SortCache: Intelligent Cache Management for Accelerating Sparse Data Workloads

SRISHESHAN SRIKANTH, ANIRUDH JAIN, and THOMAS M. CONTE, Georgia Institute of Technology
 ERIK P. DEBENEDICTIS, Zettaflops, LLC
 JEANINE COOK, Sandia National Laboratories

Sparse data applications have irregular access patterns that stymie modern memory architectures. Although hyper-sparse workloads have received considerable attention in the past, moderately-sparse workloads prevalent in machine learning applications, graph processing and HPC have not. Where the former can bypass the cache hierarchy, the latter fit in the cache. This article makes the observation that intelligent, near-processor cache management can improve bandwidth utilization for data-irregular accesses, thereby accelerating moderately-sparse workloads. We propose SortCache, a processor-centric approach to accelerating sparse workloads by introducing accelerators that leverage the on-chip cache subsystem, with minimal programmer intervention.

CCS Concepts: • **Computer systems organization** → **Architectures**; • **Hardware** → **Memory and dense storage**; • **Theory of computation** → Theory and algorithms for application domains;

Additional Key Words and Phrases: Sparse matrix, intelligent cache, graph analytics, spgemma, pruned convolutions

ACM Reference format:

Sriseshan Srikanth, Anirudh Jain, Thomas M. Conte, Erik P. DeBenedictis, and Jeanine Cook. 2021. SortCache: Intelligent Cache Management for Accelerating Sparse Data Workloads. *ACM Trans. Arch. Code Optim.* 18, 4, Article 56 (September 2021), 24 pages.
<https://doi.org/10.1145/3473332>

1 INTRODUCTION

Systems software and hardware architectures traditionally leverage patterned data streams to achieve low memory access latency. However, this predictability is not present for many critical workloads including sparse data applications, which have received considerable attention for several decades [32]. The memory access irregularities in sparse applications may be (i) inherent in the algorithm, such as graph analytics [12, 14, 29] and HPC [24, 35, 37], or (ii) induced by domain experts as a computational performance optimization, such as pruned deep convolutional

New paper, not an extension of a conference paper.

Authors' addresses: S. Srikanth, A. Jain, and T. M. Conte, Georgia Institute of Technology, 266 Ferst Dr, School of Computer Science, Georgia Institute of Technology, Atlanta, GA 30332; emails: seshu6392@gmail.com, {anirudh.j, conte}@gatech.edu; E. P. DeBenedictis, Zettaflops, LLC, 1415 Canyon Rim Dr. NE, Albuquerque, NM 87112; email: erikdebenedictis@gmail.com; J. Cook, Sandia National Laboratories, P.O. Box 5800, Albuquerque, NM 87185; email: jeacock@sandia.gov.

This work is licensed under a Creative Commons Attribution International 4.0 License.

© 2021 Copyright held by the owner/author(s).

1544-3566/2021/09-ART56 \$15.00

<https://doi.org/10.1145/3473332>

```

for i, j, k = ...:
    partial = a[i] * b[j] // Sparse reduction on c
    c[k] = c[k] + partial // key=k, value=partial

```

Fig. 1. High-level view of *sparse reduction*, a kernel commonly seen in sparse data applications.

Fig. 2. Percentage of useful words in a cache block load (higher is better) on average.

networks in modern machine learning workloads [6, 34, 57] that are able to tolerate strategically *dropped* computations.

Through an Amdahl analysis on real hardware, we found that each of the sparse applications we analyzed spends a *significant amount of time* (49–90%) performing the *gather-apply-scatter* transformation, incrementally, on sparse data. In this transformation, the index address of the gathered data is commonly identical to that of the scattered data, meaning that the access pattern of the gather-apply-scatter transformation is equivalent to *sparse reduction*. A high-level view of this sparse reduction is shown in Figure 1 and is the focus of this article.

Modern vector processors, such as Intel’s Icelake, are capable of reading an entire cache block in parallel [36]. Furthermore, the capacity and bandwidth of on-chip caches has been on the rise and is predicted to continue to increase [33]. However, state-of-the-art implementations of important sparse data applications are unable to effectively utilize this high cache bandwidth available, as shown in Figure 2.

Only about 10% of the words are utilized per cache block access for unvectorized sparse reductions, on average. Even hand-vectorized sparse reductions, such as in pruned convolutions (SparseConv), use less than 50% of the words per block access.

This thus wastes bandwidth, requires masking, and results in function unit stalls, causing lower performance and increased energy consumption. Therefore, a majority of sparse applications leave performance on the table either without explicit vectorization or a mechanism to exploit the caches’ internal bandwidth. An alternative to relying on the programmer is hardware acceleration that pays the smaller cost of smart data re-organization to avoid the greater cost of under-utilization.

Recent research on hardware acceleration for sparse reductions [19, 38, 41, 60, 61] has focused on near-DRAM approaches. These not only face issues that are endemic to stand-alone acceleration, i.e., programmability and generality, but also that of hardware integration, because processor and memory system vendors are often separate entities. While the latter are commodity driven (i.e., capacity first with performance second), the former are more centrally focused on application performance. Furthermore, these accelerators bypass the cache hierarchy as they target hyper-sparse

data that have a very low cache hit ratio. However, an important class of sparse applications, such as those evaluated in this article, exhibit a high cache hit ratio. Therefore, utilizing these near-DRAM accelerators for such moderately sparse workloads is inefficient, because off-chip accesses are more expensive than on-chip accesses, even if the high on-chip bandwidth is under-utilized.

There are significant advantages to designing an accelerator that augments a general-purpose processor rather than deploying a stand-alone accelerator. These advantages come in the form of ease of adoptability, integration as well as improved generality. Adopting increasingly diverse fixed-function and programmable hardware into the traditional datapath has been prevalent and successful for decades, ranging from floating point computations to vector extensions to tensor augmentations [55]. *This article makes the observation that near-processor cache management via intelligent data marshalling can improve bandwidth utilization for data-irregular accesses, thus accelerating moderately sparse workloads.*

Sparse reductions as defined above can be viewed as a stream of (key, value) pairs (*kv*-pairs), where the key is equivalent to a matrix index, and the value is its associated non-zero value. An effective method of maximizing cache bandwidth utilization is to *extract* as much sequential locality as possible from this sparse stream of data. This is more commonly done in the slower, block-structured forms of memory using an extract-by-construction paradigm, such as B⁺-trees or vectorized *n*-ary forests [15, 50, 61]. This article leverages the prior work to enhance on-chip cache subsystems via dynamic sparse-to-dense conversion based on **vectorized binary search trees (VBST)**.

A VBST, as its name suggests, extends a BST such that each node in the tree is a constant-sized, sorted vector of *kv*-pairs. Each node is associated with a *pivot* key, and these pivots in turn form a binary tree. Operations on a VBST are at the granularity of a node. The size of a node is set to be an integral multiple of a cache block (e.g., 64 bytes) to guarantee that VBST operations maximize the internal and external cache bandwidth.

Using VBSTs as a framework, we propose *SortCache*, a near-processor approach to accelerating sparse workloads. The key contributions of this article are as follows:

- (1) SortCache is the first work capable of maximizing cache bandwidth utilization across a broad range of deep learning, graph analytics and HPC workloads, thanks to its ability to perform parallel sparse reductions efficiently.
- (2) As an augmentation to on-chip cache subsystems, SortCache is an easily integrable sparse workload accelerator that is minimally intrusive, does not disrupt the processor to memory interface, and requires little programmer intervention.
- (3) Unlike prior near-memory approaches, SortCache can accelerate moderately sparse workloads, since it is implemented on chip, closer to the cores.

Using a diverse set of workloads, our experiments show that SortCache, when compared to an aggressive baseline, is able to accelerate vector workloads by 75% on average, and unvectorized workloads by an average of 2.2×, while also reducing energy consumption by 31% for the former and 9.8× for the latter.

Paper outline: Section 2 summarizes the challenges and opportunities in accelerating sparse reductions. Section 3 provides an overview of VBSTs and Section 4 describes in detail the design of a VBST-based SortCache microarchitecture as well as a presentation of its easy system integration. Section 5 describes how a broad range of important real-world sparse applications are accelerated using SortCache as they are manifestations of sparse reductions. Sections 6 and 7 present the experimental methodology and empirical evaluations, respectively. Section 8 provides a conclusion with possible future work.

