Sparsepipe: Sparse Inter-operator Dataflow Architecture with Cross-Iteration Reuse

Yunan Zhang
University of California, Riverside
USA
yzhan828@ucr.edu

Po-An Tsai
NVIDIA
USA
poant@nvidia.com

Hung-Wei Tseng
University of California, Riverside
USA
htseng@ucr.edu

2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO) | 979-8-3503-5057-9/24/$31.00 ©2024 IEEE | DOI: 10.1109/MICRO61859.2024.00090

***Abstract*— Sparse Tensor Algebra (STA) applications are limited by data movement and can benefit from better data reuse. Prior research has focused on intra-operator data reuse, such as better dataflow or caching technique for a single STA operation, missing other reuse opportunities at the application level. By expressing STA applications as sparse tensor dataflow graphs, we first identify two unexplored inter-operator data reuse opportunities: 1) producer-consumer reuse and 2) cross-iteration reuse. Producer-consumer reuse combines multiple operations to reduce data movement for intermediate results. Cross-iteration data reuse, a new opportunity identified in this paper, reduces the data movement for the shared sparse data (e.g., graph) across iterations. We then propose Output-stationary-Element-wise-Input-stationary (OEI) dataflow, a novel dataflow to capture both reuse opportunities in STA applications, and Sparsepipe, a sparse dataflow architecture to support the OEI dataflow and maximize data reuse. Evaluation results show that Sparsepipe with OEI dataflow is 19.82×/4.65× faster than CPU/GPU and 1.77× faster than an ideal sparse accelerator that cannot exploit inter-operator reuse.**

***Index Terms*—domain-specific accelerators , graph processing, high-performance and scientific computing.**

I. Introduction

Sparse tensor algebra (STA) is the key building block of scientific computing, graph analytics, and machine learning applications. STA operators such as SpMSpM (sparse matrix-sparse matrix multiplication), SpMM (sparse matrix-dense matrix multiplication), and SpMV (sparse matrix-dense vector multiplication) contribute to the majority of the runtime of these applications [25], [44]. Unlike dense tensor algebra, the data movement limits performance of STA applications due to their low arithmetic intensity. Thus, maximizing data reuse is the key to accelerating STA applications.

Prior research focuses on exploiting *intra-operator data reuse* to reduce data movement. For example, SpMSpM accelerators [25], [38], [70] propose dataflows with specialized format and microarchitecture support to minimize data movement in SpMSpM operations. Other work improves cache locality [8], [9] of SpMV operations in graph analytics. These ideas push the system closer to the roofline [62], where STA operations can fully utilize the available memory bandwidth. However, the intrinsic low arithmetic intensity in STA operations still causes many applications to reside in the bandwidth-bound region of the roofline even with existing optimizations [42]. Any innovative reduction in data movement is still attractive.

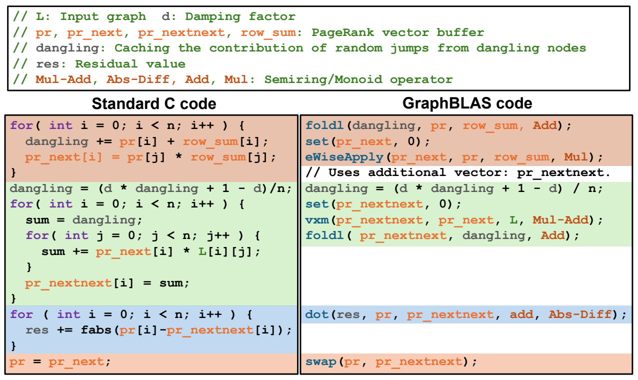
Conventionally, implementing STA applications requires hand-written and format-specific code with nested loops and application-specific logic, muddling opportunities to reduce data movements. Fortunately, recent developments in domain-specific STA languages and compilers [11], [19], [35], [51] alleviate the programming burden by generating low-level code for STA applications. Frameworks with tensor and dataflow abstractions, such as TensorFlow [2], PyTorch [28], GraphBLAS [16], and ALP [69], offer new abstractions for STA applications. Such novel abstraction with a dataflow graph presents data reuse opportunities *beyond a single operator*.

We find that there are two unexplored, inter-operator reuse opportunities for STA applications. First, **producer-consumer data reuse** reduces data movement by combining multiple tensor operations into a single, large fused operation. Prevalent in dataflow graphs, producer-consumer reuse is typically captured by forming pipelines of operations to keep intermediate results in on-chip buffers. While producer-consumer data reuse has been exploited widely in dense tensor algebra [3], limited work has explored this reuse opportunity in architecture for STA.

Second, **cross-iteration data reuse** is a *new reuse opportunity* revealed in this paper, which extends beyond single or adjacent STA operations. By unrolling loops or stages in STA applications, it is possible to fuse multiple identical operations (e.g., SpMV in a while-loop) and reduce memory traffic across iterations. No prior work has identified this opportunity, and harnessing cross-iteration data reuse requires a novel dataflow (Section III) and corresponding hardware supports (Section IV).

To exploit these inter-operator reuse opportunities, this paper proposes (a) OEI dataflow, which facilitates inter-operator data reuse, and (b) Sparsepipe–Sparse Inter-operator Dataflow Architecture, which incorporates key features:

- A dynamic execution pipeline with compute cores for each stage of the OEI dataflow. These cores support the diverse semiring operations prevalent in common STA applications, extending its applicability beyond HPC/DNN.
- An efficient on-chip buffer that streamlines the data supply to compute cores in the OEI dataflow.
- A set of intelligent control and management policies to schedule computation and data access tasks in sub-tensor manners to maximize data reuse, targeting the producer-consumer and cross-iteration reuse opportunities.
- A sparse tensor preprocessing algorithm, including blocking



```
// L: Input graph  d: Damping factor
// pr, pr_next, pr_nextnext, row_sum: PageRank vector buffer
// dangling: Caching the contribution of random jumps from dangling nodes
// res: Residual value
// Mul-Add, Abs-Diff, Add, Mul: Semiring/Monoid operator
```

Standard C code

```
for( int i = 0; i < n; i++ ) {
  dangling += pr[i] + row_sum[i];
  pr_next[i] = pr[j] * row_sum[j];
}
dangling = (d * dangling + 1 - d)/n;
for( int i = 0; i < n; i++ ) {
  sum = dangling;
  for( int j = 0; j < n; j++ ) {
    sum += pr_next[i] * L[i][j];
  }
  pr_nextnext[i] = sum;
}
for ( int i = 0; i < n; i++ ) {
  res += fabs(pr[i]-pr_nextnext[i]);
}
pr = pr_next;
```

GraphBLAS code

```
foldl(dangling, pr, row_sum, Add);
set(pr_next, 0);
eWiseApply(pr_next, pr, row_sum, Mul);
// Uses additional vector: pr_nextnext.
dangling = (d * dangling + 1 - d) / n;
set(pr_nextnext, 0);
vxm(pr_nextnext, pr_next, L, Mul-Add);
foldl( pr_nextnext, dangling, Add);

dot(res, pr, pr_nextnext, add, Abs-Diff);

swap(pr, pr_nextnext);
```

Fig. 1. Inner loop of `PageRank` algorithm. For simplicity, the c implementation assumes dense tensors.

and reordering, to improve inter-operator reuse.

We evaluate Sparsepipe on a set of STA applications and compare its performance against CPU- and GPU-based STA implementations. On average, Sparsepipe is 19.82× better than CPU and 4.65× better than GPU. We also compare Sparsepipe with an ideal sparse tensor accelerator but cannot exploit inter-operator reuse. Sparsepipe is up to 3.59 × faster than the ideal sparse accelerator and saves 54.98% dynamic energy.

In summary, our contributions are:

- Identifying *cross-iteration* data reuse in STA applications,
- A novel dataflow to exploit inter-operator data reuse, and
- The first sparse dataflow architecture to accelerate the complete algorithm in STA applications, instead of a specific operator or domain.

II. Background and Challenges

This section first reviews how to represent STA applications in modern sparse tensor frameworks. With this representation, this section will highlight the required hardware architecture ingredients and potential data reuses to motivate the proposed OEI dataflow and our Sparsepipe architecture.

A. STA applications as tensor dataflow graphs

Implementing high-performance STA applications traditionally requires various manual optimizations and is rarely portable. To improve programmers' productivity, recent advances in language and compiler design thus leverage and extend the abstraction of BLAS [1] and Einsum [35] to allow programmers to represent their STA applications as tensor dataflow graph. For example, in Fig. 1, we compare two implementations of a classic STA application, `PageRank`, one in standard C, and the other in the GraphBLAS [34], tensor-based abstraction.

There are three advantages of the dataflow representation. First, the building block is a set of well-define, *semiring* tensor operators, such as `vxm` or `mxm` (vector/matrix matrix multiplication), and a series of `e-wise` (element-wise) operations. For example, in `PageRank`, the used operators are `vxm` with `Mul-Add` as the semiring operation, and `set`, `fold`,

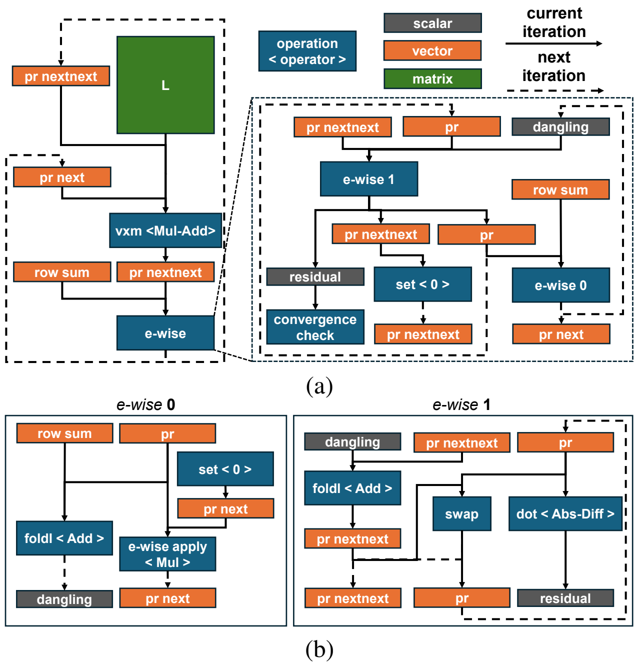


Fig. 2. Inner loop compute graph of `PageRank` algorithm, (a) abstracted compute graph fusing all `e-wise` operations with `e-wise 1` and `e-wise 2`. (b) shows further break down of fusing `e-wise 1` and `e-wise 2` with orignal tesnor operations used in Figure 1.

`dot` (vector-vector dot product), `swap` as `e-wise` operations, and other STA applications use different combinations (see Table. III). The implementation details of the operators, including storage format and tensor traversal order, are hidden from the programmer. This separation of concerns [50] lets programmers focus on expressing the applications and leaves how to optimize operators to system designers.

Second, under this abstraction, the tensor dataflow and dependency between operators is clear. Fig. 2 shows an abstracted compute graph of `PageRank`'s inner loop. The `vxm` operator takes vector `pr_next` and matrix `L` as input and produces vector `pr_nextnext` as output, while other vectors serve as inputs, outputs, or intermediates for fused `e-wise`. As shown in Fig. 2 (b), two groups of `e-wise` can be fused by identifying connected components of operations and data nodes, yielding two new sub-graphs. This abstracted compute graph enhances visibility of data dependencies across each loop iteration, facilitating exploration of cross-iteration data reuse opportunities.

