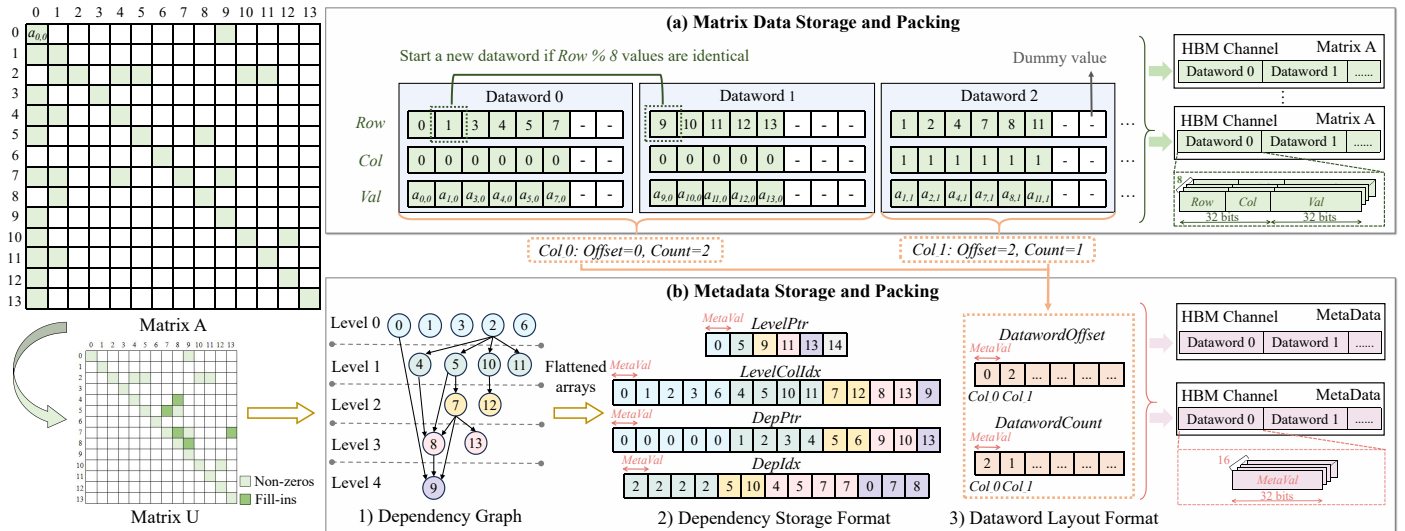


Fig. 1: Overall flow of the dataflow preparation framework. Fig. 1(a) details matrix data storage and packing process: Matrix A is stored in COO format (Row, Col, FloatVal), then packed into 512-bit datawords. A row-index modulo-based reordering mitigates on-chip access conflicts. Fig. 1(b) explains metadata storage and packing process: Matrix U (from symbolic factorization of Matrix A) is converted into a Dependency Graph. Dependency metadata (LevelPtr, LevelColIdx, DepPtr, DepIdx) and Matrix A layout metadata (DatawordOffset, DatawordCount) are stored. These arrays, consisting of 32-bit MetaVal entries, are then grouped into 512-bit datawords.

age and packing designs, this irregularity wastes bandwidth, hindering effective parallelism. To address this, we propose an HBM-compatible sparse data format and vectorized packing scheme, with customized strategies for matrix and metadata.

A. Matrix Data Storage and Packing

Coordinate (COO) format, explicitly storing non-zero elements with their row and column indices, efficiently retrieves scattered entries and suits sparse matrices' irregular access patterns. We encode each non-zero element in 64 bits: a 16-bit column index, a 16-bit row index and a 32-bit single-precision floating-point value. Through vectorized packing, we merge multiple 64-bit elements into 512-bit HBM datawords (Fig. 1(a)). To prevent on-chip BRAM conflicts, inspired by LevelST [29], we reorder input data via row-index modulo, assigning each element to a target slot. If occupied, the word is padded with dummies, initiating a new packing cycle.

B. Metadata Storage and Packing

Metadata comprises two parts: dependency metadata (level information and fine-grained dependency list) and matrix metadata (dataword offsets and counts for Matrix A and L/U data).

