MetaStrider: Architectures for Scalable Memory-centric Reduction of Sparse Data Streams

SRISESHAN SRIKANTH, ANIRUDH JAIN, JOSEPH M. LENNON, and THOMAS M. CONTE, Georgia Institute of Technology
ERIK DEBENEDICTIS and JEANINE COOK, Sandia National Laboratories

Reduction is an operation performed on the values of two or more key-value pairs that share the same key. Reduction of sparse data streams finds application in a wide variety of domains such as data and graph analytics, cybersecurity, machine learning, and HPC applications. However, these applications exhibit low locality of reference, rendering traditional architectures and data representations inefficient. This article presents MetaStrider, a significant algorithmic and architectural enhancement to the state-of-the-art, SuperStrider. Furthermore, these enhancements enable a variety of parallel, memory-centric architectures that we propose, resulting in demonstrated performance that scales near-linearly with available memory-level parallelism.

CCS Concepts: • **Hardware** → **Memory and dense storage**; • **Computer systems organization** → *Special purpose systems*; • **Theory of computation** → *Shared memory algorithms*;

Additional Key Words and Phrases: Memory-centric architectures, DRAM, sparse

ACM Reference format:
Sriseshan Srikanth, Anirudh Jain, Joseph M. Lennon, Thomas M. Conte, Erik Debenedictis, and Jeanine Cook. 2019. MetaStrider: Architectures for Scalable Memory-centric Reduction of Sparse Data Streams. *ACM Trans. Archit. Code Optim.* 16, 4, Article 35 (October 2019), 26 pages.
https://doi.org/10.1145/3355396

1 INTRODUCTION

The utility of sparse data is well known to the community due to its prevalence in machine-learning algorithms (e.g., support vector machines (SVM) [58], text analytics [55]), in scientific-computing applications [29] (e.g., Schur complement method in hybrid linear solvers [78], algebraic multigrid (AMG) methods [12], finite element analysis [39], molecular dynamics [41], many-atom systems [46]), in graph analytics (e.g., breadth-first-search (BFS) [34], PageRank [16], minimum spanning tree (MST), single source shortest path (SSSP) [19], matching [62], contraction [18],

New article, not an extension of a conference paper.
Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the US Department of Energy's National Nuclear Security Administration under Contract No. DE-NA-0003525. The views expressed in the article do not necessarily represent the views of the U.S. Department of Energy or the United States Government.
Authors' addresses: S. Srikanth, A. Jain, and J. M. Lennon, Georgia Institute of Technology, KACB 2337, 266 Ferst Dr, Atlanta, GA 30332; emails: {seshan, anirudh.j, jlennon7}@gatech.edu; T. M. Conte, Georgia Institute of Technology, KACB 2334, 266 Ferst Dr, Atlanta, GA 30332; email: conte@gatech.edu; E. DeBenedictis and J. Cook, Sandia National Laboratories, P.O. Box 5800, MS 1319, Albuquerque, NM 87185; {epdeben, jeacook}@sandia.gov.

1544-3566/2019/10-ART35
https://doi.org/10.1145/3355396

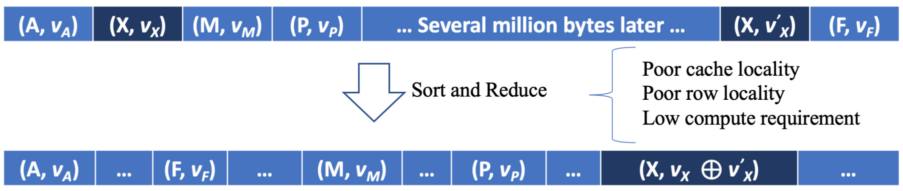


Fig. 1. Reduction of key-value pairs in a sparse data stream.

reachability [65], clustering [71], triangle counting [10], and cycle detection [79]), and in cyber-security [8, 45]. Furthermore, novel deep-learning algorithms that are being proposed employ network pruning and effectively "sparsify" [9, 37, 38, 51, 53, 77] them to reduce memory footprint as well as required computation.

A significant and important operation for sparse data streams occurs during associative array reduction (Figure 1). This is a set of associative operations performed on the values of two or more key-value pairs that share the same key. General terms for *key-value pairs* are typically *kv-pairs*, *records*, *index-nonzeros* (for sparse matrices), or *tuples*, where the key, value, and reduction operator are application dependent.

We focus on important applications of reduction with a variety of workloads (Table 1) that span several domains:

- General sparse matrix-matrix multiplication (**SpGEMM**) is used in various domains mentioned above [57]. Sparse reduction occurs during partial products accumulation.
- **Firehose** [8] is a collection of open-source stream processing benchmarks designed to represent cyber-security applications. It models a variety of sparse network event distributions and their subsequent (soft) real-time analysis, *requiring incremental updates to sparse data.* Sparse reduction occurs in the form of counter increments of a given IPv6 address.

Low locality of reference, low degree of reuse, and low compute-to-communicate ratios are intrinsic characteristics of sparse data streams, rendering even large caches ineffective. Therefore, main memory (DRAM) access latency and redundant data movement through the memory hierarchy are fundamental bottlenecks.

Numerous proposals to improve the status quo in sparse data research have been made by parallelizing the application to maximize use of memory level parallelism (MLP) and parallel compute resources (CPU [7, 27, 40, 61, 68, 70, 75] and GPU [22, 27, 35, 50, 54, 59]). With intelligent and careful design, these techniques are by and large successful in hiding memory latency, although *they come at the cost of increased data movement, a significant fraction of which is often redundant.*

For instance, as representative examples, consider three recent SpGEMM implementations: a CPU-GPU performance portable framework [27], a GPU-only implementation [50], and an Intel KNL-based analysis [57]. It is known that the outer-product algorithm (OP) for SpGEMM is more efficient [60] than Gustavson's original row-wise algorithm (RW) [36], because RW does significantly less useful work per unit input data read compared to OP. Yet, all of these works (as well as a majority of the literature [57]) employ RW instead of OP simply because caches cannot hold the large number of temporaries (partial products) that OP generates. Such favoring of redundant gather for optimized scatter is understandable as their focus is to maximize cache usage. However, these are not necessarily tailored towards DRAM operational inefficiencies such as row activation (**ACT**) and precharge (**PRE**) overheads.

Proposals in the hardware accelerator space recognize that even large caches do not capture locality in such sparse data streams, and are often designed in a DRAM-centric manner. However, these either specifically target SpGEMM [49, 60, 69] or are not suited for handling incremental/

streaming updates to sparse data (GraFBoost [43]), and therefore are incapable of accelerating applications such as Firehose. An exception to this is SuperStrider [24, 42, 72], although it still suffers from significant DRAM overheads despite being a memory-centric design.

SuperStrider (Section 2.3) employs a vectorized binary search tree in DRAM and a bitonic merge network near memory. To maximize useful bytes read per ACT (CAS per ACT), a DRAM row is treated as a first-class citizen and is the fundamental unit of operation. The row logically forms a node in the tree, where each node contains a vector of records sorted by key, as well as a pivot (key) and a pair of child pointers.

The tree is built in two phases. Phase 1 (`Addvec()`) sends records across a dynamically determined path of the tree from the root node to a leaf node while performing any possible reductions along that path. As a result, `Addvec()` results in possibly duplicate keys in different paths of the tree, that are yet to be reduced. Phase 2 (`Normalize()`) performs two depth-first-searches (DFS) on the entire tree to reduce and de-duplicate such records.

`Addvec()` is repeated multiple times to incrementally build the tree as new input that needs to be reduced is streamed in. `Normalize()` is performed once, at the end. However, our analysis indicates that the energy overhead of `Normalize()` is equivalent to re-running `Addvec()` on the entire input stream again, a significantly excessive overhead.

Contributions

This article proposes novel algorithms and scalable architectures, known as MetaStrider, to perform energy-efficient, memory-centric acceleration of sparse reductions. When compared with state-of-the-art hardware accelerators for reduction, namely, SuperStrider [24, 42, 72] and GraFBoost [43], MetaStrider realizes un-core energy savings of 5.3× and 14.9×, while also improving performance by 11% and 30% on average across all workloads, respectively (Section 7.2). The key contributions of MetaStrider are three-fold:

Algorithmic improvements. MetaStrider uses expressive metadata (Section 3) to significantly enhance SuperStrider operation, especially its `Normalize()` functionality. When combined with metadata decoupling described below, a 10× reduction in DRAM ACT is seen in this context for a 5.2× reduction in `Normalize()` runtime (Section 7.3).

Architectural improvements. First, MetaStrider decouples metadata from the key-value store (DRAM). Therefore, the latter is accessed only when absolutely necessary. This also enables tree density improving techniques such as tree balancing (Section 3) that would otherwise have been significantly more expensive in SuperStrider, where no such decoupling is done. As a result, MetaStrider's performance is *within 8%* of that of an idealized accelerator that performs reduction (compute and memory) in zero time.

Second, a light-weight merger is designed that performs merging, reduction and de-duplication in O(K) time rather than O(K lg K) time (SuperStrider's bitonic network). Furthermore, our merger is designed to work on natively incremental data bursts (such as 64-/128-bit DRAM bursts) to fundamentally maximize overlap of logic and memory (Section 4).

Leveraging MLP. Once the fundamental impact of DRAM latency on performance has been significantly reduced, scalable algorithms and architectures are co-designed to further hide latency via parallelism (Section 5). MetaStrider is able to achieve high bandwidth utilization and its performance scales near-linearly.

Paper outline. Related work is summarized in Sections 1 and 2. Section 2 also presents characterizations of the two benchmark sets used in this article, and details state-of-the-art software and hardware approaches for sparse reduction. Section 3 introduces enhancements MetaStrider makes to the metadata of SuperStrider. Section 4 describes the underlying architecture and formalizes system integration via an API. Section 5 presents a systematic design space exploration for extracting MLP efficiently. Experimental evaluation is presented in Sections 6 and 7. We conclude in Section 8.

2 BACKGROUND AND RELATED WORK

2.1 Workload Characterization

SpGEMM. As described in Section 1, $C = A \times B$ can be performed using row-wise (RW) or outer-product (OP). Both of these algorithms generate a *sparse set of partial products* that need to be reduced (summed / accumulated, in this case).

With RW, C is computed one row at a time by fetching one row of A and all rows of B in a row-wise manner such that $C_{i,:} = \sum_k A_{i,k} \times B_{k,:}$. As a result, all partial products generated in this manner prior to reduction (accumulation) are confined to a single output row of C.

With OP, the kth column of A is multiplied by the kth row of B to generate the kth partial product matrix. Reduction of all such partial product matrices results in the final output matrix such that $C = \sum_k C^k = \sum_k A_{:,k} \times B_{k,:}$.

Observe that OP fetches the input exactly once but generates a large space of temporaries (partial products), whereas RW fetches the input several times although it enables caching of temporaries. It can be theoretically shown that OP fetches a factor of N fewer inputs than RW assuming uniform random matrices. When these are simulated with real input considered in this article, 30,000× fewer input accesses are seen with OP. This insight is also demonstrated by [60]. Given its higher ops per byte fetched, OP is chosen as the choice of implementation. *MetaStrider is designed to fundamentally be capable of handling the large number of temporaries that OP generates.*

Firehose models soft real-time events of cyber-security applications. It consists of three stream generator algorithms as its front-end, namely power law, active set and two-level, to simulate a variety of network traffic. As with SpGEMM, each generator produces a stream of key-value pairs. Here, key is a proxy for an IPv6 address and value is a counter (+bias bits) associated with each key. The goal of the benchmark suite is to track the number of identical keys received, and whenever the number of identical keys exceeds a certain threshold, the key is flagged (and the bias bits associated with such keys are then tested against the ground-truth bias bits included in the input).