Table 1. Summary of Advantages and Limitations of Prior Work on Accelerating Sparse Reductions

Type	Approach	Primary advantage	Primary impediment to efficient sparse reduction
Sparse representation	Implicit	Mostly-dense data	Redundant storage and data movement
	Ordered COO (vector) [25]	Sparse gather	Data movement for sparse scatter
	Ordered COO (list) [44]	No explicit data movement	Requires traversing the entire structure for every operation potentially
	Unordered COO [44]	Constant time insertion	
	CSR / CSC [31]	Sparse gather, lower storage for several sparse patterns	Data movement for sparse scatter unless performed in key order
	DCSR / DCSC [13]		
Sparse to dense hardware convertor	GraFBoost [41]	DRAM row conscious	Unable to handle incremental updates, data-agnostic all-all comparison
	SpiDRE [9]	Sparse gather	Sparse scatter not supported
	PHI [53]	Lower DRAM traffic	Scalar updates don't leverage cache b/w
In-memory accelerators [4, 26, 27, 58]		Reduced data movement	Requires higher level data layout strategy
Specialized On-chip accelerators [67]		Reduced data movement	Not general purpose

2 BACKGROUND AND RELATED WORK

2.1 Anatomy of Sparse Reductions

Sparse reductions consist of three components: sparse gather from a given key (read $c[k]$ from cache in Figure 1), apply an associative reduction operation to the key's value ($+$ partial), and sparse scatter to the key (write $c[k]$ to cache).

Therefore, sparse reductions are a tightly-dependant chain of three operations per instance. In the absence of parallelism within a sparse reduction, the upshot is that latency of a cache word access is on the critical path. A promising approach to amortizing this is to leverage two characteristics: (i) Sparse reductions across different keys are independent and parallelizable, and (ii) Modern vector processors can access data at a cache block granularity in a single cycle thanks to increasingly higher cache bandwidth interfaces. SortCache is able to leverage both of these, i.e., *parallel reductions* as well as *high cache bandwidth*.

The remainder of this section captures how prior work tackles aspects of the sparse reduction problem, their limitations, and how some of them complement SortCache. A summary is available in Table 1.

2.2 Software-based Data Reorganization

For dense data applications, compiler assisted data transformations can improve cache locality [21, 42]. Other approaches repurpose the cache real estate as a software-managed scratchpad memory to reduce redundant data transfers in the context of CPUs, GPUs, and embedded systems [8, 47, 52, 63].

For sparse data applications, prior work has relied on novel representation. There are several sparse representations that are widely used today [11, 13, 23, 44, 66]. These can be broadly classified as (i) implicit (e.g., dense array), (ii) coordinate (e.g., COO), and (iii) compressed (e.g., CSR). While zeros are typically no longer represented for sparse representations and this saves tremendous redundant storage and bandwidth, there are challenges to (ii) and (iii) as well. In particular, during the sparse scatter phase, these approaches require significant data movement unless the writes are performed in key order. This is because COO- and CSR-like representations are vector-based and insertion into the middle of a vector requires significant data copying [28]. This limitation can be overcome in two ways: (a) static allocation, requiring prior knowledge of the size of the input, which is either not possible or expensive, or (b) append-only insertions, requiring all push-based (outer-product) algorithms to being constrained to their often sub-optimal pull-based (inner-product) equivalents [56].

Therefore, while conventional sparse representations are optimized for sparse gather, they suffer from issues fundamental to sparse scatter, both of which are important to efficient sparse reductions. Furthermore, as these approaches are fundamentally scalar, they are unable to fully leverage *parallel reductions* and *high cache bandwidth*.

2.3 Hardware-based Transformation Accelerators

For dense data applications, like software-assisted transformations, hardware-based transformations of cache data layout to improve locality have also been proposed, in the form of reducing conflict and capacity misses using alternate indexing and address transformations [45, 62].

For sparse data applications, transformation-based hardware acceleration has gathered significant interest in the near-data (DRAM) processing community [19, 38, 41, 60, 61, 70]. Most of these approaches bypass caches and target hyper-sparse applications and are relatively harder to adopt by virtue of requiring off-chip computation.

HTA [68] and SPiDRE [9] are recent examples of cache-based sparse-to-dense hardware. These approaches extract cache locality and do not require off-chip computation. HTA targets irregular accesses that arise from hash tables, by making use of a cache-conscious data layout. This is similar to SortCache in that sequential locality is leveraged. However, no dynamic ordering among cache blocks is necessary for their target domains [46, 51]. SortCache's parallel merge unit and locality optimized data storage techniques give it the ability to outperform HTA that limits the location of a key-value pair to any location within a single cache line only. SPiDRE performs LLC prefetching via a programmer-specified rearrange function, therefore improving cache hit ratios and also useful cache bandwidth. Although this improves sparse-gather performance, sparse-scatter is not accelerated. However, SortCache can benefit from SPiDRE as it can potentially accelerate the front-end generation of partial products for **sparse-matrix by dense-matrix multiplication (SpMDM)**, **sparse-matrix by dense-vector multiplication (SpMV)**, and **sparse-matrix by sparse-matrix multiplication (SpGEMM)** (Section 5).

PHI [53] is an orthogonal approach to the sparse reductions problem. In particular, PHI focuses on push-based updates in the context of graph analytics, with an emphasis on reducing traffic to main memory rather than maximizing cache bandwidth utilization. It leverages temporal locality in keys via coalescing, i.e., by treating portions of the cache hierarchy as a write buffer, and further lowers main memory traffic via batched updates. However, during their in-cache update phase, which is fundamentally scalar in nature, unless the stream of sparse reductions has significant sequential locality to begin with (which, according to Figure 2, is not common unless the program was explicitly vectorized), PHI is unable to effectively utilize the wide cache bandwidth available in modern processors, and this is where SortCache excels. Therefore, PHI and SortCache, when used together can benefit from each other. Both approaches reduce synchronization overheads as multiple cores can afford stateless offloads of their sparse scatters. As an alternative to software tiling, SortCache can leverage PHI to reduce memory traffic. PHI can leverage SortCache for more efficient in-cache sparse updates.

GAMMA [67] belongs to the class of dedicated on-chip SpGEMM accelerators and relies on a combination of Gustavson's algorithm (row-wise product), on-chip SRAM-based storage for data reuse, and high-radix mergers for performing sparse reductions. Its primary drawback is its rigid reliance on Gustavson's algorithm due to the expensive partial-product SRAM cache, and large crossbars that feed its high-radix mergers. In contrast, SortCache is general purpose and can be as easily used with Gustavson's algorithm as the outer-product algorithm, and other techniques that generate sparse partial products.

Fig. 3. Example of a VBST with $K = 4$.

2.4 In-memory Accelerators

Another class of cache-based acceleration is in the form of in-cache [4, 26, 27] or near-cache computation [22, 49]. These approaches do not transform the data layout but instead perform useful computation via the cache hierarchy. Such computation either leverages emerging SRAM circuit technology (*bitline computing*) or introduces a co-processor to the cache hierarchy. Fundamentally, bitline computing [39, 43] activates multiple word-lines in parallel and infers an *and* (*nor*) operation by sensing the shared bitline (complement). However, without a higher-level strategy for data layout, such approaches are unable to single-handedly improve sparse reductions because of their inherent data-irregularity. However, SortCache has the potential to benefit from such parallel activations as it can accelerate the key-compare and value-reduce operations, thus making its preconditioning and merge units (Section 4.2) more efficient.

3 EXTRACTING SEQUENTIAL LOCALITY VIA VBST

SortCache re-orders sparse data during gather-scatter operations to extract sequential locality by intelligent construction. This mechanism is provided via VBSTs, inspired by n -ary forests proposed in the context of DRAM-based hypersparse accelerators [61]. This section presents the detailed operations that can be performed on VBSTs.

A VBST is defined by four static parameters: K , the maximum number of kv -pairs in a VBST node; sz , the size (bytes) of each kv -pair; N_0 , address of the root node; and op , the reduction operator (*add* in Section 5). Each node, N_i , logically consists of three *tag* fields in addition to the K kv -pairs: the pivot, p_i , and the addresses of the left and right sub-VBSTs. Two fundamentally critical invariants are enforced at each N_i to scalably extract sequential locality: (i) *All keys in its left sub-VBST are smaller than p_i , which is smaller than all keys in its right sub-VBST*, and (ii) *all of its kv -pairs are sorted by key*. These invariants can also be observed in Figure 3.

3.1 Insertion

To efficiently enforce the required invariants, a VBST is built incrementally over time. First, K unsorted kv -pairs (*records*) are supplied by the input as a batch to the VBST. Second, these records are propagated down a dynamic path through the VBST, from root to leaf. En route, reductions are performed in-situ, and duplicate keys are coalesced with their reduced value. If there are insufficient repeated keys, then a new leaf node is created. As a result, all the input records end up being written into the VBST in addition to sparse reductions. These are described via an example below.

Consider the state of the tree shown in Figure 3(a). Assume op is the $+$ operation and the records in the new batch from the input are $R_{input} = [(12, 3), (25, 8), (47, 42), (125, 7)]$. The records R_0 from the root node N_0 are *merged* with R_{input} to obtain a temporary vector $R_{temp} = [(12, 3), (13, 20), (18, 22), (25, 3 + 8), (37, 10), (47, 42), (125, 7)]$. This merge operation sorts by key the records in R_{input} and R_0 , while also performing a reduction operation (according to op) on the values of records with matching keys. After reduction, the redundant records are compressed out, resulting in a total length of at most $2K$ for R_{temp} .