Dependency metadata is critical for guiding computation order and preserving pipeline efficiency. Our proposed storage format utilizes the Dependency Graph (Fig. 1(b-1)), constructed from non-zero structure of matrix U after static symbolic factorization. It is represented as a directed acyclic graph (DAG) $GE(V, E)$, where the node set $V = \{1, 2, \dots, n\}$ corresponds to all columns and the edge set $E = \{(j, k) | U(j, k) \neq 0, j < k\}$ denotes the dependency from column j to column k . Nodes without conflicts are grouped into the same level based on their dependency depth, where the level of node k is computed as:

$$\text{level}(k) = \max(-1, \text{level}(j_1), \text{level}(j_2), \dots) + 1 \quad (3)$$

where j_1, j_2, \dots denote row indices of all off-diagonal non-zero elements in column k of U (i.e., $U(j_i, k) \neq 0$). Given the Dependency Graph's irregular storage and non-consecutive access during level traversal in hardware, we encode these dependencies into four linearized arrays (Fig. 1(b-2)): LevelPtr: stores the starting offset of each level; LevelColIdx: stores the column indices per level; DepPtr: stores the offset of each column's dependency list, with the difference between two entries giving the number of dependencies; DepIdx: stores the column indices on which a given column depends.

To transform the logical structure of the sparse matrix into accessible data blocks in HBM, we introduce matrix metadata. We employ a dataword layout format with two arrays, DatawordOffset and DatawordCount (Fig. 1(b-3)), storing each column's starting offset and the occupied datawords, respectively. Following this, all metadata entries are grouped into 512-bit HBM datawords.

V. OVERALL ARCHITECTURE

The overall architecture of SCALER is illustrated in Fig. 2. Customized HBM-compatible data streams, preprocessed from Matrix A and metadata, enter the FPGA via HBM. The Dispatcher aggregates metadata and column data, streaming them to Dedicated Parallel Processing Engines for MAC/DIV computation. Processing Element Groups (PEGs) first check dependent L/U factors in Local Memory (LM). An LM miss routes requests to Shared Memory (SM). SM processes these, gathers PEG-computed L/U factors and streams results to the host via the L/U Writer, completing sparse LU factorization. Further details on each component are provided in this chapter.

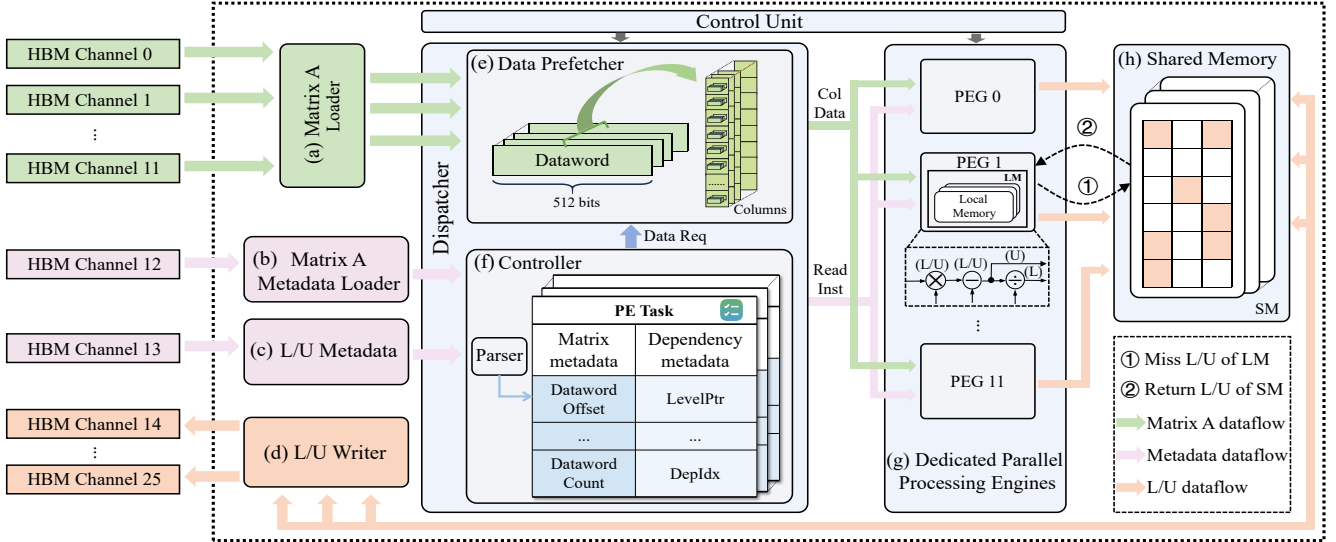


Fig. 2: SCALER's overall hardware architecture.