As can be expected of anomalous network events, these three generators are designed to be unpredictable and to have a varied dynamic range. Therefore, the incoming key-value stream is sparse and anomaly detection can be performed by applying a reduction (increment) operator on elements with identical keys.

The reduction operator for SpGEMM is integer/floating-point addition (i.e., accumulation), while Firehose requires saturating counter-increment and comparison operations. Note that other applications that can be implemented as associative array reduction kernels may use other operators as the reducer (known as collision function) [21, 31]. For example, the Bellman-Ford algorithm [14] for single source shortest path (SSSP) can benefit from *min* as a collision function. MetaStrider is designed to accelerate the reduction of any of these sparse data streams. For the remainder of this article, *32-bit addition is used as the default reducer* as an example.

Table 1 presents key features of the workloads used in this article. Matrices for SpGEMM are obtained from the SuiteSparse collection [23], with the matrix selection being guided by prior research in the field [22, 35, 60, 70] to span a wide variety of domains. Inputs for Firehose are generated to insert a similar number of non-zeros into the sparse stream. Note that the primary focus of this article is on addressing the fundamental memory latency problem (in a manner that also scales gracefully). Therefore, the initial configuration is constrained to use a single memory channel, which is fixed at 128MB for commodity high bandwidth memory (HBM), and the input size is chosen accordingly to fit in memory. For the software baselines (below), an LLC (last level cache) size of 4MB is used, which is a relatively massive cache given the small amount of memory.

Table 1. Sparse Input and Application Properties

Application	Input	Domain	nnz_{stream}	% repeated keys	Amdahl's fraction	LLC Miss Ratio of kernel
SpGEMM	2cubes_sphere (2cu)	Electromagnetics	6.3M	50	0.8	0.24
	belgium_osm (bel)	Road Networks EU	1.5M	<1	0.5	0.49
	ca-CondMat (caC)	Undirected Graph	700K	28	0.9	0.36
	mario002 (mar)	2D/3D Problem	2.7M	22	0.8	0.58
	netherlands_osm (net)	Road Networks EU	2M	<1	0.5	0.57
	p2p-Gnutella31 (p2p)	Directed Graph	500K	<1	0.7	0.45
	patents_main (pat)	Wt. Directed Graph	2.6M	12	0.8	0.56
	roadNet-CA (roa)	Road Networks US	3M	1.5	0.6	0.55
Firehose	Power Law (pl)	Cyber Security	500K/1M/5M	88/91/96	n/a	0.01
	Active Set (as)			5/8/17	n/a	0.33/0.41/0.54
	Two Level (tl)			28/30/31	n/a	0.29/0.38/0.53

nnz_{stream} denotes the number of non-de-duplicated kv-pairs in the input stream, out of which *%repeated* of the pairs have identical keys whose values have to be reduced. The Amdahl's fraction is calculated using a scalar binary tree-based reducer (described in Section 2.2), and indicates a high relative importance from accelerating reduction (compute operations and associated memory reads/writes). Finally, all workloads, with the exception of *pl*, exhibit low locality of reference in their streams. The *pl* stream has a high degree of reuse owing to the fact that it was generated using a relatively low range of static keys (100,000) by default. Therefore, we report MetaStrider results for *pl*, but *omit them from averages.*
*The Amdahl's fraction for Firehose is not shown because the kernel is the application.

MetaStrider exhibits over an order of magnitude improvement in performance for all these workloads when compared to the baseline (with the exception of *pl*, for which traditional caching is clearly more suited—see Table 1).

2.2 Software Reducers

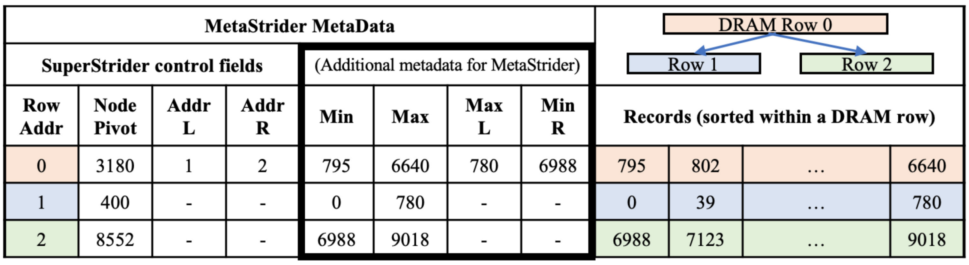
Given that MetaStrider is a vectorized binary search tree, a **scalar binary tree** (where the keys in the sparse stream form the nodes of the tree) forms a natural baseline to compare against. `std::map` is a CPU-only associative container that contains unique key-value pairs, and is implemented as a balanced red-black binary tree [1], allowing for lg N insertion complexity, where N is the number of unique key-value pairs inserted.

A similar possibility is std::unordered_map. However, its performance with real workloads considered in this article is similar to, or worse than std::map. This is attributed to an inefficient default hash function with long chains. Instead, a non-STL reducer built using the kokkos framework is a more suitable hashmap, and is in-fact the *state-of-the-art* software reducer for SpGEMM.

The **Kokkos HashMap Accumulator** [27] is representative of a CPU-GPU sparse reducer, consisting of four parallel arrays where the hashmap is stored in a linked list structure. The *Ids* and *Values* arrays store the keys and values, respectively. The *Begins* array holds the beginning indices of the linked list corresponding to the hash values, and the *Nexts* array holds the indices of the next elements within the list. While this approach is significantly better than the baseline, it is not explicitly designed to reduce DRAM ACT/PRE overheads.

2.3 Hardware Reducers

SuperStrider [72] introduces a novel, explicitly managed DRAM layout for sparse data to improve row locality, where each DRAM row forms a node in a vector binary tree. To manage the tree,


DRAM Row 0
Row 1
Row 2

MetaStrider MetaData								Records (sorted within a DRAM row)			
SuperStrider control fields				(Additional metadata for MetaStrider)							
Row Addr	Node Pivot	Addr L	Addr R	Min	Max	Max L	Min R				
0	3180	1	2	795	6640	780	6988	795	802	…	6640
1	400	-	-	0	780	-	-	0	39	…	780
2	8552	-	-	6988	9018	-	-	6988	7123	…	9018

Fig. 2. Data layout of records and control fields/metadata in a DRAM-centric vectorized binary tree. The Metadata fields of MetaStrider are more expressive than the control fields of SuperStrider and are decoupled from the kv-rows. This results in direct and indirect benefits as described qualitatively in Section 3 and quantitatively in Section 7. The values are omitted (in the figure) from the kv-rows for brevity.

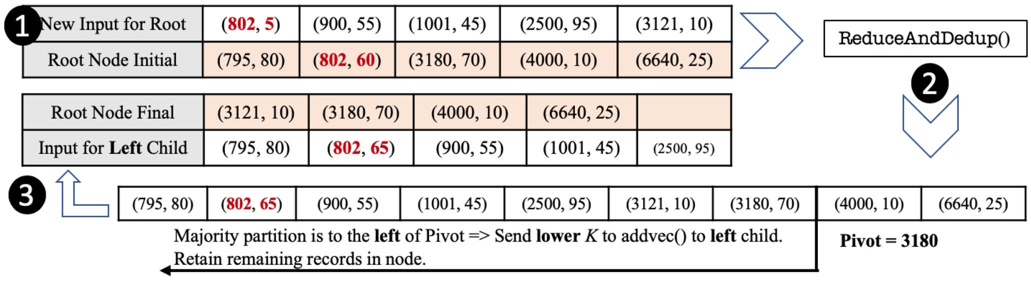


Fig. 3. Example functionality of Addvec() with $K = 5$.
Step 1: Front-end supplies a pre-sorted 5-vector to insert and reduce.
Step 2: ReduceAndDedup() is applied on the root node, given the input. In this example, records with key = 802 were reduced and de-duplicated.
Step 3: The pivot of the root node partitions the resultant vector, to determine which records need to be written back to the root, and which serve as input to recursively apply ReduceAndDedup() to one of the node's children. The relative direction of the larger partition determines the direction of recursion.

control fields in the form of a pivot (key) and a pair of child pointers are needed at each node, i.e., for every block of K-sorted key-value pairs that constitute the node, as shown in Figure 2. **K=170** is used as the default in this article unless otherwise mentioned. This is derived assuming a 2KB row [2] consisting of records whose keys are 64-bit and values are 32-bit (Section 2.1).

In general, two properties are desired of the tree:

(1) *Property 1*: For each node, all records in its left subtree have keys smaller than the **pivot key** of the node, which, in turn, is smaller than the keys of its right subtree.
(2) *Property 2*: For each node, all keys in its left subtree are less than **all keys** of the node, which are, in turn, less than those of its right subtree.

Property 1 is guaranteed to be true of the tree at any time including during incremental construction. *Property 2*, which is more strict, is guaranteed to be true for the final state of the tree, such that no duplicate keys exist.

The operational granularity being that of a DRAM row, the tree is built K records at a time. K pre-sorted records are added to the tree (and reduced and de-duplicated with existing records in the tree) via an operation known as Addvec(), which recursively applies a ReduceAndDedup() function along a path P of nodes (rows) of the tree, as shown in Figure 3. This function takes two

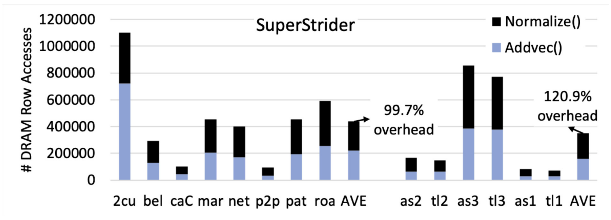


Fig. 4. Overhead of `Normalize()` in SuperStrider is $>100\%$ on average, thus motivating an improved, energy-efficient design. (See Figures 11 and 14 for a quick look at how much MetaStrider reduces this.)

K-sorted vectors and generates a deduplicated vector of size at-most $2K$, post reduction. Depending on the pivot key at that node, the larger partition (of size K, in general) is sent as an input vector to be applied recursively to the left or right child of the node, and the smaller partition (of size $\leq K$, in general) is written back to the node. This acyclic path P traverses the tree starting at the root and ends in a leaf of the tree (potentially creating a new leaf), or when there are no records left as a result of `ReduceAndDedup()`, whichever is earlier. When a new leaf node is created, a constant pivot key is chosen for that node as the median of its associated records.

In traversing P, a key observation that enables SuperStrider performance is that `Addvec()` visits at most 1 node (row) per level in the tree while simultaneously performing the reduction operation and avoiding *random* accesses to DRAM. The downside, however, is that there may be duplicates along other paths of the tree, which have to be consolidated. A detailed example is available in our previous work [72], but is omitted here for space constraints. In the operation of `Addvec()`, *Property 1* is always maintained. As a result, duplicates along different paths of the tree can be reduced and deduplicated as a final post-processing step that fixes any nodes in violation of *Property 2*.

This step, known as `Normalize()`, consolidates the left and right subtrees of each node in a DFS manner. This is done by traversing the left (right) subtree, extracting the leaf that houses the maximum (minimum) key, and performing an `Addvec()` at the parent node with the contents of the leaf as the input vector whenever *Property 2* above is violated. However, this tree traversal for `Normalize()` in SuperStrider is rather expensive in terms of energy, as quantified in Figure 4.

Furthermore, the latest version of SuperStrider employs a bitonic merge-based systolic network to realize the `ReduceAndDedup()` functionality [42]. The control logic (and stride management) is such that the inputs to the network are always K-sorted, an invariant that is re-enforced by `Addvec()`. This means that `ReduceAndDedup()` can be done in O(K) steps rather than O(K lg K) steps that the bitonic merger would require.

