VIA: A Smart Scratchpad for Vector Units with Application to Sparse Matrix Computations

Julián Pavón[1,2], Ivan Vargas Valdivieso[1,2], Adrián Barredo[1,2], Joan Marimon[1,2], Miquel Moreto[1,2], Francesc Moll[2], Osman Unsal[1], Mateo Valero[1,2] and Adrian Cristal[1,2]
[1]*Barcelona Supercomputing Center*
firstname.lastname@bsc.es
[2]*Universitat Politècnica de Catalunya*
firstname.lastname@upc.edu
Barcelona, Spain

***Abstract*—Sparse matrix operations are critical kernels in multiple application domains such as High Performance Computing, artificial intelligence and big data. Vector processing is widely used to improve performance on mathematical kernels with dense matrices. Unfortunately, existing vector architectures do not cope well with sparse matrix computations, achieving much lower performance in comparison with their dense counterparts.**

To overcome this limitation, we present the Vector Indexed Architecture (VIA), a novel hardware vector architecture that accelerates applications with irregular memory access patterns such as sparse matrix computations. There are two main bottlenecks when computing with sparse matrices: irregular memory accesses and index matching. VIA addresses these two bottlenecks with a smart scratchpad that is tightly coupled to the Vector Functional Units within the core.

Thanks to this structure, VIA improves locality for sparse-dense computations and improves the index matching search process for sparse computations. As a result, VIA achieves significant performance speedup over highly optimized state-of-the-art C++ algebra libraries. On average, VIA outperforms sparse matrix vector multiplication, sparse matrix addition and sparse matrix matrix multiplication kernels by 4.22×, 6.14× and 6.00×, respectively, when evaluated over a thousand sparse matrices that arise in real applications. In addition, we prove the generality of VIA by showing that it can accelerate histogram and stencil applications by 4.5× and 3.5×, respectively.

***Index Terms*—Sparse Algebra, Vector Computing, Scratchpad Memory**

I. Introduction

Since the end of Dennard's scaling and the subsequent stagnation of CPU clock frequency, both computer architects and software developers are forced to exploit parallelism to improve performance. While instruction- and thread-level parallelism are extensively studied, there are still many unexplored opportunities to achieve significant performance and energy improvements from Data-Level Parallelism (DLP). DLP can be exposed to the hardware by vector computations [5], [26], where a Single Instruction operates over Multiple Data streams (SIMD) [3], [37].

Many applications can potentially benefit from vectorized execution for better performance, higher energy efficiency and greater resource utilization [34]. Ultimately, the effectiveness of a vector architecture depends on the quality of the vectorized code [60]. Sparse matrix operations are a clear example of computations difficult to vectorize [8], [19]. Such computations are a key kernel in High Performance Computing (HPC), Artificial Intelligence (AI) and big data workloads. In particular, two such killer-applications are Sparse Matrix Vector multiplication (SpMV) and Sparse Matrix Matrix Multiplication (SpMM). SpMV is an important component for the High Performance Conjugate Gradient (HPCG) code that has become an alternative to LINPACK for rating supercomputers [20]. SpMV is also fundamental in AI applications such as Support Vector Machine computations via gradient descent [27]. Last but not least, SpMV is an essential part of graph kernels used in big data and is the most important kernel in the GraphBLAS specification [42]. On the other hand, SpMM is an equally important kernel. SpMM is widely used in numerous application domains such as data analytics [46], graph processing [30], [58] and particularly for pruned neural networks [69]. Therefore, an efficient execution of SpMV and SpMM will be transformative for HPC, AI and big data communities.

In these relevant domains, many applications use sparse matrices with less than 1% of non-zero values [17]. To make an efficient usage of memory and compute power, many different compressed representations for sparse matrices have been proposed [9], [10], [50]. However, none of them copes well with current vector extensions. Thus, there are two intertwined obstacles against efficient execution of sparse matrix computations on vector architectures: (1) existing sparse matrix representations are not easily vectorizable, and (2) current vector hardware implementations are not optimized for sparse matrix operations.