A. Channel Allocation

In terms of managing access patterns for different data types (Matrix A, L/U factors and metadata), SCALER allocates 26 HBM channels: Non-zeros of sparse Matrix A are mapped to 12 channels, while L/U data are stored in another 12. Given metadata size is less than core computation data, SCALER allocates 2 HBM channels: one for dependency metadata and Matrix A's layout and the other for L/U data's layout. Loaders and Writers perform streaming access, facilitating highly concurrent dataflow across dedicated HBM channels.

B. Control Unit

High HBM access latency and complex data dependencies are impediments in sparse LU factorization. To address these, SCALER designs a Dispatcher, composed of a Data Prefetcher (Fig. 2(e)) and a Controller (Fig. 2(f)), coordinating the accelerator's data and task flow. The Data Prefetcher efficiently prefetches requested raw data via Matrix A Loader (Fig. 2(a)) to avoid unnecessary HBM access. Simultaneously, the Controller (Fig. 2(f)) organizes metadata from the Metadata Loader (Fig. 2(b),(c)) into parallel PE Task packets. Finally, the Dispatcher distributes these tasks and corresponding column data to the 12 PEGs within the Dedicated Parallel Processing Engines (Fig. 2(g)) for numerical column computations.

C. Hierarchical Memory Module

L/U factors have random and frequent access requirements. If only LM is set up, its limited capacity and coverage struggle to capture dependent L/U factors, leading to frequent misses. More critically, if a required L/U factor resides in another PEG's LM, direct cross-PEG communication faces challenges because an all-to-all crossbar communication structure between the PEGs is impractical due to quadratic circuit complexity and frequency degradation. Direct access to HBM incurs high latency. Therefore, we propose a two-tier hierarchical memory module to enhance on-chip data reuse: the extremely fast local access (LM) and the large-capacity sharing (SM) (Fig. 2(h)).

LM: LM stores L/U factors in sparse column memory entry arrays built from ultra RAM (URAM). Column index mapping tables (record column ID corresponding to each memory slot) and validity flag arrays (indicate whether the data in a slot is valid) are stored in corresponding arrays using Block RAM (BRAM). Its fully partitioned design maps each memory slot to different physical storage units (independent BRAM/URAM banks). LM holds recently computed L/U factors or fetches from SM/HBM. Leveraging dual-port BRAM, LM supports parallel PE access, effectively preventing access conflicts.

SM: To avoid capturing distant cross-PEG dependencies and improve overall memory hit rates while reducing HBM access, SM serves as the large-capacity memory shared by all PEGs. SM uses URAM resources to store L/U factors and BRAM resources for the global column index mapping table and validity flag array. Its internal storage units also feature a fully partitioned design, supporting high concurrent access and maximizing both temporal and spatial locality. Upon computation and storage in LM, L/U factors are simultaneously copied to SM and asynchronously written back to HBM via L/U Writer (Fig. 2(d)). Overlapping with the PEG's computation pipeline, the HBM write-back operation hides latency.

VI. SCHEDULING STRATEGY

To further accelerate sparse LU factorization, we propose a multi-stage pipelined prefetching with a stream-aware synchronization scheduling strategy, transforming irregular dependencies into hierarchical streaming tasks, facilitating parallel MAC/DIV computations by PEs.

A. Multi-stage Pipelined Prefetching

Traditional prefetching, when confronted with irregular off-chip sparse memory access patterns, may fetch unneeded data. To mitigate this, SCALER proposes a multi-stage pipelined prefetching mechanism, efficiently enhancing off-chip communication. Equipped with an asynchronous interface, Data Prefetcher accesses all 12 HBM channels for original Matrix A in parallel. Guided by the Controller's metadata, it strictly

fetches data for columns within the current processing level, thereby preventing unnecessary HBM accesses. Its multi-stage pipeline concurrently manages multiple HBM read requests, meticulously partitioning access into stages (e.g., address sending, internal processing and data return) to facilitate temporal overlap. Furthermore, the Prefetcher precisely tracks HBM requests and responses, ensuring data integrity. This approach ensures multiple parallel-processed columns within the same level can promptly access required data without blocking.