GraFBoost [43] is the most recent, *state-of-the-art*, out-of-core approach (such as GraphChi [48], X-stream [66], etc.) that accelerates random key-value pair updates through a specialized sort-reduce accelerator, which sequentializes fine-grained random accesses for data pairs stored in secondary (Flash) storage. The accelerator makes use of increasingly larger levels of reductions, all of which are based on multi-level merge sort. The in-memory reducer relevant to this work is a 16-1 merge tree composed of 2-1 bitonic mergers. GraFBoost requires the entirety of the data stream that has to be reduced be made available in memory before the in-memory reducer can be launched, rendering it unsuitable for accelerating the Firehose benchmark set. As such, this article performs comparison for the SpGEMM workloads alone in this regard, with a "batch mode" configuration, where all the partial products are available *a priori* in memory. Even so, the MetaStrider approach of carefully managing a vector binary tree is superior (by about 30% in performance and 15× in energy) to re-reading large chunks of memory in the style of a hierarchical merge-sort (GraFBoost).

Table 2. Description of MetaStrider's Metadata Fields

MetaStrider "metadata"					
SuperStrider "control fields"			*Additional fields unique to MetaStrider*		
Field	**Description**	**Bytes**	**Field**	**Description**	**Bytes**
RowAddr	DRAM row ID of node	2	Min	Min key of node	8
Pivot	Pivot key associated with node	8	Max	Max key of node	8
Addr L	DRAM row ID of left child	2	Max L	Max key of left subtree	8
Addr R	DRAM row ID of right child	2	Min R	Min key of right subtree	8

2.4 Key Takeaways for Metastrider Design

Key design decisions based on analysis of the state-of-the-art are now summarized.

First, sparse streams have low locality of reference, rendering automatic caching ineffective. MetaStrider chooses to *bypass caches* to save energy.

Given that DRAM accesses are on the critical path of the application, maximizing row locality is key. The *vectorized binary tree* approach proposed by SuperStrider for optimizing data layout is a promising one, but it can be made better via improved Metadata and its management (Section 3).

DRAM access occurs in bursts (64-bit for DDR4 and 128-bit for HBM). `ReduceAndDedup()` is the fundamental compute operation in MetaStrider, and is implemented using a *light-weight merge network* at the memory controller (Section 4). Designing the merge network such that it incrementally constructs the output by consuming the input in this manner would natively hide merge network latency without additional mechanisms such as write buffer, row re-map memory, and so on [72], as described in Section 4.

Given that the merge network is small, it can be replicated to enable *parallel operations on the tree* (Section 5) to leverage MLP. Finally, two variants of MetaStrider are considered:

- One that uses the memory-centric design without any extra hardware, at the cost of potentially increased energy consumption due to traffic on the system interconnect. In such a design, `ReduceAndDedup()` is performed by the front-end core instead of dedicated hardware.
- One that uses dedicated logic for merging "near" memory, at the DRAM controller. Such near data processing (*NDP*) not only reduces data movement but also enables an even tighter logic-memory integration for future variants as technologies evolve in the third dimension [30]. Therefore, unless otherwise mentioned, we assume this variant (and not the non-NDP variant) to be the default for MetaStrider.

3 METASTRIDER METADATA

Recall from Section 2.3 that SuperStrider's `Normalize()` is rather inefficient. Our analysis reveals the reason for this as redundant DRAM row activations in traversing down a given subtree in determining whether or not *Property 2* is violated at the root node of the subtree. As a solution, MetaStrider chooses to "memoize" certain range information such that testing for the property can be done without pointer chasing. If the range information indicates that *Property 2* is not violated, then no further actions are necessary (meaning no DRAM ACT) at that node, and the DFS superstep can continue. Typically, it is found that this is the majority scenario, which means that such extra "metadata" is worth the extra spatial overhead (a classic speed vs space tradeoff).

Table 2 describes the resultant Metadata fields with numerical examples apparent in Figure 2. Observe that *Property 2* can now be translated as follows: maxL < min < max < minR for any node.

To avoid confusion, in this article, the terminology of Metadata is unique to MetaStrider, and "control fields" are used to describe the equivalent for SuperStrider. Since SuperStrider's `Normalize()` is now updated to benefit from these Metadata as described above, it is called `GlobalReduce()` in the context of MetaStrider.

The **primary advantage**, a direct one, therefore stems from the introduction of these extra fields. The overhead of DRAM ACT is paid only when absolutely necessary.

The **secondary advantage**, an indirect one, stems from decoupling the Metadata from the *kv*-rows into a separate Metadata Store. This helps in improving structural properties of the tree such as density and worst-case guarantees. For example, with AVL tree balancing [33],[1] the tree is guaranteed to be dense irrespective of nature of sparse input, and such balancing can be done efficiently on the Metadata Store alone (without disturbing the *kv*-rows). We find that decoupling and AVL balancing alone reduce the average number of DRAM ACTs by over 20% when compared to an implementation where all the Metadata fields above are present but are not decoupled. Formally, the number of ACTs per call to `Addvec()`, irrespective of input, is bounded by the number of levels of the tree, which, in turn, thanks to the AVL property, is bound to lg N, where $N = \frac{nnz}{K}$ is the number of nodes in the tree.

Such decoupling also enables other downstream tree-optimizations such as partitioning, fanout, and so on (Section 5) to be made without having to re-build the system from scratch.

The **overhead** due to the Metadata Store is a small fraction of the data being stored. From Table 2, each node needs 46 bytes' worth of Metadata, 32 bytes of which are key-range information. For a 2 KB wide row being treated as a node, the overhead is therefore just about 2%. This overhead can be reduced via partial omission or compression of fields, the details of which are beyond the scope of this article. These mechanisms may be designed based on the following insights: (a) MetaStrider performance is not very sensitive to average access times of the Metadata Store (Section 7), (b) key-range information accounts for a significant fraction of the Metadata overhead (70%), (c) ranges are useful only during `GlobalReduce()`, (d) lower levels of the tree are less likely to be accessed, and (e) range re-computation can replace memoization for nodes with shallow sub-tree depths.

4 ARCHITECTURE AND INTEGRATION

Figure 5 presents, at a high-level, the micro-architecture that MetaStrider implements. Recall from Section 2.4 that the non-NDP variant utilizes the CPU to perform the functionality of the NDP unit. It can be deployed using off-the-shelf devices given the following two capabilities: ability to bypass caches, and ability to control row management (possibly via partial knowledge of row address interleaving). The remainder of this section focuses on describing the components of the NDP unit and then discussing techniques of integrating MetaStrider with conventional system software.

Merger Unit is responsible for performing `ReduceAndDedup()`, which is the recurrent substep used in `Addvec()`. It reads records in bursts from DRAM on one side and the front-end on the other. The ALU then compares the keys and performs reduction on their values if necessary. If no reduction is performed (i.e., distinct keys are input from both sides), then the record with the higher key is retained in the ALU input register for subsequent comparison, and that with the lower key is pushed to the tail of the output FIFO. This process is repeated until K records are

[1] An Adelson-Velsky and Landis (AVL) self-balancing binary search tree ensures that the heights of the two child subtrees of any nodes differ by at most one. In the context of MetaStrider's vectorized binary tree, AVL balancing is applied on the pivots of the nodes, without moving actual row contents.

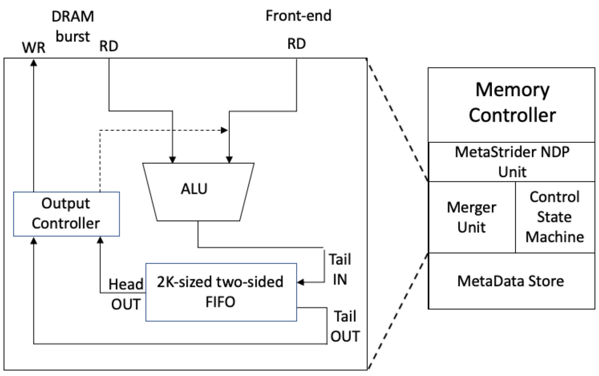


Fig. 5. A schematic of MetaStrider microarchitecture, implemented as part of the memory controller. In this article, the memory controller is designed to be able to access both, system memory and MetaStrider memory (pre-segmented at the beginning of the program using the `Init()` API).

read from each side. As described in Section 2.3, the output controller then sends the first/last few (≤K) records (depending upon the pivot) back to the open DRAM row, and retains the remaining (typically K) records in the FIFO. These latter records are then internally streamed to the ALU for the next sub-step when records from one of the child DRAM rows are streamed-in.

For example, in the example from Figure 3, the first record read from DRAM and the input, respectively, contain the keys 795 and 802. The ALU subsequently retains 802 and pushes 795 to the tail of the FIFO. Next, the record with key 802 is read from DRAM, and the ALU detects a collision and accordingly updates the associated value. Then, the record with key 900 is read from the input, causing the ALU to retain 900 and push the record with key 802 and its updated value to the FIFO. This process is repeated until both, the DRAM row and the input batch are read, and this results in the FIFO containing 9 records in this example. As the output controller finds that a majority of records are less than the pivot, it retains the lower 5 records in the FIFO to stream as the input batch for the next `ReduceAndDedup()` operation (on the left child). The higher 4 records are written back to the open DRAM row.

Note that this design, unlike SuperStrider's bitonic network, interleaves logic (ALU) and memory operations at the granularity of a DRAM burst. Furthermore, the complexity is linear in K, as opposed to log-linear (SuperStrider).

Metadata Store is akin to an SRAM cache in utility and dimensions, although it is explicitly managed as a scratchpad. Ideally, it would be housed on-chip with the memory-controller for fast access. Alternately, the LLC, which is largely unused for MetaStrider, can be used for this purpose. However, either of these strategies may not be possible when the size of the tree grows such that the total Metadata overhead is over several MB. In such a scenario, realizing the following key insight is helpful: the first few levels of the tree (and therefore their Metadata) are significantly more likely to be accessed frequently. Therefore, it would be advisable to "cache" these on-chip, and spill the rest off-chip, perhaps into system memory. A detailed analysis of off-chip Metadata Store is beyond the scope of this article. However, Section 7.3 demonstrates that nominal values of average access latencies have an insignificant impact (<0.5%) on overall performance.

Control State Machine. The control logic is able to issue load and store commands to the Metadata Store and to the memory controller, in addition to Merger Unit data routing. These functions are well within the capabilities of modern, smart memory controllers that perform sophisticated scheduling, queue coalescing, address interleaving, idle row closure decision making, and so on [56, 74].

Table 3. MetaStrider Application Programming Interface (API)

API	Comment
Init (sz, n, f)	MetaStrider configures its K, MLP-strategy (Section 5), reserves corresponding segment(s) of memory.
EnqReduce (k, v)	Each core sends to memory K sorted records one by one.
Addvec()	Each core signals Addvec() on its tree after K EnqReduce() or end of input, whichever is earlier.
GlobalReduce()	Each core signals end of input.
Lookup(key)	Lookup the value of key or initiate in-order traversal.

API. Table 3 summarizes a simple, offload-based blocking API to integrate with the MetaStrider engine. Note that the front-end producer of sparse data may be a general purpose CPU, accelerator, external flash device [43] or network I/O [8]. Without loss of generality, "core" is used to abstract these producers of sparse data. Sufficient memory is assumed available to house all records.