Next, R_{temp} is partitioned by p_0 , into two vectors $R_{large} = [(25, 11), (37, 10), (47, 42), (125, 7)]$ and $R_{small} = [(12, 3), (13, 20), (18, 22)]$. Note that the terms R_{large} and R_{small} indicate the sizes of the resultant partitions, and not the relation of their entries to the pivot element; R_{large} is of length at least K ; and R_{small} is of length at most K . The partitions are re-adjusted if necessary such that R_{large} has length exactly K . In this example, R_{large} happens to be of length K so a re-adjustment is not necessary. Subsequently, R_{small} is written back to N_0 as R_0 . If the remaining keys in R_{large} are greater than or equal to p_0 , then the above procedure is repeated with the right sub-VBST (root at N_2)—else with the left sub-VBST (root at N_1)—with R_{large} as the new “ R_{input} ” to the sub-VBST. This is repeated until $R_{large} = \phi$, which may require creating a new leaf as part of the last step. In this example, the path taken is $[N_0, N_2, N_5]$. Upon leaf creation, at N_5 , p_5 is assigned as the median key of R_5 , resulting in the updated state of the VBST as shown in Figure 3(b).

Insertion of a K -batch visits each level of the VBST at most once, resulting in $\sim \lg \frac{|R_{stream}|}{K}$ nodes being visited per insertion, where R_{stream} is the number of unique records in the VBST at the time of this insertion. *Observe that as part of the insertion, sparse reduction is also performed, while maximizing the number of consecutive words accessed.*

On consumption of the entire input, a small fraction of duplicate records may exist outside of the dynamic paths chosen for each batch, such as the pairs $(12, 3)$ in N_0 and $(12, 13)$ in N_1 , but these are resolved via a simple depth-first-search after the input has been consumed. Furthermore, several applications are fundamentally tolerant to such partial reductions either due to their approximate nature (i.e., deep learning) or their iterative nature. This post-processing step is omitted for space limitations.

3.2 Lookup

If a record was gathered for the sole purpose of apply-scatter, as is the common case in the workloads evaluated, then that is a sparse reduction and is more efficiently performed via the insertion procedure above. A *Lookup* is a gather that is not followed by an apply-scatter. There are two variants of lookup seen in applications: *random* and *ordered*.

Random scalar lookup for the value associated with a key κ is achieved via a VBST walk from the root. If $R_0[0] \leq \kappa \leq R_0[K - 1]$, then κ , if it exists, exists in N_0 , and a binary search within N_0 may be performed, since R_0 is sorted. Else if $\kappa < R_0[0]$, then the scalar lookup procedure can be performed recursively the left sub-VBST. Else, when $\kappa > R_0[K - 1]$, a lookup can be performed on the right sub-VBST. Therefore, this visits at most $\lg N$ nodes, where N is the number of nodes

Fig. 4. An overview of a traditional cache augmented with SortCache hardware. For simplicity, two nodes are assumed to fit in the cache, and ways, tag compare, pre-decoders, mats, and so on, are omitted from the figure.

in the tree at the time of lookup. This can be improved further by storing the range and address information of each node in a separate compact data structure (see Section 4.1).

Ordered lookup is more straightforward and efficient. From Figure 3, observe that an in-order, depth-first-search traversal of the VBST starting at the left-most leaf results in records being read out in increasing key order.

3.3 Deletion

Although not necessary for the workloads evaluated in this article, we briefly discuss deletion. Batch deletion is useful when certain records need to be “aged out” for capacity constraints, such as for real-time intrusion detection in cybersecurity [5]. Records are deleted via the insertion procedure above where the values in R_{input} set to 0. During the merge step performed on R_{temp} , when a 0-value is detected, the reduction operation can then omit the associated key(s) entirely, or propagate them to a backup VBST via a write-back insertion procedure.

For the remainder of this article, VBST or SortCache operations refer to insertion, unless otherwise specified.

4 AUGMENTING TRADITIONAL CACHES FOR SPARSE REDUCTIONS

This section presents how the SortCache design implements a VBST. Figure 4 is a high-level overview of SortCache components from the perspective of a single cache.

4.1 Data Layout

Recall that the goal of SortCache is to make effective use of cache bandwidth by maximizing intra-cache-block locality. Since records within a VBST node are contiguously accessed, they should not cross cache block boundaries. Therefore, the records of a VBST node are logically mapped onto an integral multiple of cache blocks. The question of finding the right integral multiple is addressed below in Section 4.4.

The VBST itself is partitioned across the cache hierarchy either statically or dynamically. Observe that higher level nodes of the VBST are more likely to be accessed than the lower (leaf) level nodes. Therefore, nodes closer to the root must be installed higher up in the cache hierarchy. This can be achieved in two ways: (i) *Static*: Assign a logical node-index to each node via a top-down topological sort. Given L1, L2, and L3 cache capacities *available to SortCache* as C_1 , C_2 , and C_3 bytes, and a node size of $sz_{node} = sz_{record} \times K$ bytes, assign the first $\frac{C_1}{sz_{node}}$ nodes to L1, the next $\frac{C_2}{sz_{node}}$

nodes to L2, and the remaining to L3. (ii) *Dynamic*: Perform demand-based block eviction/install to occur at the granularity of a node. For this, the existing LRU tracking is re-purposed to operate at a node-granularity rather than at a block-granularity basis (recall that nodes span an integral multiple of blocks). The existing pseudo-LRU replacement policy implementation provides $\frac{S-1}{S} + \log_2 S$ (where S is the set-associativity of the cache) bits per cache block [3]. The replacement bits of the reserved partition of the cache hierarchy now operate on all the SortCache nodes in the partition. Even for a relatively small value of KK ($=64$) where $sz_{node} = 4$ bytes, and a typical 32 KB L1 cache (8-way set-associative with 64B blocks), the existing bit budget is sufficient. For the worst-case scenario where the entire cache is allocated to SortCache, the maximum possible nodes in the above L1 cache is 128, and the number of bits required for Node-level LRU is only 1,023 as against the available 3,968 bits.

Given the skewed access distribution of the nodes, the dynamic approach tends to mimic the static partitioning strategy described above, while also being able to account for unexpected regularity in the otherwise irregular sparse input stream. Node-level LRU counter operation is necessary, because it is absolutely essential that all records within a node migrate together. During SortCache operation, due to insitu reduction, holes towards the end of nodes may dynamically appear and disappear. Using regular LRU may therefore result in partial eviction and tracking these would result in more complexity than using LRU counters that operate at a node granularity.

The per-node *tag* fields (pivot, addresses of left and right children) are stored in the cache's tag store fields associated with the nodes. This is due to three reasons: (i) the traditional tag bits are unnecessary for hardware managed *kv*-pairs; (ii) tag access in parallel with data access, like in traditional caches, lowers access latency; and (iii) there is sufficient tag storage in traditional caches, as will be explained in more detail below, offering enough headroom for future SortCache implementations that may need more *tag* than that required by the VBSTs described in this article. For example, consider a typical $C_1/C_2/C_3$ of 32 KB/256 KB/16 MB (total of 16.28125 MB available for the VBST) and relatively small $K = 64$. Note that a smaller K results in a larger number of VBST nodes, requiring more number of bits to encode addresses of children as well as a higher number of *tag* instances—thus making this example a worst-case scenario. Assume $sz_{record} = 4$ bytes, the pivot key requires 2 bytes, $sz_{node} = 4 \times 64 = 256$ bytes, number of nodes $N = \frac{16.28125 \times 1024 \times 1024}{256} = 66688$. In practice, not all of the cache hierarchy would be available to SortCache as the non-reduction portions of the application would require some caching. Therefore, 16 bits (2 bytes) are sufficient to address all nodes for this example. In the implementation, the child addresses are not stored as node indices, but are encoded according to their 16-bit geometrical coordinate locations L in the cache hierarchy. L is of the form {cacheId, indexId, wayOffset}. With 8-way caches, a 3-bit wide wayOffset suffices. The width of indexId increases with cache size, i.e., from L1 to L2 to L3. The remaining MSB bits form the cacheId field, to help decode whether the node is in L1, L2, or L3. This results in the *tag* fields requiring 2 (pivot) + 2×2 (child addresses) = 6 bytes per node, or $\frac{32 \times 1024}{256} \times 6 = 768$ bytes, $\frac{256 \times 1024}{256} \times 6 = 6$ KB, $\frac{16 \times 1024 \times 1024}{256} \times 6 = 64$ KB for the *tag* fields of L_1 , L_2 , and L_3 respectively. Assuming 8-way caches, these are much smaller than the already available tag storage (2.875 KB, 21.5 KB, 1.15625 MB, respectively).

