SMASH: Co-designing Software Compression and Hardware-Accelerated Indexing for Efficient Sparse Matrix Operations

Konstantinos Kanellopoulos[1] Nandita Vijaykumar[2,1] Christina Giannoula[1,3] Roknoddin Azizi[1] Skanda Koppula[1] Nika Mansouri Ghiasi[1] Taha Shahroodi[1] Juan Gomez Luna[1] Onur Mutlu[1,2]

[1]ETH Zürich [2]Carnegie Mellon University [3]National Technical University of Athens

ABSTRACT

Important workloads, such as machine learning and graph analytics applications, heavily involve sparse linear algebra operations. These operations use sparse matrix compression as an effective means to avoid storing zeros and performing unnecessary computation on zero elements. However, compression techniques like Compressed Sparse Row (CSR) that are widely used today introduce significant instruction overhead and expensive pointer-chasing operations to discover the positions of the non-zero elements. In this paper, we identify the discovery of the positions (i.e., indexing) of non-zero elements as a key bottleneck in sparse matrix-based workloads, which greatly reduces the benefits of compression.

We propose SMASH, a hardware-software cooperative mechanism that enables highly-efficient indexing and storage of sparse matrices. The key idea of SMASH is to explicitly enable the hardware to recognize and exploit sparsity in data. To this end, we devise a novel software encoding based on a *hierarchy of bitmaps*. This encoding can be used to efficiently compress any sparse matrix, regardless of the extent and structure of sparsity. At the same time, the bitmap encoding can be directly interpreted by the hardware. We design a lightweight hardware unit, the Bitmap Management Unit (BMU), that buffers and scans the bitmap hierarchy to perform highly-efficient indexing of sparse matrices. SMASH exposes an expressive and rich ISA to communicate with the BMU, which enables its use in accelerating any sparse matrix computation.

We demonstrate the benefits of SMASH on four use cases that include sparse matrix kernels and graph analytics applications. Our evaluations show that SMASH provides average performance improvements of 38% for Sparse Matrix Vector Multiplication and 44% for Sparse Matrix Matrix Multiplication, over a state-of-the-art CSR implementation, on a wide variety of matrices with different characteristics. SMASH incurs a very modest hardware area overhead of up to 0.076% of an out-of-order CPU core.

KEYWORDS

sparse matrices, compression, hardware-software cooperation, accelerators, memory, efficiency, specialized architectures, linear algebra, graph processing

ACM Reference Format:
Konstantinos Kanellopoulos, Nandita Vijaykumar, Christina Giannoula, Roknoddin Azizi, Skanda Koppula, Nika Mansouri Ghiasi, Juan Gomez Luna, Taha Shahroodi and Onur Mutlu. 2019. SMASH: Co-designing Software Compression and Hardware-Accelerated Indexing for Efficient Sparse Matrix Operations. In *The 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52), October 12–16, 2019, Columbus, OH, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3352460.3358286

1 INTRODUCTION

Sparse linear algebra operations are widely used in modern applications like recommender systems [33, 49, 61], neural networks [29, 50], graph analytics [8, 12], and high-performance computing [10, 20, 24, 25, 34]. The matrices involved in these operations are very large in size and highly sparse, i.e., the vast majority of the elements are zeros. For example, the matrices that represent Facebook's and YouTube's social network connectivity contain 0.0003% [75] and 2.31% [45] non-zero elements, respectively. These highly sparse matrices lead to significant inefficiencies in both storage and computation. First, they require an unnecessarily large amount of storage space, which is largely occupied by zeros. Second, computation on highly sparse matrices involves a large number of unnecessary operations (such as additions and multiplications) on zero elements. The traditional solution to these inefficiencies is to compress the matrix and store only the non-zero elements, and then operate only on the non-zero values.

Prior works take two major approaches to designing such compression schemes. The first approach is to devise general compression *formats* or encodings [38, 44, 53, 68, 72, 86, 89, 93], such as CSR [53], COO [72], BCSR [38], and CSR5 [53]. Such formats essentially store the non-zero elements and their positions within the matrix using additional data structures and different encodings. Such encodings are general in applicability and are highly-efficient in storage, with high compression ratios. However, this approach involves repositioning and packing the non-zero values in the matrix, which leads to significant computation overheads that diminish the overall benefit. Determining the positions of the non-zero elements in the compressed encoding (i.e., **indexing**) requires a series of pointer-chasing operations in memory that, as we demonstrate, are highly inefficient in modern processors and memory hierarchies.

The second approach taken by prior work is to leverage a certain known structure in a given type of sparse matrix to avoid the cost of discovering the non-zero regions of the sparse matrix [7, 16, 30, 41, 42, 46, 77]. For example, the DIA format [7] is highly effective in matrices where the non-zero values are concentrated along the diagonals of the matrix. Specializing the compression scheme to patterns in the sparsity can be efficient in both computation and storage but it is highly specific to certain types of matrices and inapplicable when the structure and extent of sparsity are not known a priori.

Our **goal** in this work is to enable efficient and general sparse matrix computation with a technique that satisfies **three major requirements**: 1) high computation and storage efficiency by storing and operating on only non-zero elements; 2) minimal overheads from the compression scheme (e.g., efficient discovery of non-zero elements) and 3) generality and applicability to any sparse matrix, regardless of its structure or the extent of its sparsity.

Our **key idea** is a new hardware-software co-design where we enable the hardware to *recognize* and *exploit* the compression encoding used in software for any sparse matrix. This allows us to add hardware support for highly-efficient storage and retrieval of non-zero values in sparse matrices, avoiding the overheads of software indexing (requirement 2). Our software encoding is designed to maintain low storage requirements (requirement 1) and be generally applicable to any sparse matrix without any assumption of structure or extent of sparsity (requirement 3).

We propose SMASH (**S**parse **MA**trix **S**oftware/**H**ardware), a general hardware-software cooperative mechanism that efficiently compresses and operates on sparse matrices. The key construct behind SMASH is the use of efficiently encoded *hierarchical bitmaps* to express sparsity, where each bit represents a region of non-zero values. These bitmaps are recognized by both hardware and software. On the software side, sparse matrices of any form are flexibly encoded using our hierarchical bitmap representation. SMASH adapts to each matrix's sparsity characteristics by supporting multiple compression granularities throughout the bitmap hierarchy. On the hardware side, the bitmap representation allows us to use a lightweight hardware unit, the Bitmap Management Unit (BMU), to perform highly-efficient scans of the bitmap hierarchy. The BMU hardware enables low-cost indexing (that avoids expensive pointer-chasing lookups) and efficient sparse matrix computation.

To enable wide applicability, SMASH exposes five new instructions that enable the software to communicate with the Bitmap Management Unit. The new instructions enable efficient lookup of non-zero matrix regions, and are sufficiently rich to express a wide variety of operations on any type of (sparse) matrix.

We evaluate SMASH on four use cases: two sparse matrix kernels, Sparse Matrix Vector Multiplication (SpMV) and Sparse Matrix-Matrix Multiplication (SpMM), as well as two graph analytics applications, PageRank and Betweenness Centrality (BC). For our experiments, we use a collection of sparse matrices with varying structure and sparsity characteristics [19]. We compare SMASH to two state-of-the-art compression formats, CSR [53] and BCSR [38]. We find that SMASH improves average performance by 41.5% for SpMV and SpMM, across 15 matrices and by 20% for PageRank and BC, compared to a state-of-the-art CSR implementation [40]. We also compare the software-only version of SMASH against two state-of-the-art sparse matrix frameworks [1, 40] on a real system. We find that even with no hardware support, SMASH's bitmap encoding outperforms a state-of-the-art CSR implementation [40].

In this paper, we make the following **key contributions**:

- We show that discovering the positions of non-zero elements (*indexing*) is a key bottleneck in sparse matrix computation. We demonstrate that efficient indexing can boost the performance of sparse matrix operations significantly.
- We introduce SMASH, a hardware-software cooperative mechanism that enables the hardware to recognize and exploit the compression encoding used in software. SMASH consists of 1) a novel software encoding that uses a hierarchy of bitmaps to efficiently compress sparse matrices and 2) hardware support to enable highly-efficient indexing of sparse matrices that are compressed using SMASH's software encoding.
- We show how SMASH can efficiently compress sparse matrices with diverse structure and sparsity characteristics using the hierarchical bitmap encoding. We design and demonstrate the effectiveness of the Bitmap Management Unit (BMU) that efficiently buffers and scans the bitmap hierarchy in hardware to identify non-zero regions in the matrix. We introduce an expressive ISA that enables the flexible use of SMASH in a wide variety of sparse matrix operations.
- We evaluate SMASH on important sparse matrix kernels and graph analytics applications using a collection of matrices with diverse structure and sparsity. We find that SMASH provides significant performance improvements compared to state-of-the-art CSR implementations while incurring a very modest area overhead in a modern out-of-order CPU.

2 MOTIVATION

Sparse matrix operations are widely used in a variety of applications including sparse linear algebra [39, 69], graph processing [8, 12], convolutional neural networks (CNNs) [50], and machine learning (ML) [33, 49, 61, 97]. These applications involve matrices with very high sparsity, i.e., a large fraction of zero elements. Using a compression scheme is a straightforward approach to avoid unnecessarily 1) storing zero elements and 2) performing computations on them. To this end, a variety of sparse matrix representation formats (e.g., [38, 44, 53, 68, 72, 86, 89, 93]) have been proposed to compress the sparse matrix. The most widely used state-of-the-art format is Compressed Sparse Row (CSR) [53]. In this section, we describe the CSR format and demonstrate its inefficiency.

2.1 Compressed Storage Formats

The Compressed Sparse Row (CSR) format [53] is widely used in many libraries that involve sparse matrix operations [1, 23, 40, 87, 92]. It consists of three one-dimensional arrays: `row_ptr`, `col_ind`, and `values`. Given an $M \times N$ matrix, the `row_ptr` array is used to store (and determine) the number of non-zero elements per row; the `col_ind` array indicates the column indices of the non-zero elements; and the `values` array stores the values of only the non-zero elements. Discovering the position of a non-zero element in row i requires streaming through `col_ind` from `col_ind[row_ptr[i]]` up to `col_ind[row_ptr[i+1]]` to discover its column index in the row. Figure 1 illustrates an example of a compressed matrix using the CSR format. In this example, in order to discover the non-zero elements of row 1 (i.e., second row from the top) of A, we search in `col_ind` starting from index `col_ind[row_ptr[1] == 1]` up to `col_ind[row_ptr[2] == 3]`.

A variant of CSR is Compressed Sparse Column (CSC). CSC stores the elements in column-major order instead of row-major. The `col_ptr` array is used to store (and determine) the number of non-zero elements per column, the `row_ind` array holds the row indices of the non-zero elements, and the `values` array stores the values of the non-zero elements themselves.

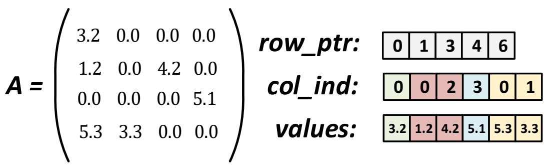

Figure 1: Compressed Sparse Row format for a 4×4 matrix with 6 non-zero elements. We count the number of non-zero elements of row i by computing (row_ptr[$i+1$]-row_ptr[i]). col_ind holds the column index of each non-zero element.

CSR significantly reduces the amount of memory needed to store a sparse matrix, especially when the matrix is large and its sparsity is high. However, CSR and schemes with CSR-like structures [53, 72] have one major requirement: in order to determine where the non-zero elements are located in the original matrix, the corresponding indices need to be retrieved from the row_ptr and col_ind data structures. Accessing these data structures adds many additional instructions and requires a series of indirect data-dependent memory accesses. These overheads reduce the benefits of avoiding the computation on zero elements. Hence, even though CSR-like formats reduce storage requirements and avoid needless computation, discovering the positions of the non-zero elements still is an unsolved challenge that causes performance and efficiency overheads.