The most straightforward use-case scenario is where the programmer (or in certain cases, the compiler) identifies a sparse data stream to be reduced, and accordingly initializes MetaStrider with the size of records, number of cores supplying packed sparse data in parallel, and the desired reduction operator. MetaStrider accordingly configures itself based on available resources, reserves corresponding segment(s) of memory and returns the value of K to the program. Each core then scatters the gathered sparse data to be reduced by first pre-sorting these, K at a time, and then dispatching them in-order via `EnqReduce()`. Recall from Section 2.3 that this pre-sorting is a necessary requirement for scalable `Addvec()` functionality. Note that standard sparse input formats (see below) are pre-sorted by themselves. If the application is such that it is not possible to obtain a pre-sorted input, then an incrementally sorted paradigm, such as a priority queue, may be used while K records are gathered and buffered at the front-end. Even if it is decided to perform a $K\lg K$ sort as a separate pre-processing step, our analysis in Section 7.4 reveals that the core frequency required for this to be in parallel with a preceding `Addvec()` is modest (<1GHz) for typical values of K (recall that `Addvec()` typically involves reading and writing multiple DRAM rows in succession, and is therefore a relatively longer latency operation).

After K such `EnqReduce()` calls, an `Addvec()` call signals the MetaStrider controller that the end of this batch (of size K or end of input) is reached. This process is repeated until the input is consumed. `GlobalReduce()` is then called to ensure the entire dataset is reduced and de-duplicated, thereby priming the data for downstream operations.

For clarity, the above is also explained via a specific example in the form of how the MetaStrider API may be used in the context of a single-core SpGEMM, as shown in Listing 1.

Conversion to/from other representations. Commonly used linear sparse formats include ordered-coordinate, (doubly) compressed sparse row (D)CSR [15, 17, 28, 44, 76]. For hyper-sparse matrices such as those of interest in this article, CSR offers little benefit over ordered-coordinate as there are few non-zeros per row. DCSR offers benefits only when a group of consecutive rows are all zero. Thus, ordered-coordinate (typically stored as arrays of keys and values) are most commonly used in this space [3]. All of these representations have the disadvantage of not being able to support dynamic updates efficiently as they fundamentally require insertion into the middle of an array. Nevertheless, MetaStrider operation is independent of and is compatible with inputs in any of these formats. Note that the MetaStrider representation most closely resembles ordered-coordinate with DRAM-friendly enhancements.

Conversion from these formats into MetaStrider format is a direct application of the API when the original input is streamed record-by-record into the MetaStrider engine. Thus, no explicit conversion is required as conversion can be combined (implicitly) with downstream operations.

```
1  // A and B may be stored in conventional sparse formats or in the MetaStrider format.
2  SpGEMM_OuterProduct (Matrix A, Matrix B):
3    // Initialize MetaStrider memory.
4    K = MetaStrider.Init(12, 1, "+")
5    colA = A.readNextNonZeroColumn()
6    rowB = B.readNextNonZeroRow()
7    // Assume that the column index of colA and the row index of rowB are identical (k). If
         not, advance accordingly until they are (not shown).
8    for (i, k, A_ik) in colA:
9      for (k, j, B_kj) in rowB:
10       key = makeKey(i, j)
11       value = A_ik * B_kj
12       sortedQueue.pushAndSort((key, value))
13       if (sortedQueue.size() == K or EOF):
14         // Stream records for Addvec().
15         while (sortedQueue.size() > 0):
16           MetaStrider.EnqReduce(sortedQueue.popMin())
17         MetaStrider.Addvec()
18   // Repeat lines 5–17 until A or B consumed. Then:
19   MetaStrider.GlobalReduce()
20   // Perform gather for downstream operations as needed.
21   MetaStrider.Lookup()
```

Listing 1. Demonstration of MetaStrider API.

Converting back to a conventional linear ordered format from the MetaStrider format can be achieved easily by reading out the records back from the tree via an in-order DFS traversal. However, further operations on MetaStrider data do not require conversion to a traditional format as long as the operations are associative. In other words, downstream algorithms can stream in data from the MetaStrider tree in a DFS or BFS manner directly without having to store an intermediate linear matrix format into memory. Furthermore, several hash-based applications do not even require ordered input data [57]. As such, we envision such explicit write-out to occur only once all processing on data is complete, and only if necessary for backward compatibility.

Explicit or implicit conversion to/from these formats can thus be done on the fly, in parallel with downstream operations. Therefore, there is negligible impact on performance due to conversion.

Lookup. Finally, one can extract the value associated with key κ via `Lookup(`κ`)`, where the Metadata is consulted to perform a binary search over the nodes of the tree (using the pivots) to locate the node whose key-range includes κ. If such a node is found, then the corresponding DRAM row is opened. Then, a binary search within the row is performed to locate κ as the records in the row are already sorted.

5 LEVERAGING MLP

So far, the article has focussed on MetaStrider from the perspective of a single bank of memory (MLP = 1). Modern memory systems have significant MLP available via bank, rank, and channel level parallelism. For example, HBM has 8 channels, each of which comprises 8 banks, resulting in an MLP of 64. This section explores several mechanisms for parallelizing a MetaStrider tree to leverage available MLP and obtain improved performance (assuming the input rate (front-end) can keep up).

Figure 6(a) depicts the baseline MetaStrider tree (N nodes and lg N levels) sprawling a single bank of DRAM, as well as parallel approaches described below.

This section focuses on `Addvec()` and not on `GlobalReduce()`. This is because our Metadata improvements (Section 3) have significantly reduced the overhead of the latter (Section 7.3) such that `Addvec()` is now the most critical step. Furthermore, parallelizing `GlobalReduce()` can benefit from years of parallel DFS research [63] and is therefore not necessarily a novel contribution of this article in itself.

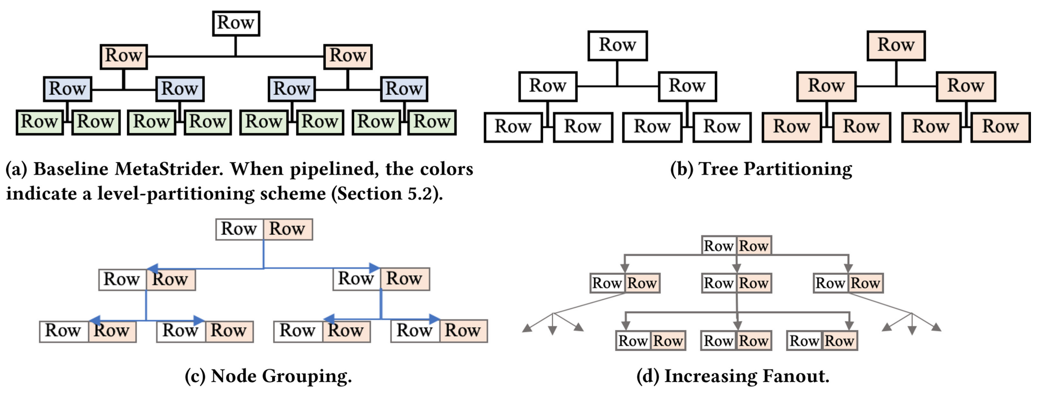

Row
(a) Baseline MetaStrider. When pipelined, the colors indicate a level-partitioning scheme (Section 5.2).
(b) Tree Partitioning
(c) Node Grouping.
(d) Increasing Fanout.

Fig. 6. Four novel techniques of vectorized binary tree management to leverage MLP. Different colors indicate different banks of memory that can be accessed in parallel.

Note that with the exception of Tree Partitioning approach, none of the other three approaches require the programmer or compiler to make any code changes. If a power user wishes to override the default strategy, then it is sufficient to simply extend the `Init()` API to specify a bit-mask to indicate which mode(s) to deploy.

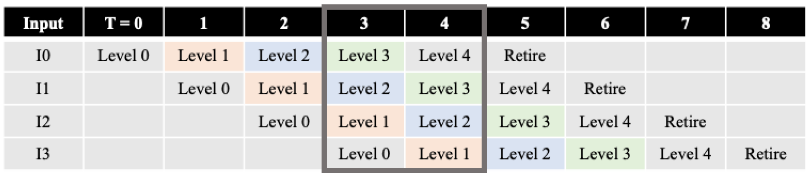
5.1 Tree Partitioning

The single tree can be spatially partitioned by key into T trees, each containing $\frac{N}{T}$ nodes and $\lg \frac{N}{T}$ levels. This enables the trees to function independently and in parallel (Figure 6(b) shows $T = 2$). Furthermore, the unit of work for each tree is reduced as they each have fewer nodes and levels are significantly reduced when compared to the baseline MetaStrider tree. For these benefits to fully manifest, the following are desirable:

Load balancing. For shortest critical path, all the trees should have approximately an equal number of records to reduce. A round-robin distribution of keys (tree t gets keys such that $key\%T == t$, where % is the modulo operator) is a simple, yet effective approach at this. Load balancing can be further improved via hashing, provided the hash is perfect and preferably minimum and uniformly distributing.

For example, it is known that the Residue Number System (RNS) [32] can help distribute contiguous data to a wider range [25, 26, 73]. Such a hash function would be computed by concatenating the set of residues (moduli) $R(\kappa)$ of key κ against a pre-determined set of co-prime bases B such that $R(\kappa) = \{\kappa\%b, b \in B\}$. For keys generated by indexing into a matrix of at most $100M \times 100M$ dimensions, it can be shown that $B = \{45, 46, 47, 49, 53, 59, 61, 67\}$, $B = \{2049, 2050, 2051, 2053\}$, or $B = \{4194305, 4194306\}$ for $T = 8, 4, 2$, respectively, are suitable for such RNS hashing. Other similar hash functions that help extract MLP in modern systems, such as XOR-based hashes [64, 80] may also help improve load balancing. Finally, note that any such record-to-tree assignment is to be applied by the front-end prior to the call to `EnqReduce()` to satisfy the pre-sorted input condition. Our evaluation therefore is bounded by $T = 8$ to limit the amount of intra-front-end state communication.

Parallel NDP and front-end hardware. The front-end should be capable of feeding all trees in parallel for maximum benefit. In other words, the front-end should be able to issue T `Addvec()` calls in parallel. Next, the control logic, Merger Unit, and Metadata Store need to be replicated at some level. Note that the former two are relatively light-weight by design and can therefore be replicated.



Input	T = 0	1	2	3	4	5	6	7	8
I0	Level 0	Level 1	Level 2	Level 3	Level 4	Retire			
I1		Level 0	Level 1	Level 2	Level 3	Level 4	Retire		
I2			Level 0	Level 1	Level 2	Level 3	Level 4	Retire	
I3				Level 0	Level 1	Level 2	Level 3	Level 4	Retire

Fig. 7. The “EX” stage can be sub-pipelined to leverage MLP.

The Metadata Store need not increase in size as the total number of nodes (hence total Metadata) doesn’t change with tree partitioning. For example, if the unpartitioned tree has 100 nodes, then with $T = 4$ partitions, each partition would nominally have 25 nodes (resulting in each partition being a shallower/faster tree). However, it is desirable to implement the Metadata Store as a multi-banked store for parallel access.

API usage. The programmer or compiler must perform the following for updating T trees in parallel, to benefit from the tree partitioning strategy:

(1) Specify $n = T$ in `Init()`.
(2) Create T threads, each with a thread-private version of `sortedQueue` (cf. Listing 1).
(3) Each thread produces records, applies the load-balancing hash to its keys and then sends to appropriate thread.
(4) Each thread then calls `EnqReduce()`, `Addvec()`, and `GlobalReduce()`.

5.2 Pipelining

Given sufficient *resources*, pipelining is a compelling mechanism of extracting parallelism from a stream of independent *instructions*. For pipelining MetaStrider, *resource* is equivalent to MLP and *instructions* are equivalent to recurrent calls to `Addvec()`. Recall that each such call typically traverses several levels of the tree, accessing exactly one unique DRAM row (assume $F = 2$) per level (hence, no data stalls are possible). Therefore, given sufficient MLP (read no resource stalls), consecutive calls to `Addvec()` can occur at every time step (τ) as shown in Figure 7, where τ is the time required to perform `ReduceAndDedup()` on a row. This is similar to deep-pipelining or sub-pipelining the function units in the execute stage of a traditional processor.