4.2 Merge Hardware

To maintain the two VBST invariants for efficient operation, efficient merge is a requirement (Section 3). Therefore, SortCache relies on explicit hardware support for merging the records of the input batch R_{input} , with those of a given node R_i (R_0 to start with).

Recall that R_i is always sorted, and the intermediate results R_{temp} are also sorted. This means that for the majority of operation, merging takes two sorted vectors as input, which is significantly more efficient than generic sorting. However, R_{input} is unsorted as no such data transformations

Fig. 5. SortCache serial merge and preconditioning hardware.

are assumed to be applied by the main application. Thus, *SortCache incrementally preconditions R_{input} (i.e., sorts the input batchwise) such that further merging is more efficient.*

One method to sort unsorted data is to break it in two and repeatedly merge it. This allows the same merge hardware used for merging sorted vectors to be used to sort R_{input} , with more iterations. If the merge hardware is replicated, then preconditioning on a given input batch can be done in parallel with a preceding series of merge operations (resulting from the previous input batch). Such replication results in lower hardware necessary than having a dedicated sorter, without lost performance. This is because a single preconditioning is typically followed by several downstream merge operations, potentially down to the leaf node, enabling preconditioning and subsequent merging to be pipelined.

Two variants for the underlying merge hardware are now presented, followed by the control logic. While the latter is distributed across the cache controllers, the merge hardware is assumed to be close to the L1 cache controller to save power as it is more likely to access records in L1.

4.2.1 Linear Merge. Since the fundamental merge operation is performed on two sorted vectors, it can be done in linear time. For example, say $R_{input} = \{(1, 1), (2, 24), (5, 42), (10, 35)\}$ and $R_0 = \{(2, 5), \dots\}$. First, the keys from (1, 1) and (2, 5) are compared. Since $1 < 2$, (1, 1) is written out to R_{temp} and the keys from (2, 24) and (2, 5) are compared. Next, a reduction is performed and (2, 24 op 5) is written out to R_{temp} . This process is repeated, advancing a record from either vector until all records are consumed.

R_{input} is buffered in a $2K$ -record sized FIFO (called *BUF*) with two input and one output port. *BUF* also doubles as the real estate for R_{temp} , with an advancing head pointer to separate the two. Recall from Section 4.1 that R_0 is housed in a K -dependent number of contiguous cache blocks.

Post the wordline, bitline and sense-amplifier operations, the **word select latch (WSL)** has a cache block's worth of records, which are read one-by-one by the ALU (i.e., compute unit that performs comparison and reduction *op*) and populated into *BUF* (see Figure 5). Therefore, WSL_{read}/BUF_{read} , ALU and BUF_{write} form pipeline stages that can operate without stalls for the lifetime of the current cache block, after which the entire process is repeated. At design time, SortCache examines the latencies of each stage and generates a two-stage or three-stage pipeline accordingly. At the end of each node-merge step, R_{small} is written back from *BUF* back to the most recently read cache blocks.

4.2.2 Parallel Bitonic Merge. Since the WSL loads a cache block in parallel, rather than serially reading from WSL, reading and merging in parallel using a bitonic merge network is possible. This is a classic power-performance tradeoff between the two approaches.

Fig. 6. SortCache parallel merge and preconditioning hardware with $C = 4$ as an example.

The bitonic merge network takes two K -record sized sorted vectors and passes them through a C -wide and $\lg C$ -deep bitonic network of ALUs, where $C < K$ (see Figure 6). The records are read in $\lg K$ strides of $K, \frac{K}{2}, \dots, 1$. For each stride, C records are routed to the $\lg C$ -deep network in a depthwise pipelined manner.

Preconditioning of R_{input} can be done by repeating the above operation, with 1 stride in the first iteration, 2 strides in the second iteration, $\dots$, $\lg K$ strides in the final iteration.

4.2.3 Control Logic. The control logic of SortCache is responsible for two major class of operations: (i) SortCache mode macro-level semantics, such as routing records to the appropriate cache (Section 4.1), setting/resetting node-level LRU/install/evict logic, way-partitioning and lazy writebacks (Section 4.3); and (ii) VBST operation micro-level semantics, such as routing records between cache blocks and merge networks above and dynamically deciding the root-to-leaf path to be taken after each node-merge (Section 3).

4.2.4 Cache Coherence. During the merge process, nodes may migrate from the VBST portion of one level of the cache hierarchy to another. When this happens, the cache controller initiates a series of block installs into the pre-reserved VBST portion of the target cache. SortCache does not generate any additional coherence requests for accessing the VBST portion of the caches (including private caches). It is up to the programmer to ensure that VBST instances across private caches are coherent. One way to achieve this is by performing reduce operations across VBST's in a log-hierarchical manner and preventing any intermediate reads until all records are accumulated in DRAM. More details of this operation are outside the scope of this article.

4.3 System Integration

SortCache operation can be triggered using a simple RED R_{key}, R_{value} instruction, where R_{key} and R_{value} are registers containing the key and value per record that is to be reduced. The sparse

```

loop:
  MUL R_value, R_a, R_b
  LD R_tmp, [R_c], offset
  ADD R_tmp, R_tmp, R_value
  ST R_tmp, [R_c], offset

```

Fig. 7. Pseudo-assembly for the sparse reduction kernel.

```

loop:
  MUL R_value, R_a, R_b
  MOV R_key, offset2
  RED R_key, R_value

```

Fig. 8. The RED instruction instructs SortCache to perform *gather-apply-scatter*. `offset2` is different from `offset` as it has to account for storage of keys alongside the values (rather than in a separate vector such as in CSR). If size of key and value are same, then `offset2 = 2*offset1`.

```

#pragma SortCache INIT K=256, sz=4, op='+'
#pragma SortCache RED base_pointer=c
for i, j, k = ...:
  partial = a[i] * b[j]
  c[k] = c[k] + partial
...
print(c)
#pragma SortCache END base_pointer=c

```

Fig. 9. Programmer annotations to leverage SortCache.

reduction kernel introduced in Figure 1 can then be transformed from the pseudo-assembly in Figure 7 to that in Figure 8 by a compiler (or in rare cases manually by a programmer). Pragma-style programmer annotations are used to specify static VBST parameters (K , sz , reduction operator), the base pointer and the scope of reduction, as shown in Figure 9. The compiler identifies these Pragmas and generates RED instructions, one per key-value pair. More details on the compiler's operation are outside the purview of this article. However, one possible implementation of such a toolchain operating on a high-level Python-like language can be found in Reference [69].

Although sparse reductions account for the majority of cache bandwidth usage, applications may need to retain a portion of the cache capacity for other memory accesses. Automatically determining the general fraction of cache capacity required for regular memory accesses is beyond the scope of this work. However, for the memory footprint and tile sizes in the workloads evaluated in this article, we found reserving the L1, half of L2 and half of L3 for sparse reductions is sufficient to prevent L3 evictions for the VBST, while also not introducing regular memory accesses onto the critical path of the application. Reserving portions of a cache can be done via static way-partitioning or address-based set-partitioning. In this article, all 8-ways of L1 and the last four ways of 8-way L2/L3 caches are assumed reserved for VBSTs (during SortCache operation). As the sequence of node accesses is available only one node at a time for a VBST, all the cache MSHRs are available to the other portions of the application (outside of the sparse reduction operation).

On encountering the first RED instruction, the portions of the cache hierarchy known to be available for SortCache operation are transformed into SortCache mode, wherein the modified

node-level LRU counters and install/evict logic are used for these reserved cache portions. Lazy flush (dirty writeback or clean invalidate) of L1 and the reserved ways of L2/L3 are also performed when nodes are installed into the cache. At the end of a RED region, rather than reverting to traditional cache mode immediately, contents of the reserved portions are retained so that downstream lookup operations can be performed on the VBST directly (Section 3.2). If no more operations on the reduced sparse data are necessary, as can be signaled by the SortCache END pragma, then reservations are lazily invalidated and released for regular cache operation.

4.4 Choosing K

Recall from Section 3 that increasing K , the number of kv -pairs in a node, reduces the average VBST depth and therefore increases performance. However, it also increases the amount of data to be merged on a per-node basis as well as the preconditioning (pre-sorting) of the input batch of records for efficient subsequent merging (Section 4.2).

The number of serial comparison operations grows as $O(K \lg K)$ for preconditioning and $O(K)$ for linear merge at a node. As there are (ideally) $O(\lg \frac{R_{stream}}{K})$ nodes that need to be visited per input-batch (R_{stream} is the number of unique records already in the tree at time of insertion), the cost to inserting a batch of records may be viewed, to a first order, as $O(K \lg K + K \log \frac{R_{stream}}{K})$. Upon performing a differential analysis over K , and assuming that the pre-conditioning is overlapped with (previous) merge series, it can be seen that a higher R_{stream} favors a higher K for high performance.