Lastly, under the dataflow representation, STA applications contain multiple iterations/stages of *the same subgraph*, with each iteration advancing towards a convergent result. The loop body can generally be divided into a BLAS-2/3 operation (`vxm` or `mxm`), and a series of `e-wise` operations. Very often, the matrix in the `vxm` or `mxm` operator is a constant sparse tensor (e.g., the graph `L` in `PageRank`), which accounts for the most of the data movement and is shared across iterations.

B. Architectural support to accelerate STA applications

To accelerate dataflow-based STA application like `PageRank` shown in Fig. 2, we identify several key architectural supports missing from existing solutions.

First, supporting and accelerating configurable *semiring* tensor operations is a must. STA applications implemented in frameworks like GraphBLAS require a larger set of semiring operators (Table. III). Prior sparse accelerators, such as GAMMA [70] for scientific applications, SCNN [45] for sparse DNNs, optimize the data reuse within a STA operator, but only support multiply-add as the basic computation. SIMD^2 [71] extends dense tensor accelerator for general semiring computation, but no prior work in sparse accelerators has the required semiring support.

Moreover, simply combining ideas in optimizing intra-operator reuse and SIMD^2 does not capture the full reuse opportunity in Fig. 2. To fully capture the inter-operator reuse, the sparse architecture should support an explicit data staging between operators. ISOSceles [68] proposes hardware support to capture such producer-consumer reuse, but only for sparse CNNs. ALP and GraphBALS' nonblocking execution method lets the programmer exploits producer-consumer reuse of STA applications in CPUs, but the lack of an explicit buffer control in hardware prevents the programmer from exploiting the reuse opportunity. Such hardware support is similar to prior accelerators [47] for dense tensor dataflow graphs, but need to specialize for the dataflow and dynamism of STA applications.

Finally, despite that the sparse matrix is very often reused across multiple iterations or stages in STA applications, the footprint of this sparse matrix and the long reuse distance (i.e., cross-iteration) prevent any prior on-chip buffer or cache optimizations from capturing such reuse. To address this, the system needs to store only a small portion of sparse tensors at any time and *executes work in different iterations in a short time window* to exploit the potential reuse across iterations, ensuring that stored data is quickly consumed to make room for new data. Therefore, the system must closely monitor the on-chip buffer and schedule work to maximize reuse.

Implementing all above required support in purely software can be both challenging and inefficient, negating the potential benefits of inter-operator data reuse. These opportunities and challenges motivate us to propose the OEI dataflow and develop the Sparsepipe architecture.

C. Sparsepipe v.s. Dataflow architectures

Sparsepipe has roots from decades of work in dataflow architectures [5], [52], [6], [56] and runtime systems [10] that treat data dependencies as first-class citizens and form pipelines between processing units to limit control and synchronization overheads. Both static [17] and dynamic [5] dataflow machines contrast the von Neumann architecture with fine-grained dependency tracking (e.g., tokens) techniques. More recent work instead implements task-based dataflow runtime systems on commercial ISAs (e.g., Cell [10] and x86 [18], [24]) with no or limited changes to the hardware.

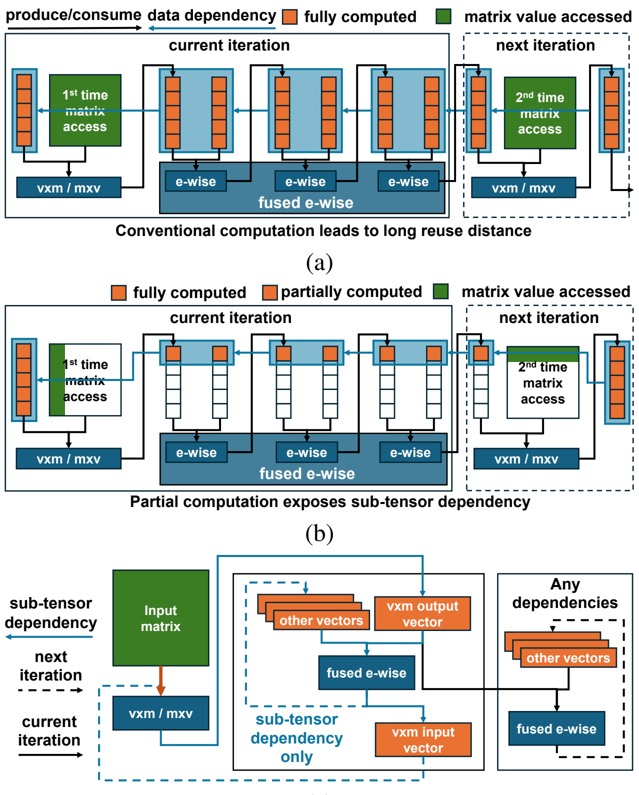


Fig. 3. Generalized compute graph of STA applications. (a) Data dependencies of STA application using conventional computation. (b) Data dependencies of STA application using partial computation. (c) Abstracted STA compute graph with isolation of sub-tensor dependency only region.

Sparsepipe advances the concept from prior dataflow architectures and runtime systems for tensor workloads in the following aspects. First, this work targets the *sparse tensor abstraction* and builds on recent advances in tensor-based programming models to track dependency at the (sub-)tensor level. Without Sparsepipe, prior work can only exploit the dataflow in scalar and task-based programming models, missing the advantages in productivity of the tensor-based dataflow graph in Fig. 1 that is critical in STA. Second, the Sparsepipe architecture with the OEI dataflow captures the long-distance data reuse (i.e., reuse the sparse matrix across iterations) in STA. In contrast, prior dataflow architectures and programs cannot efficiently exploit cross-iteration reuse even with programmers' careful crafting of loop orders and applying tiling algorithms [14].

III. Exploiting Cross-Iteration Data Reuse

This section details a novel dataflow tailored for cross-iteration data reuse. We start by formulating an abstract view of STA applications to identify cross-iteration data reuse.

A. Abstracting sparse algorithms

STA algorithms typically can be decomposed into two components: leading matrix (e.g.,`vxm`/`mxv`) operations and sub-

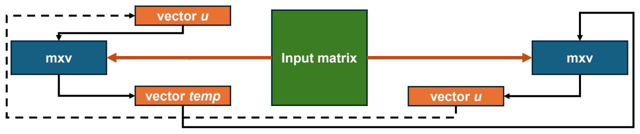


Fig. 4. Inner loop compute graph of `KNN`.

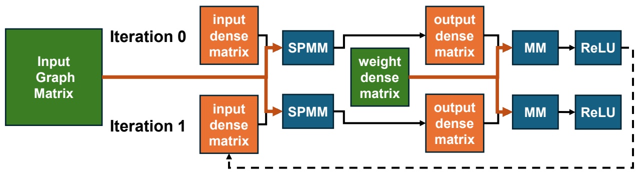


Fig. 5. Inner loop compute graph of `GCN`.

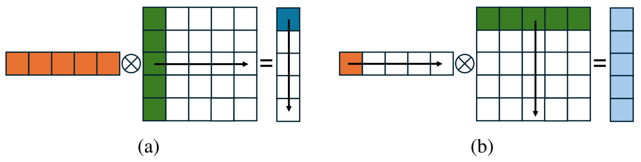


Fig. 6. (a) Output stationary `vxm` dataflow. (b) Input stationary `vxm` dataflow.

sequent `e-wise` operations. By fusing all `e-wise` operations, the originally complex compute graph simplifies, revealing clear data dependencies between input and output tensors.

To exploit cross-iteration reuse, a dataflow schedule must simultaneously execute operations spanning multiple loop iterations. That is, to reuse the input sparse matrix, it is crucial that the `vxm` and fused `e-wise` from the current iteration are fused with the `vxm` of the subsequent iteration.

However, conventional dataflow schedule of STA applications executes operators sequentially (i.e., `vxm` has to finish before `e-wise` starts). Such schedule leads to a long reuse distance between two consecutive `vxm` operations. Figure 3 (a) shows the unrolled compute graph of an arbitrary STA application including `vxm` in two iterations. To provide the input vector of the second `vxm`, the current iteration needs to access the entire input matrix for the first `vxm` to produce the output vector, and fully compute fused `e-wise` operations. Data dependencies on the entire output vector lead to long reuse distance, preventing efficient fusing of two `vxm` operations.

Fortunately, for STA in dataflow representation, the loop traversal order is hidden from the programmer. The system can optimize the schedule arbitrarily and perform partial computation, so long as it acknowledge the finest-data dependency (as small as a scalar). Figure 3 (b) demonstrates the advantage of partial computation, which reveals finer granularity of data dependencies. If the schedule aims to produce *just a single input element* for the subsequent `vxm`, the current iteration only needs to partially access the input matrix and compute corresponding elements of fused `e-wise` operation. We define such finer data dependency as **sub-tensor dependency**.

Sub-tensor dependency reduces the reuse distance of two consecutive `vxm` operations, as the top-left element of the sparse matrix can be reused after the first `vxm` access only a column of the sparse matrix, instead of the full matrix. Isolating the **sub-tensor dependency-only-region** of the dataflow thus reveals cross-iteration data reuse opportunity in any STA applications. Figure 3 (c) shows the generalized STA compute graph after fusing `e-wise` operations. For any STA compute graph, if there exists a subgraph that includes both input and output vector of `vxm`, and all operations within the subgraph exhibit sub-tensor dependency, fusing two `vxm` can leverage cross-iteration data reuse. For example, in `PageRank`, a valid subgraph can be structured by `vxm` $\rightarrow$ `e-wise` 1 $\rightarrow$ `e-wise` 0 $\rightarrow$ `vxm`, which exposes sub-tensor dependency for all operations fused in `e-wise` 1 and `e-wise` 0.

Several other structurally different compute graphs also reveal the benefits of this generalized STA abstraction. KNN (K-nearest neighbors), as shown in Figure 4, incorporates two `vxm` (or `mxv`) within the same iteration. Despite its unique structure, the circular dependency between the two `vxm` across iterations forms a subgraph: `vxm` $\rightarrow$ `no-op` $\rightarrow$ `vxm`, making reuse of input matrix possible.

In addition, as shown in Figure 5, Graph Convolutional Neural Networks (GCNs) can be represented as subgraphs of `SpMM` $\rightarrow$ `MM` (Dense Matrix Multiplication) $\rightarrow$ `ReLU`. Since no value in the input dense matrix is blocked by `MM` and `ReLU`, and `SpMM` can be implemented as multiple `vxm`, it is possible to fuse SpMM operations from different stages to exploit cross-iteration data reuse.

Based on this observation, so long as an STA algorithm can be abstracted with the generalized compute graph, cross-iteration data reuse applies regardless of whether the fused operations occur within a single loop iteration or span across multiple iterations.

B. *OEI dataflow*

Within the isolated subgraph, exposing sub-tensor dependencies ensures that subsequent `vxm` must execute concurrently with preceding `vxm` to avoid blockage. Given that all other operations within the subgraph do not interfere with each other, any output generated by the earlier `vxm` can be immediately consumed by the subsequent `vxm`.

However, single `vxm` can only choose stationarity between the input vector or the output vector. As illustrated in Figure 6, `vxm` operations exhibit two prevalent compute dataflows: (a) Output Stationary (OS) dataflow, which generates a single element in the output vector at a time, requiring access to all input vector elements, and (b) Input Stationary (IS) dataflow, which yields partial results for all output vector elements but requires only a single input vector element at a time.