Resource allocation. The intuition of our approach to minimize resource stalls stems from the following insight: assigning each pair of consecutive levels of the tree to different memory banks eliminates resource stalls for a (sub)pipeline depth of 2. It can then be seen that, for a tree of depth $D \leq MLP$, assigning each level to a different bank (level-partitioning) enables pipelines of depth MLP (Figure 6(a) shows $D = MLP = 4$).

However, beyond a sufficiently deep level $D_{thres} \leq D$, the number of rows in that level may exceed the size of a bank. For example, with a 128MB (64K x 2 KB rows) bank, $D_{thres} = 16$. To account for bank capacity, this level-partitioning approach is applied only to the first $D_{thres} - 1$ levels of the tree, and subsequent levels $d \geq D_{thres}$ are each sliced equally across the banks, resulting in batches of $\frac{2^d}{MLP}$ rows of level d per bank.

Load balancing. When $D > MLP$, the above level-partitioning mechanism requires further thought for levels at depth d, where $MLP \leq d < D_{thres}$. In other words, a relatively small MLP would result in increased resource stalls in the pipeline if the load distribution to the banks is not uniform. A simple mechanism to address this is to assign such levels to different banks in a round-robin (RR) manner. Note two properties of the tree, as one goes down the tree: (i) the number of rows in each level doubles and (ii) the probability of a level being accessed decreases. RR is not cognizant of either of these properties, thus begetting an improved heuristic.

If D were known *a priori*, then levels 0 and D would be mapped to bank 0, levels 1 and $D - 1$ to bank 1, and so on. Such an allocation factors in both of these properties above to realize a balance that is as close to the ideal as possible. However, as the size and percentage of repeated keys of the input is not known, D is not static and hence cannot guide resource allocation. Therefore, we propose an incremental variant of this heuristic where levels are assigned in a ping-pong (PP) manner such that level d maps to bank ($d\%MLP$) if $d\%(2 \times MLP) \leq MLP$, else, it maps to ($MLP - d\%MLP$).

More complicated approaches that are not level-partitioned are possible when this is expressed as a graph coloring problem. However, according to our analysis, a simple heuristic such as PP is able to leverage about 80% of available MLP on average (and reduces resource stalls by over 35% when compared to RR). Yet other designs may choose to literally cache the "rows" of the first few levels of the tree that tend to be accessed most frequently, although those approaches may not scale easily with increased number of trees.

Supporting hardware. Explicit pipeline buffers are not necessary as the DRAM rows implicitly serve as buffers. The Merger Unit needs to be replicated to take advantage of MLP, similar to Section 5.1. The Metadata Store capacity remains unchanged but requires parallel access to serve multiple inputs simultaneously. The front-end still produces a single input to `Addvec()` at a time, its rate being determined by pipeline depth (MLP).

5.3 Grouping

Yet another mechanism of leveraging multiple banks is to group two rows of the same index across two banks into the same node (Figure 6(c)). The motivation is that it would increase K logically, thus improving algorithmic efficiency because of denser trees and higher probability of parallel reduction. For example, say $K = 170$ records fit in a row. By grouping such rows across 4 banks and associating them with a single tree node, $K = 680$ is realized.

This results not only in trees with reduced depth but also in lower Metadata overhead thanks to the reduced number of nodes. However, grouping increases the load on front-end as it now has to pre-sort gathered data in batches of 680 records rather than 170 records before sending it across for scatter, although it reduces the Metadata Store overhead by 4×. Also, MLP is not utilized within a tree if the same Merger Unit as baseline MetaStrider is used. If MLP has to be utilized, then a log-hierarchical network similar to SuperStrider's bitonic merger is better suited, although it does not fully support fine-grained logic-memory overlap.

5.4 Fanout

The arity/fanout (F) of a binary tree can be increased to $F > 2$ to decrease the number of levels of the tree to $\log_F N$ (Figure 6(d) shows $F = 3$), thereby facilitating faster MetaStrider operation. However, it then becomes necessary to increase the number of DRAM rows associated with any given node to $F - 1$ to maintain the granularity of operation to be that of a DRAM row (recall from Section 2.3 that row-granularity operations are fundamental to increasing row locality in a vectorized tree design).

As an example of contradiction, assume each node in a 3-ary tree is still associated with a single DRAM row. A 3-ary node would by definition have 2 pivots instead of 1 to result in 3 partitions corresponding to its 3 children. During `ReduceAndDedup()`, consider the task of partitioning (at most) $2K$ records in the output buffer of the Merger Unit into 3 partitions. Depending on the nature of the 2 pivots, it cannot be guaranteed that at least one partition has at least K records in it. Therefore, it is not possible to propagate any single partition to a child without violating the row-granularity of operations. Propagating $< K$ records to a child would result in significantly diminishing returns (increasingly lower useful bytes accessed per subsequent ACT) as we traverse down the tree. However, propagating two partitions down the same direction would violate *Property 1* (when generalized to $F \geq 2$, see below). Instead, if 2 DRAM rows are associated with a 3-ary node, then the task of partitioning would involve $(2K + K = 3K)$ records. It can then be *guaranteed* that at least one partition has at least K records.

The conclusion is that $MLP \geq F - 1$ is necessary to retain the row-granularity of operations. Increasing F is, in essence, another way of leveraging MLP. Note, increasing fanout can leverage MLP *without* requiring a parallel front-end.

The Merger Unit architecture is similar to the $F = 2$ case, with the output buffer being of size $F \times K$ and the control logic being updated to support F partitions.

Changes to the algorithm. One may gain intuition for the general working of updated `Addvec()` and `GlobalReduce()` based on the text above and on the generalized version of *Property1, 2* (Section 2.3) below, for each F-ary node:

(1) *Property 1*: All records in its fth subtree have keys smaller than the fth **pivot key** of the node, which is, in turn, smaller than the keys of its $(f + 1)$th subtree, where $0 \leq f < F$.
(2) *Property 2*: All keys in its fth subtree are less than **all keys** of the node, which are, in turn, less than those of its $(f + 1)$th subtree, where $0 \leq f < F$.

Impact on Metadata. An F-ary node requires storage of $(F - 1)$ pivots, addresses of its (F) children, and key-ranges of each of the following: (F) partitions of the node, $(F - 1)$ rows associated with the node and (F) subtrees of the node (some of these metadata overlap and are rendered redundant when $F = 2$). As a result, when compared to the per node Metadata size in bytes of $F = 2$ (46 bytes: Section 3), an F-ary node's Metadata experiences an increase by a factor of about $0.7F$ where $F > 2$. However, note that upon increasing F, the number of nodes reduces, because each node now consists of upto $(F - 1)K$ records. As a result, the total overhead of Metadata in bytes, when compared to $F = 2$, is an input-dependent factor α, where $\alpha \in [\frac{0.7F}{F-1}, 0.7F]$ and $F > 2$. A detailed derivation of these details is omitted for space constraints.

Summary of Section 5. In this section, 4 orthogonal approaches to leveraging MLP are proposed, all of which can be used in conjunction with each other. This results in a tradeoff-rich design space, given hardware constraints and availability. Based on recent industry trends, the authors' preference of these are as follows: Partitioning > Pipelining > Grouping > Fanout. In particular, a combination of Partitioning and Pipelining is found to be most practical and efficient. A more quantitative treatment is available in Section 7.4.

6 EVALUATION METHODOLOGY

Recall from Section 2.1, the focus of this work is to tackle main memory latency for sparse data. As such, the first-round of simulations deploy a single channel of 128MB of HBM main memory with a relatively liberal 4MB of LLC to favor the software reducers conservatively. The software baselines benefit from bank-level parallelism (8 banks in a channel), whereas MetaStrider conservatively does not leverage this in these first round of simulations. As is standard practice, an FR-FCFS memory scheduler, and an open-adaptive row management policy are used. Although the reducers in this article (including MetaStrider) would perform with any DRAM without significant changes in efficiency, HBM is chosen because its stacked architecture naturally lends scalability to near-data processing (NDP), allowing for easier comparisons in future works.

Since the workloads have low compute intensity, the LLC and DRAM are driven with an inorder core (with 8-way 32KB L1 I/D caches). The system is simulated using the gem5 full system simulator. In particular, MetaStrider is implemented as a separate MemObject connected to the system interconnect. MetaStrider API is implemented using gem5's pseudo-instructions at the front-end.

Table 4. Simulation Infrastructure

	Section	Tool
Baseline, kokkos	2.2	Unmodified gem5 [13]
MetaStrider	4	gem5 + cache bypass
SuperStrider	2.3	+ dedicated accelerator memory
GraFBoost	2.3	+ hardware at memory controller
Rapid design space exploration	5	Cycle-accurate, in-house simulator using HBM timing parameters [2]
Power, area	Logic	ALU [52, 67]
	Memory	SRAM-CACTI [11], DRAM [20]
(22nm models)	System Interconnect	OpenSMART [47]

A second round of simulations are designed to evaluate the scalability of MetaStrider with available MLP. For simulation speed, a cycle-accurate, in-house simulator is used for this purpose.

A summary of the simulation infrastructure is in Table 4. This article makes an effort to compare MetaStrider against other state of the art reducers (Sections 2.2, 2.3) across various domains (CPU, GPU, accelerator, out-of-core) to provide reasonable insights. For fair comparison, we have re-implemented these original works using the infrastructure above to better suit them for sparse associative reduction, making assumptions in their favor where possible (for example, ignoring the overhead of SuperStrider's control fields and row-remap memory). Finally, to compare against GraFBoost, which requires all the partial products to be available in memory *a priori*, a "batch" mode is also implemented.

7 EXPERIMENTAL RESULTS

7.1 DRAM Performance

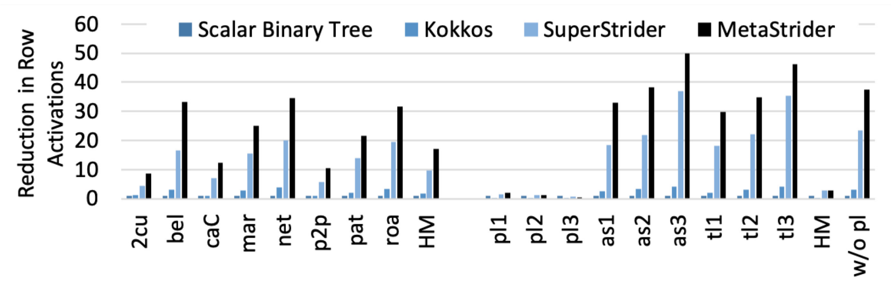
Row activation. Figure 8(a) shows a significant reduction in ACTs (and subsequent PRE). An average reduction of over 17× and 37× is seen for kernels in SpGEMM and Firehose, respectively, when compared to the baseline. Furthermore, these are 1.8× and 1.6× better than SuperStrider thanks to MetaStrider's improved Metadata capability.

Row hit rate and CAS per ACT. Bytes read per ACT of a 2KB DRAM row is close to 64 for the baseline, because the default read-out is the cache line size. Kokkos significantly improves locality characteristics, reading over 300 bytes per ACT, for a row hit rate of close to 80%. MetaStrider and SuperStrider achieve a row hit rate in excess of 95% and bytes per ACT of over 2000.