However, from an energy stand point, a higher value of K is more detrimental to the log-linear, buffer-heavy bitonic network when compared to the linear merger network. As such, our evaluations in Section 7.5 find that the linear mergers favor a higher K than the bitonic network.

5 SPARSE REDUCTIONS IN REAL-WORLD APPLICATIONS

The real-world applications evaluated in this article are expressions of *repeated* matrix-matrix multiplications of varying shapes and sparsities, all of which *require efficient sparse reductions*.

For example, graph analytics applications can be represented as $x_{i+1} = A^T x_i$ for a given iteration i , where A represents the edge weights as the adjacency matrix of the graph, and x_i is a column vector denoting properties of each vertex in the graph. With outer-product, the n th column of A^T is “multiplied” with the n th entry of x_i to generate the n th partial product column vector x_{i+1}^n . The final column vector for the next iteration is generated by “adding” or applying sparse reduction to each of these partial column vectors, i.e., $x_{i+1} = \sum_n x_{i+1}^n$. The nature of the multiplication operation, addition operation and value associated with an element of x are described in detail later in this section.

Upon examining the above more closely, one notices that partial product generation via outer product can be done efficiently due to packed sparse representations such as coordinate and compressed that are commonly used today. The same applies to VBSTs as well, as they merely build upon these representations at a higher granularity. For example, reading a non-zero column of A^T benefits greatly from sequential locality as the non-zeros are typically stored in a contiguous manner in all of these representations. A similar argument holds for reading x , as they are accessed in increasing order of packed non-zeros. As a result, the front-end generation of these partial products is efficient.

However, when one has to apply sparse reductions to these partials, since each column of A^T generates partials across a wide range of keys (i.e., one vertex may push to an arbitrary set of vertices in the graph), performing the gather-apply-scatter sequence (Section 2.1) of a sparse

```

for n in oc:
    for weight, (r, s) in nzw:
        offset = f(n, c, r, s, ifm.dim)
        for y in ofm.height_dim:
            for x in ofm.width_dim:
                ofm[n][y][x] += weight * ifm[offset]

```

Fig. 10. Sparse Convolution as a sparse-matrix-dense-matrix multiplication, adapted from Reference [57]. *oc*, set of output channels where *nzw* (set of non-zero-weights (sparse filters of dimension $R \times S$ each)) represented in compressed sparse row (CSR) format [44]; *ofm*, output feature map and *ifm*, input feature map; *offset* is a linear combination *f* of *n*, input channel (*c*), non-zero-weight index (*r, s*), and the *ifm* dimensions.

reduction results in a highly-irregular cache access pattern. Prior work has also arrived at a similar conclusion to demonstrate the relative importance of accelerating sparse reductions [41].

5.1 Deep Learning

Deep convolutional neural networks requires a large number of parameters (AlexNet has 60 million [48], for instance). The fully connected layers' parameters are primarily responsible for the memory footprint of these networks. The operations due to convolution layers' parameters are primarily responsible for computation time. Accordingly, *pruning* techniques [6, 34, 57] have been proposed with an objective of reducing memory footprint and/or improving performance, but result in moderately sparse structures, the latter of which is the focus of SortCache.

Highly compressed convolutions, or sparse convolutions, can be viewed as a SpMDM, as shown in Figure 10. More formally, $O(n, y, x) = \sum_{c=0}^{C-1} \sum_{r=0}^{R-1} \sum_{s=0}^{S-1} W(n, c, r, s) I(c, y + r, x + s)$, where there are *C* input channels and $R \times S$ filters. When *W* and *I* are flattened appropriately, the values at the same given (*y, x*)-position of all *N* output channels can be computed collectively as a sparse-matrix by dense-vector multiplication, which when applied to all output positions results in SpMDM described above [57].

Notice how this is similar to sparse reduction as depicted in Figure 1, where the key is a linear combination of *n, y* and *x*, and the value is the non-zero partial product obtained by multiplying the corresponding filter weight with the input feature map.

5.2 Graph Analytics

While graph analytics is a very broad domain, this article focuses on three important applications that span various instances of sparse reductions. Each of these applications can be expressed as a generalized SpMV problem with different “multiply” (partial product generation) and “add” (reduce) operations. The sparse matrix here is the transpose of the adjacency matrix, and the vector is an application-dependent state vector spanning all vertices of the graph. SpMV maps directly to sparse reductions in that the key is the output vector index and the values are the non-zero partial products corresponding to that vector index.

There are two approaches to SpMV: inner-product and outer-product. While the former is better for locality of output vector updates (*add*), the latter is better for input vector reads (*mul*). From a graph analytics perspective, these are sparse-pull-dense-push (or just *pull*) and dense-pull-sparse-push (or just *push*), respectively. Whether to push or pull or employ a hybrid is an active research area [10]. Push is particularly attractive, because it reduces locality-based inefficiencies, performance-ambiguities and redundancies in reading the input vector, especially in algorithms where not all vertices are active in a given iteration, as is demonstrated by GraphMat [64], a well-cited software framework for easily-programmable, matrix-based graph analytics.

The drawbacks of push are poor locality and potential synchronization costs with multiple pushers for sparse reductions. However, hardware accelerators that explicitly manage data (such as References [41, 53, 61]) are able to overcome both of these drawbacks, at various levels of the memory hierarchy. Finally, these graph problems are repeated applications of SpMV, and the reduction operators are associative. This allows for asynchronous processing of sparse reductions, which SortCache leverages in a manner that maximizes effective cache bandwidth utilization. In summary, push-based SpMV is efficient in gathering the input and offers sparse reductions as the bottleneck, making it a prime candidate for hardware acceleration.

5.2.1 Breadth-First Search. BFS begins at a given source vertex and iteratively explores all vertices in its component. From an SpMV perspective, the state vector maintains the first parent node obtained during traversal. The vertex index of such a parent node is specified as a result of the *mul* operation (assignment), and the *add* operation (conditional assignment) assigns the parent node to the corresponding element in the state vector if and only if it was previously unassigned.

5.2.2 Single-Source Shortest Path. SSSP is an example of another iterative-search application. In addition to performing graph traversal, it computes the minimum cumulative distance at each frontier. Formally, at each iteration, for a vertex v , $d(v) = \min_{u \in N^-(v)} \{d(u) + w(u, v)\}$, where d is the current minimum distance to a given vertex from the source vertex, w is the edge weight, and $N^-(v)$ is the incoming neighborhood of v . From an SpMV perspective, the state vector represents the distance to each vertex from the source. The *mul* operation is summation (of distance to neighbor and edge weight), and the *add* operation is the minimum operator (to find the current minimum distance at this iteration).

5.2.3 PageRank. PageRank computes the probability of a random walk through edges ending in a given vertex. Formally, at each iteration i , $P_{i+1}(v) = r + (1 - r) \sum_{u \in N^-(v)} \frac{P_i(u)}{|N^+(u)|}$, where r is a constant normalized randomness fraction, $N^-(u)$ and $N^+(u)$ are the incoming and outgoing neighbors of vertex u . From an SpMV perspective, the state vector maintains the PageRank of each vertex. Unlike BFS and SSSP, PageRank has a consistently dense state vector. The *mul* operation is division (of PageRank of incoming neighbor and its out-degree), and the *add* operation is summation (over all incoming neighbors).

The equivalence between graph analytics and sparse linear algebra is well studied [44], and similar transformations from graph to SpMV problems can also be found in Reference [41]. For brevity, initial conditions, convergence criteria, and so on, are omitted from the above.

5.3 High Performance Computing

Efficient generalized SpGEMM is critical to several HPC and graph analytics applications [54]. In a manner similar to SpMDM and SpMV described above, SpGEMM maps directly to sparse reductions in that the key is the output matrix index and the values are the non-zero partial products corresponding to that matrix index.

6 EVALUATION METHODOLOGY

6.1 Sparse Applications Evaluated

A wide variety of sparse data applications are simulated in this study, with multiple inputs used for each workload category introduced in Section 5, SparseConv, BFS, SSSP, PageRank, and SpGEMM. These are described here.