In this paper, we propose the *Vector Indexed Architecture* (VIA), a vector architecture that aims at accelerating sparse matrix computation. VIA features a smart scratchpad memory specially designed to cope with sparse-dense (SpMV) and sparse-sparse (SpMM) matrix computations with irregular DLP. For SpMV, VIA manages the high-locality dense vector in the scratchpad, thus reducing memory traffic and releasing memory bandwidth to read the low-locality sparse matrix from main memory more efficiently. For SpMM, VIA features a specialized parallel lookup structure in the scratchpad that enables the functional units to efficiently compute index matching operations, one of the main bottlenecks in SpMM kernels. This allows the vector functional units to exploit the irregular DLP of these computations efficiently.

While existing FPGA or ASIC-based accelerators focus on a single sparse algebra computation [57], [59] with a specific sparse matrix compressed representation [41], VIA is a general solution aiming to improve performance over several different sparse computations using a diverse set of sparse matrix compressed representations. Moreover, VIA can be efficiently integrated in an out-of-order processor pipeline with reduced hardware cost.

This paper makes the following contributions beyond the state-of-the-art:

- The VIA design, a novel vector architecture that accelerates sparse matrix computations. VIA requires minimal hardware support, mainly a smart scratchpad memory and a vector unit that accelerates sparse matrix operations. A detailed design space exploration is performed to size the VIA hardware. Register-Transfer Level (RTL) implementation and associated synthesis results confirm that VIA is area- and power-efficient (0.515mm^2 and 0.5mW, respectively) using a 22nm technology node.
- A rich set of novel instructions to work with VIA. These instructions can be easily integrated in the programming model of different Vector Instruction Set Architectures (ISAs).
- An exhaustive evaluation with a full system cycle-accurate simulator considering real applications and matrices. Our evaluation shows that VIA achieves significant performance speedups of 4.22×, 6.14×, and 6.00× on average for SpMV, Sparse Matrix Addition and SpMM. These three kernels are evaluated over one thousand different sparse matrices that arise in real applications. Moreover, VIA architecture is applicable to other important application domains such as histograms and stencil computation. For these two applications we achieve 4.51× and 3.39× speedup respectively.

This paper is organized as follows: Section II motivates the problem. Section III and IV describe the key ideas and the implementation of the proposed hardware, respectively. Section V presents the simulation environment. Next, Section VI performs a design space exploration of the VIA implementation, while Section VII shows the performance and energy results of our proposal. Section VIII reviews the related work and Section IX summarizes the main conclusions of this work.

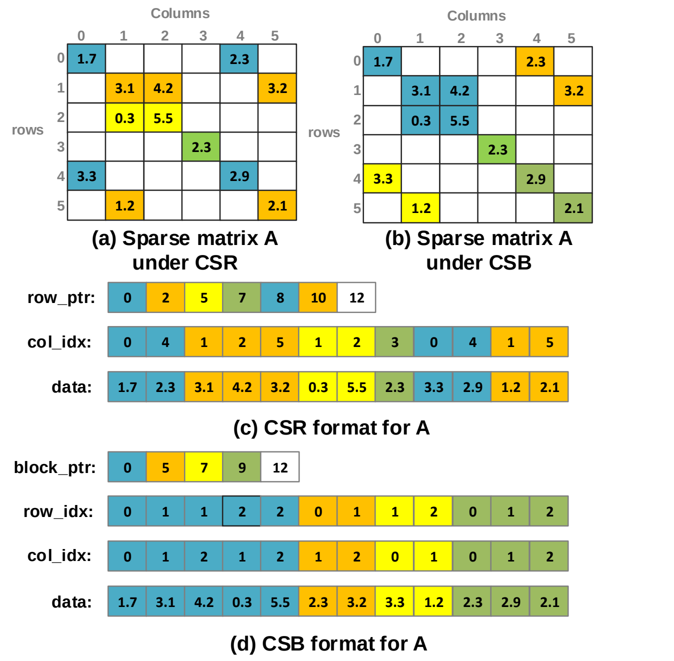
II. Background and Motivation

This section reviews current sparse matrix representations, discusses challenges in vectorizing sparse matrix kernels and motivates the need for efficient vectorization through hardware-software co-design.