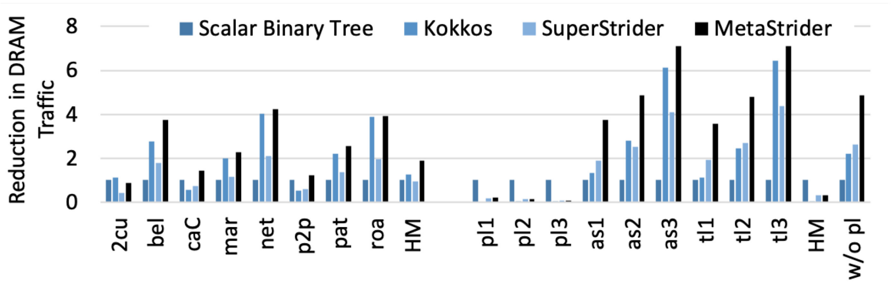
DRAM bytes read. As a result of the above observations, MetaStrider reduces DRAM pressure by 1.9× and 4.9× for SpGEMM and Firehose, respectively, when compared with the baseline, as shown in Figure 8(b). These are significantly superior when compared with the other reducers (both hardware and software). In fact, SuperStrider ends up reading more bytes from DRAM on average for SpGEMM workloads on average when compared to the baseline because of its row-heavy operations, especially during its `Normalize()` phase.

7.2 Full System Results

Energy Savings. For space constraints, a summary of un-core kernel energy savings due to MetaStrider is depicted in Figure 9(a), when averaged across all workloads. (For fairness, for the non-NDP MetaStrider variant, the front-end core's contribution to `ReduceAndDedup()` is viewed as un-core energy, and is denoted by "Merger Unit"). Clearly seen, when compared to the baseline, MetaStrider reduces energy consumption significantly for all system components. The added

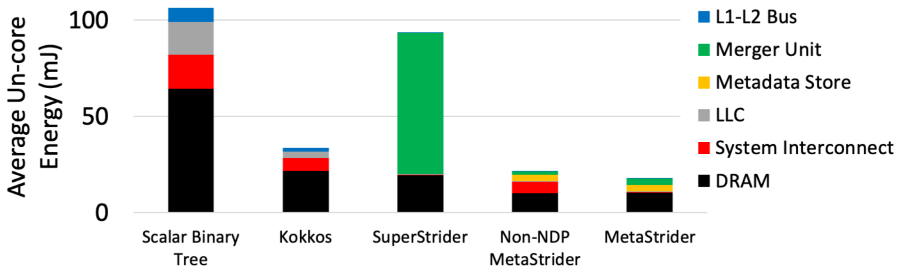


(a) Reduction in DRAM ACTs.

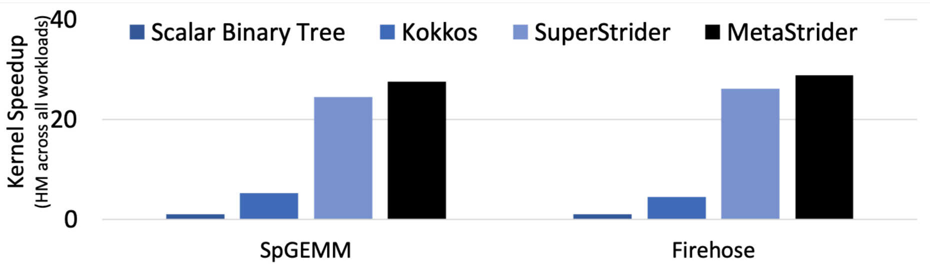


(b) Reduction in total bytes read from DRAM.

Fig. 8. MetaStrider significantly reduces the number of activates as well as DRAM traffic, when compared to the baseline (higher reduction is better), thanks to its memory-centric design and intelligent Metadata.



(a) A breakdown of average un-core energy.



(b) Overall kernel speedup.

Fig. 9. The most energy efficient reducer is the NDP variant of MetaStrider, realizing 5.3× energy savings when compared with the next best reducer, upon averaging across all workloads. Furthermore, it does so with a 11% performance improvement.

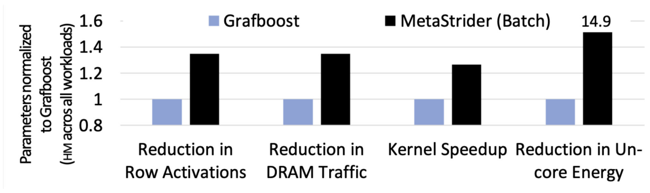


Fig. 10. When all of the input kv-pairs are already available in memory, MetaStrider reduces un-core energy consumption by almost 15× and is 30% faster than the state-of-the-art approach for such merging because of its superior handling of DRAM rows.

overheads due to the Merger Unit and Metadata Store are dwarfed by these savings. Even when compared to SuperStrider, there is significant improvement in energy savings due to DRAM and the Merger Unit.

Area Overhead. The Metadata Store requires 7.74mm^2 for 3MB of Metadata required, which can be significantly reduced if techniques such as grouping (Section 5.3) or compression (Section 3) are used. Furthermore, the store can also use existing LLC-SRAM or system DRAM if a dedicated budget is not available, as described in Section 4. The Merger Unit requires a mere 0.06 mm^2, which, when compared to an HBM die size of 40–100mm^2 [4–6], is a very small fraction. Therefore, techniques that leverage MLP (Section 5) that may require some form of replication of the Merger Unit can also be easily realized on the same die. Note that this is significantly less resource intensive when compared with SuperStrider's bitonic merge network (0.24mm^2) and GraFBoost's hierarchical merge tree (2.36mm^2).

Kernel Speedup. Figure 9(b) depicts the kernel speedup over the baseline when averaged across all the workloads. The relative trend among the configurations can be explained easily based on the detailed breakdown presented thus far. For SpGEMM workloads, MetaStrider approaches the Amdahl limit with a deficit of less than 8%. Note that in the case of the Firehose benchmark set, the kernel *is* the application.

Batch Mode. When all the input kv-pairs are already available in memory, i.e., when the input is no longer an incremental stream of records, the state-of-the-art approach for such merging is GraFBoost. Note that while such batch-mode input may be possible for SpGEMM workloads, it is infeasible for other incremental / streaming applications such as Firehose, for which only MetaStrider would be applicable. Figure 10 shows that in batch mode, MetaStrider reduces un-core energy consumption by almost 15× and while also improving performance by 30%, when compared with GraFBoost, thanks to its improved DRAM behavior and more efficient merger unit. Recall that GraFBoost uses a 16-1 merge network (and successively merges exponentially increasing batches of records), contributing to over 94% of its un-core energy.

7.3 Sensitivity Analysis

Overhead of GlobalReduce(). The performance overhead incurred due to `GlobalReduce()` is just about 2% on average for MetaStrider, shown in Figure 11. Thanks to enhanced Metadata, this is over 5.2× faster than SuperStrider's `Normalize()`.

Benefit of near data processing. Figure 12 shows that even the non-NDP variant of MetaStrider reduces traffic on the system interconnect when compared to the baseline by 1.5× and 3.2× for kernels in SpGEMM and Firehose workloads, respectively. Upon deploying NDP, there is a further reduction of over 22× in traffic that is compounded. NDP helps further reduce energy consumption (without affecting performance despite its slower logic, because the merger unit operation is completely overlapped with DRAM operation).

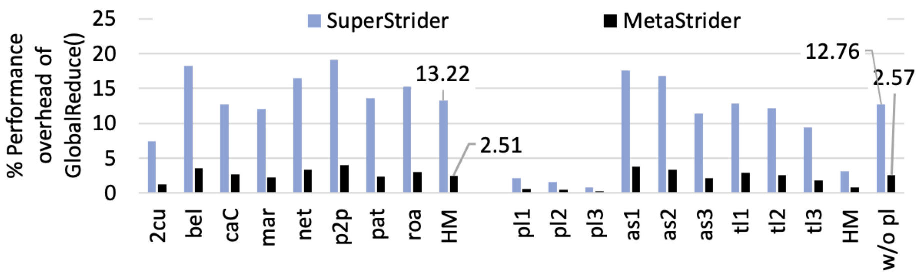


Fig. 11. The % performance overhead incurred due to `GlobalReduce()` (lower is better) is just over 2% for MetaStrider, when averaged across all workloads.

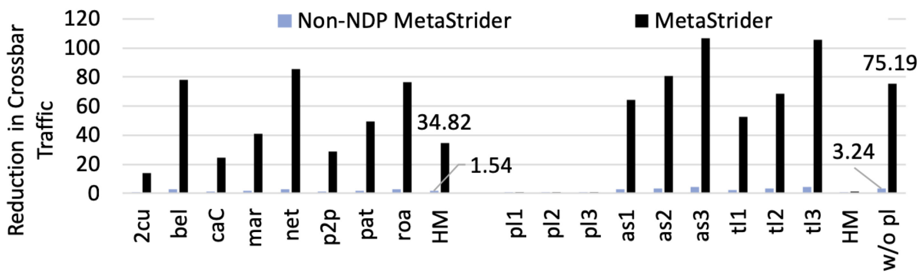


Fig. 12. Primary benefit of near data processing (NDP) portrayed via the reduction in bytes of traffic on the system interconnect (higher reduction is better). Note that MetaStrider reduces traffic even without NDP.

Speed of Metadata Store. When the Metadata size exceeds area budget, the Metadata Store needs to be adapted to use a combination of slow-fast memories (Section 4), or pay the penalty of extracting key-ranges via decompression or re-computation (Section 3). In either approach, an increase in average access time is the result. However, no significant impact (<0.5%) on kernel speedup was observed when the following representative access times were simulated across all workloads: (i) Idealized, where there is no overhead in accessing the store, (ii) SRAM, and (iii) DRAM, where all the Metadata is in an SRAM/DRAM store, respectively.

Real-time capability. When `GlobalReduce()` is omitted, a call to extract the value for a key κ via `Lookup(`κ`)` may yield one of the following cases: (i) Exact match (record with key=κ found and correct value returned), (ii) Partial match (key found but incorrect value, prompting incomplete reduction), and (iii) Missing key (key not found). This is of specific importance to Firehose, where real-time flagging is necessary. In this context, MetaStrider achieves a favorable distribution of 96.5%, 2% and 1.5%, respectively. When no tree balancing is done, case (iii) is no longer a possibility. Recall that after a call to `GlobalReduce()`, both (ii) and (iii) would be eliminated.

Benefit of Metadata decoupling and tree balancing. Decoupling the Metadata from the kv-rows and tree-balancing independently improve DRAM efficiency. However, tree balancing is practical only with decoupling, because of the metadata-heavy nature of re-balancing. As a result, MetaStrider deploys a combination of tree balancing and Metadata decoupling. Figure 13 shows that a 20% reduction in DRAM ACTs is observed due to AVL-balancing of the tree (when Metadata is decoupled), upon averaging across all workloads.

7.4 Scalability Analysis

This section demonstrates quantitatively the tradeoffs described in Section 5.

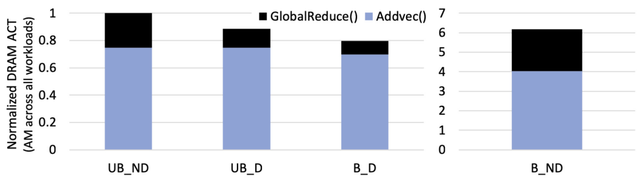


Fig. 13. Benefit of Metadata decoupling and tree balancing (lower #ACT is better). UB = UnBalanced, B = Balanced. ND = Not Decoupled, D = Decoupled.