Conventional implementations of STA algorithms adopt one dataflow type across all iterations, which prevents the cross-iteration data reuse of multiple `vxm`. For instance, when fusing two `vxm` with OS dataflow, the first `vxm` produces one output element at a time, but the second `vxm` requires entire vector output from the first `vxm` to start. Namely, the first

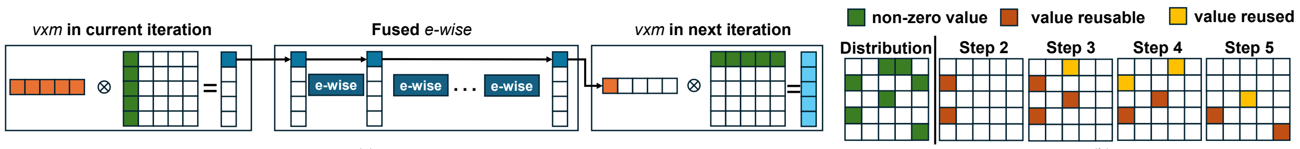


Fig. 7. Overview of OEI dataflow, (a) Fusing OS `vxm` and IS `vxm`, (b) Reuse oppertunity of an example sparse matrix

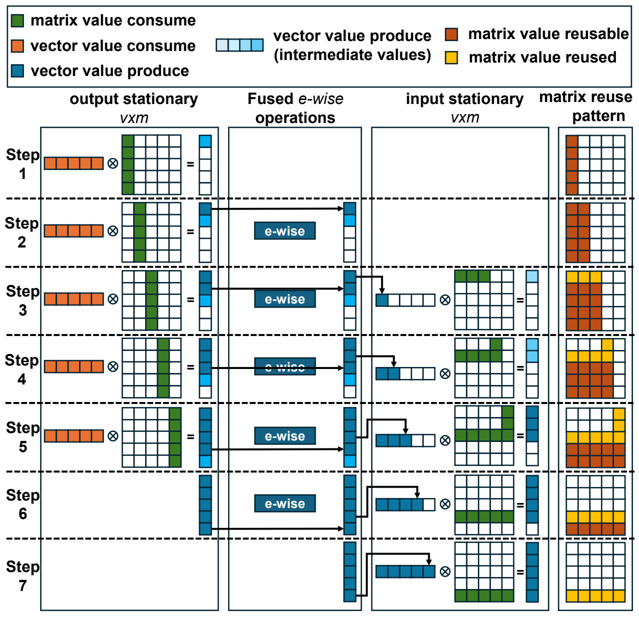


Fig. 8. Illustration of OEI dataflow for dense matrices.

TABLE I
PORTION OF SPARSE MATRIX NEED TO BE STORED ON-CHIP TO ENABLE OS-EWISE-IS DATAFLOW (SMALLER % IS BETTER)

matrix	**row/col**	**nnz**	**max (%)**	**avg (%)**
ca	18772	198110	98802 (49.9%)	65124 (32.9%)
gy	17361	178896	8661 (4.8%)	3321 (1.9%)
g2	150102	438388	15448 (3.5%)	7304 (1.7%)
co	434102	16036720	2143362 (13.7%)	1155196 (7.2%)
bu	513351	10360701	9329007 (90%)	4944897 (47.7%)
wi	3566907	45030389	17422630 (38.7%)	10450514 (23.2%)
ad	6815744	13624320	1143568 (9.4%)	694064 (5.1%)
ro	23947347	28854312	557694 (1.9%)	281769 (1.0%)
eu	50912018	54054660	2338567 (4.3%)	1419430 (2.6%)

`vxm` does no expose sub-tensor dependency with the second `vxm`. Conversely, when choosing IS dataflow for both `vxm`, the input vector elements for the second `vxm` are not fully available until the completion of the first `vxm`, which also prevents the sub-tensor dependency between two `vxm`.

To address this, **our insight is to employ OS dataflow for the first `vxm` and IS dataflow for the second `vxm`.** This mixed dataflow meets the necessary condition to reuse the data from the sparse matrix. As depicted in Figure 7 (a), the first `vxm` with the OS dataflow generates an output vector element, subsequently consumed by the fused `e-wise` operation. The element produced by `e-wise` operations can be directly consumed by the second `vxm` with the IS dataflow, without any hindrance from preceding operations. We name such dataflow as OEI (OS-ewise-IS) dataflow, which facilitates the simultaneous execution of operations in the subgraph, enabling the use of large input matrices in two `vxm` across iteration.

Fig. 8 demonstrates the OEI dataflow with a 5×5 dense matrix as an example. In each step, the OS `vxm` computes one output vector element by accessing the entire input vector and a single column of the input matrix. The fused `e-wise` is delayed by one step relative to the OS `vxm` because the input vector elements required for `e-wise` are not fully available until the OS `vxm` completes its previous step. Similarly, IS `vxm` lags two steps behind the OS `vxm`, as it awaits the completion of both the OS `vxm` and the fused `e-wise`. IS `vxm` scatters the partial sum from the multiplication for each pair of elements, where the matrix value has been previously use by OS `vxm`. Therefore, the IS `vxm` avoids the computation of the full outer product in each step.

To show the data reuse, the matrix reuse pattern in Fig. 8 indicates matrix values that are either ready for reuse or are currently being reused. Figure 7 (b) highlights the matrix reuse pattern for selected execution steps after applying the OEI dataflow to a sparse matrix. For extremely sparse input matrices in STA applications, only a small portion of the matrix is required to be buffered at a time, potentially fitting within a standard on-chip buffer.

Table I demonstrates our simulation results for the maximum and average percentage of the nonzero values in a sparse matrix to be stored on-chip using the OEI dataflow. For a vast majority of the examined matrices, maintaining only a small fraction (<10%) of values is sufficient to capture the reuse opportunity in the OEI dataflow.

When the on-chip buffer can hold all the reusable matrix values across all steps, the IS `vxm` need not to load any matrix element from memory. However, given the uneven distribution of non-zero values in sparse matrices, it becomes challenging to ensure that OS `vxm` uniformly loads data from the main memory in each step. This uneven data distribution can lead to load imbalance between OS and IS stages and under-utilization of memory bandwidth in certain steps.

To address this, figure 9 shows an enhancement to the OEI dataflow, using the same sparse matrix referenced in Figure 7 (b). In step 3, the initial element of the IS `vxm` input vector

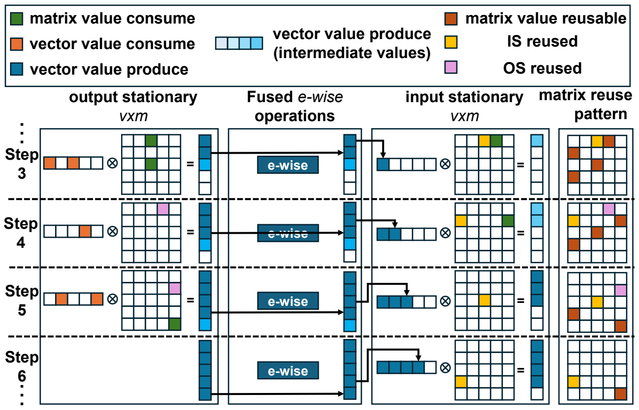


Fig. 9. Illustration of OEI dataflow for sparse matrices with eager IS execution.

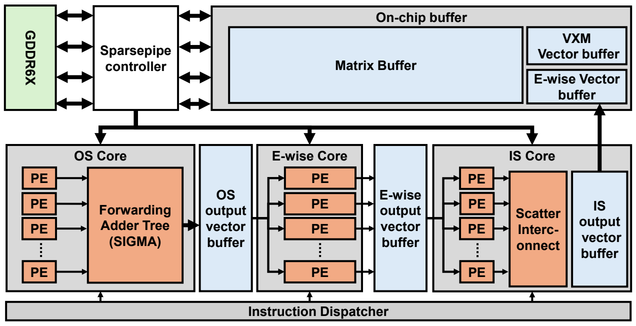


Fig. 10. High-level architecture of Sparsepipe: Pipelined OS Core, IS Core, and E-Wise Core share an on-chip buffer.

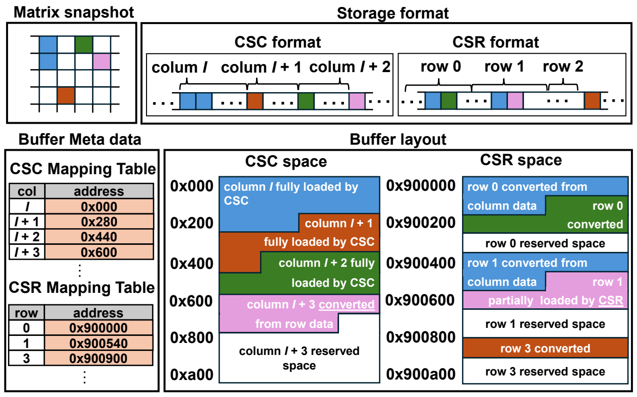


Fig. 11. Dual sparse storage and memory layout of Sparsepipe on-chip buffer

is generated by fused `e-wise` operations. Besides computing a partial sum with reusable matrix elements, the IS `vxm` also proactively loads another matrix value from memory, instead of idling. The matrix value loaded by IS `vxm` is instead reused by OS `vxm` in step 4, eliminating the need for additional main memory loading. Similar dynamics occur between steps 4 and 5. This refined approach addresses the issues of load imbalance and bandwidth utilization.

IV. Sparse Inter-operator Dataflow Architecture

This section will overview the proposed Sparsepipe architecture and describe key components and optimizations.

A. Overview of Sparsepipe

Figure 10 presents the high-level functional blocks of Sparsepipe. Sparsepipe efficiently supports sparse tensor algebra while enabling data reuse opportunities in the following aspects.

- Dual sparse storage in on-chip buffers to efficiently accommodate various data access demands in different stages of OEI dataflow.
- A dynamic scheduling pipeline that natively supports OEI dataflow and sparse tensor semiring operations through compute cores, dedicated to the demand of each stage in the OEI dataflow: the OS Core, E-Wise Core, and IS Core.
- A dataflow-aware controlling logic that schedules memory accesses, dispatches computation tasks, and prefetches data in units of sub-tensors in the control logic to efficiently use memory bandwidth and buffer space.

Sparsepipe leverages the compiler behind existing STA programming frameworks to generate efficient tensor semiring instructions. Sparsepipe and a Sparsepipe-compliant language framework partition data and schedule computation tasks in sub-tensors to more efficiently use memory bandwidth and reduce memory footprint. Sparsepipe initiates a stream of computation tasks from loading input sub-tensors that typically contain multiple columns of data for the OS Core and propagates the intermediate sub-tensors to the E-Wise Core. In the meantime, Sparsepipe can initiate another stream of computation tasks by loading another set of sub-tensors into the buffer. Sparsepipe can start OS operations for the later stream once the previous stream advanced to the E-wise operations. By executing streams of sub-tensor computation tasks in a pipeline manner among OEI compute cores, Sparsepipe exploits pipeline parallelism and cross-iteration data reuse opportunities. In addition to the architectural supports and optimizations, the language framework can further optimize data structures that facilitate program execution and improve space efficiency.

B. Dual sparse storage

In Sparsepipe architecture, both the OS and IS Cores require sub-tensor inputs from the same sparse tensor but with opposite traversal orders. Specifically, the `vxm` operations in OS Cores need sub-tensors from the input matrix in column order for the desired semiring operations with the input vector. In contrast, IS `vxm` demands row-wise access for scatter multiplication with a matrix row to yield partial outputs.