A. *Sparse Matrix Compressed Formats*

One of the main issues related to sparse matrices is how to store them efficiently in memory. In this scenario, efficiency is defined as avoiding to store the abundant unused zero values while keeping the occasional non-zero values in the matrix. Multiple compressed formats for sparse matrices have been proposed to tackle this issue.

The most common sparse matrix representations belong to the *compressed sparse* family, such as CSR and CSC (Compressed Sparse Row and Column, respectively) (see Figures 1.a and 1.c). The CSR format is widely used in many libraries that involve sparse matrix operations [23], [43], [65]. CSR makes use of three arrays to represent the matrix: an array with the column index of each non-zero value in the row(col_idx), an array with the values of all



Columns
rows
(a) Sparse matrix A under CSR
(b) Sparse matrix A under CSB
row_ptr: 0 2 5 7 8 10 12
col_idx: 0 4 1 2 5 1 2 3 0 4 1 5
data: 1.7 2.3 3.1 4.2 3.2 0.3 5.5 2.3 3.3 2.9 1.2 2.1
(c) CSR format for A
block_ptr: 0 5 7 9 12
row_idx: 0 1 1 2 2 0 1 1 2 0 1 2
col_idx: 0 1 2 1 2 1 2 0 1 0 1 2
data: 1.7 3.1 4.2 0.3 5.5 2.3 3.2 3.3 1.2 2.3 2.9 2.1
(d) CSB format for A

Fig. 1. **Comparison of compressed storage formats** — (a, b) Show the sparse matrix A in its dense representation under CSR and CSB respectively; (c) Shows the matrix A stored using the CSR format; (d) shows the matrix A stored using the CSB format

the non-zero elements in the matrix(data) and finally, an array with the beginning value for each row in the col_idx and data arrays (row_ptr). CSC makes use of a very similar representation swapping the row and column arrays.

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 resultant dense vector. Algorithm 1 represents a CSR-based implementation. The algorithm iterates only over the non-zero values of matrix A, avoiding the memory and computational overhead of storing and computing over the zero values in the dense representation of the matrix. However, CSR's performance is hindered by *pointer-chasing* effects, which happen when col_idx is loaded and then used to read the value from the input vector x (see line 3 in Algorithm 1).

Algorithm 1 CSR-Based SpMV implementation

```
1: for (i=0; i<rows; i++) do
2:     for (j=row_ptr[i]; j<row_ptr[i+1]; j++) do
3:         y[i] += data[j] * x[col_idx[j]]
4:     end for
5: end for
```

2) *Sparse Matrix Addition (SpMA):* Consider the SpMA algorithm C = A + B, where all the operands are sparse matrices with same number of rows and columns. Algorithm 2 depicts a CSR-based implementation of SpMA. Both input matrices (A and B) are traversed element by element, adding the elements or copying one of them if the indices are not the same. Then, the resulting value is inserted into the matrix C.

3) *Sparse Matrix Matrix Multiplication (SpMM):* SpMM can be executed in many different ways (e.g. outer product, inner product,

Algorithm 2 CSR-Based SpMA implementation

```
for (i=0, j=0; i<A.non-zeros.size() and j<B.non-zeros.size()) do
    if (A[i].idx == B[j].idx) then
        C.insert(A[i].value + B[j].value, A[i].idx)
        i++; j++
    else if A[i].idx > B[j].idx then
        C.insert(B[i].value, B[i].idx)
        j++;
    else
        C.insert(A[i].value, A[i].idx)
        i++;
    end if
end for
Copy tail of A or B.
```

row- and column wise). However, the classic SpMM algorithm is formulated as an inner product multiplication [57]. In the SpMM implementation shown in Algorithm 3, matrix A is compressed using CSR and B using CSC. SpMM iterates in row-major over A and in column-major over B. For each non-zero element in each row in A, we need to search through the column indices of A and the row indices of B to discover if we can multiply the element in A. This process is called *index matching* (lines 4-5 in the code). Because the index matching is performed for every product operation, a CSR-based SpMM implementation requires a large number of position-finding operations.