2.1.1 Sparse Matrix Vector Multiplication (SpMV). We consider the SpMV kernel $y := y + Ax$, where A is a sparse matrix, x is a dense vector, and y is the output vector. The naive 2D implementation of the SpMV kernel involves performing computations on every element of the two-dimensional matrix A and incurs high computational and storage overheads. Code Listing 1 presents the CSR-based implementation. In this case, the algorithm iterates over only the non-zero elements and avoids unnecessary zero-element computations. However, it introduces a pointer-chasing operation when col_ind is loaded and then used as an index to load the appropriate element of the vector (x[col_ind[j]]). Only after this complex indexing operation can we perform the multiplication with the corresponding non-zero element in matrix A (values[j]), as shown in Line 3 of Code Listing 1.

```
1.  for (i = 0; i < N; i++)
2.     for (j = row_ptr[i]; j < row_ptr[i+1]; j++)
3.         y[i] += values[j] * x[col_ind[j]]
```

Code Listing 1: CSR-based SpMV implementation. The column index of each element is needed to perform the multiplication with the x vector.

2.1.2 Sparse Matrix Matrix Multiplication (SpMM). SpMM is traditionally performed using inner product multiplication [66]. This results in a series of dot product operations between each row of the first matrix (A) and each column of the second matrix (B) to produce the final elements of the result matrix (C). The naive $O(n^3)$ SpMM implementation is prohibitively expensive due to the high number of unnecessary computation operations on zero elements. A CSR-based implementation of SpMM, shown in Code Listing 2, avoids such unnecessary computations. In SpMM, matrix A is compressed using CSR and matrix B using CSC. SpMM iterates over the rows of A and columns of B (Lines 1-2 in Code Listing 2). For each non-zero element in each row of A, we need to search through col_ind of matrix A and row_ind of matrix B to discover which elements should be multiplied during each dot product. This process is called *index matching* (Lines 4-6 in Code Listing 2). Figure 2 presents an example of index matching. We need to match the positions of the non-zero elements of matrix A at row 0 and the non-zero elements of matrix B at column 0 to perform the dot product correctly. Given that index matching is performed for every dot product operation, a CSR-based SpMM implementation requires a large number of position-finding operations for non-zero elements and thus the indexing mechanism plays a critical role in performance and efficiency.

```
1.  for each row of A
2.     for each column of B
3.        for each non-zero element in row of A
4.           k1 = search_in_col_ind_of_A()
5.           k2 = search_for_row_ind_of_B()
6.           if (k1 == k2)
7.              y[i][j] += a_val[k1] * b_val[k2]
```

Code Listing 2: CSR-based inner-product SpMM implementation. Index matching (Lines 4-6) is needed to perform the multiplication of A's rows and B's columns (line 7).

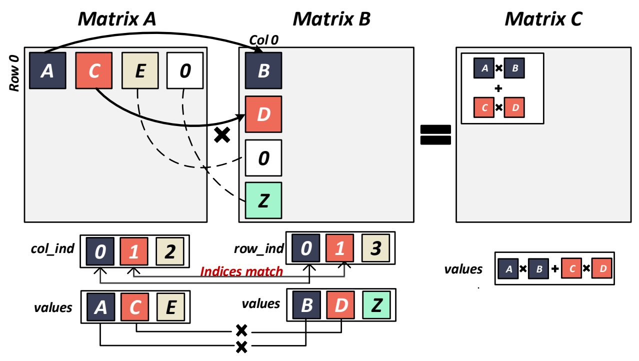

Figure 2: Index matching in SpMM. We search col_ind of matrix A and row_ind of matrix B to find indices that match. Only the indices of the first two elements of A's Row 0 and B's Column 0 match. The remaining two elements in A's Row 0 or B's Column 0 have at least one zero element in them and their indices do not match.

2.2 Limitations of Existing Compressed Storage Formats

Compressed storage formats, such as CSR [53], BCSR [38], and CSR5 [53], are effective at reducing the storage area and avoiding redundant computations on the zero elements of the matrix. However, as described above, they require additional computation and indirect memory accesses to find the indices, i.e., the row and column positions, of non-zero elements. This increases the computation burden and memory traffic, and hence lowers the potential gains from the compression scheme.

To understand the impact of this *indexing overhead* on sparse matrix processing, we conduct an experiment where we compare a state-of-the-art CSR implementation to an idealized version in which accessing the positions of non-zero elements does not incur any additional computation or memory access. Figure 3 shows the

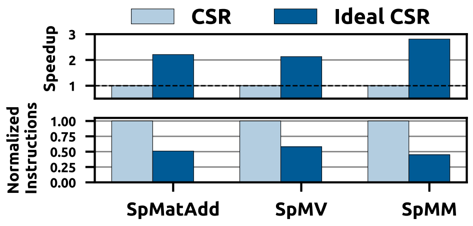

Figure 3: Speedup and normalized number of executed instructions of an ideal indexing scheme over the baseline CSR, averaged across 15 sparse matrices for Sparse Matrix Addition, SpMV, and SpMM (see Section 6 for methodology).

speedup and executed instruction count of this idealized CSR over the regular state-of-the-art CSR. As shown in the figure, eliminating the indexing part of the CSR format significantly improves performance: 2.21× for Sparse Matrix Addition, 2.13× for SpMV, and 2.81× for SpMM. These performance benefits come from the reduced number of executed instructions (by 49%, 42%, 65%, respectively) and from eliminating the expensive pointer-chasing lookups in memory.

2.3 Other Approaches to Sparse Matrix Compression

Other approaches [7, 16, 30, 41, 42, 46, 77] aim to maximize the efficiency of sparse matrix computations by specializing to particular matrix types and thus trading off generality. These approaches assume and leverage a specific matrix structure or known pattern. Saad et al. [70] assume a diagonal matrix and base the compression scheme around the assumption that all non-zero elements are only along the matrix diagonal. Kourtis et al. [41] assume matrices with few unique non-zero elements in designing the compression scheme. As a result, these approaches are not applicable to a wide range of important classes of applications like CNNs and graph processing algorithms, where such assumptions do not hold.

3 SMASH : DESIGN OVERVIEW

We introduce SMASH, a hardware-software cooperative mechanism that 1) enables highly-compressed storage and avoids unnecessary computation; 2) significantly reduces the overheads imposed by the compression scheme itself (i.e., enables efficient discovery of the positions of non-zero elements); and 3) can be used generally across a diverse set of sparse matrices and sparse matrix operations.

The **key idea** of SMASH is to enable the hardware to recognize and exploit the compression encoding used in software. We devise a new construct, recognized by both the hardware and software, to compress sparse matrices efficiently: *a hierarchy of bitmaps*. Each bitmap in the hierarchy efficiently encodes sparsity by denoting the presence/absence of non-zero values in a matrix region using a single bit. The size of the region varies with the level of the bitmap in the hierarchy and can be adjusted by software. This representation does not assume any structure in the matrix and can be used to efficiently compress a diverse set of sparse matrices. At the same time, as we demonstrate, it enables designing hardware that can exploit the known sparsity in data and hence perform highly-efficient indexing of sparse matrices.

3.1 Design Challenges

Designing SMASH involves addressing two major challenges:

Challenge 1: Efficiently Encoding Bitmaps. Representing each element in a matrix with a single bit to denote sparsity is highly-inefficient in terms of storage and computation. Hence, we need a more efficient bitmap representation that is effective regardless of the matrix sparsity and the location/distribution of the non-zero values. At the same time, hardware should be able to flexibly interpret and leverage this bitmap encoding.

Challenge 2: Flexibility and Expressiveness. To express a diverse set of sparse matrix operations in any application, we need a rich cross-layer interface between the application and the underlying hardware that 1) allows the software to flexibly manipulate and index sparse matrices encoded using our hierarchical bitmap encoding and 2) enables hardware to easily interpret the sparse matrix operations in the application and effectively accelerate those operations.

3.2 SMASH: Key Components

Hierarchy of Bitmaps. To address Challenge 1, we represent the positions of the non-zero values in any sparse matrix with a *hierarchy* of bitmaps. Our system is designed to support a certain maximum number of levels of the hierarchy. Each level of the hierarchy encodes the presence of non-zero values with a configurable compression ratio. This compression ratio is determined by the software based on the sparsity and distribution of non-zero values in a given matrix. With this representation, a zero matrix would require only one bit and one level of the bitmap encoding.

In Figure 4, we show an example with a 3-level bitmap hierarchy. Each bit in *Bitmap-2* encodes the presence of non-zero values in two consecutive regions in *Bitmap-1* (hence, the compression ratio at this level is two). *Bitmap-1*, on the other hand, encodes the presence of non-zero values in *four* consecutive regions in Bitmap-0 (hence, the compression ratio at this level is four). The selection of compression ratio at any level is a tradeoff between computation efficiency and storage efficiency. With a higher compression ratio, we store fewer bits to encode the presence/absence of non-zero elements but may perform unnecessary computations on zero elements. With a lower compression ratio, on the other hand, we can compute only on non-zero elements, but this would incur a higher storage overhead.

The non-zero elements of the matrix are kept in a data structure called the **Non-Zero Values Array (NZA)**. The granularity at which they are stored in memory depends on the compression ratio of the lowest level of the bitmap hierarchy (Bitmap-0). As we show in the example bitmap hierarchy of Figure 4, Bitmap-0 encodes the 4-element non-zero blocks of the NZA using a single bit (i.e., the compression ratio is 4:1).

Our hierarchical bitmap compression mechanism enables efficient representation of matrices with varying degrees of sparsity and varying distributions of non-zero elements by adjusting the bitmap representation granularity (i.e., compression ratios at all levels of the bitmap hierarchy). It can also be flexibly and efficiently interpreted by hardware, as we describe next.

Bitmap Management Unit (BMU). We design a new hardware unit that buffers and efficiently scans the bitmap hierarchy to quickly find the non-zero elements. The BMU is responsible for efficiently and quickly calculating the indices of the non-zero regions

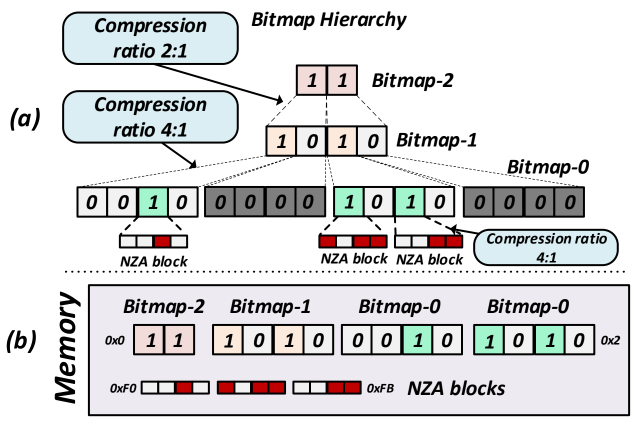

Figure 4: (a) A three-level bitmap hierarchy with different compression ratios between the levels. The NZA (Non-Zero Values Array) stores the non-zero values of the matrix. (b) We store in memory only the non-zero blocks of the bitmaps and the NZA.

in the matrix using the bitmap hierarchy. To this end, it caches and efficiently operates on the bitmaps in the hierarchy. Section 4.2 describes the BMU in detail.