Sparsepipe on-chip buffers store input sub-tensors in a dual storage strategy that utilizes both CSC and CSR formats to optimize data access for both OS and IS dataflows to address the limitation that no single sparse matrix storage format optimally supports both row and column data access simultaneously.

Sparsepipe does not take the design option of using a single buffer to store orientation-neutral data format like Coordinate List (COO) because the orientation-neutral design can only ensure efficient access for the sorted dimension.

Figure 11 depicts the high-level idea of the dual storage on-chip buffers. Each buffer contains a CSC space for data in CSC format and CSR space for data in CSR format. As CSC format consecutively places column data between *col_start* and *col_end* of the coordinate and data arrays, each *val* shares the same *col_idx* but exhibits unique *row_coord*. Sparsepipe stores the same column consecutively in the on-chip buffer, providing straightforward access for OS Cores. The practice of consecutive fetching and storing column data fetched from CSC format seamlessly extends to managing row data fetched from CSR format. Storing row data fetched from CSR format into the CSR space follows a similar strategy. The starting memory address of each row and column are recorded in the CSC/CSR mapping table for subsequent access.

As data in the same column always belongs to unique rows, the on-chip buffer must store the converted row data non-consecutively in the CSR space the original input is in CSC format. To address this, Sparsepipe determines the necessary space for each row using *row_start* - *row_end* from the CSR index array, reserving space upon receiving the first converted row data from CSC data. Specifically, when column data (*col_idx, row_coord, val*) from CSC format is converted to row data as (*row_coord, col_idx, val*), equivalent to (*row_idx, col_coord, val*) in CSR format, Sparsepipe allocates exact space for converting non-zero values in each row and records the starting memory addresses in CSR mapping table.

As STA applications demand column data in the order from lower to higher *col_idx*, the row data in which an earlier fetching operation brings should always have lower *col_coord* compared the later ones. Therefore, the first non-zero element of any row can always trigger space reservation in advance, allowing for consecutive and ascending storage of subsequently fetched row data within its reserved space. Additionally, when row data are eagerly loaded from the CSR format to utilize the remaining memory bandwidth, they are converted to column data and stored in the CSC space following the same logic.

C. The OEI compute pipeline

Sparsepipe's compute pipeline features the OS Core, the E-Wise Core, and the IS Core to efficiently execute the OEI dataflow.

1) OS Core: The OS Core executes operations on each matrix column and vector pair like dot-products. Beyond the `mul-add` operation, Sparsepipe extends each processing element (PE) to additionally support frequently used semiring operations in sparse tensor algebra, including `and-or`, `min-add`, `aril-add`. Each PE in the OS Core can execute of a semiring operation on a sub-tensor/column simultaneously with other PEs. Sparsepipe uses the Forwarding Adder Tree from SIGMA [49] to handle the varying number of non-zero elements per column, allowing flexible sets of PEs to communicate during the reduction phase of each vector-column dot product. The OS Core stores the generated intermediate sub-tensors (vectors) in a buffer for the later stage.

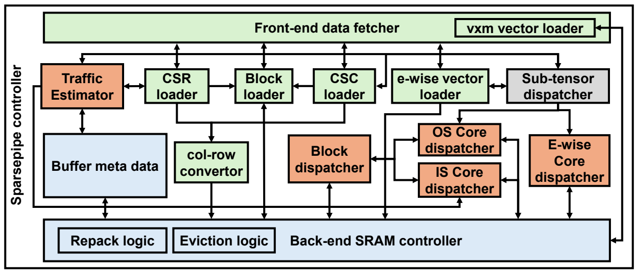


Fig. 12. Sparsepipe control logic

2) E-Wise Core: E-Wise Core processes `e-wise` operations on the OS `vxm`'s output buffer, where the sub-tensor size directly corresponds to the number of elements managed by the E-Wise Core in each step of OEI dataflow. The E-Wise Core processes sub-tensor elements concurrently in Single Instruction, Multiple Data (SIMD) model. Sparsepipe uses offline compilation to pre-generate instructions for fused `e-wise` operations specific to an application. Each PE in the E-Wise Core is identical to the PEs in the OS Core. E-Wise Core stores the results of sub-tensors into the e-wise vector buffer.

3) IS Core: The IS Core performs outer product calculation, where a single vector element multiplies against multiple row elements and accumulates with intermediate values generated in E-Wise Cores. Sparsepipe includes an output vector buffer to store partial results from the IS Core and employs a scatter network to integrate the most recent products. Sparsepipe writes each fully computed output vector element back to the main memory. Similar to OS Core, IS Core configures semiring operators specific to each application before execution, eliminating any additional runtime overhead.

D. Control logic

The control unit in Sparsepipe targets utilizing all available memory bandwidth beyond controlling compute resource operations to push performance to the roofline. Figure 12 shows Sparsepipe's control logic that serves the following purposes:

- Regulating each OEI dataflow stage's functional units.
- Estimating bottleneck component of Sparsepipe at each step within the OEI dataflow.
- Managing the selection of CSC/vector sub-tensors for loading in the forthcoming steps of the OEI dataflow.
- Determining when to load/prefetch CSR data to maximize memory bandwidth utilization.

1) Pipeline control: The pipeline control logic generates control signals for each datapath element in the OEI pipeline. Figure 13 shows the concept of Sparsepipe's pipeline control. In Figure 13, the sub-tensor with index I represents the first sub-tensor in the STA program's compute order, followed by sub-tensors with indexes $I+1$, $I+2$, and $I+3$, respectively. The horizontal axis of Figure 13 represents the time as steps

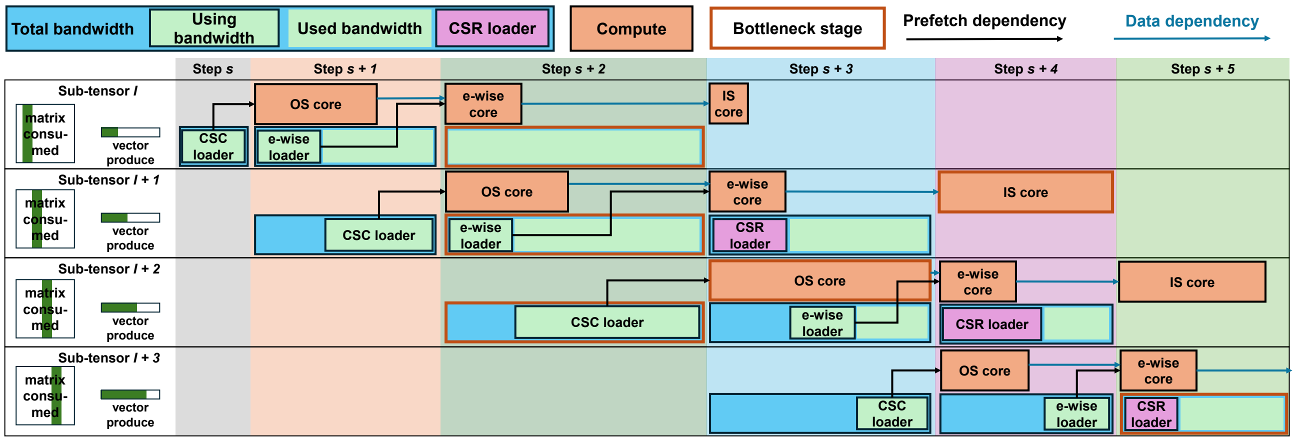


Fig. 13. Runtime behavior of Sparsepipe's pipeline and stages in OEI dataflow.

corresponding to the progress of Sparsepipe's pipeline in processing sub-tensors. Each sub-tensor in the pipeline will go through four steps in the following order. (1) the CSC loader, (2) the OS stage and e-wise data loader, (3) the e-wise stage and CSR data loader (optional), and finally, (4) the IS stage.

Based on the sub-tensor workflow, the pipeline control sets the operations of each core unit accordingly. For example, at time step $s+2$, if the E-Wise Core dispatcher receives tasks on sub-tensor I, then the OS Core dispatcher should receive sub-tensor $I+1$, and in the meantime, loading the input vector of element-wise operations that the pipeline will compute with the OS output of sub-tensor $I+1$ before the future step $s+3$ arrives. Also, in step $s+2$, the CSC loader can start loading sub-tensor $I+2$ since sub-tensor $I+2$ is the input for the OS in step $s+3$.

Unlike OS `vxm` and `e-wise`, which operates on determinant sub-tensor in each step of OEI dataflow, the runtime behavior of the IS `vxm` can vary based on several conditions: (1) Sparsepipe opts to load CSR data only when there's leftover bandwidth. (2) The IS dataflow limits its computation to element-row scatter multiplication exclusively for the rows in the on-chip buffer. (3) Scatter multiplication computations by the IS dataflow are contingent upon the completion of OS `vxm` and `e-wise`.

2) Sub-tensor loading and prefetching: Sparsepipe contains three dedicated data loaders for each OEI dataflow's compute stage: the CSC loader for the OS Core/stage, E-wise vector loader for E-wise stage, and CSR loader for the IS Core/stage. As Figure 13 depicted, these data loaders load data a step before the step when the corresponding compute stage occurs.

If the sub-tensor has a size of T columns, the OS Core dispatcher will load columns and vector elements with indexes ranging from $(I+1)*T$ to $(I+2)*T$, whereas the E-Wise Core dispatcher loads elements of e-wise vectors with indexes ranging from $I*T$ to $(I+1)*T$. On the other hand, IS Core dispatcher, theoretically, can eagerly compute scatter multiplication up to the s-th rows. However, to prevent bottlenecks in the IS Core from affecting other stages and to align with data prefetched by the CSR loader, the IS Core conservatively fetches up to R (received from the traffic estimator) row data in all rows along with the corresponding vector elements.

Both the CSC and e-wise vector loaders directly receive sub-tensor indexes from the sub-tensor dispatcher and generate fetch commands for CSC and vector inputs, respectively. The loader issues commands to the DRAM if a demanding portion of the sub-tensor does not exist in on-chip buffers. For each loaded sub-tensor, the col-row convertor flips the column and row index from the received row/column data. With accessing space reservation size from buffer meta data, col-row convertor issues commands to SRAM controller and stores fetched data to on-chip buffer.

The CSR loader receives a parameter R from the traffic estimator, indicating the quantity of desired sub-tensor data. As on-chip buffer stores all row data in consecutive and ascending from each row's first non-zero element, the CSR loader can access all rows eligible for IS `vxm` computation from the buffer. Suppose memory bandwidth saturates due to the loading of CSC data for the OS dataflow and vector data for E-wise operations. In that case, the IS Core will prioritize computing scatter multiplications solely on data already fetched or converted to CSR format. When bottlenecks arise during the compute phase and access to the CSC and vector data under-utilizes the available memory bandwidth, Sparsepipe prioritizes loading CSR data to mitigate this imbalance.

Sparsepipe also prefetches CSR data heuristically. For each row r has an index greater than the highest fully computed row index S by the IS `vxm`, yet smaller than the fully computed index E, the CSR loader decides the number of elements to prefetch per row by considering (a) the total number of fetched row data does not surpass R and, (b) for each row r, where $S<r<E$, the CSR loader calculates $P(r)=\frac{\sum_{i=S}^{r}T(i)}{r}$, where $T(i)$ denotes the count of already fetched row data for row i. This heuristic ensures that (a) the loaded row data fully utilizes the remaining memory bandwidth, and (b) load balance of IS

`vxm`. The CSR loader then issues fetch commands for each row r to retrieve consecutive data starting from the next non-zero at *col_coord* to *col_coord*$+P(r)$.