Algorithm 3 CSR-Based SpMM implementation

```
 1: for each row in A do
 2:     for each col in B do
 3:         for each elem in row do
 4:             k = col.search_idx(elem.idx)
 5:             if (k != -1) then
 6:                 C[elem.row, col[k].column] += elem.value * col[k].value);
 7:             end if
 8:         end for
 9:     end for
10: end for
```

B. Block Based Formats

A problem when using the CSR and CSC representations is the challenge to map each element in a 2D space when all the values and indices are placed one after other. Generally the sparse matrices from real-world applications have clustered non-zero values forming block sub-structures. All computations with sparse matrices can greatly benefit from these structures.

This issue is addressed by means of Block-Based sparse matrix compressed Formats (BBF). For example, Buluc et al. [10], [11] propose CSB (Compressed Sparse Block), a storage format based on blocking techniques. The CSB format does not give preference to rows over columns (or vice versa), providing a symmetric representation of the matrix. It consists of four one-dimensional arrays: data, row_idx, col_idx and block_ptr. The first three represent the in-block structure by storing the non-zero elements with their in-block relative row and column indices. The last one stores the initial position in the other three arrays for each block and it is used to determine the non-zero elements per block too (See Figures 1.b and 1.d). To reduce the memory footprint of having both row and column index arrays, a single in-block index array can be created, merging the row and column indices of each value.

The SpMV, SpMA, and SpMM algorithms will need specific changes to execute with BBF representations. The main difference is in the loop-end conditions and the necessity of new inner-loops. In the SpMV algorithm, line 1 changes to iterate on the blocks instead of rows. Then, the inner loop in line 2 is changed to iterate in the block. Finally, we need to add an extra step to line 3 to calculate the matrix-relative row and column indices of the values. Using BBF increases a bit the complexity of the indexing stages of sparse matrices in the three evaluated algorithms. However, data locality within a block significantly increases, reducing off-chip memory requests in the inner loops as a result.

C. Vectorizing Sparse Matrix Kernels

Vector architectures are highly used to improve the performance and energy efficiency of many scientific applications due to their capacity to exploit DLP.

Many important scientific kernels perform computation on sparse matrices. However, vectorizing these data structures implies several challenges to their compressed representations. The main difficulties for current commercial Vector ISAs, such as AVX [37] and SVE [61] are: (1) how to access data without relying on expensive memory indexed instructions such as scatter-gather [62]. (2) How to efficiently execute index matching with current vector instructions.

To optimize vectorization, BBF-based software techniques restructure data using block sizes that match the Vector Length (VL, i.e. the Vector Length in AVX) offered by commodity vector hardware [9], [44], [50], [64], [71]. Still, these approaches require (1) increased writes to memory to update partial results because the matrix is split into small chucks with a smaller range of rows or columns. This process is called store-load forwarding and increases the memory traffic, (2) sparse to dense block transformation [9], and (3) Zero padding techniques [44]. In all the cases the performance of BBF-based software implementations suffer from poor utilization of the vector ALUs due to the abundant number of zero values and the memory bandwidth reduction.

D. Hardware-Software Co-Design

Due to the performance benefits that vector architectures can achieve, many software and hardware approaches have been proposed to properly exploit their capacities over sparse matrix kernels. On the one hand, software-oriented solutions have focused on developing specialized data structures to facilitate how values are read from the matrix [9], [50], [71].

On the other hand, hardware-oriented solutions have looked at the way data is collected from memory, to facilitate its re-ordering and execution [22], or doing near-memory data layout transformations to mitigate the sparsity problem [6]. Both solutions focus on how data is stored in memory and how it is loaded to be executed by the vector units.