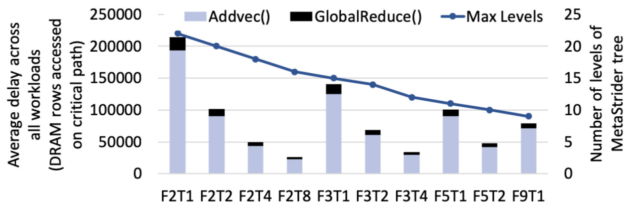


Fig. 14. Performance of various fanout/partitioning configurations when averaged across all workloads, in terms of DRAM rows accessed (lower is better). MLP = (F − 1) * T, where F = fanout, T = #trees. Performance fundamentally improves with either technique, even in the absence of tree balancing. Furthermore, `GlobalReduce()` overhead is significantly reduced when compared to SuperStrider (Figure 4).

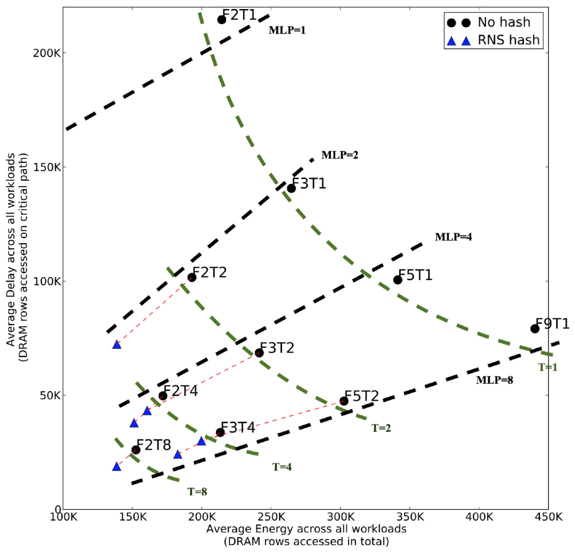
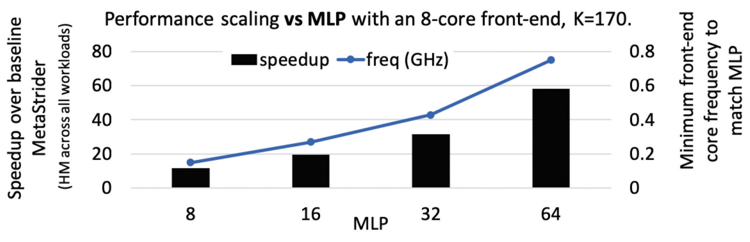
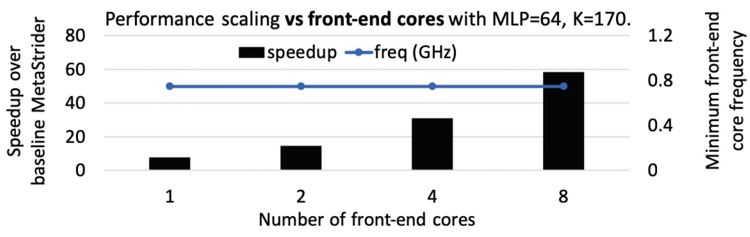


Fig. 15. A pareto-style tradeoff analysis in DRAM delay and energy for various combinations of tree partitioning and fanout, when averaged across all workloads.

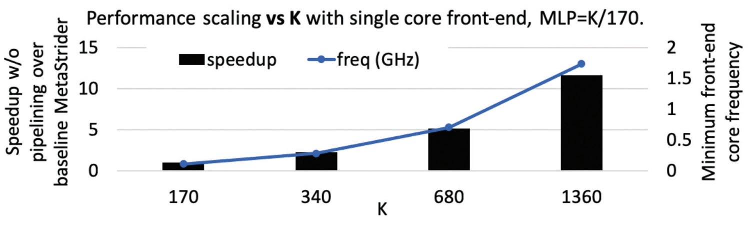
Partitioning vs Fanout. Figure 14 evaluates various design points that combine partitioning and fanout, while also depicting the scalability of the performance overhead of `GlobalReduce()`. Although the height of the tree decreases with increasing fanout, note that fragmentation increases as each node in an F-tree spans $(F - 1)$ rows, resulting in an increased average depth per node insertion. Increasing T rather than F is therefore favorable, assuming sufficient front-end core parallelism is available to scatter to each partition.



(a) Scalability with MLP.



(b) Scalability with cores.



(c) Scalability with K.

Fig. 16. MetaStrider performance scales near-linearly with available hardware resources using techniques from Section 5. The number of partitions (trees) is always equal to the number of front-end cores. A sub-1 GHz core frequency is sufficient to drive MetaStrider, unless a large K is needed. From Figure 15, partitioning is favored over fanout. Therefore, in (a, b), partitioning+pipelining is used. In (c), the impact of row-grouping (alone) is depicted.

Figure 15 shows that increasing T rather than F is more favorable from both, an energy stand-point and performance stand-point. Furthermore, a load balancing hash for the partitions such as RNS hashing further improves gains. In the figure, note that iso-T frontiers are indicated with green dashed lines and iso-MLP with black dashed lines.

Pipelining. Ideally, one would therefore allocate a partition for every unit of MLP. However, in a system that is constrained by the number of front-end cores driving MetaStrider, that may not be possible. Instead, pipelining across banks may be used, although perfect MLP utilization would be traded off. Figures 16(a) and 16(b) demonstrate that this is still largely effective in providing near-linear scalability in performance.

Grouping. As the reader may have guessed, the effect of grouping is similar to that of logically increasing K of the system. This is attractive provided the front-end can provide batches of K-sorted vectors accordingly. Figure 16(c) shows super-linear benefits of increasing K when a correspondingly fast front-end core is used.

8 CONCLUSION

Sparse data applications are widely used in several domains today. Because of their unique characteristics of having low compute-to-communicate ratio and little-to-no locality of reference, these sparse data streams perform poorly when traditional algorithms and architectures are used. MetaStrider addresses these fundamental latency-bound inefficiencies in these streams in a manner that also scales with available parallel resources. The proposed memory-centric architectures and algorithms are significantly more efficient than comparable approaches to the problem. Furthermore, the resultant performance of sparse applications is within 8% of that of an idealized accelerator that performs the compute and memory operations associated with sparse reduction in zero time. MetaStrider has been designed such that future technologies that further improve logic-memory integration would result in even more efficient operation. The authors hope that this work will motivate even more novel, memory-centric architectures that tackle sparse, data-irregular streams in the future.

ACKNOWLEDGMENTS

The authors thank Dr. Lavanya Subramanian, Dr. Vinson Young, and Abhinav Jain for their feedback at various stages of this work.

REFERENCES

[1] Standard map reference implementation. [n.d.]. GCC std::map. Retrieved from https://gcc.gnu.org/onlinedocs/libstdc/libstdc-html-USERS-3.4/stl__map_8h-source.html.
[2] JEDEC High Bandwidth Memory Specification. [n.d.]. High bandwidth memory (HBM) DRAM. Retrieved from https://www.jedec.org/standards-documents/results/HBM.
[3] Matrix Market. [n.d.]. Retrieved from https://math.nist.gov/MatrixMarket/.
[4] High Bandwidth Memory Characterization 1. [n.d.]. Retrieved from https://www.amd.com/Documents/High-Bandwidth-Memory-HBM.pdf.
[5] High Bandwidth Memory Characterization 2. [n.d.]. Retrieved from https://www.anandtech.com/show/9969/jedec-publishes-hbm2-specification.
[6] High Bandwidth Memory Characterization 3. [n.d.]. Retrieved from https://www.gamersnexus.net/news-pc/2972-amd-vega-frontier-edition-tear-down-die-size-and-more.
[7] Kadir Akbudak and Cevdet Aykanat. 2017. Exploiting locality in sparse matrix-matrix multiplication on many-core architectures. *IEEE Trans. Parallel Distrib. Syst.* 28, 8 (2017), 2258–2271.
[8] Karl Anderson and Steve Plimpton. 2015. *FireHose Streaming Benchmarks.* Technical Report. Sandia National Laboratory.
[9] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. 2017. Structured pruning of deep convolutional neural networks. *ACM J. Emerg. Technol. Comput. Syst.* 13, 3 (2017), 32.
[10] Ariful Azad, Aydin Buluç, and John Gilbert. 2015. Parallel triangle counting and enumeration using matrix algebra. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSW'15).* IEEE, 804–811.
[11] Rajeev Balasubramonian, Andrew B. Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New tools for interconnect exploration in innovative off-chip memories. *ACM Trans. Archit. Code Optim.* 14, 2 (June 2017). DOI: https://doi.org/10.1145/3085572
[12] Nathan Bell, Steven Dalton, and Luke N. Olson. 2012. Exposing fine-grained parallelism in algebraic multigrid methods. *SIAM J. Sci. Comput.* 34, 4 (2012), C123–C152.
[13] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7. DOI: https://doi.org/10.1145/2024716.2024718
[14] John Adrian Bondy, Uppaluri Siva Ramachandra Murty et al. 1976. *Graph Theory with Applications.* Vol. 290. Citeseer.
[15] Urban Borštnik, Joost VandeVondele, Valéry Weber, and Jürg Hutter. 2014. Sparse matrix multiplication: The distributed block-compressed sparse row library. *Parallel Comput.* 40, 5–6 (2014), 47–58.
[16] Sergey Brin and Lawrence Page. 1998. The anatomy of a large-scale hypertextual web search engine. *Comput. Netw. ISDN Syst.* 30, 1–7 (1998), 107–117.

[17] Aydin Buluc and John R. Gilbert. 2008. On the representation and multiplication of hypersparse matrices. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS'08)*. IEEE, 1–11.

[18] Aydın Buluç and John R. Gilbert. 2011. The combinatorial BLAS: Design, implementation, and applications. *Int. J. High Perform. Comput. Appl.* 25, 4 (2011), 496–509.

[19] Timothy M. Chan. 2010. More algorithms for all-pairs shortest paths in weighted graphs. *SIAM J. Comput.* 39, 5 (2010), 2075–2089.

[20] N. Chatterjee, M. O'Connor, D. Lee, D. R. Johnson, S. W. Keckler, M. Rhu, and W. J. Dally. 2017. Architecting an energy-efficient DRAM system for GPUs. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'17)*. 73–84. DOI: https://doi.org/10.1109/HPCA.2017.58

[21] Jan Ciesko, Sergi Mateo, Xavier Teruel, Xavier Martorell, Eduard Ayguadé, Jesús Labarta, Alex Duran, Bronis R. de Supinski, Stephen Olivier, Kelvin Li et al. 2015. Towards task-parallel reductions in OpenMP. In *Proceedings of the International Workshop on OpenMP*. Springer, 189–201.

[22] Steven Dalton, Luke Olson, and Nathan Bell. 2015. Optimizing sparse matrix–matrix multiplication for the gpu. *ACM Trans. Math. Software* 41, 4 (2015), 25.

[23] Timothy A. Davis and Yifan Hu. 2011. The university of florida sparse matrix collection. *ACM Trans. Math. Software* 38, 1 (2011), 1.

[24] Erik P. DeBenedictis, Jeanine Cook, Sriseshan Srikanth, and Thomas M. Conte. 2017. Superstrider associative array architecture: Approved for unlimited unclassified release: SAND2017-7089 C. In *Proceedings of the IEEE High Performance Extreme Computing Conference (HPEC'17)*. IEEE, 1–7.

[25] Bobin Deng, Sriseshan Srikanth, Eric R. Hein, Thomas M. Conte, Erik Debenedictis, Jeanine Cook, and Michael P. Frank. 2018. Extending Moore's law via computationally error-tolerant computing. *ACM Trans. Architect. Code Optim.* 15, 1 (2018), 8.