Cross-layer interface. To flexibly accelerate a diverse range of operations on any sparse matrix, we expose to the software a rich set of primitives that 1) are general, 2) can control the BMU to quickly find the locations of non-zero elements, and 3) enable the processing core to skip unnecessary computations. These primitives are implemented as five new ISA instructions that are designed to 1) communicate the parameters of a sparse matrix and its bitmap hierarchy to the BMU, 2) load the bitmaps into the BMU, 3) scan the bitmaps to determine the location of the non-zero elements in the matrix, and 4) communicate the <row, column> positions of the non-zero elements in the original sparse matrix back to the application. These instructions are sufficiently expressive to be used for a diverse set of sparse matrix computations, such as Sparse Matrix Vector Multiplication, Sparse Matrix Matrix Multiplication, Sparse Matrix Addition, and more [39, 69], thereby addressing Challenge 2 (Section 3.1). Section 4.3 describes the SMASH ISA primitives in detail. Section 5 shows examples of how the primitives can be used in software applications.

4 SMASH: DETAILED DESIGN

In this section, we provide a detailed description of the different components of SMASH and their operation.

4.1 Software Compression (Hierarchical Bitmap Compression)

The hierarchy of bitmaps is the key construct of SMASH that enables 1) highly-compressed matrix storage and 2) efficient discovery of the positions of the non-zero regions of the matrix. The Non-Zero Values Array (NZA) holds the actual values of the sparse matrix. Every set bit of the last-level Bitmap-0 corresponds to one non-zero *block* in the NZA. The *size* of each non-zero block in the NZA (that is encoded by a single set bit in Bitmap-0) depends on the compression ratio used for Bitmap-0.

There are two major factors that impact the effectiveness of our hierarchical bitmap compression scheme: 1) the selected compression ratio at each level of the bitmap hierarchy (Section 4.1.1) and 2) the distribution of non-zero elements in the matrix (Section 4.1.2).

4.1.1 *Impact of compression ratio.*

Figure 5 demonstrates the impact of choosing different compression ratios for Bitmap-0 using a simplified example. We show two cases where the bitmap encodes the same 4×4 matrix using two different compression ratios between Bitmap-0 and the NZA. In case ❶, the bitmap uses a single bit to encode 8 consecutive elements of the matrix, i.e., the compression ratio is 8:1. The bitmap requires 2 bits to encode the entire matrix and the NZA holds one 8-element non-zero block (consisting of 2 non-zero and 6 zero elements). In case ❷, the bitmap uses a *single* bit to encode 4 consecutive elements of the matrix, i.e, the compression ratio is 4:1. In this case, the bitmap requires 4 bits to encode the entire matrix and the NZA holds one 4-element block (consisting of 2 non-zero and 2 zero elements).

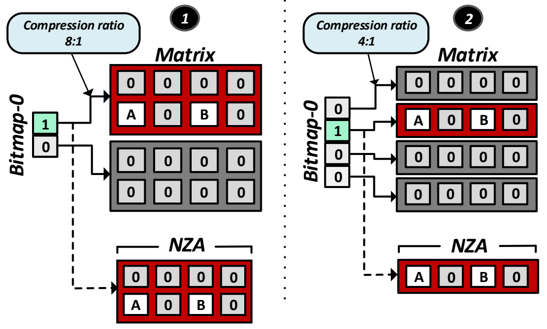

Figure 5: Two bitmaps that compress the matrix with two different compression ratios. Bitmap ❶ encodes blocks of 8 elements with a single bit, while Bitmap ❷ encodes blocks of 4 elements with a single bit.

As we demonstrate with this example, a higher compression ratio reduces the size of the bitmap and thus scanning it becomes more efficient. However, with a higher compression ratio, the NZA unnecessarily stores more zero elements. Since zeroes *within* the block cannot be identified a priori, the processor unnecessarily performs computation on them.

Hence, the compression ratio forms a tradeoff between 1) smaller bitmaps that can be quickly scanned and 2) more zero elements in the NZA storage and unnecessary computations on them. This major tradeoff also applies to the higher levels of the bitmap hierarchy.

4.1.2 *Impact of distribution of non-zero elements in the sparse matrix.*

The *distribution* of non-zero elements across the matrix also affects the size of the NZA. On the one hand, if the non-zero elements of the matrix are closely clustered, the number of non-zero blocks of the matrix decreases and the NZA stores fewer blocks. On the other hand, if the non-zero elements of the matrix are distributed more uniformly across the matrix, the number of non-zero blocks of the matrix increases and the NZA may need to hold more non-zero blocks (that also contain more zeros).

4.1.3 *Conversion to the hierarchical bitmap format.*

If the input data to any user application is already stored using another compression format (such as CSR), the application needs to convert the sparse matrices to the hierarchical bitmap encoding and the NZA used in SMASH. This is done using three steps. First, the application identifies all the non-zero blocks in the matrix using the indexing mechanism required by the original format. The size of the block depends on the assumed size of the non-zero blocks in the NZA. Second, the application creates the NZA by appending these non-zero blocks contiguously in memory. Third, the application creates the bitmap hierarchy, starting with Bitmap-0. To create Bitmap-0, the application determines the locations of the non-zero blocks of the matrix (where the block size is equal to the Bitmap 0 compression ratio). For each one of these locations, the application sets the corresponding bit of Bitmap-0 to 1. Next, the application creates the higher levels of the bitmap hierarchy: each level i of the hierarchy is created based on the compression ratio of Bitmap-i and the corresponding set bits of Bitmap-$(i-1)$. Assuming the chosen compression ratio of level i is 8:1, we set each bit in Bitmap-i if there are one or more set bits in the corresponding 8 elements in Bitmap-$(i-1)$.

We note that this conversion process from any format to our hierarchical bitmap encoding can be automated in software, and, if needed, accelerated with hardware support.

4.2 Hardware Indexing (Bitmap Management Unit)

The Bitmap Management Unit (BMU) provides the key functionality of scanning the bitmap hierarchy to quickly identify the non-zero elements of the matrix in a highly-efficient manner. It recognizes the bitmap encoding used in software based on the parameters of the bitmap (and sparse matrix) provided to it via the SMASH software-hardware interface (Section 4.3).

4.2.1 BMU components. Figure 6 demonstrates the structure of the BMU. It consists of four key components: 1) the SRAM buffers that hold the bitmaps when they are being scanned, 2) the hardware logic that scans the buffers to find the non-zero blocks, 3) programmable registers that hold configuration parameters, such as the matrix dimensions and the compression ratios of the bitmap hierarchy, and that effectively orchestrate the operation of the hardware logic, and 4) two registers to store the row and column indices of the non-zero elements determined by the BMU. The BMU supports multiple *groups* of the components presented in Figure 6, to enable the indexing of multiple sparse matrices, where each group is dedicated to buffering and scanning a single sparse matrix.

The SRAM buffers hold the bitmaps one block at a time. In our implementation, each buffer is 256 bytes in size. The compression ratio (the number of bytes encoded by a single bit) at each level of the bitmap, including between Bitmap-0 and the NZA, must be less than or equal to the bitmap buffer size. This is to avoid loading the buffers multiple times from memory to scan a single block, which would be expensive and inefficient. For example, with a 256-byte SRAM buffer size, the maximum compression ratio supported in the BMU is 256 × 8 = 2048:1.

4.2.2 BMU operation. As depicted in Figure 6, identifying the location of a non-zero block in the matrix involves three steps. First, the hardware logic scans the bitmap buffers to find the set bits and discover the positions of the non-zero blocks ❶. Second, the hardware logic reads the matrix dimensions and the compression ratios from the programmable registers to calculate the row and column indices of the current non-zero block ❷. Third, it updates the output registers ❸ that hold the row ❹ and column ❺ indices of the non-zero block. These registers can be repeatedly read by the CPU to iteratively find the location of the non-zero elements. To find the next non-zero element, the hardware logic looks for the next set bit in the bitmap block or loads the next bitmap block from the memory hierarchy to perform the search for the non-zero blocks. These operations are controlled by software with five new ISA instructions ❻, which we describe in Section 4.3.

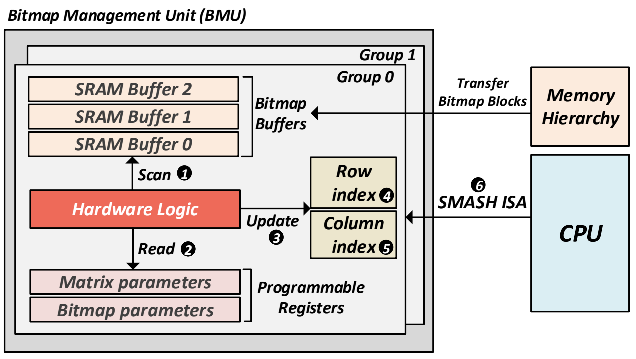

Figure 6: Bitmap Management Unit consists of four key components: 1) SRAM buffers to store portions of the bitmaps that are being operated on, 2) hardware logic that scans the buffers, 3) registers to hold the matrix and bitmap parameters, and 4) output registers to store the row ❹ and column indices ❺ of the non-zero blocks. The BMU communicates with the CPU using the SMASH ISA ❻.

4.2.3 Efficient indexing with the BMU. The BMU iteratively communicates to the CPU the row and column indices of the non-zero elements in the sparse matrix. Since we use multiple levels of indirection with the hierarchical bitmap, finding each non-zero element involves traversing the bitmap hierarchy in a depth-first manner. Every time a set bit is encountered at any bitmap level, we save that bit's index within the bitmap and then traverse the lower-level bitmap associated with that set bit. The BMU traverses the hierarchy in this manner (saving the index of the set bit at each level) until it reaches Bitmap-0. Any set bit in Bitmap-0 directly maps to a block of elements in the sparse matrix that has at least one non-zero element. Using the saved indices of the set bits at each level of the hierarchy, as well as the corresponding compression ratios, the BMU calculates the index of the non-zero block in the original sparse matrix.

The final index $= row\_index * matrix\_columns + column\_index$, is computed using the hierarchy of bitmaps in the following way: $Index = \sum_{i=0}^{levels-1}((\prod_{0}^{j=i} comp(j)) * index\_bit(i))$ where $comp(j)$ is the compression ratio of Bitmap-j and $index\_bit(i)$ is the index of the encountered set bit while scanning Bitmap-i. The row and column indices of the non-zero block in the original sparse matrix are calculated as follows: $row\_index = Index/matrix\_columns$ and $column\_index = Index\%matrix\_columns$. These indices are stored in the two output registers dedicated to the row index and the column index in the BMU. They are retrieved by the CPU using new ISA instructions (described below). The application iterates through the non-zero blocks of the NZA to compute on the non-zero values. For each consecutive non-zero block, the application reads the row and column indices from the BMU registers to identify the <row,column> location of the non-zero block in the original sparse matrix.

4.3 SMASH ISA: Software/Hardware Interface

We introduce five new instructions in the ISA to control the functionality provided by the BMU. Table 1 shows the instructions. These instructions are designed to i) communicate parameters of the matrix and the bitmap hierarchy to the BMU (`MATINFO,BMAPINFO`), ii) load the bitmaps into the BMU buffers (`RDBMAP`), iii) iteratively scan the bitmaps to determine the location of the next non-zero element in the matrix (`PBMAP`), and iv) communicate the row and column indices (in the original sparse matrix) of the non-zero element back to the application (`RDIND`).

`MATINFO`: This instruction communicates the dimensions of the matrices to the BMU. It has three source operands: `row,col,` and `grp`. `row` represents the number of rows and `col` represents the number of columns of the matrix. `grp` is used to select the group of the BMU. If the user application involves two sparse data structures, `MATINFO` needs to be executed twice (one for each matrix) before performing other operations with the BMU.