If a column in CSC space has reserved its space but a newly converted column data is not the next non-zero element the computation task needs, Sparsepipe discards this conversion. Even if storing the converted column were feasible, any not-requested data with a lower column index would lead to separate fetches, potentially causing the CSC loader to initiate multiple commands for non-consecutive data in the same column, thus diminishing memory utilization efficiency.

Upon receiving a fetch command from CSC and CSR loaders, the front-end data fetcher retrieves the data and computes a set of *vxm* vector indexes by intersecting all received *row_idx* and *row_coord*. It retrieves the necessary vector data from memory and stores them in the on-chip buffer. Lastly, the e-wise vector loader directly issues fetch commands since e-wise PEs use all e-wise data in a single step of the dataflow.

3) Data eviction and repacking: OEI dataflow facilitates data reuse by only storing a fraction of the input matrix in the on-chip buffer. Sparsepipe further reduces the buffer size required with an efficient eviction policy and a buffer repacking mechanism. In the CSC buffer, Sparsepipe evicts entire column data immediately after the OS Core processes them to free up reserved space. Using sub-tensor execution further prevents fragmentation by fetching and evicting multiple columns concurrently.

Since IS Core operations consume each data element in a row individually, Sparsepipe maintains additional metadata for each stored row to monitor the total count of consumed elements. Upon surpassing a predetermined threshold of total consumed elements, the controller initiates a buffer repacking process that discards fully computed sub-tensors and places remaining sub-tensors in a contiguous CSR space.

When Sparsepipe encounters Out-Of-Memory (OOM) conditions without any available repacking opportunities, Sparsepipe adheres to the reuse patterns outlined in Figure 8, prioritizes eviction for rows with higher *row_idx*. The control logic will reload the evicted row when later steps of OEI dataflow need that row.

E. Sparse tensor preprocessing

Sparsepipe implements two offline optimizations designed to preprocess input sparse matrices.

1) Row reorder: Sparsepipe employs row reordering to enhance the locality of the non-zero distribution of sparse matrices. As any converted row data may trigger CSR space reservation, string too much unconsumable row data with high *row_idx* causes frequent Out-Of-Memory in the on-chip buffer, leading to memory ping-ponging in later steps. Sparsepipe favors fetching row data with higher *row_idx* in later steps of the OEI dataflow.

Like prior works optimizing the SpMSpM operation [70], Sparsepipe utilizes the GraphOrder algorithm [61] to rearrange rows for input data. Additionally, Sparsepipe incorporates a straightforward vanilla reorder algorithm as an alternative, which aims to reorder the sparse matrix towards an upper triangular matrix with simple heuristics.

2) Blocked sparse storage: Sparsepipe also adopts blocked sparse storage to reduce the storage overhead due to dual sparse storage. Dual sparse storage has two main drawbacks: (a) CSC and CSR formats use redundant data arrays (with different orders). (b) Overhead in indexing structure as each coordinate requires at least 4 bytes. Using the FiberTree notation proposed in Sparseloop [65], [66], Sparsepipe uses UOP-CP-CP format to compress dual sparse storage and reduce storage size. Specifically, each value in the data array of CSC and CSR points to a non-zero block of the original sparse matrix, which offers two advantages: (a) A single byte can store a coordinate within any block that has a size up to 256, which saves $4\times$ space compared with using 4-bytes coordinates; (b) quantity of non-zero blocks is significantly less than non-zero values, allowing CSR and CSC format to have less redundancy by storing pointers of sprase blocks and sharing the same data and coordinate array.

Beyond storage efficiency, blocks of non-zeros also provide Sparsepipe performance improvements for specific input matrices and applications. Leveraging these benefits, Sparsepipe opts for blocked sparse storage. When utilizing blocked storage, CSC and CSR loaders additionally transmit the loaded block ID to the block loader, which then facilitates the Front-end data fetcher in loading the required non-zero blocks. Similarly, in the compute stage, OS and IS Core dispatchers load prefetched non-zero blocks and unpacked row/column data that compute cores will later use for processing.

F. Code generation and optimization

Sparsepipe leverages the nonblocking execution pattern, the code generation, and the optimization approach similar to ALP/GraphBLAS. However, as Sparsepipe provides hardware support for the OEI dataflow, Sparsepipe can perform static compilation of a tensor program instead of runtime code generation.

The offline compilation process begins with a data dependence analysis on the tensor-based program (e.g., GraphBLAS program), separating it into sub-tensor dependence groups and all other operation groups. For element-wise operations within each group, Sparsepipe merges consecutive operations to exploit inter-operator reuse and generates fixed vector instructions for the `e-wise` core. Unlike the dynamic data dependency analysis used in ALP/GraphBLAS, Sparsepipe's OEI dataflow enables looser conditions for fusing element-wise operations, incurring no runtime overhead.

Based on the semi-ring operator for each application, the compiler generates opcodes for the OS and IS core operations. At runtime, all cores execute identical operations with preloaded opcodes, relying on sub-tensor indexes received from the sub-tensor dispatcher for each sub-tensor of the input matrix and vectors. Sparsepipe can either operate on a fixed sub-tensor size for an already optimized configuration or explore the optimal sub-tensor size in the initial steps of the OEI dataflow.

TABLE II
MEMORY CONFIGURATION EVALUATED

	Bandwidth (GB/s)	**Latency (Read/Write) (ns)**	**DRAM tech**
CPU (AMD 5800X3D)	40	13.75/12.5	DDR4
GPU (NVIDIA 4070)	504	12.0/5.0 [41]	GDDR6X
Sparsepipe (iso-CPU)	40	13.75/12.5	DDR4
Sparsepipe (iso-GPU)	504	12.0/5.0	GDDR6X

TABLE III
BENCHMARK STA APPLICATIONS

Algorithm	**`vxm` Semiring**	**Reuse Pattern**	**Domain**
PageRank (pr)	Mul-Add	cross-iteration, producer-consumer	Graph Analytics
Kcore Decomposition (kcore)	Mul-Add		
Breadth First Search (bfs)	And-Or		
Single Source Shortest Path (sssp)	Min-Add		
Kmeans Initialisation (kpp)	Aril*-Add		Clustering
K-Nearest Neighbors (knn)	And-Or		
Label Propagation (label)	Mul-Add		Machine Learning
Graph Convolutional Neural network (gcn)	Mul-Add		
Generalized Minimal Residuals (gmres)	Mul-Add		Solver, HPC
Conjugate Gradient (cg)	Mul-Add	producer-consumer	
Biconjugate Gradients Stabilized (bgs)	Mul-Add		

*Assigns the **r**ight-hand input **if** the **l**eft-hand input evaluates true.

V. METHODOLOGY

This paper evaluates Sparsepipe through a custom-built simulator and a set of STA algorithms. This section highlights the simulated configurations and workloads.

A. *Simulation framework*

We developed an event-driven, cycle-accurate simulator to assess the performance of Sparsepipe. The simulator generates accurate cycle counts and other statistics, including memory accesses and bandwidth utilization, by iteratively modeling each pipeline stage of the OEI dataflow, sub-tensor fetching, on-chip buffer management (e.g., caching, eviction), and computation events within Sparsepipe. The memory subsystem of our simulator models a GDDR6X memory controller [37] and interacts with real input data. We evaluated the energy consumption of compute units and memory components using Cacti [39] and Accelergy [63] with the Aladdin [53] plug-in. We also scale the dynamic energy consumption based on factors reported in prior work [30] [37]. In this paper, we simulate an Sparsepipe architecture with 1024 PEs for each compute core, a 64 MB on-chip buffer, and on-device DRAM at 504GB/s bandwidth. Table II demonstrates detailed memory configuration of Sparsepipe, CPU, GPU. In addition to an iso-GPU Sparsepipe configuration, we also evaluated iso-CPU configuration of Sparsepipe with limited memory bandwidth, discussed in Section VI-B.

B. *Evaluated STA algorithms and systems*

We evaluate a total of 10 applications from ALP/GraphBLAS. Table III lists these applications, including their semiring operations, data reuse patterns, and application domains. Eight applications can leverage cross-iteration data reuse. We also include two applications that benefit solely from inter-operator data reuse to assess the performance of Sparsepipe on applications without OEI dataflows. 4 out of 10 applications employ graph analytics algorithms, while the remaining six power scientific computing and machine learning applications.

To thoroughly investigate the performance across all applications, we selected all nine representative sparse matrices, each with unique row/column size, sparsity, and non-zero distributions. Table I provides the list of these datasets.

We compare the performance of Sparsepipe with two baselines. (1) CPU baseline with large on-chip memory. We run the same workloads and datasets and collect performance counter numbers from a machine with AMD 5800X3D CPU, featuring a 96 MB 3D stacked V-cache and 128GB of dual-channel DDR4 main memory with measured memory bandwidth at 44 GB/s. (2) An idealized sparse accelerator (baseline) that utilizes the same compute and memory bandwidth as Sparsepipe, but does not exploit inter-operator data reuse. **This idealized sparse accelerator always has the throughput as its roofline**, representing the upper bound of prior sparse accelerators. Additionally, we chose `bfs, kcore, pr, sssp` to compare Sparsepipe's performance against an NVIDIA 4070 GPU with GDDR6X memory bandwidth at 504 GB/s.

VI. EXPERIMENTAL RESULT

A. *Performance over an idealized sparse accelerator*

Sparsepipe (iso-GPU) achieves up to 3.59 $\times$ in end-to-end latency over the baseline accelerator across all benchmark applications. For applications with OEI dataflow presented, Sparsepipe achieves a geometric mean speedup ranging from 1.21$\times$ to 2.62$\times$. Figure 14 detailed the speedup of end-to-end latency in each application.

Despite the baseline accelerator in Figure 14 always delivering performance at the roofline of STA operations, the baseline does not exploit inter-operator data reuse nor data reuse in OEI dataflow. Even for applications without OEI dataflow (i.e., cg and bgs), Sparsepipe can still exploit producer-consumer reuse and achieve the same level speedup as the baseline accelerator, ranging from 0.75$\times$ to 1.20$\times$.

Figure 15 explains Sparsepipe's performance by showing the memory bandwidth breakdown in each phase of execution using four representative workloads. Figure 15 (a) demonstrates a well-performing case of running `sssp` with the input matrix `bu`. With evenly distributed non-zero elements, all three Sparsepipe's OEI pipeline stages can maintain high memory bandwidth utilization, achieving a 2.9$\times$ speedup compared to the baseline accelerator. Additionally, Sparsepipe actively reclaims unused bandwidth to load CSR data during the IS

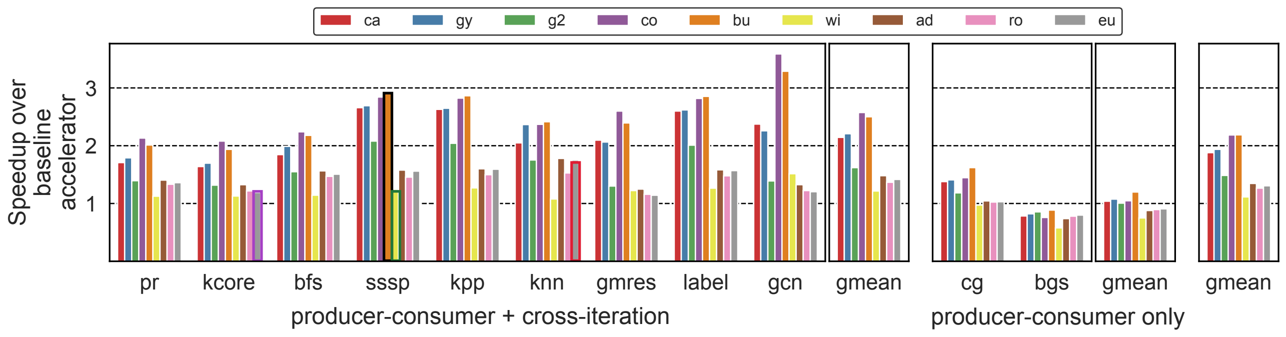


Fig. 14. Speedup of Sparsepipe over baseline accelerator. 4 highlighted bars with colored edges, `kcore-eu`, `sssp-bu`, `sssp-wi`, `knn-eu`, are representative benchmarks discussed in Figure 15 with corresponding highlighted color.