SparseConv: A caffe-based [40] implementation of pruned, deep convolutional neural networks, SkimCaffe [1, 57], was used as the framework for real-world sparse convolutions. Each of the four sparse CNNs that resulted from the pruned AlexNet [48] topology were considered

Table 2. Summary of Sparse Workloads Evaluated

Name	Application	Domain	Num Sparse Reductions	% non-zeros	Amdahl's %
conv2	Sparse Convolution	Machine Learning	8.2 G	14.4	55%
conv3			1.7 G	6.9	74%
conv4			1.2 G	8.2	61%
conv5			1.2 G	11.5	75%
2cubes_sphere	BFS, SSSP, PageRank	Electromagnetics	1.5 M, 3.0 M, 49.5 M	6.6, 3.4, 0.2%	78, 74, 49%
ca-CondMat		Collaboration N/W	0.2 M, 0.2 M, 7.5 M	11.7, 9.2, 0.3%	71, 80, 64%
email-Enron		Email N/W	0.4 M, 0.7 M, 16.5 M	9.3, 4.9, 0.2%	67, 81, 53%
m133-b3		Combinatorics	1.6 M, 20.5 M, 48.01 M	12.5, 1.0, 0.4%	86, 77, 53%
mario002		2D/3D Problem	1.9 M, 4.4 M, 57.9 M	20.9, 8.8, 0.7%	65, 61, 57%
p2p-Gnutella31		Peer N/W	0.3 M, 0.5 M, 11.5 M	21.2, 13.5, 0.5%	65, 90, 66%
patents_main		US Patent Citations	1.1 M, 1.7 M, 46.0 M	20.8, 13.8, 0.5%	75, 72, 60%
TSOPF_RS_b39_c7		Power N/W	0.5 M, 0.9 M, 17.1 M	3.0, 1.6, 0.1%	70, 51, 51%
wiki-Vote		Wiki Votes N/W	0.2 M, 0.4 M, 8.1 M	3.5, 1.8, 0.1%	51, 69, 51%
CAG_mat364		Combinatorics	586 K	10.2%	72%
ex2	SpGEMM	Fluid Dynamics	464 K	13.8%	68%
filter2D		Model Reduction	23 K	0.4%	67%
Journals		Undirected Graph	203 K	78.5%	66%
LeGresley_4908		Power N/W	237 K	0.1%	75%
mhda416		Electromagnetics	188 K	5%	63%
qc324		Electromagnetics	388 K	25.5%	66%
rotor2		Structural	85 K	1.7%	68%
Si2		Quantum Chemistry	113 K	3%	76%
Trefethen_700		Combinatorics	63 K	2.6%	74%
west0381		Chemical Process Sim	12 K	1.5%	75%

The number of sparse reductions refer to the number of input records presented to Sortcache.

The percentage of non-zeros refers to the ratio of non-zero elements in the corresponding input matrix. The

Amdahl's percentage refers to the contribution of sparse reductions to application overall runtime, excluding I/O.

(the one dense CNN was excluded), with inputs from the ImageNet ILSVRC-2012 dataset [20] provided to the network. Note that, among the evaluated workloads, this is the only category where *the framework included carefully hand-optimized vectorized code by domain experts*.

BFS, SSSP, PageRank: Graphmat [64], an SpMV-based engine, was used as the framework for graph analytics.

SpGEMM: The most work-optimal algorithm for SpGEMM, outer-product [56], was used. For the SpMV- and SpGEMM-based workloads, the input matrices were selected from the SuiteSparse repository [18]. This was guided by prior sparse research [17, 30, 56, 59] and by the goal to demonstrate SortCache on a wide spectrum of domains.

Table 2 summarizes several key characteristics of the workloads evaluated. Most notable is the relatively *high Amdahl's percentage of sparse reductions* across multiple domains. The Amdahl's analysis was performed via native execution on a 2.3-GHz, 4-issue AMD Opteron 6276 with 2 MB LLC per core.

Finally, evaluations are performed for single threaded execution. Since, the focus of this work is to maximize cache bandwidth utilization for irregular data access applications, and there are per core L1, L2, and L3 (per-core slices) caches, results from our evaluation will scale for per-core instances of SortCache. Across SortCache instance reductions can explicitly be performed by the programmer on these instances. The details of this operation are omitted from this article due to space constraints.

6.2 Simulation Infrastructure and Configurations

The sparse reduction kernel was evaluated using a cycle-accurate, trace-based simulator. Timing, power and area for the caches (baseline and SortCache) and buffers (SortCache) were modeled in detail using CACTI 7 [7]. Those for the computational logic (SortCache) were modeled using

Table 3. Baseline Configuration

Parameter	Value
Front-end Core	Infinite issue width
	Perfect memory disambiguation
Cache Policy	Inclusive, LRU replacement
Block Size	64 bytes
Granularity	1 byte to 64 bytes per access
L1	32 KB, 8-way, 10 MSHR [36]
	2 Read-ports, 1 Write-port
L2	256 KB, 8-way, 20 MSHR
L3	16 MB

Table 4. SortCache Configuration

Parameter	Value
Cache Policy	Static, Dynamic (Section 4.1)
Record Size (<i>sz</i>)	Application dependent
Records per Node (<i>K</i>)	64–8192
Node Size	$K \times sz$ bytes
Bitonic Network Size (<i>C</i>)	2–64
Reduction <i>op</i>	+, min, predicated assignment
Preconditioning and Merger Unit Architectures	Linear-Linear, Linear-Bitonic, Bitonic-Bitonic

15nm-based RTL synthesis results from Synopsys Design Compiler [16] using the FreePDK15 standard cell library [2].

Recall that SortCache focuses on extracting maximum cache-*hit* bandwidth for parallel reductions from modern vector processors that have wide cache interfaces. To isolate this, the simulator models a front-end core with infinite issue width and perfect memory disambiguation, connected to a typical cache hierarchy. Given the ubiquity of software prefetching and cache tiling in modern applications including those evaluated in this article [1, 57, 64], the simulation of the sparse reduction kernel assumes that they hit in the cache hierarchy, for both the baseline and SortCache. For the *baseline* (Table 3), the L1 has 1 write-port and 2 read-ports (true dual porting) with a 64 byte wide interface, as used in modern vector processors such as Intel’s Icelake [36]. For SortCache (Table 4), these parameters are retained, except that the caches have only 1 read-port (and 1 write-port) each. Specifics of cache partitioning/reservation for SortCache operation has been described in Section 4.3.

To simulate SortCache, the frameworks from Section 6.1 were modified according to Figure 9 to generate a dynamic RED instruction stream. Both static and dynamic VBST node placement strategies (Section 4.1) were evaluated. *K* was varied from 64 to 8,192 (log-scale). Exploring higher *K* values would have caused a node to not fit entirely in L1, whereas lower *K* values would have resulted in trees with too many levels to be practical. The record size (*sz*) was determined based on the desired key range to be encoded. Most workloads require 4 bytes in practice. The reduction *op* $\in \{+, \min, \text{predicated assignment}\}$ was found sufficient to evaluate all workloads. The preconditioning and merger unit was simulated as combinations of linear or bitonic mergers.

Fig. 11. Speedup of Sparse Convolutions and SpGEMM vs. the baseline.

Fig. 12. Speedup of BFS, SSSP, and PageRank vs. the baseline.

7 EXPERIMENTAL RESULTS

7.1 Finding Universally Beneficial SortCache Configurations

Before presenting results, we present details of the SortCache configuration that benefits all of the workloads while not being over-fitted for any particular workload.

First, in terms of the merger approach, the linear merger was sufficient to extract significant performance benefits for BFS-, SSSP-, PageRank-, and SpGEMM-based workloads. The more powerful bitonic merger was used for SparseConv, as it had to compete against a more aggressive baseline, which, as we discussed above, benefitted from optimized hand-vectorized code. Thus both mergers are included with SortCache.

Second, both the preconditioning unit and the merger unit were assumed to be identical (linear-with-linear, bitonic-with-bitonic), as increasing design complexity with heterogeneous units did not yield any significant benefits.

Third, a high $K = 8,192$ was set for the linear merger and a low $K = 128$ was set for the bitonic merger (with $C = 32$) (for more details see Sections 7.5, 7.6).

Finally, the dynamic node placement strategy (Section 4.1) was used as it resulted in the most number of higher-level cache hits (for more details see Section 7.7).

7.2 Performance Results

Figure 11 presents SortCache's speedup over the baseline, both for the sparse reduction kernels and the entire application for SparseConv and SpGEMM. Figure 12 presents those for BFS, SSSP and PageRank.

Referring to the results, all of the workloads achieve at least a 15% speedup, with the exception of SpGEMM/rotor (6%). This is because the optimal K for rotor is 2048, which would have yielded a 20% speedup. In general, each of the workloads have their own optimal point for K , depending upon the nature of the input, or more precisely, on the density distribution of the resulting VBST nodes. When reduction (repeated keys) manifests uniformly over time (batches) and space (keys),

Fig. 13. SortCache achieves significant energy savings due to its intelligent data management.

a large fraction of VBST nodes result in empty space in them, implying that a lower K would have been more efficient.

There are two solutions (whose details are beyond the scope of this article) to improve the performance of such tail workloads: (i) Periodically “defragment” the VBST and (ii) Leverage compiler-assisted inspector-executor strategies [65] in determining the optimum K for a given program phase.