`BMAPINFO`: This instruction communicates the compression ratio of each bitmap to the BMU. It has three source operands: `comp` for the compression ratio, `lvl` selects the bitmap level in the hierarchy, and `grp` selects the group of the BMU.

Table 1: SMASH instructions

ISA instruction	Functionality
`matinfo row,col,grp`	Loads the matrix dimensions into the registers of the BMU.
`bmapinfo comp,lvl,grp`	Loads the compression ratio `comp` that bitmap at `lvl` operates with.
`rdbmap [mem],buf,grp`	Loads bitmap starting from `[mem]` into SRAM buffer `buf`.
`pbmap grp`	Signals the BMU to scan the SRAM buffers and find the row and column indices of the next non-zero block.
`rdind rd1,rd2,grp`	Loads into `rd1` and `rd2` the row and column indices of the current non-zero block that are stored in the row index and column index output registers of the BMU.

`RDBMAP`: This instruction loads the bitmap into the bitmap buffers in the BMU. It has three source operands: a `[mem]` location, a `buf` selector, and a `grp` selector. It loads a bitmap block starting from the address pointed by `[mem]` into the buffer `buf` of the group `grp` .

`PBMAP`: This instruction signals the hardware logic of the BMU to scan the bitmap buffers and find the index of the next non-zero block. The hardware logic updates the row index and column index output registers with the position of the non-zero block. This instruction has one source operand: `grp` selects the group of the BMU.

`RDIND`: This instruction communicates to the CPU the row and column indices of the current non-zero block that was identified by the BMU after the execution of `PBMAP`. This essentially involves reading the row index and column index output registers of the BMU. `RDIND` has two destination registers: `rd1` and `rd2`, and a `grp` selector. `RDIND` loads the row index and column index into `rd1` and `rd2`, respectively. `grp` is used to select the group of the BMU.

4.4 An Alternative: Software-only SMASH

The hierarchical bitmap encoding of SMASH can be used entirely in software, as a pure software compression mechanism without any hardware support. In this case, a sparse matrix is represented using the hierarchy of bitmaps but the indexing is still performed entirely in software, i.e., the BMU and the ISA are *not* used to accelerate the indexing. If used entirely in software, one 64-byte block of the bitmap needs to be loaded using four memory load instructions. A Count Leading Zeros (e.g., CLZ in x86) bitwise instruction is needed to find the first most-significant set bit. For every set bit that is found, one bitwise AND is needed to mask the set bit and then search for the next one. This adds more computation to find the set bits compared to the mechanism we describe in Section 4.2. In contrast to Software-only SMASH, the BMU loads the whole block at once and does not need AND operations to find set bits. As we show in Section 7.2, Software-only SMASH cannot leverage the full benefits of the hierarchical bitmap encoding and incurs additional computational overhead. Even so, as we also show in Section 7.2, Software-only SMASH still outperforms CSR on average, because it uses fewer instructions overall.

5 SMASH EXAMPLE USE CASES

We describe in detail how the SpMV and SpMM operations are performed using SMASH. We assume a 3-level bitmap hierarchy for SpMV and, for simplicity of explanation, a 1-level bitmap for SpMM. Our SpMM example consists of two sparse data structures.

5.1 Example Use Case 1: SpMV

Figure 7 and Algorithm 1 describe the execution flow for SpMV using SMASH. SpMV involves only one sparse matrix and a dense vector, x. Therefore, it utilizes only one group of the BMU's components. `MATINFO` is used in the beginning of the algorithm to communicate the dimensions of the matrices ❶ to the BMU (line 2 in Algorithm 1). `BMAPINFO` is executed once for each bitmap to communicate the compression ratios ❷ (lines 3-5). `RDBMAP` is executed three times at the beginning to load the bitmap hierarchy into the bitmap buffers ❸ (lines 6-8). Whenever a non-zero block is found, `PBMAP` is used to search in the SRAM buffers to find the row and column indices of the next non-zero block of the NZA ❹ (line 11) and `RDIND` returns these indices of the non-zero block back to the application (line 12). The processor loads the block from the NZA, and multiplies it with the x vector's block at the row index returned by `RDIND` ❺ (lines 15-16).

5.2 Example Use Case 2: SpMM

Figure 8 and Algorithm 2 describe the execution flow for SpMM using SMASH. We describe SpMM in the case where a 1-level bitmap hierarchy is used for each of the two sparse data structures. In the initialization phase of the algorithm, we need to execute `MATINFO` twice, one for each sparse matrix ❶ (lines 2-3 in Algorithm 2). We also need to execute `BMAPINFO` twice, one for each bitmap ❷ (lines 4-5). The program iterates over the rows of matrix A and columns of matrix B. For each row of matrix A, we load the bitmap at the correct offset using `RDBMAP` as follows: `[mem]`=*bitmapA+rowOffset*,

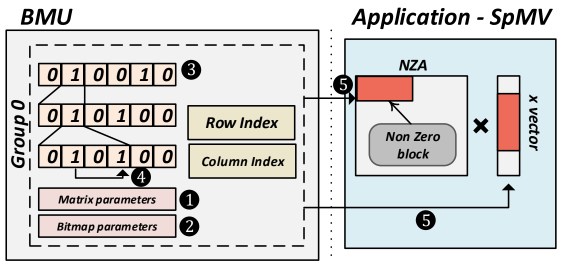

Figure 7: SpMV flow of execution. One matrix is compressed using a 3-Level Bitmap Hierarchy.

Algorithm 1 : SpMV using SMASH

```
1  # Operation: A * x = C
2  matinfo rows,columns,0 # Load dimensions to BMU
3  bmapinfo comp2,2,0
4  bmapinfo comp1,1,0 # Load compression ratios to BMU
5  bmapinfo comp0,0,0
6  rdbmap [bitmap2],2,0
7  rdbmap [bitmap1],1,0 # Load 3 bitmaps in BMU buffers
8  rdbmap [bitmap0],0,0
9  ctrNZ = 0 # Initialize counter of NZ blocks
10 for all non-zero blocks of the sparse matrix
11     pbmap 0 # Scan the bitmaps
12     rdind rowInd,colInd,0 # Read index of the NZ block
13     ctrElmt = 0 # Initialize counter of elements
14     for all elements of block (rowInd,colInd)
15         NZA_ind = ctrNZ*comp0 + ctrElmt
16         C[rowInd+ctrElmt]+=NZA[NZA_ind]*x[colInd+ctrElmt]
17         ctrElmt+=1 # Point to the next element
18     ctrNZ+=1 # Point to the next NZ block
```

where buf=0 and grp=0 ❸ (line 7). For each column of matrix B, we load the bitmap at the correct offset in the second group of buffers using RDBMAP as follows: [mem]= *bitmapB+colOffset*, where buf=0 and grp=1 (line 9). Index matching requires the use of the PBMAP instructions ❹ (lines 10-11) to search the bitmaps of both matrices and determine the matching indices of NZA_A's and NZA_B's blocks. RDIND instructions ❺ (lines 12-13) are executed to load the indices of the non-zero blocks into registers so that the column index of A can be compared to the row index of B (line 14). If the indices match, the algorithm performs the inner product between the corresponding blocks of NZA_A and NZA_B (line 15). If the row index of A is greater than the current row or if the column index of B is greater than the current column of B (line 16), the algorithm skips the remaining inner-product computation (lines 10-15) between the current row of A and the current column of B. This implies the absence of anymore non-zero elements in the current row of A (or column of B) and thus, unnecessary computation on zero elements is avoided.

The index matching phase of the algorithm requires calculating the indices of each non-zero element in both A and B for each inner product (line 10-16). A format like CSR would incur extra computations and expensive indirect memory accesses to perform this step (i.e., the index matching). With SMASH, we leverage the BMU to accelerate the index matching. We demonstrate the benefits of our scheme in Section 7.

5.2.1 *Generality of SMASH for other use cases.*

In this work, we evaluate SMASH using two sparse matrix kernels, SpMV and SpMM, that are central to many important applications (e.g., [50, 67, 83]). However, SMASH is generally applicable to *any* sparse matrix computation. Sparse matrix computations operate only on the non-zero elements of the matrix and hence fundamentally require 1) identifying the indices (row and column) of the non-zero elements and 2) retrieving those non-zero elements from memory. With the proposed ISA instructions, we can efficiently determine the location (in the matrix) of the non-zero blocks of the NZA, regardless of 1) the computation that will be performed on the non-zero elements and 2) the structure and the extent of sparsity in the matrix. Other examples of widely used operations on sparse matrices that SMASH can accelerate include Sparse LU Decomposition [69], Sparse Eigenvalue Calculation [21, 39] and Sparse Iterative Solvers [70].

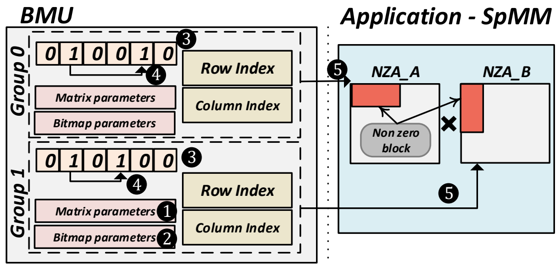

Figure 8: SpMM flow of execution. Two matrices are compressed, each using a single-level Bitmap Hierarchy.

Algorithm 2 : SpMM using SMASH

```
1  # Operation: A * B = C
2  matinfo rowsA,colsA,0 # Load dimensions to BMU
3  matinfo rowsB,colsB,1
4  bmapinfo comp0_A,0,0 # Load compression ratios to BMU
5  bmapinfo comp0_B,0,1
6  for i in [0 .. rowsA) # Iterate over rows of A
7      rdbmap [bitmapA+rowOffset],0,0 # Load bitmap_A
8      for j in [0 .. colsB) # Iterate over columns of B
9          rdbmap [bitmapB+colOffset],0,1 # Load bitmap_B
10         do:  pbmap 0 # Find next non-zero block
11              pbmap 1 # in both matrices
12              rdind rowIndA,colIndA,0 # Read NZ index in A
13              rdind rowIndB,colIndB,1 # Read NZ index in B
14              if (colIndA == rowIndB) # Index matching
15                 C[rowIndA][colIndB]+=inner_pr(NZA_A,NZA_B)
16         while (rowIndA < i) && (colIndB < j)
```

6 EXPERIMENTAL SETUP

We model and evaluate SMASH using the zsim simulator [71]. Table 2 provides the system configuration we evaluate. We simulate each workload until completion. We use an Intel Xeon system [2] to perform experiments on real hardware. Table 5 provides the configuration of the real system.

Workloads: Sparse Matrix Kernels. We evaluate SMASH using two sparse matrix kernels, Sparse Matrix Vector Multiplication and Sparse Matrix Matrix Multiplication. We use the TACO library's [40] respective implementations of these kernels as our baseline.