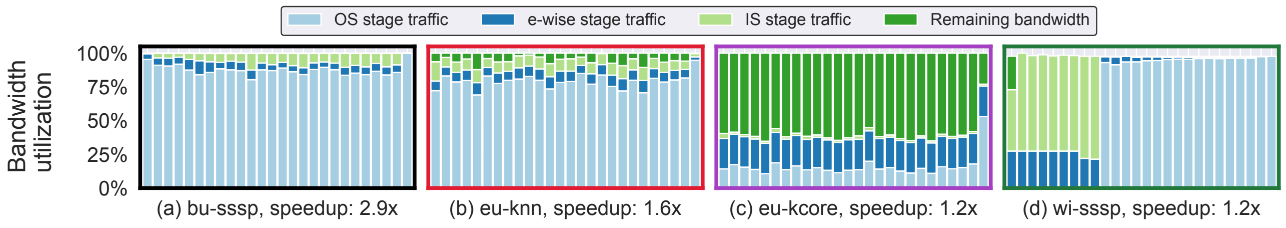


Fig. 15. Highlighted memory bandwidth utilization during Sparsepipe execution. Each bar represents the sampled utilization at every 4% interval of the simulation. The leftmost bar corresponds to the first sample step of the OEI dataflow, and the rightmost bar represents the final sampled step.

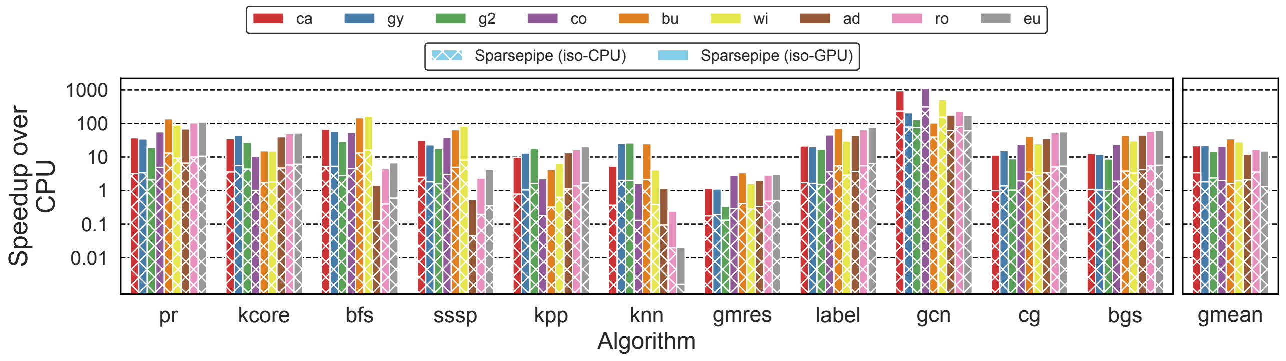


Fig. 16. Speedup of Sparsepipe over CPU implementation of STA algorithms.

stage, improving performance when processing our largest input matrix `eu`, as shown in Figure 15 (b).

Two cases could make Sparsepipe less effective. The first case is when the workload is more compute-intensive. Figure 15 (c) depicts Sparsepipe's utilization when executing the relative compute-intensive `kcore`, containing many `e-wise` operations. Using the largest input matrix `eu` requires processing all `e-wise` operations on very large vectors. While Sparsepipe makes every effort to fetch CSR data and utilize unused memory bandwidth, the limited buffer size and long compute latency result in low bandwidth utilization for most steps. Consequently, executing large input matrices yields fewer benefits from the OEI dataflow, resulting in only a 1.18× speedup.

The uneven distribution of non-zero values in sparse matrices is the other case that can make Sparsepipe less effective. Figure 15 (d) illustrates running `sssp` with the matrix `wi`, which exhibits a skewed distribution of non-zero values even after row reordering. Due to the minimal CSC data fetched in the early steps of the OEI dataflow, even with CSR data fetching to maintain high memory bandwidth, the limited on-chip buffer space and heavy CSC data traffic cause memory ping-ponging in later steps. This sparsity characteristic of the matrix `wi` leads to lower performance across all algorithms.

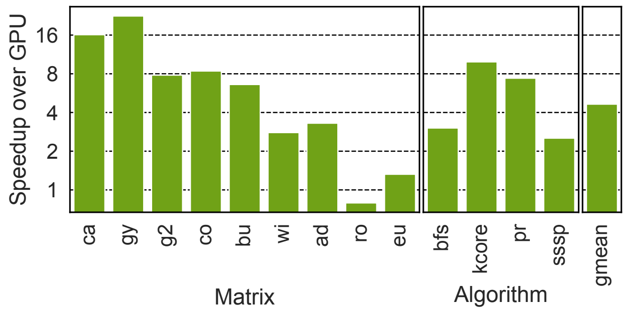


Fig. 17. Speedup of Sparsepipe over GPU implementation, each bar shows geometric mean of speedup.

B. Performance over CPU and GPU implementations

Figure 16 compares Sparsepipe with implementations using ALP/GraphBLAS running on a multicore CPU-based STA framework. The CPU implementations exploit non-blocking execution patterns for producer-consumer data reuse. In contrast, Sparsepipe benefits from both OEI dataflow and a higher memory bandwidth utilization. Excluding graph convolution neural networks (GCN), where Sparsepipe also benefits from dp4a-like instructions, Sparsepipe (iso-GPU) achieves up to a 164.84× speedup. Across all applications and matrices, Sparsepipe performs 12.20× to 35.14× better, surpassing the theoretical benefit ratio of 12.6× on system configuration with higher memory bandwidth. Sparsepipe (iso-CPU) with identical memory bandwidth can still bring 1.31× to 3.57× speedup compare to CPU framework, which demonstrates pure advantages of OEI dataflow by exploting cross-iteration data reuse.

While our focus primarily lies in presenting evaluation results compared to accelerator-based approaches, we also chose the four algorithms as a representative benchmark to study Sparsepipe's advantages over GPUs. We utilized the implementation from GraphBLAST [67] and Gunrock [60], GPU-based STA frameworks. As Figure 17 (a) demonstrates, Sparsepipe achieves a geometric mean of 4.65× speedup across all matrices.

C. Effectiveness in exploiting cross-iteration data reuse

To evaluate the effectiveness of Sparsepipe in exploiting cross-iteration and OEI dataflow data reuse opportunities, we modeled an oracle STA accelerator that assumes that all elements of the input sparse matrix are always ready when reuse opportunities across iterations present, fully exploiting all inter-operator data reuse opportunities irrespective of on-chip buffer size. Such an oracle accelerator presents the theoretical performance upper limit for STA applications. As shown in Figure 18, on average, Sparsepipe achieves 66.78% of the oracle accelerator's performance, utilizing only a 64MB on-chip buffer to process sparse matrices as large as 1.3GB (with 64-bit datatype).

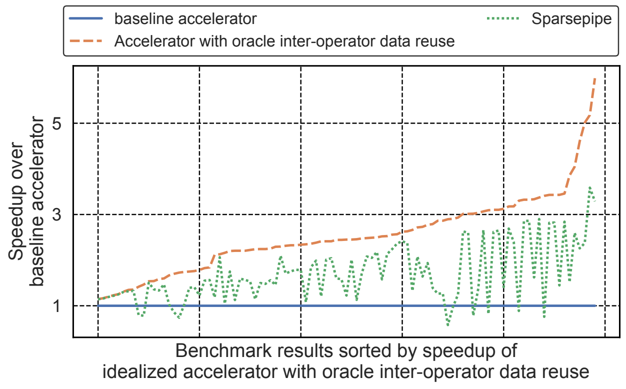


Fig. 18. The performance of Sparsepipe compared with an accelerator with perfect inter-operator reuse.

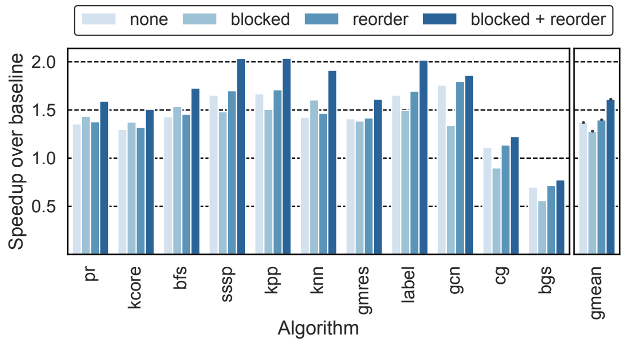


Fig. 19. Sensitivity study for the benefit of data optimization.

D. Impact of sparse tensor preprocessing

The accelerator architecture of Sparsepipe presents a skeleton of STA accelerators. Without any optimization, Figure 19 shows that Sparsepipe can still achieve 1.37× speedup over the baseline accelerator.

Encoding data using the blocked sparse format can help Sparsepipe to improve performance by up to 1.12×. Employing solely the optimal row reorder technique slightly boosts Sparsepipe's performance from 1.01× to 1.03×. With both optimizations increasing the locality of non-zero values, Sparsepipe can have a more efficient data access pattern during the OS-wise-IS dataflow, leading to 1.05× to 1.34× speedup from the Sparsepipe without data optimizations.

Using the blocked sparse format reduces the size of the dual sparse storage of input matrices. Figure 20 (a) illustrates the storage benefits of the blocked dual sparse format. Regardless of the reorder technique employed, blocked formats enhance the storage efficiency of dual storage, decreasing the storage requirements to 39.2% of the non-blocked dual-storage format.

E. Memory bandwidth utilization in Sparsepipe

Sparsepipe can effectively use available memory bandwidth for bandwidth-sensitive STA applications. Figure 21 shows that

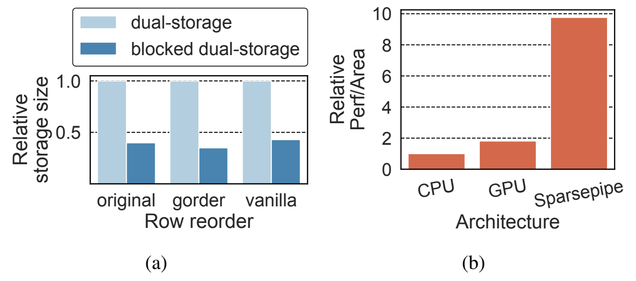


Fig. 20. (a) Storage improvement of blocked format. (b) Relative performance-per-area comparision normalized to CPU.