B. Stream-Aware Synchronization Scheduling

Parallel scheduling is profoundly intricate due to random inter-column dependencies. To maximize concurrent execution, our stream-aware synchronization scheduling strategy establishes a predefined order. Using a column index modulo strategy, the Dispatcher streams independent level 0 column tasks and their data to different PEGs for parallel processing. Within each PEG, PEs collaboratively streamline dependency parsing and perform MAC/DIV operations for the current column. This stream-aware synchronization scheduling strictly adheres to hierarchical order: only after column tasks within the current level are processed, the Dispatcher streams tasks for the next level. This ensures all required dependencies from preceding levels are completed and available via the two-tier memory module when a PEG processes its current column.

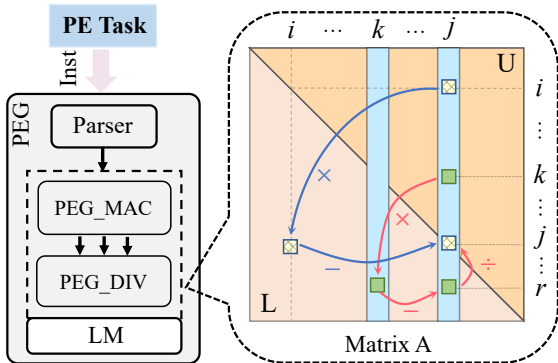


Fig. 3: Parallel LU factorization process in PEG.

MAC operation: Each PE receives MAC input packets and employs vectorized processing, leveraging 8 floating-point values and their packed elements to execute parallel MAC operations. DSP resources are used for floating-point multiplication and addition, with a deep pipeline design to optimize their utilization. Its internal loop unrolling achieves an Initiation Interval (II) = 1. As Fig. 3 illustrates, the MAC unit computes products like $L_{ji} \times U_{ij}$ and $L_{rk} \times U_{kj}$ in parallel, subtracting them from the corresponding elements A_{jj} and A_{rj} of the current column j to update its dense representation.

DIV operation: Following the MAC stage, PEs collaboratively process and normalize the current column’s diagonal element. Streaming pipelined floating-point dividers (typically 1-2) compensate for long delays. The process also addresses near-zero diagonal elements, ensuring numerical stability. After MAC operations are complete, the diagonal element A_{jj} of the current column j becomes U_{jj} . PEs in DIVIDER unit then divide elements A_{rj} below the diagonal in column j (where

$r > j$) by U_{jj} in parallel to yield the final L factor L_{rj} .

VII. EVALUATION

A. Experiment Setup

We evaluate performance of SCALER on a Xilinx Alveo U280 FPGA board. SCALER is implemented using Xilinx HLS C++ and prototyped with Vitis 2022.2 toolbox. The HLS code is built on top of the stream-based framework TAPA [35].

1) GPU Baseline: We run sparse LU factorization on NVIDIA RTX 4090 and NVIDIA Tesla V100 GPUs for comparison. We use the NVIDIA cuDSS [36] Level 2 API `CUDSS_PHASE_FACTORIZATION` with CUDA 12.4. cuDSS, an official CUDA high-performance library for sparse LU factorization, is considered the state-of-the-art (SOTA) GPU solution. Also, we determine the time for LU factorization phase using `cudaEventElapsedTime` and power consumption is measured with `nvidia-smi`.

SCALER and compared accelerator specifications are detailed in Table I. Memory bandwidth for U280 is specific to SCALER, utilizing 26 HBM channels for most off-chip communication. For GPUs, only maximum bandwidth is shown.

TABLE I: The specifications of FPGA accelerator and GPUs.

	SCALER (ours)	RTX 4090	Tesla V100
Frequency	300 MHz	2520 MHz	1530 MHz
Bandwidth	338 GB/s	1008 GB/s	900 GB/s
Power	60 W	123 W	134 W
Process Node	TSMC 16nm	TSMC 4 nm	TSMC 12nm

TABLE II: The information of evaluated matrices.