Tackling only one side of the problem limits its potential benefits. Hardware implementations without a software point of view lead to complex and highly dedicated designs. Similarly, pure software approaches fail to fully exploit the capabilities of the vector units.

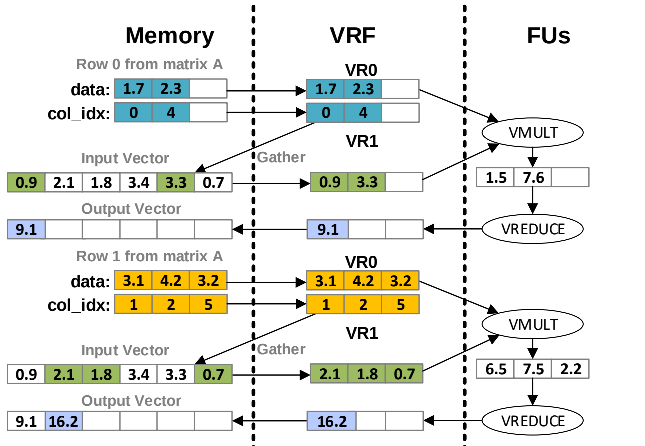


Fig. 2. Vectorized implementation of a CSR format based SpMV using a conventional vector ALU. The figure depicts the flow between the memory, the Vector Register File (VRF) and the Functional Units (FUs).

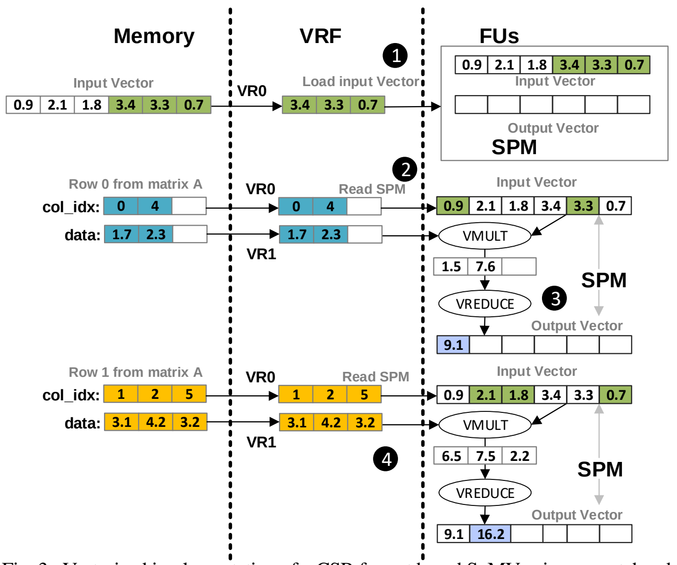


Fig. 3. Vectorized implementation of a CSR format based SpMV using a scratchpad memory (SPM). Using the SPM, the entire memory bandwidth is used to read the input matrix values.

For this reason, our proposal described in Sections III, adopts a hardware design taking into account the computational requirements of the best state-of-the-art block-based software implementations.

E. Applicability to Other Application Domains

The proposed architecture improves neighbouring data accesses and index matching operations. Thus, VIA can be directly applied beyond sparse matrix operations to other application classes that feature similar data access and computational patterns. Sections IV-F and VII-D show how VIA can significantly accelerate two such important application classes: histogram processing and stencil computations.

Histograms are a crucial part of database query planning [40], and are highly utilized in image processing [18]. But their computation is resource-intensive. Histograms are challenging to be vectorized due to the irregularity of the memory access in their computation [39].

Stencils are ubiquitous in HPC applications, including fluid dynamics, heat diffusion [16], seismic imaging [52] and climate modeling [28]. A stencil operates on a structured multidimensional grid based on the values of the local neighborhood of grid points.