7.3 Energy Results

Using the same overall SortCache configuration as above, Figure 13(a) shows the average normalized energy across workloads. The linear merger based design (used for BFS, SSSP, PageRank, SpGEMM) offers an order of magnitude lowering in energy consumption. The bitonic network used for SparseConv is also able to significantly lower energy even though it competes against the hand-optimized baseline.

The overheads due to accessing the *tag* fields, the compute logic and the buffer are insignificant in general (Figure 13(b)). Moreover, because of effective use of cache bandwidth (Figure 2), significant gains come from lower cache energy. For the bitonic network, there is a strong correlation between its relatively higher energy consumption (as compared to the linear merger) and its heavy internal buffer usage.

7.4 Area

The bitonic network requires an area of 0.26 mm^2 , of which 0.01 mm^2 is needed for the network of ALUs. This, similar to the energy consumption, speaks to the buffer-heavy nature of this design. The linear merger requires an area of 0.14 mm^2 (which is almost entirely due its BUF structure). As these networks are duplicated to perform pipelined preconditioning, a total area overhead of 0.79 mm^2 is required. This is under 5% of that of the cache hierarchy.

7.5 Impact of K

As discussed in Section 4.4, increasing K tends to improve performance, since it leads to fewer nodes accessed per insertion, thanks to a shallower VBST. However, the cost for increased K is a higher preconditioning and node-merge overhead. Given the range of K explored (Section 6.2), the benefits of the former outweigh the costs of the latter for the linear merger, as seen with the BFS-, SSSP-, PageRank-, and SpGEMM-based workloads in Figure 14(a).

However, for a similar balance to be achieved with the bitonic merger, the network throughput must be increased for higher values of K . A higher C improves the parallelism within the network, but stresses the already buffer-heavy design. With $C = 32$, an initial performance improvement

(a) Impact of K on performance.

(b) Impact of bitonic network size on performance.

(c) Effectiveness of VBST node placement strategy in terms of percentage of accesses to the cache hierarchy being L1 hits (higher is better).

Fig. 14. Sensitivity analysis of key SortCache parameters.

is seen when K is increased from 64 to 128 due to a shallower tree, but this rapidly drops as K is increased further due to higher node-merge latency.

7.6 Impact of the Bitonic Network Size (C)

Figure 14(b) shows the average performance improvement relative to the baseline for SparseConv, as a function of bitonic network size.

As discussed above, increasing C increases the parallelism available in the bitonic network. Initially, this provides performance benefits (at increased power), since a wider set of ALUs are able to operate in parallel. However, beyond $C = 32$, the performance quickly tapers off because of the routing and output buffers becoming bottlenecks. Another way of viewing these buffers is through the lens of a crossbar-like structure that routes records to and from the C ALUs. For these reasons, the sweet spot of $C = 32$ has been used for SortCache. A more complex design that deploys a hierarchical bitonic network (i.e., a merge tree of smaller bitonic networks) is a potentially scalable solution, albeit at higher area/energy costs. The study of such hierarchical structures with regards to SortCache has been left for future work.

7.7 Impact of Node Placement and Merger Replication

In deciding where to install a newly created leaf VBST node in the cache hierarchy, Section 4.1 introduced using either a dynamic node-LRU-based approach or a static approach where topologically higher nodes were placed higher in the cache hierarchy. A hybrid is also possible where only the topologically higher few nodes are placed in the L1, while L2 and L3 would resort to using a node-level LRU behavior, similar to the dynamic approach.

While the static and hybrid approaches would benefit a few workloads, the dynamic approach is better overall. This is because VBST access paths are non-uniform. Although the static approaches make strong intuitive sense in that the topologically higher nodes are more likely to be accessed,

once the access path lengths increase, the lower nodes need to be accessed (and often, for certain input patterns). The dynamic approach is able to promote these “hot” lower nodes in addition to ensuring that a majority of the topologically higher nodes (especially the root node) that are accessed most frequently remain in the L1, as seen in Figure 14(c).

It can further be observed that the L1 is able to satisfy a vast majority of SortCache accesses on average. The exception is SparseConv, which requires smaller node sizes because of its more expensive bitonic merger. As a result, we replicated neither the linear nor the bitonic mergers. However, if there is headroom in the area budget, then replicating these mergers at lower levels of the cache would reduce the energy cost of intra-chip data movement.

8 CONCLUSION

We propose SortCache, a near-processor approach to improving the performance of sparse workloads by introducing accelerators that leverage the on-chip cache subsystem. SortCache can work in parallel with the traditional processor to memory interface, and requires minimal programmer intervention. Cache bandwidth utilization is enhanced by leveraging vectorized binary search tree-based (key, value) pairing.

Sparse data applications present significant challenges to modern memory systems. Although hyper-sparse workloads have received considerable attention in the past, moderately-sparse workloads that form the core of many machine learning, graph processing and HPC workloads, have not. Where hyper-sparse workloads benefit from bypassing the cache hierarchy, the moderately-sparse workloads cannot, since they largely fit in the cache. SortCache uses intelligent cache management to improve bandwidth utilization for data-irregular accesses, thereby accelerating these critical, moderately-sparse workloads.

Using a diverse set of workloads, experiments show that SortCache, when compared to an aggressive baseline, is able to accelerate vector workloads by 75% on average, and unvectorized workloads by an average of 2.2 $\times$, while also reducing energy consumption by 31% for the former and 9.8 $\times$ for the latter.

Future work will explore the benefits of periodic node defragmentation and compiler-assisted selection of K along with using hierarchical merge structures. In addition, we plan to study dynamically partitioning the cache between SortCache and traditional caching used by the rest of the application.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their valuable time and feedback.

REFERENCES

- [1] Intel Labs. SkimCaffe. Retrieved on Aug 2019 from <https://github.com/IntelLabs/SkimCaffe>.
- [2] North Carolina State University. FreePDK. Retrieved on Sep 2019 from <https://www.eda.ncsu.edu/wiki/FreePDK15:Contents>.
- [3] A. Abel and J. Reineke. 2014. Reverse engineering of cache replacement policies in Intel microprocessors and their evaluation. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'14)*. 141–142. <https://doi.org/10.1109/ISPASS.2014.6844475>
- [4] S. Aga, S. Jeloka, A. Subramanian, S. Narayanasamy, D. Blaauw, and R. Das. 2017. Compute caches. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'17)*. 481–492. <https://doi.org/10.1109/HPCA.2017.21>
- [5] Karl Anderson and Steve Plimpton. 2015. *FireHose Streaming Benchmarks*. Technical Report. Sandia National Laboratory.
- [6] 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.

- [7] 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, Article 14 (June 2017), 25 pages. <https://doi.org/10.1145/3085572>
- [8] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, Mahesh Balakrishnan, and Peter Marwedel. 2002. Scratchpad memory: A design alternative for cache on-chip memory in embedded systems. In *Proceedings of the 10th International Symposium on Hardware/Software Codesign (CODES'02)*. IEEE, 73–78.
- [9] A. Barredo, J. C. Beard, and M. Moretó. 2019. POSTER: SPiDRE: Accelerating sparse memory access patterns. In *Proceedings of the 28th International Conference on Parallel Architectures and Compilation Techniques (PACT'19)*. 483–484. <https://doi.org/10.1109/PACT.2019.00056>
- [10] Maciej Besta, Michał Podstawski, Linus Groner, Edgar Solomonik, and Torsten Hoeftler. 2017. To push or to pull: On reducing communication and synchronization in graph computations. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*. 93–104.
- [11] 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.
- [12] 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.
- [13] 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.
- [14] Timothy M. Chan. 2010. More algorithms for all-pairs shortest paths in weighted graphs. *SIAM J. Comput.* 39, 5 (2010), 2075–2089.
- [15] Shimin Chen, Phillip B. Gibbons, Todd C. Mowry, and Gary Valentin. 2002. Fractal prefetching B+-trees: Optimizing both cache and disk performance. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*. ACM, 157–168.
- [16] Synopsys Design Compiler. Retrieved from <https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test.html>.
- [17] Steven Dalton, Luke Olson, and Nathan Bell. 2015. Optimizing sparse matrix–matrix multiplication for the gpu. *ACM Trans. Math. Softw.* 41, 4 (2015), 25.
- [18] Timothy A. Davis and Yifan Hu. 2011. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.* 38, 1 (2011), 1.
- [19] 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 High Performance Extreme Computing Conference (HPEC'17)*. IEEE, 1–7.
- [20] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 248–255.
- [21] Chen Ding and Ken Kennedy. 1999. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *ACM SIGPLAN Notices*, Vol. 34. ACM, 229–241.
- [22] Filipa Duarte and Stephan Wong. 2010. Cache-based memory copy hardware accelerator for multicore systems. *IEEE Trans. Comput.* 59, 11 (2010), 1494–1507.
- [23] Iain S. Duff, Albert Maurice Erisman, and John Ker Reid. 2017. *Direct Methods for Sparse Matrices*. Oxford University Press.
- [24] 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. Softw.* 28, 2 (2002), 239–267.
- [25] Iain S. Duff and John Ker Reid. 1979. Some design features of a sparse matrix code. *ACM Trans. Math. Softw.* 5, 1 (1979), 18–35.
- [26] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer, D. Sylvester, D. Blaauw, and R. Das. 2018. Neural cache: Bit-serial in-cache acceleration of deep neural networks. In *Proceedings of the ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA'18)*. 383–396. <https://doi.org/10.1109/ISCA.2018.00040>
- [27] Daichi Fujiki, Scott Mahlke, and Reetuparna Das. 2019. Duality cache for data parallel acceleration. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA'19)*. ACM, New York, NY, 397–410. <https://doi.org/10.1145/3307650.3322257>
- [28] John R. Gilbert, Cleve Moler, and Robert Schreiber. 1992. Sparse matrices in MATLAB: Design and implementation. *SIAM J. Matrix Anal. Appl.* 13, 1 (1992), 333–356.
- [29] 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.
- [30] Felix Gremse, Andreas Hoft, 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.