ID	Name	Size	non-zeros	Density	Field
M1	add20	2,395	13,151	2.30E-3	Circuit Simulation
M2	ACTIVSg2000	4,000	28,505	1.78E-3	Power Network
M3	sherman3	5,005	20,033	8.00E-4	CFD
M4	pd	8,081	13,036	2.00E-4	Counter Example
M5	flowmeter5	9,669	67,391	7.21E-4	Model Reduction
M6	Pesa	11,738	79,566	5.77E-4	Directed Weighted Graph
M7	tuma2	12,992	49,365	2.92E-4	2D/3D
M8	mult_dcop_03	25,187	193,216	3.25E-4	Circuit Simulation
M9	poli4	33,833	73,249	6.40E-5	Economic
M10	onetone2	36,057	222,596	1.72E-4	Circuit Simulation
M11	rajat26	51,032	247,528	9.50E-4	Circuit Simulation

2) Datasets: To compare the performance of SCALER with SOTA GPUs implementations, we select 11 sparse matrices from the SuiteSparse [37] collection for evaluation. These matrices span diverse domains including circuit simulation, power networks, computational fluid dynamics (CFD), graph theory and 2D/3D problems, etc. Table II reveals detailed information about the evaluation matrices, including their size, number of non-zero elements and density, etc.

3) Metrics: (1) Throughput: Measured in Giga Floating-point Operations Per Second (GFLOP/s). (2) Energy Efficiency: Measured in throughput per Watt (MFLOP/s/W). (3) Bandwidth Efficiency: Measured in throughput per unit of memory bandwidth (MFLOP/s/(GB/s)).

TABLE III: Performance comparison between SCALER and cuDSS solver on NVIDIA V100 and RTX 4090 GPUs.

		M1	M2	M3	M4	M5	M6	M7	M8	M9	M10	M11	Geomean
Throughput (GFLOP/s)	SCALER (ours)	0.09	1.54	1.18	0.01	2.32	11.00	0.08	3.41	0.12	22.20	1.34	0.75
	Tesla V100	0.06	0.90	0.64	0.01	1.69	5.80	0.04	1.37	0.06	9.35	0.76	0.42
	RTX 4090	0.07	1.10	0.77	0.01	1.66	7.10	0.06	2.34	0.06	12.71	0.98	0.52
	Improvement	1.29×	1.40×	1.53×	1.43×	1.37×	1.55×	1.33×	1.46×	2.00×	1.75×	1.37×	1.49×
Energy efficiency (MFLOP/s/W)	SCALER (ours)	1.51	25.85	19.81	0.17	38.95	184.66	1.34	57.24	2.01	372.67	22.49	12.55
	Tesla V100	0.45	6.72	4.78	0.04	12.61	43.28	0.30	10.22	0.45	69.78	5.67	2.99
	RTX 4090	0.57	8.94	6.26	0.06	13.50	57.72	0.49	19.02	0.49	103.33	7.97	4.11
	Improvement	2.65×	2.89×	3.16×	2.95×	2.89×	3.20×	2.75×	3.01×	4.13×	3.61×	2.82×	3.07×
Bandwidth efficiency (MFLOP/s/(GB/s))	SCALER (ours)	0.24	4.56	3.49	0.03	6.86	32.54	0.24	9.11	0.36	59.36	3.58	2.14
	Tesla V100	0.07	1.00	0.71	0.01	1.88	6.44	0.04	1.52	0.07	10.39	0.84	0.47
	RTX 4090	0.07	1.09	0.77	0.01	1.65	7.04	0.06	2.32	0.06	12.61	0.97	0.52
	Improvement	3.43×	4.18×	4.53×	3.00×	3.65×	4.62×	4.00×	3.93×	5.14×	4.71×	3.69×	4.04×

B. Results

We use Xilinx Runtime (XRT) to measure kernel execution time and xbutil to measure power consumption. Table III presents a comparison of throughput, energy efficiency and bandwidth efficiency between SCALER and cuDSS solver on NVIDIA V100 and RTX 4090 GPUs. The improvement is the performance enhancement of SCALER over the best-performing of the compared GPUs. The resource utilization of SCALER is reported in Table IV.

TABLE IV: Resource utilization of SCALER on Xilinx Alveo U280 FPGA.