Table 2: Simulated system configuration

CPU 3.6 GHz, Westmere-like [43] OOO, 4-wide issue; 128-entry ROB; 32-entry LQ and SQ;	
L1 Data + Inst. Cache 32 KB, 8-way, 2-cycle; 64 B line; LRU policy; MSHR size: 10; Stride prefetcher;	
L2 Cache 256 KB, 8-way, 8-cycle; 64 B line; LRU policy; MSHR size: 20; Stride prefetcher;	
L3 Cache 1MB , 16-way, 20-cycle; 64 B line; LRU policy; MSHR size: 64; Stride prefetcher;	
DRAM 1-channel; 16-bank; open-row policy; 4GB DDR4;	

Table 3: Evaluated sparse matrices

	Name	# Rows	Non-Zero Elements	Sparsity (%)
M1:	descriptor_xingo6u	20,738	73,916	0.01
M2:	g7jac060sc	17,730	183,325	0.06
M3:	Trefethen_20000	20,000	554,466	0.14
M4:	IG5-16	18,846	588,326	0.17
M5:	TSOPF_RS_b162_c3	15,374	610,299	0.26
M6:	ns3Da	20,414	1,679,599	0.40
M7:	tsyl201	20,685	2,454,957	0.57
M8:	pkustk07	16,860	2,418,804	0.85
M9:	ramage02	16,830	2,866,352	1.01
M10:	pattern1	19,242	9,323,432	2.52
M11:	gupta3	16,783	9,323,427	3.31
M12:	nd3k	9,000	3,279,690	4.05
M13:	human_gene1	22,283	24,669,643	4.97
M14:	exdata_1	6,001	2,269,500	6.30
M15:	human_gene2	14,340	18,068,388	8.79

Table 4: Input graphs

	Graph	Vertices	Edges
G1:	com-Youtube	1.1M	2.9M
G2:	com-DBLP	317K	1M
G3:	roadNet-CA	1.9M	2.7M
G4:	amazon0601	403K	3.3M

Table 5: Real system configuration

CPU Intel Xeon Gold 5118 2.30 GHz 14nm [2]	
L1 384 KB, 8-way	
L2 12 MB, 16-way	
L3 16.5MB, 11-way	
Main memory DDR4-2400	

For input datasets, we use a diverse set of 15 sparse matrices from the Sparse Suite Collection [19]. The matrices have different sparsities and distributions of non-zero elements. The term sparsity refers to the fraction of non-zero elements in the matrix over the total number of elements. Table 3 presents these matrices, sorted in ascending order of their sparsity. We use the term M_i to refer to each matrix in Section 7. We open source SMASH's software implementations of these sparse matrix kernels [3].

Workloads: Graph Processing. We implement PageRank and Betweenness Centrality from the Ligra Benchmark Suite [9] as SpMV computation and evaluate the performance of SMASH over the default CSR-based versions of these two algorithms. PageRank [65] was first used by Google to rank website pages. Specifically, it takes as input a graph and computes the rank of each vertex, which represents the relative importance of each node (e.g., webpage). PageRank iteratively uses SpMV to calculate the ranks of nodes in the graph [91].

Betweenness Centrality [27] is a measure of the significance of each vertex based on the number of shortest paths that pass through it. Betweenness Centrality iteratively uses SpMV to perform breadth-first searches in the graph [9]. We evaluate these two workloads using a set of four graph inputs from the Sparse Suite Collection [19]. We use the term G_i to refer to each graph input in Section 7.

7 EVALUATION RESULTS

We evaluate five different mechanisms: *(i)* **TACO-CSR**: The CSR-based implementation from the TACO library [40]; *(ii)* **TACO-BCSR**: The BCSR-based implementation from the TACO library [40]; *(iii)* **MKL-CSR**: CSR-based SpMV and SpMM implementations from Intel MKL [1]; *(iv)* **Software-only SMASH**: SMASH's hierarchical bitmap encoding implemented purely in software without the BMU; and *(iv)* **SMASH**: our complete proposed scheme, with the hierarchical bitmap encoding and the BMU. The matrices we evaluate vary in sparsity and hence, for SMASH implementation of each matrix, we use different compression ratios in the bitmap hierarchy for SMASH . We denote the bitmap configuration of each matrix (and graph) i as $M_i.b2.b1.b0$, where M_i denotes the matrix id and $b2.b1.b0$ denotes the compression ratios of each level in the bitmap hierarchy. We evaluate 4 different use cases: SpMV, SpMM (Section 7.2), PageRank, and Betweenness Centrality (Section 7.3).

7.1 Software-only Approaches

We first compare the performance of three state-of-the-art sparse matrix formats (TACO-CSR [40], TACO-BCSR [40], and MKL [1]) and the hierarchical bitmap encoding used in SMASH (i.e., Software-only SMASH) on our real Intel Xeon system (Table 5). Our goals with this experiment are to 1) compare existing state-of-the-art software solutions to identify a baseline for our simulation experiments and 2) evaluate the performance of SMASH's hierarchical bitmap encoding *without* any hardware support. The TACO and MKL formats employ a range of software optimizations that are orthogonal to the matrix format itself and the indexing mechanism. To ensure a fair comparison, our implementation of Software-only SMASH includes all the software optimizations used by the TACO compiler, but uses SMASH's hierarchical bitmap encoding instead of CSR.

Figure 9 depicts the speedup of TACO-BCSR, MKL, and software-only SMASH, normalized to TACO-CSR, for SpMV and SpMM. We make two observations, averaged across the 15 matrices we evaluate. First, MKL outperforms TACO-CSR in both SpMV and SpMM by 15% and 25% respectively. MKL also outperforms TACO-BCSR, but by a smaller margin: 3% in SpMV and 4% in SpMM. While MKL uses the same CSR format as the TACO compiler, it also employs a range of proprietary software optimizations that lead to better performance than the TACO implementations. We can only compare to the TACO implementations in the simulation experiments as the additional optimizations in the closed-source MKL library cannot be added to SMASH (or to any other technique) to enable a fair and insightful comparison. Second, we observe that Software-only SMASH outperforms TACO-CSR by 5% in SpMV and 10% in SpMM, but is outperformed by both TACO-BCSR and MKL. Software-only SMASH incurs a performance overhead because 1) indexing the hierarchical bitmap *entirely* in *software* requires more instructions than indexing the CSR format and 2) unlike CSR, which

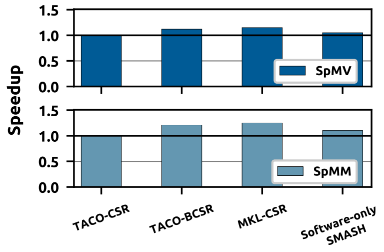

Figure 9: Performance of software-only approaches on a real Intel Xeon system (normalized to TACO-CSR).

eliminates all zeros, SMASH's hierarchical bitmap encoding may require unnecessary computations on zero values, depending on the choice of compression ratios in the bitmap hierarchy. However, despite these overheads, the hierarchical bitmap encoding avoids the expensive indexing and pointer-chasing operations in CSR and outperforms TACO-CSR.

7.2 Sparse Matrix Kernels

7.2.1 Performance Results. Figure 10 shows the speedup of TACO-CSR, TACO-BCSR, Software-only SMASH, and SMASH, normalized to TACO-CSR, for the SpMV kernel. Figure 11 shows the number of executed instructions in each mechanism, normalized to TACO-CSR. We make three observations.

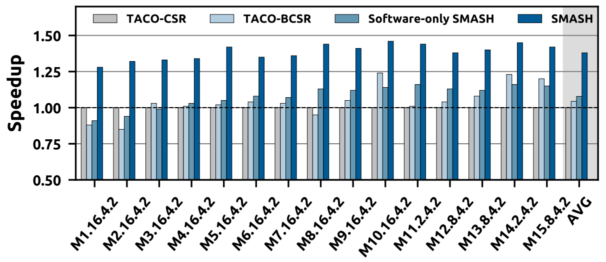

Figure 10: Speedup (normalized to TACO-CSR) for SpMV.

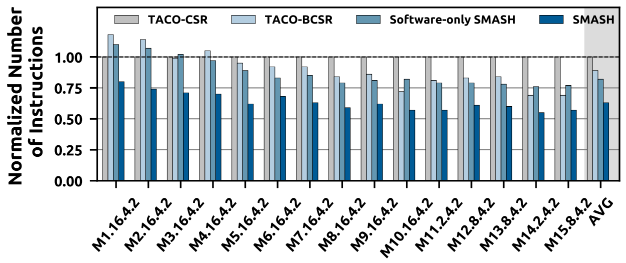

Figure 11: Number of executed instructions (normalized to TACO-CSR) for SpMV.

First, SMASH significantly outperforms all other mechanisms: it is 38% faster over TACO-CSR and 32% over TACO-BCSR, on average. SMASH's speedup is mainly due to executing fewer indexing instructions (47% less than TACO-CSR and 30% less than TACO-BCSR, on average) and avoiding expensive pointer-chasing operations in memory. Second, SMASH is highly effective regardless of the sparsity of the matrix. For the matrices with the highest sparsity in our evaluation (M_1 and M_2), TACO-BCSR is inefficient because it encodes data in blocks, which leads to unnecessary computation on zero elements. SMASH avoids such overhead by leveraging the configurability of compression ratios in our hierarchical bitmap encoding to adapt the compression ratio to the matrix sparsity. Third, we observe that the effectiveness of Software-only SMASH depends on the *sparsity* of the matrix. Software-only SMASH incurs a higher performance overhead when the sparsity is higher because Software-only SMASH performs more unnecessary computation on zero elements and executes more instructions searching the bitmaps for non-zero bits. In these cases, TACO-CSR outperforms Software-only SMASH. When the matrix is denser, the benefits of avoiding pointer-chasing and indirect indexing outweigh the additional computation and search operations on zero elements. Thus, Software-only SMASH outperforms both TACO-CSR and TACO-BCSR for less sparse matrices. This strong impact of sparsity on overall performance is not seen as strongly in SMASH because the hardware support makes the indexing highly efficient even when the matrix is extremely sparse. As a result, SMASH outperforms all other approaches regardless of the matrix sparsity.

Figures 12 and 13 depict the speedup and the number of executed instructions, respectively, for the SpMM kernel.

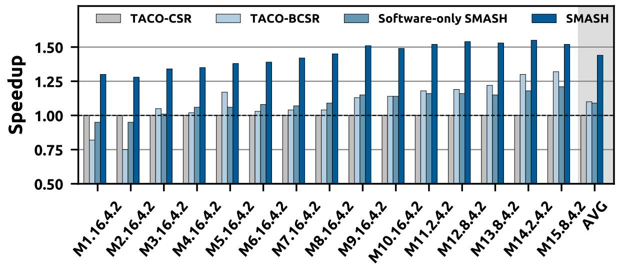

Figure 12: Speedup (normalized to TACO-CSR) for SpMM.

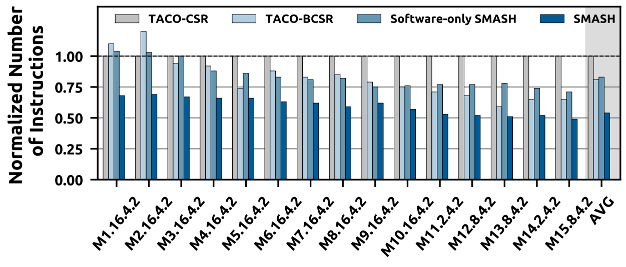

Figure 13: Number of executed instructions (normalized to TACO-CSR) for SpMM.

We observe the same trends and make the same observations for SpMM as we do for SpMV, but with an important difference. SpMM requires *twice* the number of indexing operations as SpMV per dot product. As a result, the performance benefits of SMASH are even higher over TACO-CSR for SpMM than for SpMV: SMASH outperforms TACO-CSR by 44% and TACO-BCSR by 30%, on average.

We conclude that SMASH is a highly effective mechanism to accelerate sparse matrix computation, regardless of the sparsity of the matrix, and it significantly outperforms state-of-the-art compression schemes for all matrices we evaluate.

7.2.2 Sensitivity to Compression Ratio. As discussed in Section 4.1.1, the compression ratio between Bitmap-0 and the NZA defines an important tradeoff between 1) smaller bitmaps and 2) more zero elements in the NZA that lead to unnecessary computation on zero elements. In Figures 14 and 15, we quantitatively evaluate this tradeoff for SpMV and SpMM respectively, by showing the speedups

SMASH provides with different compression ratios. The speedups are normalized to a configuration that uses a 2:1 compression ratio between Bitmap-0 and the NZA (i.e., each bit in Bitmap-0 encodes two elements in the NZA).