[26] Bobin Deng, Sriseshan Srikanth, Eric R. Hein, Paul G. Rabbat, Thomas M. Conte, Erik DeBenedictis, and Jeanine Cook. 2016. Computationally-redundant energy-efficient processing for y'all (CREEPY). In *Proceedings of the IEEE International Conference on Rebooting Computing (ICRC'16)*. IEEE, 1–8.

[27] Mehmet Deveci, Christian Trott, and Sivasankaran Rajamanickam. 2017. Performance-portable sparse matrix-matrix multiplication for many-core architectures. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW'17)*. IEEE, 693–702.

[28] Iain S. Duff, Albert Maurice Erisman, and John Ker Reid. 2017. *Direct Methods for Sparse Matrices*. Oxford University Press.

[29] Iain S. Duff, Michael A. Heroux, and Roldan Pozo. 2002. An overview of the sparse basic linear algebra subprograms: The new standard from the BLAS technical forum. *ACM Trans. Math. Software* 28, 2 (2002), 239–267.

[30] Tim Finkbeiner, Glen Hush, Troy Larsen, Perry Lea, John Leidel, and Troy Manning. 2017. In-memory intelligence. *IEEE Micro* 37, 4 (2017), 30–38.

[31] Vijay Gadepally, Jeremy Kepner, William Arcand, David Bestor, Bill Bergeron, Chansup Byun, Lauren Edwards, Matthew Hubbell, Peter Michaleas, Julie Mullen, et al. 2015. D4m: Bringing associative arrays to database engines. In *Proceedings of the High Performance Extreme Computing Conference (HPEC'15)*. IEEE, 1–6.

[32] Harvey L. Garner. 1959. The residue number system. In *Proceedings of the Western Joint Computer Conference*. ACM, 146–153.

[33] G. Georgy Adelson-Velsky and Evgenii Landis. 1962. An algorithm for the organization of information. In *Proceedings of the USSR Academy of Sciences*.

[34] John R. Gilbert, Steve Reinhardt, and Viral B. Shah. 2006. High-performance graph algorithms from parallel sparse matrices. In *Proceedings of the International Workshop on Applied Parallel Computing*. Springer, 260–269.

[35] Felix Gremse, Andreas Hofter, Lars Ole Schwen, Fabian Kiessling, and Uwe Naumann. 2015. GPU-accelerated sparse matrix-matrix multiplication by iterative row merging. *SIAM J. Sci. Comput.* 37, 1 (2015), C54–C71.

[36] Fred G. Gustavson. 1978. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Trans. Math. Software* 4, 3 (1978), 250–269.

[37] Song Han, Huizi Mao, and William J. Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. arXiv preprint arXiv:1510.00149.

[38] Song Han, Jeff Pool, John Tran, and William Dally. 2015. Learning both weights and connections for efficient neural network. In *Advances in Neural Information Processing Systems*. MIT Press, 1135–1143.

[39] Václav Hapla, David Horák, and Michal Merta. 2012. Use of direct solvers in TFETI massively parallel implementation. In *Proceedings of the International Workshop on Applied Parallel Computing*. Springer, 192–205.

[40] MKL Intel. 2007. Intel math kernel library. Retrieved from https://software.intel.com/en-us/mkl.

[41] Satoshi Itoh, Pablo Ordejón, and Richard M. Martin. 1995. Order-N tight-binding molecular dynamics on parallel computers. *Comput. Phys. Commun.* 88, 2–3 (1995), 173–185.

[42] Anirudh Jain, Sriseshan Srikanth, Erik DeBenedictis, and Tushar Krishna. 2018. Merge network for a non-von Neumann accumulate accelerator in a 3D chip. In *Proceedings of the IEEE International Conference on Rebooting Computing (ICRC'18).*

[43] Sang-Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu et al. 2018. GraFBoost: Using accelerated flash storage for external graph analytics. In *Proceedings of the ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA'18).* IEEE.

[44] Jeremy Kepner and John Gilbert. 2011. *Graph Algorithms in the Language of Linear Algebra.* SIAM.

[45] Peter M. Kogge and Shannon K. Kuntz. 2017. A case for migrating execution for irregular applications. In *Proceedings of the 7th Workshop on Irregular Applications: Architectures and Algorithms.* ACM, 6.

[46] Walter Kohn. 1996. Density functional and density matrix method scaling linearly with the number of atoms. *Phys. Rev. Lett.* 76, 17 (1996), 3168.

[47] H. Kwon and T. Krishna. 2017. OpenSMART: Single-cycle multi-hop NoC generator in BSV and Chisel. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'17).* 195–204. DOI: https://doi.org/10.1109/ISPASS.2017.7975291

[48] Aapo Kyrola, Guy E. Blelloch, and Carlos Guestrin. 2012. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12).*

[49] Colin Yu Lin, Ngai Wong, and Hayden Kwok-Hay So. 2013. Design space exploration for sparse matrix-matrix multiplication on FPGAs. *Int. J. Circ. Theory Appl.* 41, 2 (2013), 205–219.

[50] Weifeng Liu and Brian Vinter. 2014. An efficient GPU general sparse matrix-matrix multiplication for irregular data. In *Proceedings of the IEEE 28th International Parallel and Distributed Processing Symposium.* IEEE, 370–381.

[51] Xingyu Liu, Jeff Pool, Song Han, and William J. Dally. 2018. Efficient sparse-winograd convolutional neural networks. arXiv preprint arXiv:1802.06367.

[52] V. Manikandan, V. P. Muralikrishna, J. Ajayan, and V. S. Mohammed Riyas Deen. [n.d.]. Static carry skip adder designed using 22-nm strained silicon CMOS technology operating under wide range of temperatures. *Int. J. Eng. Tech. Res.* 8, 2 ([n. d.]).

[53] Huizi Mao, Song Han, Jeff Pool, Wenshuo Li, Xingyu Liu, Yu Wang, and William J. Dally. 2017. Exploring the granularity of sparsity in convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'17).*

[54] Kiran Matam, Siva Rama Krishna Bharadwaj Indarapu, and Kishore Kothapalli. 2012. Sparse matrix-matrix multiplication on modern architectures. In *Proceedings of the 19th International Conference on High Performance Computing (HiPC'12).* IEEE, 1–10.

[55] Asit K. Mishra, Eriko Nurvitadhi, Ganesh Venkatesh, Jonathan Pearce, and Debbie Marr. 2017. Fine-grained accelerators for sparse machine-learning workloads. In *Proceedings of the 22nd Asia and South Pacific Design Automation Conference (ASP-DAC'17).* IEEE, 635–640.

[56] Onur Mutlu and Lavanya Subramanian. 2015. Research problems and opportunities in memory systems. *Supercomput. Front. Innovat.* 1, 3 (2015), 19–55.

[57] Yusuke Nagasaka, Satoshi Matsuoka, Ariful Azad, and Aydın Buluç. 2018. High-performance sparse matrix-matrix products on Intel KNL and multicore architectures. arXiv preprint arXiv:1804.01698.

[58] Eriko Nurvitadhi, Asit Mishra, and Debbie Marr. 2015. A sparse matrix vector multiply accelerator for support vector machine. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'15).* IEEE, 109–116.

[59] CUDA NVIDIA. 2014. Cusparse library. *NVIDIA Corporation, Santa Clara, CA.*

[60] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. 2018. OuterSPACE: An outer product-based sparse matrix multiplication accelerator. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'18).* IEEE, 724–736.

[61] Md Mostofa Ali Patwary, Nadathur Rajagopalan Satish, Narayanan Sundaram, Jongsoo Park, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, Sergey G. Pudov, Vadim O. Pirogov, and Pradeep Dubey. 2015. Parallel efficient sparse matrix-matrix multiplication on multicore platforms. In *Proceedings of the International Conference on High Performance Computing.* Springer, 48–57.

[62] Michael O. Rabin and Vijay V. Vazirani. 1989. Maximum matchings in general graphs through randomization. *J. Algor.* 10, 4 (1989), 557–567.

[63] V. Nageshwara Rao and Vipin Kumar. 1987. Parallel depth first search. Part I. implementation. *Int. J. Parallel Program.* 16, 6 (1987), 479–499.

[64] B. Ramakrishna Rau. 1991. Pseudo-randomly interleaved memory. In *ACM SIGARCH Computer Architecture News*, Vol. 19. ACM, 74–83.

[65] Liam Roditty and Uri Zwick. 2008. Improved dynamic reachability algorithms for directed graphs. *SIAM J. Comput.* 37, 5 (2008), 1455–1471.

[66] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*. ACM, 472–488.

[67] Subhendu Roy, Mihir Choudhury, Ruchir Puri, and David Z. Pan. 2014. Towards optimal performance-area trade-off in adders by synthesis of parallel prefix structures. *IEEE Trans. Comput.-Aided Design Integr. Circ. Syst.* 33, 10 (2014), 1517–1530.

[68] Karl Rupp, Florian Rudolf, and Josef Weinbub. 2010. ViennaCL-a high level linear algebra library for GPUs and multi-core CPUs. In *Proceedings of the International Workshop on GPUs and Scientific Applications*. 51–56.

[69] Fazle Sadi, Larry Fileggi, and Franz Franchetti. 2017. Algorithm and hardware co-optimized solution for large SpMV problems. In *Proceedings of the IEEE High Performance Extreme Computing Conference (HPEC'17)*. IEEE, 1–7.

[70] Erik Saule, Kamer Kaya, and Ümit V. Çatalyürek. 2013. Performance evaluation of sparse matrix multiplication kernels on Intel Xeon Phi. In *Proceedings of the International Conference on Parallel Processing and Applied Mathematics*. Springer, 559–570.

[71] Viral B. Shah. 2007. *An Interactive System for Combinatorial Scientific Computing with an Emphasis on Programmer Productivity*. University of California, Santa Barbara, CA.

[72] Sriseshan Srikanth, Thomas M. Conte, Erik P. DeBenedictis, and Jeanine Cook. 2017. The superstrider architecture: Integrating logic and memory towards non-von Neumann computing. In *Proceedings of the IEEE International Conference on Rebooting Computing (ICRC'17)*. IEEE, 1–8.

[73] Sriseshan Srikanth, Paul G. Rabbat, Eric R. Hein, Bobin Deng, Thomas M. Conte, Erik DeBenedictis, Jeanine Cook, and Michael P. Frank. 2018. Memory system design for ultra low power, computationally error resilient processor microarchitectures. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'18)*. IEEE, 696–709.

[74] Sriseshan Srikanth, Lavanya Subramanian, Sreenivas Subramoney, Thomas M. Conte, and Hong Wang. 2018. Tackling memory access latency through DRAM row management. In *Proceedings of the International Symposium on Memory Systems*. ACM, 137–147.

[75] Peter D. Sulatycke and Kanad Ghose. 1998. Caching-efficient multithreaded fast multiplication of sparse matrices. In *Proceedings of the 1st Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (IPPS/SPDP'98)*. IEEE, 117–123.

[76] Richard Wilson Vuduc and James W. Demmel. 2003. *Automatic Performance Tuning of Sparse Matrix Kernels*. Vol. 1. University of California, Berkeley.

[77] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2016. Learning structured sparsity in deep neural networks. In *Advances in Neural Information Processing Systems*. MIT Press, 2074–2082.

[78] Ichitaro Yamazaki and Xiaoye S Li. 2010. On techniques to improve robustness and scalability of a parallel hybrid linear solver. In *Proceedings of the International Conference on High Performance Computing for Computational Science*. Springer, 421–434.

[79] Raphael Yuster and Uri Zwick. 2004. Detecting short directed cycles using rectangular matrix multiplication and dynamic programming. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, 254–260.

[80] Zhao Zhang, Zhichun Zhu, and Xiaodong Zhang. 2000. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*. ACM, 32–41.

Received April 2019; revised June 2019; accepted August 2019