BRAM	DSP	FF	LUT	URAM
1540 (38.2%)	1476 (16.4%)	516K (19.6%)	364K (27.9%)	506 (51.4%)

1) **Throughput:** SCALER’s geomean throughput gains $1.79\times$ over V100 GPU and $1.44\times$ over RTX 4090 GPU, respectively. SCALER’s throughput further achieves a peak speedup of $2.00\times$ over these GPUs on matrix ‘poli4’. Despite RTX 4090 and V100 GPUs having significantly higher frequencies than SCALER (approaching or exceeding $3\times$), SCALER still achieves a notable improvement, owing to its streaming-aware synchronization scheduling to manage dependencies and exploit on-chip parallelism, compensating lower frequencies with high concurrent data processing capabilities, thus surpassing both GPUs in overall throughput.

2) **Energy efficiency:** SCALER’s geomean energy efficiency surpasses V100 GPU and RTX 4090 GPU by $4.20\times$ and $3.05\times$, respectively. It can be observed that SCALER achieves higher throughput while maintaining lower power consumption. This is primarily due to SCALER’s efficient data prefetching and optimized on-chip memory allocation strategies, which significantly reduce high-latency HBM accesses and thus lower data movement energy consumption.

3) **Bandwidth efficiency:** Bandwidth utilization is crucial for sparse LU factorization, a memory-intensive task. SCALER’s customized HBM-compatible dataflows with dedicated vectorized data packing yields its bandwidth efficiency significantly outperforming NVIDIA V100 GPU and RTX 4090 GPU by $5.12\times$ and $4.12\times$, respectively.

4) **Preprocessing Overhead Analysis:** We also record the preprocessing overhead of SCALER’s sparse storage format

conversion and vectorized packing. Fig. 4 compares preprocessing with sparse LU factorization on the CPU. For most test matrices, preprocessing time is roughly comparable to or lower than LU factorization. Only for ‘flowmeter5’ and ‘Pesa’ does it slightly exceed or approach, yet its cost remains relatively small. Although incurring additional time, this preprocessing is fully controllable. In scenarios such as circuit simulation [32] that requires repeated numerical sparse LU factorizations, our one-time preprocessing cost becomes negligible.

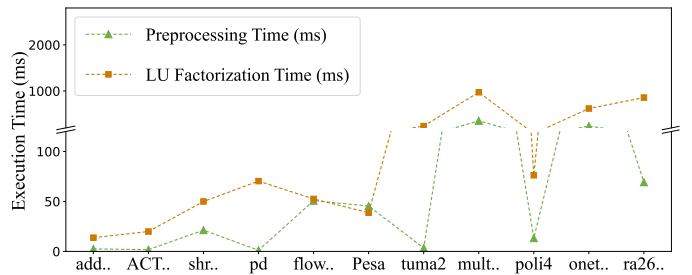


Fig. 4: Comparison of preprocessing time and LU numerical factorization time on 11 evaluated matrices.

VIII. CONCLUSION

This paper proposes SCALER, a high-performance sparse LU factorization accelerator on HBM FPGAs. We customize HBM-compatible data streams to maximize bandwidth utilization. A two-tier hierarchical memory module enhances memory access and on-chip data reuse. To manage complex dependencies and maximize parallelism, a stream-aware synchronization scheduling strategy is designed. Its multi-stage pipelined data prefetching mechanism hides latency by overlapping HBM access stages, improving off-chip communication. Evaluation shows SCALER improves throughput, energy efficiency and bandwidth efficiency compared to SOTA NVIDIA Tesla V100 and RTX 4090 GPU implementations.

ACKNOWLEDGMENT

This work was supported by NSFC (Grant No. 92473107, 12526211, 62374031), Zhejiang Provincial NSF (Grant No. D24F040002) and Jiangsu Provincial NSF (Grant No. BK20240173).