We make two observations. First, increasing the compression ratio from 2:1 to 8:1 degrades performance, by 4% on average (up to 13%) for SpMV and by 5% on average (up to 15%) for SpMM. This is a direct result of more unnecessary computations on zero elements in the NZA which cannot be skipped as a result of the high compression ratio.[1] Second, we observe that matrices with higher density can in some cases benefit from a higher compression ratio (performance increases by 18% in M_{12} and 40% in M_{14}, by going from a compression ratio of 2:1 to 8:1). These matrices exhibit a more *clustered* distribution of non-zero values (i.e., the non-zero elements are close to each other). As a result, even though the compression ratio is higher, the number of zeros in the NZA (and hence the unnecessary computation on zero elements) does not increase in proportion. Instead, SMASH benefits from scanning smaller bitmaps during the indexing operation with a higher compression ratio.

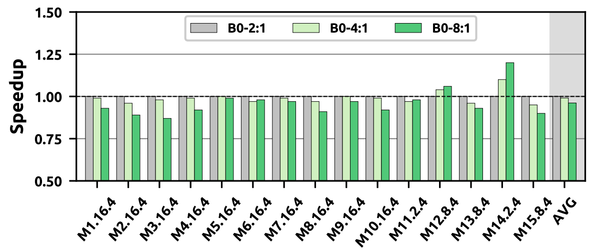

Figure 14: Sensitivity of SMASH speedup to the compression ratio between Bitmap-0 and the NZA for SpMV.

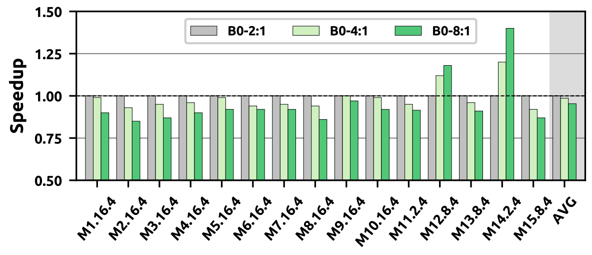

Figure 15: Sensitivity of SMASH speedup to the compression ratio between Bitmap-0 and the NZA for SpMM.

We conclude that the compression ratio between Bitmap-0 and the NZA plays an important role on the effectiveness of SMASH. Our evaluations indicate that a compression ratio of 2:1 is most effective on average and should be used when the structure of sparsity in unknown. However, a matrix that has a *known* clustered distribution of non-zero elements may significantly benefit from a higher compression ratio.

7.2.3 Locality of Sparsity. In Figures 16 and 17, we illustrate the impact of the *distribution* of non-zero elements in a sparse matrix on the effectiveness of SMASH. We define a new metric, *locality of sparsity*, which is the ratio of the average number of non-zero elements per block of the NZA to the size of each NZA block (expressed as a percentage). A matrix with a 100% locality of sparsity would have no zero elements in any NZA block (i.e., all the non-zero elements are clustered at the granularity of the NZA block size). A matrix with 12.5% locality, on the other hand, has exactly one non-zero element per NZA block assuming the NZA holds 8 elements per block.

Figures 16 and 17 compare the speedup of SMASH when the locality of sparsity is varied for three different matrices (M_2, M_8, M_{13}). The results are normalized to the performance of SMASH when the locality of sparsity is 12.5%. The three matrices are chosen to have widely different sparsities (0.06%, 0.85%, and 4.97%).

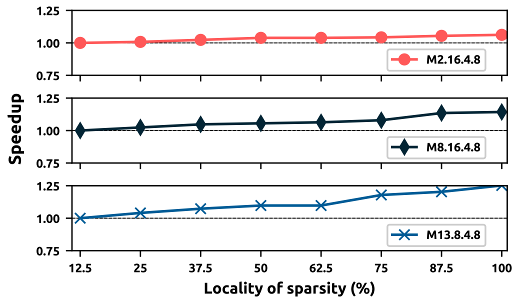

Figure 16: Sensitivity to locality of sparsity in SpMV.

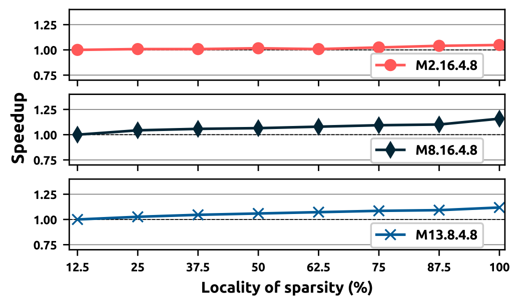

Figure 17: Sensitivity to locality of sparsity in SpMM.

We make two observations. First, the speedup of SMASH increases with an increase in locality of sparsity (by up to 25% in M_{13} for SpMV when going from 12.5% to 100% locality of sparsity). This is because the NZA blocks contain fewer zeros when locality is higher, and this leads to fewer unnecessary computations on zero elements and faster scans of bitmaps during indexing. Second, the performance impact of locality of sparsity depends on the number of non-zero elements in the matrix, i.e., sparsity. The benefits of locality diminish when the matrix is more sparse. This is because *indexing* of the matrices dominates the overall computation time when the sparsity is very low and very little time is spent on computing over the non-zero values. As a result, reducing the amount of unnecessary computation on zero elements in the NZA blocks does not provide significant performance benefit.

7.3 Graph Applications

Figure 18 compares the default CSR-based and the SMASH-based implementations of the PageRank and Betweenness Centrality applications from the Ligra benchmark suite [74] in terms of performance and the number of executed instructions.

We observe that the SMASH-based implementations outperform the CSR-based implementations by 27% and 31% respectively, for PageRank and Betweenness Centrality. Similar to SpMV and SpMM,

[1]Note that we assume there is enough memory to store matrices with any bitmap compression ratio.

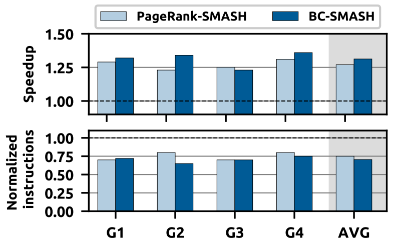

Figure 18: Speedup and normalized number of executed instructions for PageRank and Betweenness Centrality using SMASH (normalized to the CSR-based implementations).

SMASH's speedups in graph workloads come from executing fewer instructions to index the sparse matrices and avoiding the expensive pointer-chasing operations in memory. However, SMASH's benefits are lower in graph workloads than in the SpMV and SpMM kernels since sparse matrix indexing operations in PageRank and BC form a smaller component of the overall execution time. We conclude that SMASH is effective in graph applications.

7.4 Storage Efficiency

A key goal of a sparse matrix compression scheme is to efficiently store the matrix in a manner that avoids saving zero elements and minimizes the amount of metadata required. In Figure 19, we compare the storage efficiency of SMASH and CSR by measuring the ratio of the size of the original uncompressed matrix to the total size of all the data structures required to encode the matrix with SMASH or the CSR format (called the *total compression ratio*).

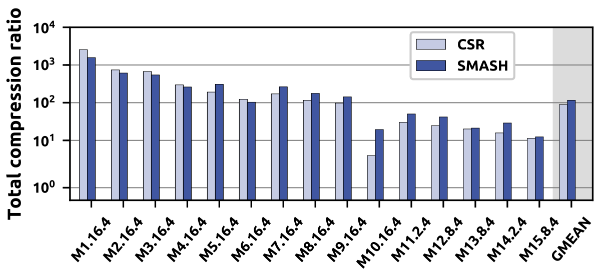

Figure 19: Total compression ratios of SMASH and CSR. Y-axis is in log scale.

CSR stores only the non-zero elements and uses one integer per non-zero element to save the non-zero element's position in the matrix, regardless of the sparsity or the locality of sparsity of the matrix. SMASH, on the other hand, encodes each block in the NZA using a *single bit* (we assume each block to hold 2 elements in Figure 19). If the non-zero elements are not contiguously located, the NZA may hold zero elements in its blocks.

As a result, when the matrices are highly sparse, CSR only stores the few non-zero elements and their positions in the matrix. In contrast, SMASH may store a large number of zero elements in the NZA and many zero bits in the bitmap hierarchy to encode the positions of the few non-zero elements. Hence, as we observe in Figure 19, CSR has a higher total compression ratio than SMASH for matrices that are highly sparse (M_1-M_4).

However, as the matrices become denser, SMASH provides either a similar compression ratio to or a much higher compression ratio than CSR. The reason is twofold. First, the non-zero elements are more likely to be located close to each other at higher matrix densities (i.e., the locality of sparsity is higher). SMASH, as a result, is likely to unnecessarily store fewer zero elements in the NZA. Second, at higher matrix densities, the cost of storing one additional integer index per non-zero element in the CSR format becomes higher than encoding the zero and non-zero regions of the sparse matrix using bitmaps. Hence, in Figure 19, we observe that SMASH generally has a significantly higher (up to 2.48x) total compression ratio than CSR for matrices with higher densities. In some cases (e.g., $M13$, $M15$), even though the matrix has high density, SMASH does not provide a greatly higher compression ratio than CSR. This is because these matrices have low locality of sparsity, which causes SMASH to store more zero elements in the NZA and more zero bits in the bitmap hierarchy.

We conclude that SMASH provides high compression ratios when encoding sparse matrices, enabling highly-efficient storage of sparse matrices in memory. The hierarchical bitmap structure is highly effective in exploiting any locality of sparsity in the matrix to provide even higher compression ratios.

7.5 Format Conversion Overhead

In cases where the sparse matrix is already stored in another format (such as CSR), it first needs to be converted to the hierarchical bitmap encoding in order to leverage the indexing benefits of SMASH. Figure 20 depicts the total time spent on such conversion operations relative to the computation itself for SpMV, SpMM, and PageRank, assuming that the sparse matrix has to be stored in the CSR format but processed using SMASH. In SpMM and PageRank, the total time spent on conversion from CSR to SMASH and vice versa is only a small fraction of the overall computation (10% in SpMM and 0.5% in PageRank), and hence imposes only a small conversion overhead. SpMV, on the other hand, is a short-running kernel and hence, the conversion time dominates SpMV's total execution time (55% of the overall execution time).

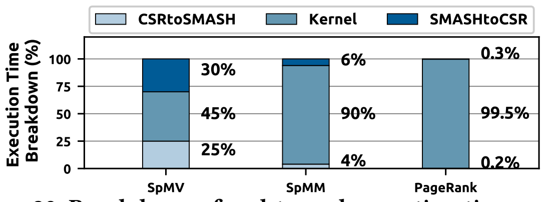

Figure 20: Breakdown of end-to-end execution time, assuming the matrix has to be stored in CSR format but processed using SMASH.

We conclude that the conversion overhead is negligible compared to the demonstrated benefits of SMASH for long running applications and cases where SpMV and SpMM are invoked multiple times (e.g., in neural networks and graph applications). However, for short-running kernels that are invoked only once, the benefits of SMASH may not justify the conversion overhead from another format, assuming that other format is necessary for storage.

7.6 SMASH Area Overhead

The major area overhead in SMASH comes from the buffers and the registers in the BMU. Assuming a BMU with 4 groups of 3 bitmap buffers (where the size of each buffer is 256 bytes), the total additional SRAM required for the bitmaps is 3KB and 140 bytes for the registers. We evaluate the area overhead of the BMU using CACTI 6.5 [47]. In a modern Intel Xeon E5-2698 CPU core, with a 32 KiB L1, 256 KiB L2 and 2.5 MiB L3 Cache, the BMU incurs an area overhead of at most 0.076%.