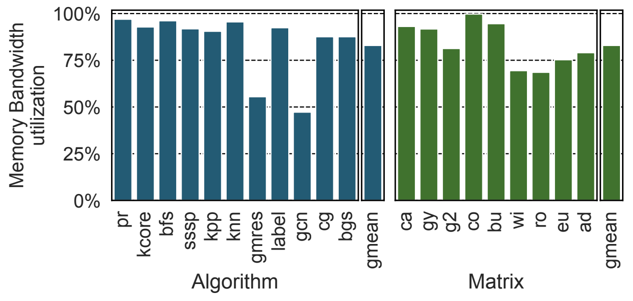


Fig. 21. Bandwidth utilization of Sparsepipe, geometric mean across algorithms and sparse matrices.

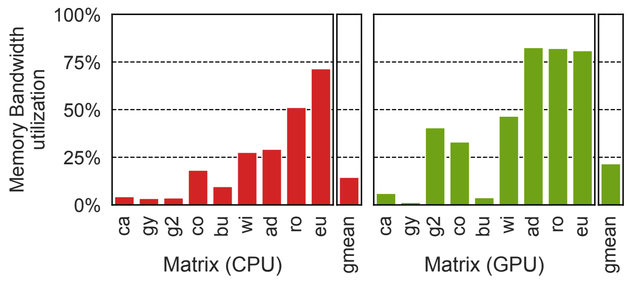


Fig. 22. Bandwidth utilization of STA application on CPU and GPU, geometric mean across sparse matrices.

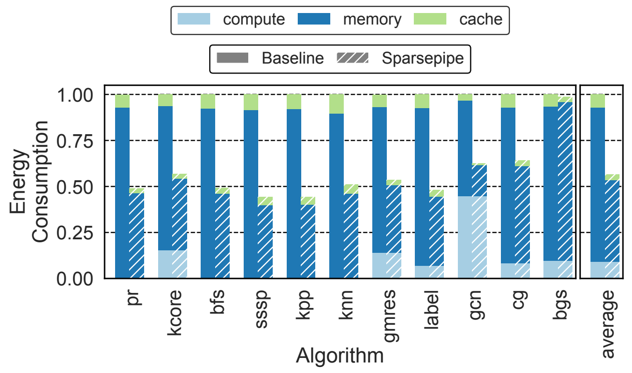


Fig. 23. Relative energy consumption of Sparsepipe separating compute, memory, and cache operations.

Sparsepipe maintains 82.93% memory bandwidth utilization. When considering only naturally memory-bound applications (excluding *gmres* and *gcn*), Sparsepipe efficiently utilized a geometric mean of 92.94% of system memory bandwidth. Figure 22 illustrates the memory bandwidth utilization of CPU and GPU frameworks, Sparsepipe demonstrates superior utilization of available bandwidth across all matrices. Despite multiple cache levels reducing memory traffic benefits CPU and GPU frameworks to exhibit lower bandwidth utilization for smaller input matrices, when processing large matrices, these frameworks do not leverage the benefits of OEI dataflow, requiring repetitive loading of the input matrix without data reuse. Consequently, this prevents the translation of high memory bandwidth utilization into actual performance gains.

F. Energy savings with Sparsepipe

The primary advantage of Sparsepipe lies in reducing the memory traffic of STA applications through cross-iteration data reuse. Since memory operations dominate the energy consumption for a significant portion of the selected STA applications, Sparsepipe achieves a significant energy saving. It reduces energy consumption by an average of 54.98% across all applications compared with the baseline accelerator. Specifically, Sparsepipe saves 50.32% of energy on memory operations and 39.45% on cache/on-chip buffer operations.

G. Area Efficiency of Sparsepipe

Under the identical hardware configurations as our performance simulation, the Sparsepipe architecture resembles the size of a consumer-grade, mid-tier GPU but with significantly better area efficiency than GPUs for STA. We estimated Sparsepipe's area from the RTL code and synthesized with the Synopsys Design Compiler using the 45 nm technology library. By scaling the design to the TSMC N5 process, Sparsepipe takes 253.95 mm^2, close to the 294 mm^2 die size of an RTX 4070 GPU. The on-chip buffer contributes 78% of the chip area.

Figure 20 (b) plots the average relative performance-per-area comparison across all architectures running STA applications. Sparsepipe occupies 253.95 mm^2 and achieves a 5.38× improvement in relative performance per area over the GPU, and a 9.84× improvement over the CPU.

VII. Related Work

In proposing the OEI dataflow and the microarchitecture, Sparsepipe exploits inter-operator reuse and complements prior work in 1) STA system software, 2) hardware accelerators for STA, and 3) techniques for inter-operator reuse.

Sparse tensor algebra framework and compiler. Our proposed OEI dataflow can optimize existing STA compiler frameworks that share the same front-end tensor programming models, including Sparse MLIR [11], TACO [15], [35], and COMET [29], [40], [58]. These compilers can generate software-only implementations that exploit OEI dataflow across CPU/GPUs. However, without Sparsepipe architecture, the software-only optimization will incur buffer and work management overhead. The proposed OEI dataflow also applies to Sparse Abstract Machine [26]'s Einsum-based programming model to generate more efficient hardware accelerators for STA.

Sparse dataflows and accelerators. Prior sparse tensor accelerators focused on intra-operator reuse for SpMSpM, including DRT [42], OuterSPACE [44], SpArch [72], MatRaptor [55], ExTensor [25], Gamma [70], HSS [64], and RM-STC [27]. Sparsepipe complements prior accelerators in exploring inter-operator reuse. Prior sparse accelerators also propose more generic, reconfigurable pipelines, including ALRESCHA [7], SMASH [31], Flexagon [38], SIGMA [49], Symphony [46], and MAERI [36] and efficient storage formats [48] to accelerate STA. While prior accelerators target intra-operator optimizations, Sparsepipe's OEI dataflow can help prior work exploiting inter-operator optimizations.

Exploiting inter-operator reuse. Sparsepipe targets STA instead of dense tensor algebra as prior work in DNNs intensively investigated exploiting producer-consumer data reuse in dense tensor algebra and proposed special dataflows to optimize for producer-consumer reuse. Examples include FLAT [32] for the multi-head attention kernel in Transformers, Fused-Layer CNN Accelerators [3] for CNNs and Pipelayer [54], TANGRAM [20], ARCHON [43], and Atomic dataflow [73], for producer-consumer reuse across spatial hardware units. As optimizing for fusion requires an extensive design space search, prior work explored tools and modeling techniques for fusion. Convfusion [59], DNNFuser [33], MultiFuse [13], and Stream [57] provide optimization tools to search for the best fusion decisions automatically. SET [12] and LoopTree [23] propose polyhedral abstraction to reason about fusion decisions more effectively.

Very recent research started the awareness of producer-consumer reuse for a specific STA domain, including ISOSceles [68] for sparse CNNs, GOGETA [21] for Conjugate Gradient (CG) in HPC, and [22] in exploiting pipeline dataflows for GNNs. Sparsepipe supports a wide range of STA applications beyond existing work and exploits cross-iteration reuse, a new type of data reuse. Zhuang and Casas [74] presents iteration-fusing techniques for pipelined CG, enabling computation overlapping and improving concurrency. Nonetheless, without Sparsepipe's explicit buffer control and the OEI dataflow, prior work cannot capture all data reuse opportunities to address the most critical bottleneck in STA.

VIII. Conclusion

As computer architects seek performance scaling with new process technologies through hardware accelerators and software developers demand efficient and more descriptive programming frameworks, the evolution of being "domain-specific" is clearly the future for architecture and software design. Compared to conventional programming paradigms and hardware/software interfaces at the scalar/vector instructions level, domain-specific languages/interfaces make the high-level architecture of applications more visible to compilers and hardware. Therefore, the rapid adoption of domain-specific designs opens up new opportunities for performance optimizations that prior work cannot easily exploit.

This paper identifies and exploits two inter-operator reuse opportunities that domain-specific languages polish in sparse dataflow graphs: consumer-producer reuse and cross-iteration reuse, to reduce data movement, maximize data reuse, and accelerate STA applications. In contrast, prior work exploits solely intra-operator reuse due to the limitation of conventional paradigms. We propose the OEI dataflow to capture inter-operator reuse and the Sparsepipe architecture to support the OEI dataflow with limited buffer space. The evaluation shows Sparsepipe's superior performance, energy, and area efficiency over CPUs/GPUs and state-of-the-art accelerators.

Beyond our specific Sparsepipe implementation, the proposed OEI dataflow opens up other possibilities to improve STA applications. For example, how to implement the OEI dataflow on general-purpose hardware (e.g., GPGPU), and design the extra hardware support to facilitate the buffer management and synchronization across stages? How can we leverage the modern compiler framework [4] for tensor applications to automatically find applications with cross-iteration reuse and accelerate them with the OEI dataflow? Can the same concept of Sparsepipe apply to other application domains besides STA? We leave these exciting new avenues to future work.

Acknowledgments

The authors would like to thank the anonymous reviewers for their helpful comments. This work was sponsored by two National Science Foundation (NSF) awards, CNS-2007124 and CNS-2231877. This work was also supported by new faculty start-up funds from University of California, Riverside.

References

[1] BLAS (Basic Linear Algebra Subprograms). http://www.netlib.org/blas/, 2004.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[3] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. Fused-layer CNN Accelerators. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[4] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, et al. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2024.

[5] Arvind and David E Culler. Dataflow architectures. *Annual review of computer science*, pages 225–253, 1986.

[6] Arvind and Rishiyur S Nikhil. Executing a program on the mit tagged-token dataflow architecture. *IEEE Transactions on Computers*, 39(3):300–318, 1990.

[7] Bahar Asgari, Ramyad Hadidi, Tushar Krishna, Hyesoon Kim, and Sudhakar Yalamanchili. ALRESCHA: A Lightweight Reconfigurable Sparse-Computation Accelerator. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[8] Vignesh Balaji, Neal Crago, Aamer Jaleel, and Brandon Lucia. P-opt: Practical Optimal Cache Replacement for Graph Analytics. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021.

[9] Vignesh Balaji, Neal C Crago, Aamer Jaleel, and Stephen W Keckler. Community-based Matrix Reordering for Sparse Linear Algebra Optimization. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2023.

[10] Pieter Bellens, Josep M Perez, Rosa M Badia, and Jesus Labarta. Cellss: a programming model for the cell be architecture. In *2006 ACM/IEEE Conference on Supercomputing (CS)*, 2006.

[11] Aart Bik, Penporn Koanantakool, Tatiana Shpeisman, Nicolas Vasilache, Bixia Zheng, and Fredrik Kjolstad. Compiler Support for Sparse Tensor Computations in MLIR. *ACM Trans. Archit. Code Optim.*, 19(4), 2022.

[12] Jingwei Cai, Yuchen Wei, Zuotong Wu, Sen Peng, and Kaisheng Ma. Inter-layer Scheduling Space Definition and Exploration for Tiled Accelerators. In *the 50th Annual International Symposium on Computer Architecture (ISCA)*, 2023.

[13] Chia-Wei Chang, Jing-Jia Liou, Chih-Tsun Huang, Wei-Chung Hsu, and Juin-Ming Lu. MultiFuse: Efficient Cross Layer Fusion for DNN Accelerators with Multi-level Memory Hierarchy. In *2023 IEEE 41st International Conference on Computer Design (ICCD)*, 2023.

[14] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *the 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[15] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. Format Abstraction for Sparse Tensor Algebra Compilers. *Proc. ACM Program. Lang.*, 2(OOPSLA), 2018.

[16] Timothy A Davis. Algorithm 1000: SuiteSparse: GraphBLAS: Graph algorithms in the language of sparse linear algebra. *ACM Transactions on Mathematical Software (TOMS)*, 45(4):1–25, 2019.