- [31] Fred G. Gustavson. 1972. Some basic techniques for solving sparse systems of linear equations. In *Sparse Matrices and Their Applications*. Springer, 41–52.
- [32] Fred G. Gustavson. 1978. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Trans. Math. Softw.* 4, 3 (1978), 250–269.
- [33] P. Hammarlund, A. J. Martinez, A. A. Bajwa, D. L. Hill, E. Hallnor, H. Jiang, M. Dixon, M. Derr, M. Hunsaker, R. Kumar, R. B. Osborne, R. Rajwar, R. Singhal, R. D'Sa, R. Chappell, S. Kaushik, S. Chennupati, S. Jourdan, S. Gunther, T. Piazza, and T. Burton. 2014. Haswell: The fourth-generation intel core processor. *IEEE Micro* 34, 2 (Mar. 2014), 6–20. <https://doi.org/10.1109/MM.2014.10>
- [34] Song Han, Huizi Mao, and William J. Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. arXiv:1510.00149. Retrieved from <https://arxiv.org/abs/1510.00149>.
- [35] 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.
- [36] Intel. 2019. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Technical Report. Intel. Retrieved from <https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf>.
- [37] 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.
- [38] Anirudh Jain, Sriseshan Srikanth, Erik DeBenedictis, and Tushar Krishna. 2018. Merge network for a non-von neu-mann accumulate accelerator in a 3D chip. In *Proceedings of the IEEE International Conference on Rebooting Computing (ICRC'18)*.
- [39] Supreet Jeloka, Naveen Bharathwaj Akes, Dennis Sylvester, and David Blaauw. 2016. A 28 nm configurable memory (tcam/bcam/sram) using push-rule 6t bit cell enabling logic-in-memory. *IEEE J. Solid-State Circ.* 51, 4 (2016), 1009–1021.
- [40] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe. *Proceedings of the ACM International Conference on Multimedia (MM'14)*. <https://doi.org/10.1145/2647868.2654889>
- [41] Sang-Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, et al. 2018. GraFBoost: Using accelerated flash storage for external graph analytics. In *2018 Proceedings of the ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA'18)*. IEEE.
- [42] Mahmut Kandemir, J. Ramanujam, and Alok Choudhary. 1999. Improving cache locality by a combination of loop and data transformations. *IEEE Trans. Comput.* 48, 2 (1999), 159–167.
- [43] Mingu Kang, Eric P. Kim, Min-sun Keel, and Naresh R. Shanbhag. 2015. Energy-efficient and high throughput sparse distributed memory architecture. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS'15)*. IEEE, 2505–2508.
- [44] Jeremy Kepner and John Gilbert. 2011. *Graph Algorithms in the Language of Linear Algebra*. SIAM.
- [45] Mazen Kharbutli, Keith Irwin, Yan Solihin, and Jaejin Lee. 2004. Using prime numbers for cache indexing to eliminate conflict misses. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture (HPCA'04)*. IEEE, 288–299.
- [46] Onur Kocerber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. 2013. Meet the walkers: Accelerating index traversals for in-memory databases. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 468–479.
- [47] Rakesh Komuravelli, Matthew D. Sinclair, Johnathan Alsop, Muhammad Huzaifa, Maria Kotsifakou, Prakash Srivastava, Sarita V. Adve, and Vikram S. Adve. 2015. Stash: Have your scratchpad and cache it too. In *ACM SIGARCH Computer Architecture News*, Vol. 43. ACM, 707–719.
- [48] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*. 1097–1105.
- [49] Patrick A. La Fratta and Peter M. Kogge. 2009. Design enhancements for in-cache computations. In *Proceedings of the Workshop on Chip Multiprocessor Memory Systems and Interconnects*, Vol. 1.
- [50] Yinan Li, Bingsheng He, Qiong Luo, and Ke Yi. 2009. Tree indexing on flash disks. In *Proceedings of the IEEE 25th International Conference on Data Engineering*. IEEE, 1303–1306.
- [51] Pall Melsted and Jonathan K. Pritchard. 2011. Efficient counting of k-mers in DNA sequences using a bloom filter. *BMC Bioinform.* 12, 1 (2011), 333.
- [52] Maryam Moazeni, Alex Bui, and Majid Sarrafzadeh. 2009. A memory optimization technique for software-managed scratchpad memory in GPUs. In *2009 Proceedings of the IEEE 7th Symposium on Application Specific Processors*. IEEE, 43–49.
- [53] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. 2019. PHI: Architectural support for synchronization-and bandwidth-efficient commutative scatter updates. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 1009–1022.

- [54] Yusuke Nagasaka, Satoshi Matsuoka, Ariful Azad, and Aydın Buluç. 2018. High-performance sparse matrix-matrix products on Intel KNL and multicore architectures. arXiv:1804.01698. Retrieved from <https://arxiv.org/abs/1804.01698>.
- [55] NVIDIA. 2019. Retrieved from <https://www.nvidia.com/en-us/data-center/tensorcore/>.
- [56] 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.
- [57] Jongsoo Park, Sheng Li, Wei Wen, Ping Tak Peter Tang, Hai Li, Yiran Chen, and Pradeep Dubey. 2016. Faster cnns with direct sparse convolutions and guided pruning. arXiv:1608.01409. Retrieved from <https://arxiv.org/abs/1608.01409>.
- [58] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. 1997. A case for intelligent RAM. *IEEE Micro* 17, 2 (1997), 34–44.
- [59] 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.
- [60] 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.
- [61] 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. Arch. Code Optim.* 16, 4 (2019), 35.
- [62] 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.
- [63] Vivy Suhendra, Tulika Mitra, Abhik Roychoudhury, and Ting Chen. 2005. WCET centric data allocation to scratchpad memory. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS'05)*. IEEE, 10.
- [64] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R. Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. 2015. Graphmat: High performance graph analytics made productive. *Proc. VLDB Endow.* 8, 11 (2015), 1214–1225.
- [65] Anand Venkat, Mary Hall, and Michelle Strout. 2015. Loop and data transformations for sparse matrix code. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 521–532.
- [66] Richard Wilson Vuduc and James W. Demmel. 2003. *Automatic Performance Tuning of Sparse Matrix Kernels*. Vol. 1. University of California, Berkeley.
- [67] Guowei Zhang, Nithya Attaluri, Joel S. Emer, and Daniel Sanchez. 2021. Gamma: Leveraging gustavson's algorithm to accelerate sparse matrix multiplication. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*. Association for Computing Machinery, New York, NY, 687–701. <https://doi.org/10.1145/3445814.3446702>
- [68] Guowei Zhang and Daniel Sanchez. 2019. Leveraging caches to accelerate hash tables and memoization. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'52)*. ACM, New York, NY, 440–452. <https://doi.org/10.1145/3352460.3358272>
- [69] Tong Zhou, Jun Shirako, Anirudh Jain, Sriseshan Srikanth, Thomas M. Conte, Richard Vuduc, and Vivek Sarkar. 2020. *Intrepydd: Performance, Productivity, and Portability for Data Science Application Kernels*. Association for Computing Machinery, New York, NY, 65–83. <https://doi.org/10.1145/3426428.3426915>
- [70] Q. Zhu, T. Graf, H. E. Sumbul, L. Pileggi, and F. Franchetti. 2013. Accelerating sparse matrix-matrix multiplication with 3D-stacked logic-in-memory hardware. In *Proceedings of the IEEE High Performance Extreme Computing Conference (HPEC'13)*. 1–6. <https://doi.org/10.1109/HPEC.2013.6670336>

Received June 2020; revised January 2021; accepted June 2021