8 RELATED WORK

To our knowledge, this is the first work to propose a hardware-software cooperative compression scheme and indexing mechanism that 1) enables highly-compressed storage of sparse matrices, 2) provides highly-efficient hardware-supported indexing of sparse matrices to accelerate sparse matrix kernels, and 3) is general and applicable to any sparse matrix operation and matrices with any sparsity and structure.

Sparse matrix operations have been extensively studied in the past. Prior work in this area falls into three categories: 1) sparse matrix compression formats; 2) software optimizations for sparse matrix kernels; and 3) hardware accelerators and hardware-software cooperative mechanisms for sparse matrix kernels. In this section, we briefly discuss these prior works and contrast them with SMASH.

Sparse Matrix Compression Formats. Prior works propose a range of compression formats for sparse matrices [38, 44, 53, 68, 72, 86, 89, 93], including the state-of-the-art formats such as CSR [53], CSR5 [53], and BCSR [38] that are widely used today. These formats are designed to be highly-efficient in storage and can be generally applied to any sparse matrix. However, as we demonstrate in this paper, these formats incur significant performance overheads as a result of inefficient indexing. We quantitatively compare SMASH to CSR [44] and BCSR [44] in Section 7 and demonstrate that SMASH's hierarchical bitmap compression mechanism, along with its hardware support for indexing, significantly outperforms these formats by enabling highly-efficient storage and indexing of sparse matrices.

Several sparse matrix compression formats [7, 16, 30, 41, 42, 46, 77] leverage specific characteristics of the sparse matrix to enable more efficient storage and indexing. For example, CSX [42] first processes each matrix to detect patterns such as dense diagonals or repeated values and then encodes them using compression formats tailored to the detected pattern. CSR-DU [41] and CSR-VI [41] leverage repetition or similarity among non-zero values to compress the sparse matrix more efficiently. These works have three major shortcomings. First, they are limited in applicability to matrices that possess specific patterns in sparsity. SMASH, in contrast, can be used to compress *any* sparse matrix and accelerate any operation on it. Second, these approaches require expensive preprocessing of matrices to identify patterns in sparsity. Third, similar to the general compression formats, these approaches incur significant performance overheads due to inefficient indexing.

Software Optimizations for Sparse Matrix Kernels. There is a large body of prior work on software optimizations to accelerate sparse matrix computation by making sparse matrix kernels more memory or cache efficient [6, 15, 62, 78] and more amenable towards parallelization [1, 13, 54, 58]. A range of prior works also include general software *frameworks*, such as MKL [1] and TACO [40], that leverage these optimizations to produce more efficient code for CPUs [1, 14, 15, 22, 23, 34, 37, 40, 56, 57, 62, 79, 87, 88, 90], GPUs [10, 18, 31, 35, 51, 91, 93], or heterogeneous CPU-GPU systems [52, 55]. SMASH is orthogonal to these software optimizations and can be seamlessly integrated into software frameworks that employ such optimizations to further improve the performance and energy efficiency of sparse matrix computation.

Hardware Accelerators and Hardware-Software Cooperative Mechanisms for Sparse Matrix Kernels. Prior works propose a range of hardware accelerators [59, 63, 64, 66, 94, 96, 98] or FPGA designs [26, 32, 48, 82] for sparse matrix computation. Several of these works also leverage emerging memory technologies such as memristors [17] and 3D-stacked memories [99] to accelerate sparse matrix kernels. These prior approaches are either 1) only applicable to certain applications, such as SpMV [32] or sparse neural networks [96, 98], or 2) assume hardware dedicated to a specific type of sparse matrix kernel only. SMASH, in contrast, can be generally applied across a diverse set of sparse matrix operations and does *not* require fully-dedicated hardware to any particular matrix/kernel type. The hierarchy of bitmaps in SMASH can be used purely in software *without* hardware support in any system, and the hardware unit proposed in SMASH can be added with low overhead to *general-purpose* processors such as CPUs and GPUs.

Prior works propose a range of hardware-software cooperative mechanisms [4, 5, 11, 28, 36, 60, 73, 76, 80, 81, 84, 85, 95, 100] to accelerate memory-bound operations and can be applied to accelerate sparse matrix computations. These approaches are designed to be generally applicable to a very wide range of applications. SMASH, in contrast, addresses the challenges of designing a hardware-software cooperative mechanism for sparse matrix computation. Hence, SMASH is largely orthogonal to these mechanisms and can be integrated into them to accelerate sparse matrix computation.

9 CONCLUSION

We introduce SMASH, a general hardware-software cooperative mechanism that accelerates sparse matrix operations and enables highly-efficient indexing and storage of sparse matrices in memory. The key idea of SMASH is to explicitly enable the hardware to recognize and exploit data sparsity. To this end, we develop a flexible and efficient sparse matrix encoding, based on a hierarchy of bitmaps, that is recognized by both hardware and software. This encoding enables efficient compression of any sparse matrix, regardless of the structure of its sparsity. We develop a hardware mechanism that can directly interpret sparse matrices encoded with hierarchical bitmaps and accelerate computation on those matrices. The bitmap representation, along with the hardware support, greatly reduces the performance overheads of the expensive indexing operations that make state-of-the-art sparse matrix formats inefficient in computation. The expressive SMASH ISA provides programmability of the hardware support and generality across a wide variety of sparse matrix kernels. Our evaluation over a diverse set of 15 matrices and four graphs demonstrates that SMASH significantly improves the performance of SpMV, SpMM, and two graph algorithms (PageRank and Betweenness Centrality), compared to the state-of-the-art CSR format. We believe that the new ideas introduced in SMASH are applicable beyond CPUs and can be a good fit for GPUs, hardware accelerators, and processing in/near memory engines.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for feedback. We thank the SAFARI Research Group members for feedback and the stimulating intellectual environment they provide. We acknowledge the generous gifts provided by our industrial partners: Alibaba, Facebook, Google, Huawei, Intel, Microsoft, and VMware. This research was supported in part by the Semiconductor Research Corporation. Christina Giannoula is funded for her postgraduate studies from the General Secretariat for Research and Technology (GSRT) and the Hellenic Foundation for Research and Innovation (HFRI).

REFERENCES

[1] Intel Math Kernel Library. http://software.intel.com/en-us/articles/intel-mkl/

[2] Intel Xeon Gold 5118. https://ark.intel.com/content/www/us/en/ark/products/120473/intel-xeon-gold-5118-processor-16-5m-cache-2-30-ghz.html.

[3] SMASH code. https://github.com/CMU-SAFARI/SMASH

[4] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing," in *ISCA*, 2015.

[5] J. Ahn, S. Yoo, O. Mutlu, and K. Y. Choi, "PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture," 2015.

[6] K. Akbudak and C. Aykanat, "Exploiting Locality in Sparse Matrix-Matrix Multiplication on Many-Core Architectures," *TPDS*, 2017.

[7] M. Belgin, G. Back, and C. J. Ribbens, "Pattern-Based Sparse Matrix Representation for Memory-Efficient SMVM Kernels," in *ISC*, 2009.

[8] M. Besta, F. Marending, E. Solomonik, and T. Hoefler, "SlimSell: A Vectorizable Graph Representation for Breadth-First Search," in *IPDPS*, 2017.

[9] M. Besta, M. Podstawski, L. Groner, E. Solomonik, and T. Hoefler, "To Push or To Pull: On Reducing Communication and Synchronization in Graph Computations," in *HPDC*, 2017.

[10] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder, "Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid," *SIGGRAPH*, 2003.

[11] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Ranganathan, and O. Mutlu, "Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks," ser. ASPLOS, 2018.

[12] S. Brin and L. Page, "The Anatomy of a Large-scale Hypertextual Web Search Engine," in *WWW*, 1998.

[13] A. Buluc and J. R. Gilbert, "Challenges and Advances in Parallel Sparse Matrix-Matrix Multiplication," in *ICPP*, 2008.

[14] A. Buluç and J. R. Gilbert, "Parallel Sparse Matrix-Matrix Multiplication and Indexing: Implementation and Experiments," *SISC*, 2012.

[15] A. Buluc, S. Williams, L. Oliker, and J. Demmel, "Reduced-Bandwidth Multithreaded Algorithms for Sparse Matrix-Vector Multiplication," in *IPDPS*, 2011.

[16] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel Sparse Matrix-Vector and Matrix-Transpose-Vector Multiplication Using Compressed Sparse Blocks," in *SPAA*, 2009.

[17] J. Cui and Q. Qiu, "Towards Memristor based Accelerator for Sparse Matrix Vector Multiplication," in *ISCAS*, 2016.

[18] S. Dalton, L. Olson, and N. Bell, "Optimizing Sparse Matrix-Matrix Multiplication for the GPU," *TMS*, 2015.

[19] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *TOMS*, 2011.

[20] J. Dongarra, A. Lumsdaine, X. Niu, R. Pozoz, and K. Remington, "Sparse Matrix Libraries in C++ for High Performance Architectures," 1994.

[21] A. Dziekonski and M. Mrozowski, "A GPU Solver for Sparse Generalized Eigenvalue Problems with Symmetric Complex-Valued Matrices Obtained Using Higher-Order FEM," *IEEE Access*, 2018.

[22] A. Elafrou, G. Goumas, and N. Koziris, "Performance Analysis and Optimization of Sparse Matrix-Vector Multiplication on Modern Multi-and Many-Core Processors," in *ICPP*, 2017.

[23] A. Elafrou, V. Karakasis, T. Gkountouvas, K. Kourtis, G. Goumas, and N. Koziris, "SparseX: A Library for High-Performance Sparse Matrix-Vector Multiplication on Multicore Platforms," *TOMS*, 2018.

[24] R. D. Falgout, "An Introduction to Algebraic Multigrid," *Computing in Science Engineering*, 2006.

[25] R. D. Falgout and U. M. Yang, "hypre: A Library of High Performance Preconditioners," in *ICCS*, 2002.

[26] J. Fowers, K. Ovtcharov, K. Strauss, E. S. Chung, and G. Stitt, "A High Memory Bandwidth FPGA Accelerator for Sparse Matrix-Vector Multiplication," in *FCCM*, 2014.

[27] L. Freeman, "A Set of Measures of Centrality Based on Betweenness," *Sociometry*, 1977.

[28] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory," in *ASPLOS*, 2017.

[29] B. Graham, "Spatially-Sparse Convolutional Neural Networks," *arXiv*, 2014.

[30] J. L. Greathouse and M. Daga, "Efficient Sparse Matrix-vector Multiplication on GPUs Using the CSR Storage Format," in *SC*, 2014.

[31] F. Gremse, A. Höfter, L. O. Schwen, F. Kiessling, and U. Naumann, "GPU-Accelerated Sparse Matrix-Matrix Multiplication by Iterative Row Merging," *SIAM*, 2015.

[32] P. Grigoras, P. Burovskiy, E. Hung, and W. Luk, "Accelerating SpMV on FPGAs by Compressing Nonzero Values," in *FCCM*, 2015.

[33] U. Gupta, X. Wang, M. Naumov, C. Wu, B. Reagen, D. Brooks, B. Cottel, K. M. Hazelwood, B. Jia, H. S. Lee, A. Malevich, D. Mudigere, M. Smelyanskiy, L. Xiong, and X. Zhang, "The Architectural Implications of Facebook's DNN-based Personalized Recommendation," *CoRR*, 2019.

[34] P. Hénon, P. Ramet, and J. Roman, "PASTIX: A High-Performance Parallel Direct Solver for Sparse Symmetric Positive Definite Systems," *PMAA*, 2002.

[35] C. Hong, A. Sukumaran-Rajam, B. Bandyopadhyay, J. Kim, S. E. Kurt, I. Nisa, S. Sabhlok, Ü. V. Çatalyürek, S. Parthasarathy, and P. Sadayappan, "Efficient Sparse-Matrix Multi-Vector Product on GPUs," in *HPDC*, 2018.