[17] Jack B Dennis. Data flow supercomputers. *Computer*, 13(11):48–56, 1980.

[18] Yoav Etsion, Felipe Cabarcas, Alejandro Rico, Alex Ramirez, Rosa M Badia, Eduard Ayguade, Jesus Labarta, and Mateo Valero. Task superscalar: An out-of-order task pipeline. In *the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2010.

[19] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, and Tianqi Chen. TensorIR: An Abstraction for Automatic Tensorized Program Optimization. In *the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[20] Mingyu Gao, Xuan Yang, Jing Pu, Mark Horowitz, and Christos Kozyrakis. Tangram: Optimized coarse-grained dataflow for scalable nn accelerators. In *the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[21] Raveesh Garg, Michael Pellauer, Sivasankaran Rajamanickam, and Tushar Krishna. Exploiting Inter-Operation Data Reuse in Scientific Applications using GOGETA. *arXiv preprint arXiv:2303.11499*, 2023.

[22] Raveesh Garg, Eric Qin, Francisco Muñoz-Matrínez, Robert Guirado, Akshay Jain, Sergi Abadal, José L. Abellán, Manuel E. Acacio, Eduard Alarcón, Sivasankaran Rajamanickam, and Tushar Krishna. Understanding the Design-Space of Sparse/Dense Multiphase GNN dataflows on Spatial Accelerators. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2022.

[23] Michael Gilbert, Yannan Nellie Wu, Angshuman Parashar, Vivienne Sze, and Joel S. Emer. LoopTree: Enabling Exploration of Fused-layer Dataflow Accelerators. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2023.

[24] Gagan Gupta and Gurindar S Sohi. Dataflow execution of sequential imperative programs on multicore architectures. In *the 44th annual IEEE/ACM international symposium on Microarchitecture (ISCA)*, 2011.

[25] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W. Fletcher. ExTensor: An accelerator for sparse tensor algebra. In *the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019.

[26] Olivia Hsu, Maxwell Strange, Ritvik Sharma, Jaeyeon Won, Kunle Olukotun, Joel S. Emer, Mark A. Horowitz, and Fredrik Kjølstad. The Sparse Abstract Machine. In *the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS)*, 2023.

[27] Guyue Huang, Zhengyang Wang, Po-An Tsai, Chen Zhang, Yufei Ding, and Yuan Xie. Rm-stc: Row-merge dataflow inspired gpu sparse tensor core for energy-efficient sparse acceleration. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2023.

[28] Sagar Imambi, Kolla Bhanu Prakash, and GR Kanagachidambaresan. PyTorch. *Programming with TensorFlow: Solution for Edge Computing Applications*, pages 87–104, 2021.

[29] Geonhwa Jeong, Gokcen Kestor, Prasanth Chatarasi, Angshuman Parashar, Po-An Tsai, Sivasankaran Rajamanickam, Roberto Gioiosa, and Tushar Krishna. Union: A unified hw-sw co-design ecosystem in mlir for evaluating tensor operations on spatial accelerators. In *the 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2021.

[30] Norman P. Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B. Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, Thomas Norrie, Nishant Patil, Sushma Prasad, Cliff Young, Zongwei Zhou, and David Patterson. Ten Lessons From Three Generations Shaped Google's TPUv4i : Industrial Product. In *the 48th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, 2021.

[31] Konstantinos Kanellopoulos, Nandita Vijaykumar, Christina Giannoula, Roknoddin Azizi, Skanda Koppula, Nika Mansouri Ghiasi, Taha Shahroodi, Juan Gomez Luna, and Onur Mutlu. SMASH: Co-designing Software Compression and Hardware-Accelerated Indexing for Efficient Sparse Matrix Operations. In *the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019.

[32] Sheng-Chun Kao, Suvinay Subramanian, Gaurav Agrawal, Amir Yazdanbakhsh, and Tushar Krishna. FLAT: An Optimized Dataflow for Mitigating Attention Bottlenecks. In *28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[33] Kao, Sheng-Chun and Huang, Xiaoyu and Krishna, Tushar. DNNFuser: Generative pre-trained transformer as a generalized mapper for layer fusion in dnn accelerators. *arXiv preprint arXiv:2201.11218*, 2022.

[34] Jeremy Kepner, Peter Aaltonen, David Bader, Aydin Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, et al. Mathematical Foundations of the GraphBLAS. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, 2016.

[35] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The Tensor Algebra Compiler. *Proc. ACM Program. Lang.*, (OOPSLA), 2017.

[36] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Reconfigurable Interconnects. In *the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[37] Micron. Gddr6x memory. https://www.micron.com/products/memory/hbm/gddr6x. [Accessed 22-06-2024].

[38] Francisco Muñoz Martínez, Raveesh Garg, Michael Pellauer, José L. Abellán, Manuel E. Acacio, and Tushar Krishna. Flexagon: A Multi-dataflow Sparse-Sparse Matrix Multiplication Accelerator for Efficient DNN Processing. In *the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[39] Naveen Muralimanohar, Rajeev Balasubramonian, and Norm Jouppi. Optimizing NUCA Organizations and Wiring alternatives for Large Caches with CACTI 6.0. In *the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2007.

[40] Erdal Mutlu, Ruiqin Tian, Bin Ren, Sriram Krishnamoorthy, Roberto Gioiosa, Jacques Pienaar, and Gokcen Kestor. Comet: A Domain-specific Compilation of High-performance Computational Chemistry. In *International Workshop on Languages and Compilers for Parallel Computing*, 2020.

[41] Mike O'Connor, Donghyuk Lee, Niladrish Chatterjee, Michael B. Sullivan, and Stephen W. Keckler. Saving pam4 bus energy with smores: Sparse multi-level opportunistic restricted encodings. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022.

[42] Toluwanimi O. Odemuyiwa, Hadi Asghari-Moghaddam, Michael Pellauer, Kartik Hegde, Po-An Tsai, Neal C. Crago, Aamer Jaleel, John D. Owens, Edgar Solomonik, Joel S. Emer, and Christopher W. Fletcher. Accelerating Sparse Data Orchestration via Dynamic Reflexive Tiling. In *the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[43] MohammadHossein Olyaiy, Christopher Ng, and Mieszko Lis. Accelerating DNNs Inference with Predictive Layer Fusion. In *ACM International Conference on Supercomputing (ICS)*, 2021.

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

[45] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.

[46] Michael Pellauer, Jason Clemons, Vignesh Balaji, Neal Crago, Aamer Jaleel, Donghyuk Lee, Mike O'Connor, Angshuman Parashar, Sean Treichler, Po-An Tsai, Stephen W. Keckler, and Joel S. Emer. Symphony: Orchestrating Sparse and Dense Tensors with Hierarchical Heterogeneous Processing. *ACM Trans. Comput. Syst.*, 41(1–4), dec 2023.

[47] Michael Pellauer, Yakun Sophia Shao, Jason Clemons, Neal Crago, Kartik Hegde, Rangharajan Venkatesan, Stephen W Keckler, Christopher W Fletcher, and Joel Emer. Buffets: An Efficient and Composable Storage Idiom for Explicit Decoupled Data Orchestration. In *the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[48] Eric Qin, Geonhwa Jeong, William Won, Sheng-Chun Kao, Hyoukjun Kwon, Sudarshan Srinivasan, Dipankar Das, Gordon E. Moon, Sivasankaran Rajamanickam, and Tushar Krishna. Extending Sparse Tensor Accelerators to Support Multiple Compression Formats. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021.

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

[50] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. *Acm Sigplan Notices*, 48(6), 2013.

[51] Amit Sabne. XLA : Compiling Machine Learning for Peak Performance, 2020.

[52] Shuichi Sakai, Y Yamaguchi, Kei Hiraki, Yuetsu Kodama, and Toshitsugu Yuba. An architecture of a dataflow single chip processor. *ACM SIGARCH Computer Architecture News*, 17(3):46–53, 1989.

[53] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *the 41st International Symposium on Computer Architecture (ISCA)*, 2014.

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

[55] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020.

[56] Steven Swanson, Ken Michelson, Andrew Schwerin, and Mark Oskin.

[58] Ruiqin Tian, Luanzheng Guo, Jiajia Li, Bin Ren, and Gokcen Kestor. A High Performance Sparse Tensor Algebra Compiler in MLIR. In *2021 IEEE/ACM 7th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, 2021.

Wavescalar. In *the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2003.

[57] A. Symons, L. Mei, S. Colleman, P. Houshmand, S. Karl, and M. Verhelst. Stream: A Modeling Framework for Fine-grained Layer Fusion on Multi-core DNN Accelerators. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2023.

[59] Luc Waeijen, Savvas Sioutas, Maurice Peemen, Menno Lindwer, and Henk Corporaal. ConvFusion: A Model for Layer Fusion in Convolutional Neural Networks. *IEEE Access*, 9, 2021.

[60] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. Gunrock: A high-performance graph processing library on the gpu. In *the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2016.

[61] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. Speedup Graph Processing by Graph Ordering. In *2016 International Conference on Management of Data (SIGMOD)*, 2016.

[62] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[63] Yannan Nellie Wu, Joel S Emer, and Vivienne Sze. Accelergy: An Architecture-level Energy Estimation Methodology for Accelerator Designs. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2019.

[64] Yannan Nellie Wu, Po-An Tsai, Saurav Muralidharan, Angshuman Parashar, Vivienne Sze, and Joel Emer. Highlight: Efficient and flexible dnn acceleration with hierarchical structured sparsity. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2023.

[65] Yannan Nellie Wu, Po-An Tsai, Angshuman Parashar, Vivienne Sze, and Joel S Emer. Sparseloop: An analytical, energy-focused design space exploration methodology for sparse tensor accelerators. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2021.

[66] Yannan Nellie Wu, Po-An Tsai, Angshuman Parashar, Vivienne Sze, and Joel S. Emer. Sparseloop: An Analytical Approach To Sparse Tensor Accelerator Modeling. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022.

[67] Carl Yang, Aydin Buluç, and John D. Owens. GraphBLAST: A high-performance linear algebra-based graph framework on the GPU. *CoRR*, abs/1908.01407, 2019.

[68] Yifan Yang, Joel Emer, and Daniel Sanchez. ISOSceles: Accelerating Sparse CNNs through Inter-Layer Pipelining. In *the 29th international symposium on High Performance Computer Architecture (HPCA)*, 2023.

[69] A. N. Yzelman, D. Di Nardo, J. M. Nash, and W. J. Suijlen. A C++ GraphBLAS: specification, implementation, parallelisation, and evaluation. Preprint, 2020.

[70] Guowei Zhang, Nithya Attaluri, Joel S. Emer, and Daniel Sanchez. Gamma: Leveraging Gustavson's Algorithm to Accelerate Sparse Matrix Multiplication. In *the 26th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[71] Yunan Zhang, Po-An Tsai, and Hung-Wei Tseng. SIMD2: A Generalized Matrix Instruction Set for Accelerating Tensor Computation beyond GEMM. In *the 49th Annual International Symposium on Computer Architecture (ISCA)*, 2022.

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

[73] Shixuan Zheng, Xianjue Zhang, Leibo Liu, Shaojun Wei, and Shouyi Yin. Atomic Dataflow based Graph-Level Workload Orchestration for Scalable DNN Accelerators. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022.

[74] Sicong Zhuang and Marc Casas. Iteration-fusing conjugate gradient. In *2017 International Conference on Supercomputing (ICS)*, 2017.