REFERENCES

- [1] X. Fu, B. Zhang, T. Wang, W. Li, Y. Lu, E. Yi, J. Zhao, X. Geng, F. Li, J. Zhang, Z. Jin, and W. Liu, "Pangulu: A scalable regular two-dimensional block-cyclic sparse direct solver on distributed heterogeneous systems," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 1–14, 2023.
- [2] Y. Lu, Y. Luo, H. Lian, Z. Jin, and W. Liu, "Implementing lu and cholesky factorizations on artificial intelligence accelerators," *CCF Transactions on High Performance Computing (CCF THPC)*, vol. 3, no. 3, pp. 286–297, 2021.
- [3] J. Zhao, Y. Wen, Y. Luo, Z. Jin, W. Liu, and Z. Zhou, "Sflu: Synchronization-free sparse lu factorization for fast circuit simulation on gpus," in *ACM/IEEE Design Automation Conference (DAC)*, pp. 37–42, 2021.
- [4] Z. Jin, W. Li, Y. Bai, T. Wang, Y. Lu, and W. Liu, "Machine learning and gpu accelerated sparse linear solvers for transistor-level circuit simulation: A perspective survey," in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 96–101, 2024.
- [5] G. Feng, H. Wang, Z. Guo, M. Li, T. Zhao, Z. Jin, W. Jia, G. Tan, and N. Sun, "Accelerating large-scale sparse lu factorization for rf circuit simulation," in *European Conference on Parallel Processing (Euro-Par)*, pp. 182–195, 2024.
- [6] D. Niu, Y. Tao, Z. Jin, Y. Dong, C. Wang, and C. Sun, "Islu: Indexing-efficient sparse lu factorization for circuit simulation on gpus," in *ACM/IEEE International Conference on Computer-Aided Design (ICCAD)*, pp. 1–9, 2024.
- [7] G. Feng, H. Wang, Z. Guo, M. Li, T. Zhao, Z. Jin, W. Jia, G. Tan, and N. Sun, "Efficient large-scale sparse lu factorization for fast radio frequency circuit simulation," *The International Journal of High Performance Computing Applications (IJHPCA)*, pp. 1–18, 2025.
- [8] M. I. Yang, R. Q. Liu, H. W. Gao, and X. Q. Sheng, "On the \mathcal{H} -lu-based fast finite element direct solver for 3-d scattering problems," *IEEE Transactions on Antennas and Propagation (TAP)*, vol. 66, no. 7, pp. 3792–3797, 2018.
- [9] M. Paszynski, D. Pardo, and V. M. Calo, "A direct solver with reutilization of lu factorizations for h-adaptive finite element grids with point singularities," *Computers & Mathematics with Applications*, vol. 65, no. 8, pp. 1140–1151, 2013.
- [10] G. Zhou, R. Bo, L. Chien, X. Zhang, F. Shi, C. Xu, and Y. Feng, "Gpu-based batch lu-factorization solver for concurrent analysis of massive power flows," *IEEE Transactions on Power Systems (TPWRS)*, vol. 32, no. 6, pp. 4975–4977, 2017.
- [11] Z. Yun, X. Cui, and K. Ma, "Online thevenin equivalent parameter identification method of large power grids using lu factorization," *IEEE Transactions on Power Systems (TPWRS)*, vol. 34, no. 6, pp. 4464–4475, 2019.
- [12] D. Yang, Y. Zhao, Y. Niu, W. Jia, E. Shao, W. Liu, G. Tan, and Z. Jin, "Mille-feuille: A tile-grained mixed precision single-kernel conjugate gradient solver on gpus," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 1–16, 2024.
- [13] M. Fan, X. Chen, D. Yang, Z. Jin, and W. Liu, "Recg: Reram-accelerated sparse conjugate gradient," in *ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2024.
- [14] Y. Lu, L. Zeng, T. Wang, X. Fu, W. Li, H. Cheng, D. Yang, Z. Jin, M. Casas, and W. Liu, "Amgt: Algebraic multigrid solver on tensor cores," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 1–16, 2024.
- [15] M. Fan, X. Tian, Y. He, J. Li, Y. Duan, X. Hu, Y. Wang, Z. Jin, and W. Liu, "Amgr: Algebraic multigrid accelerated on reram," in *ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2023.
- [16] Y. Zhao, X. Yang, Y. Bai, L. Zeng, D. Niu, W. Liu, and Z. Jin, "Csp: Comprehensively-sparsified preconditioner for efficient nonlinear circuit simulation," in *ACM/IEEE International Conference on Computer-Aided Design (ICCAD)*, pp. 1–9, 2024.
- [17] I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct methods for sparse matrices*. Oxford University Press, 2017.
- [18] N. Kapre and A. DeHon, "Parallelizing sparse matrix solve for spice circuit simulation using fpgas," in *International Conference on Field-Programmable Technology (FPT)*, pp. 190–198, 2009.