[36] K. Hsieh, S. M. Khan, N. Vijaykumar, K. K. Chang, A. Boroumand, S. Ghose, and O. Mutlu, "Accelerating Pointer Chasing in 3D-Stacked memory: Challenges, Mechanisms, Evaluation," 2016.

[37] E.-J. Im, K. Yelick, and R. Vuduc, "Sparsity: Optimization Framework for Sparse Matrix Kernels," *IJHPCA*, 2004.

[38] E.-J. Im and K. A. Yelick, "Optimizing Sparse Matrix Vector Multiplication on SMP." in *PPSC*, 1999.

[39] J. Kestyn, V. Kalantzis, E. Polizzi, and Y. Saad, "PFEAST: A High Performance Sparse Eigenvalue Solver Using Distributed-memory Linear Solvers," in *SC*, 2016.

[40] F. Kjolstad, S. Chou, D. Lugato, S. Kamil, and S. Amarasinghe, "taco: A Tool to Generate Tensor Algebra Kernels," in *ASE*, 2017.

[41] K. Kourtis, G. Goumas, and N. Koziris, "Optimizing Sparse Matrix-Vector Multiplication using Index and Value Compression," in *CF*, 2008.

[42] K. Kourtis, V. Karakasis, G. Goumas, and N. Koziris, "CSX: An Extended Compression Format for SpMV on Shared Memory Systems," in *PPoPP*, 2011.

[43] N. Kurd, S. Bhamidipati, C. Mozak, J. L. Miller, T. M. Wilson, M. Nemani, and M. Chowdhury, "Westmere: A Family of 32nm IA Processors," in *ISSCC*, 2010.

[44] D. Langr and P. Tvrdik, "Evaluation Criteria for Sparse Matrix Storage Formats," in *TPDS*, 2016.

[45] J. Leskovec and R. Sosič, "Snap: A General-Purpose Network Analysis and Graph-Mining Library," *TIST*, 2016.

[46] J. Li, G. Tan, M. Chen, and N. Sun, "SMAT: An Input Adaptive Auto-tuner for Sparse Matrix-vector Multiplication," in *PLDI*, 2013.

[47] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "CACTI-P: Architecture-Level Modeling for SRAM-Based Structures with Advanced Leakage Reduction Techniques," in *CAD*, 2011.

[48] C. Y. Lin, N. Wong, and H. K.-H. So, "Design Space Exploration for Sparse Matrix-Matrix Multiplication on FPGAs," *FPT*, 2013.

[49] G. Linden, B. Smith, and J. York, "Amazon.com Recommendations: Item-to-Item Collaborative Filtering," *IC*, 2003.

[50] B. Liu, M. Wang, H. Foroosh, M. Tappen, and M. Pensky, "Sparse Convolutional Neural Networks," in *CVPR*, 2015.

[51] W. Liu and B. Vinter, "An Efficient GPU General Sparse Matrix-Matrix Multiplication for Irregular Data," in *IPDPS*, 2014.

[52] W. Liu and B. Vinter, "A Framework for General Sparse Matrix-matrix Multiplication on GPUs and Heterogeneous Processors," *JPDC*, 2015.

[53] W. Liu and B. Vinter, "CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication," in *ICS*, 2015.

[54] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, "Efficient Sparse Matrix-Vector Multiplication on x86-Based Many-Core Processors," in *SC*, 2013.

[55] K. K. Matam, S. R. K. B. Indarapu, and K. Kothapalli, "Sparse Matrix-Matrix Multiplication on Modern Architectures," in *HiPC*, 2012.

[56] J. Mellor-Crummey and J. Garvin, "Optimizing Sparse Matrix-Vector Product Computations using Unroll and Jam," *IJHPCA*, 2004.

[57] D. Merrill and M. Garland, "Merge-Based Parallel Sparse Matrix-Vector Multiplication," in *SC*, 2016.

[58] D. Merrill and M. Garland, "Merge-Based Sparse Matrix-Vector Multiplication (SpMV) using the CSR Storage Format," in *PPoPP*, 2016.

[59] A. K. Mishra, E. Nurvitadhi, G. Venkatesh, J. Pearce, and D. Marr, "Fine-Grained Accelerators for Sparse Machine Learning Workloads," in *ASP-DAC*, 2017.

[60] A. Mukkara, N. Beckmann, M. Abeydeera, X. Ma, and D. Sanchez, "Exploiting Locality in Graph Analytics Through Hardware-Accelerated Traversal Scheduling," in *MICRO*, 2018.

[61] M. Naumov, D. Mudigere, H. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C. Wu, A. G. Azzolini, D. Dzhulgakov, A. Mallevich, I. Cherniavskii, Y. Lu, R. Krishnamoorthi, A. Yu, V. Kondratenko, S. Pereira, X. Chen, W. Chen, V. Rao, B. Jia, L. Xiong, and M. Smelyanskiy, "Deep Learning Recommendation Model for Personalization and Recommendation Systems," *CoRR*, 2019.

[62] R. Nishtala, R. W. Vuduc, J. W. Demmel, and K. A. Yelick, "When Cache Blocking of Sparse Matrix Vector Multiply Works and Why," *AAECC*, 2007.

[63] E. Nurvitadhi, A. Mishra, and D. Marr, "A Sparse Matrix Vector Multiply Accelerator for Support Vector Machine," in *CASES*, 2015.

[64] E. Nurvitadhi, A. Mishra, Y. Wang, G. Venkatesh, and D. Marr, "Hardware Accelerator for Analytics of Sparse Data," in *DAC*, 2016.

[65] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank Citation Ranking: Bringing Order to the Web." Stanford InfoLab, Tech. Rep., 1999.

[66] S. Pal, J. Beaumont, D.-H. Park, A. Amarnath, S. Feng, C. Chakrabarti, H.-S. Kim, D. Blaauw, T. Mudge, and R. Dreslinski, "OuterSPACE: An Outer Product Based Sparse Matrix Multiplication Accelerator," in *HPCA*, 2018.

[67] G. Penn, "Efficient Transitive Closure of Sparse Matrices over Closed Semirings," *AMiLP*, 2006.

[68] A. Pinar and M. T. Heath, "Improving Performance of Sparse Matrix-Vector Multiplication," in *SC*, 1999.

[69] L. Ren, X. Chen, Y. Wang, C. Zhang, and H. Yang, "Sparse LU Factorization for Parallel Circuit Simulation on GPU," in *DAC*, 2012.
[70] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2003.
[71] D. Sanchez and C. Kozyrakis, "ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems," in *ISCA*, 2013.
[72] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, "Scan Primitives for GPU Computing," in *GH*, 2007.
[73] V. Seshadri, G. Pekhimenko, O. Ruwase, O. Mutlu, P. B. Gibbons, M. A. Kozuch, T. C. Mowry, and T. Chilimbi, "Page Overlays: An Enhanced Virtual Memory Framework to Enable Fine-grained Memory Management," in *ISCA*, 2015.
[74] J. Shun and G. E. Blelloch, "Ligra: A Lightweight Graph Processing Framework for Shared Memory," in *PPoPP*, 2013.
[75] A. Smith. 6 New Facts About Facebook. http://mediashift.org
[76] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "GraphR: Accelerating Graph Processing using ReRAM," in *HPCA*, 2018.
[77] B.-Y. Su and K. Keutzer, "clSpMV: A Cross-Platform OpenCL SpMV Framework on GPUs," in *ICS*, 2012.
[78] P. D. Sulatycke and K. Ghose, "Caching-Efficient Multithreaded Fast Multiplication of Sparse Matrices," in *IPPS*, 1998.
[79] S. Toledo, "Improving the Memory-System Performance of Sparse-Matrix Vector Multiplication," *IBM Journal of research and development*, 1997.
[80] P.-A. Tsai, Y. L. Gan, and D. Sanchez, "Rethinking the Memory Hierarchy for Modern Languages," in *MICRO*, 2018.
[81] P.-A. Tsai and D. Sanchez, "Compress Objects, Not Cache Lines: An Object-Based Compressed Memory Hierarchy," in *ASPLOS*, 2019.
[82] Y. Umuroglu and M. Jahre, "An Energy Efficient Column-major Backend for FPGA SpMV Accelerators," in *ICCD*, 2014.
[83] S. Van Dongen, "Graph Clustering via a Discrete Uncoupling Process," *SIMAX*, 2008.
[84] N. Vijaykumar, E. Ebrahimi, K. Hsieh, P. B. Gibbons, and O. Mutlu, "The Locality Descriptor: A Holistic Cross-Layer Abstraction to Express Data Locality In GPUs," in *ISCA*, 2018.
[85] N. Vijaykumar, A. Jain, D. Majumdar, K. Hsieh, G. Pekhimenko, E. Ebrahimi, N. Hajinazar, P. B. Gibbons, and O. Mutlu, "A Case for Richer Cross-layer Abstractions: Bridging the Semantic Gap with Expressive Memory," in *ISCA*, 2018.
[86] R. W. Vuduc and H.-J. Moon, "Fast Sparse Matrix-vector Multiplication by Exploiting Variable Block Structure," in *HPCC*, 2005.
[87] Q. Wang, X. Zhang, Y. Zhang, and Q. Yi, "AUGEM: Automatically Generate High Performance Dense Linear Algebra Kernels on x86 CPUs," in *SC*, 2013.
[88] J. B. White and P. Sadayappan, "On Improving the Performance of Sparse Matrix-Vector Multiplication," in *HiPC*, 1997.
[89] J. Willcock and A. Lumsdaine, "Accelerating Sparse Matrix Computations via Data Compression," in *ICS*, 2006.
[90] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms," in *SC*, 2007.
[91] T. Wu, B. Wang, Y. Shan, F. Yan, Y. Wang, and N. Xu, "Efficient PageRank and SpMV Computation on AMD GPUs," in *ICPP*, 2010.
[92] Z. Xianyi, W. Qian, and Z. Yunquan, "Model-driven Level 3 BLAS Performance Optimization on Loongson 3A processor," in *ICPADS*, 2012.
[93] S. Yan, C. Li, Y. Zhang, and H. Zhou, "yaSpMV: Yet Another SpMV Framework on GPUs," in *PPoPP*, 2014.
[94] L. Yavits and R. Ginosar, "Sparse Matrix Multiplication on CAM Based Accelerator," *CoRR*, 2017.
[95] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian, "GraphP: Reducing Communication for PIM-based Graph Processing with Efficient Data Partition," in *HPCA*, 2018.
[96] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-X: An Accelerator for Sparse Neural Networks," in *MICRO*, 2016.
[97] Y. Zhao, J. Li, C. Liao, and X. Shen, "Bridging the Gap between Deep Learning and Sparse Matrix Format Selection," in *PPoPP*, 2018.
[98] X. Zhou, Z. Du, Q. Guo, S. Liu, C. Liu, C. Wang, X. Zhou, L. Li, T. Chen, and Y. Chen, "Cambricon-S: Addressing Irregularity in Sparse Neural Networks Through a Cooperative Software/Hardware Approach," in *MICRO*, 2018.
[99] Q. Zhu, T. Graf, H. E. Sumbul, L. Pileggi, and F. Franchetti, "Accelerating Sparse Matrix-Matrix Multiplication with 3D-Stacked Logic-in-Memory Hardware," in *HPEC*, 2013.
[100] Y. Zhuo, C. Wang, M. Zhang, R. Wang, D. Niu, Y. Wang, and X. Qian, "GraphQ: Scalable PIM-Based Graph Processing," in *MICRO*, 2019.