- [19] L. Zhuo and V. K. Prasanna, "Sparse matrix-vector multiplication on fpgas," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (ISFPGA)*, pp. 63–74, 2005.
- [20] Y. Mahajan, S. Obla, M. K. Nambhoorthiripad, M. J. Datar, N. N. Sharma, and S. B. Patkar, "Fpga-based acceleration of lu decomposition for analog and rf circuit simulation," in *International Conference on VLSI Design and International Conference on Embedded Systems (VLSID)*, pp. 131–136, 2020.
- [21] K. Lu, Z. Li, L. Liu, J. Wang, S. Yin, and S. Wei, "Redesk: A reconfigurable dataflow engine for sparse kernels on heterogeneous platforms," in *ACM/IEEE International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8, 2019.
- [22] H. Jun, J. Cho, K. Lee, H.-Y. Son, K. Kim, H. Jin, and K. Kim, "Hbm (high bandwidth memory) dram technology and architecture," in *International Memory Workshop (IMW)*, pp. 1–4, 2017.
- [23] 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 *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (ISFPGA)*, pp. 54–64, 2022.
- [24] L. Song, Y. Chi, L. Guo, and J. Cong, "Serpens: A high bandwidth memory based accelerator for general-purpose sparse matrix-vector multiplication," in *ACM/IEEE Design Automation Conference (DAC)*, pp. 211–216, 2022.
- [25] E. Yi, Y. Duan, Y. Bai, K. Zhao, Z. Jin, and W. Liu, "Cuper: Customized dataflow and perceptual decoding for sparse matrix-vector multiplication on hbm-equipped fpgas," in *Design, Automation and Test in Europe Conference (DATE)*, pp. 1–6, 2024.
- [26] L. Song, Y. Chi, A. Sohrabzadeh, Y.-k. Choi, J. Lau, and J. Cong, "Sextans: A streaming accelerator for general-purpose sparse-matrix dense-matrix multiplication," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (ISFPGA)*, pp. 65–77, 2022.
- [27] Y. Gao, L. Gong, C. Wang, T. Wang, X. Li, and X. Zhou, "Algorithm/hardware co-optimization for sparsity-aware spmm acceleration of gnns," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 42, no. 12, pp. 4763–4776, 2023.
- [28] E. Yi, J. Bai, Y. Nie, D. Niu, Z. Jin, and W. Liu, "Leda: Leveraging tiling dataflow to accelerate spmm on hbm-equipped fpgas for gnns," in *ACM/IEEE International Conference on Computer-Aided Design (ICCAD)*, pp. 1–9, 2024.
- [29] Z. He, L. Song, R. F. Lucas, and J. Cong, "Levelst: Stream-based accelerator for sparse triangular solver," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (ISFPGA)*, pp. 67–77, 2024.
- [30] T. Wang, W. Li, H. Pei, Y. Sun, Z. Jin, and W. Liu, "Accelerating sparse lu factorization with density-aware adaptive matrix multiplication for circuit simulation," in *ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2023.
- [31] C.-W. Ho, A. Ruehli, and P. Brennan, "The modified nodal approach to network analysis," *IEEE Transactions on Circuits and Systems (TCAS)*, vol. 22, no. 6, pp. 504–509, 1975.
- [32] X. Chen, "Ckts: High-performance parallel sparse linear solver for general circuit simulations," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 44, no. 5, pp. 1887–1900, 2025.
- [33] J. R. Gilbert and T. Peierls, "Sparse partial pivoting in time proportional to arithmetic operations," *SIAM Journal on Scientific and Statistical Computing (SISC)*, vol. 9, no. 5, pp. 862–874, 1988.
- [34] Y.-k. Choi, Y. Chi, W. Qiao, N. Samardzic, and J. Cong, "Hbm connect: High-performance hls interconnect for fpga hbm," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (ISFPGA)*, pp. 116–126, 2021.
- [35] L. Guo, Y. Chi, J. Lau, L. Song, X. Tian, M. Khatti, W. Qiao, J. Wang, E. Ustun, Z. Fang, Z. Zhang, and J. Cong, "Tapa: A scalable task-parallel dataflow programming framework for modern fpgas with co-optimization of hls and physical design," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 16, no. 4, pp. 1–31, 2023.
- [36] NVIDIA, "NVIDIA cuDSS (Preview): A High-Performance CUDA Library for Direct Sparse Solvers – NVIDIA cuDSS documentation," <https://docs.nvidia.com/cuda/cudss/index.html>, 2024.
- [37] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, pp. 1–25, 2011.