III. VIA: Key Design Ideas

In this section, we analyze the challenges that prevent current solutions using sparse matrix compressed formats from fully exploiting data-level parallelism with vector computing. Next, we present the intuition behind our idea and outline how these insights can help us to tackle the challenges presented. These insights will then drive the design presented in Section IV.

A. Sparse Matrix Computation Challenges

Sparse matrix kernels present a set of challenges to exploit DLP:

Challenge 1: Avoiding inefficient memory indexed instructions. As shown in the previous section, sparse matrix computations intensively rely on *pointer chasing* operations. To vectorize these operations, conventional vector ISAs make use of scatter-gather instructions. These instructions are not the best choice to target high performance [39], [56], since they increase the memory traffic and add extra latency. For example, a gather operation in modern Intel out-of-order processors takes 22 cycles in the best case when all the vector operands are already in Level 1 data cache. Note that the usual case in sparse matrix operations is much worse as gather instructions typically need to access higher levels of the memory hierarchy. We illustrate this challenge in Figure 2, which depicts the execution of the inner loop in Algorithm 1 for two rows from an input sparse matrix A. In every iteration, a gather instruction is executed; this reduces the memory bandwidth and affects the efficiency of the traditional vector architectures.

Challenge 2: Computing index matching efficiently. The major challenge when using two sparse matrices is how to efficiently compute the index matching operations within the vector ALU. Vectorizing index matching operations with the current state-of-the-art vector ISAs requires several in-VRF (Vector Register File) data transformations or reordering through the memory that affects the performance of the vector ALU. Some proposals use zero-padding [41] as a way to alleviate this issue. However, this reduces the efficiency of the vector operations by adding zero elements to the input data.

B. Scratchpads Applicability to Sparse Algebra Computations

Scratchpad Memories (SPM) are high-speed internal storage structures. Scratchpads have simpler allocation and control mechanisms compared to general cache memories and can be implemented with different mapping techniques [2]. The two challenges presented in Section III-A can be appropriately addressed using scratchpad memories. However, both challenges present different requirements that must be considered separately first. Next, we discuss two scratchpad approaches suitable for sparse matrix operations that address the identified challenges. Section IV details how these two approaches can be combined in the VIA architecture.

The key idea in VIA is to attach a smart SPM next to the vector functional units. Input sparse matrices in application domains such as HPC, AI and data analytics are highly sparse (e.g. the

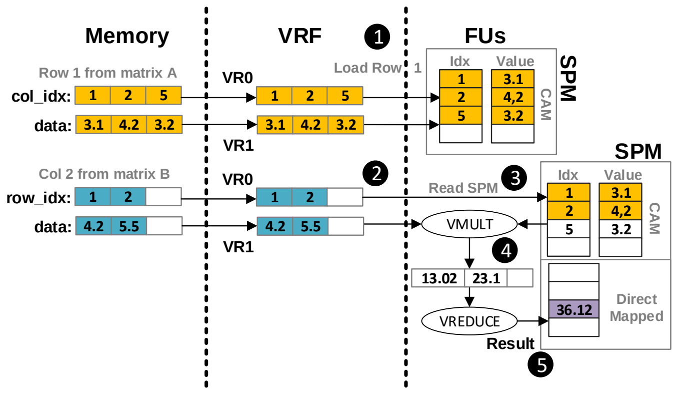


Fig. 4. Vectorized implementation of a CSR format based SpMM using a scratchpad memory (SPM) as building block. Index matching operations are executed while reading the SPM.

matrix that represents Youtube's social network connectivity has only 0.00023% of non-zero elements [47]). With such a high sparsity, computations as SpMV and SpMM become memory-bound. The regular connection of the SPM in the memory hierarchy will generate traffic between requests of the cache memories and the SPM. By placing the SPM next to the functional units, the entire memory bandwidth is used to read the low locality values (from the sparse matrix), while the SPM stores data with high locality and accumulates partial results that will be later reused.

1) Direct-Mapped SPM for Sparse-to-Dense Transformation: Challenge 1 becomes an issue when computing with two data structures where one structure is sparse while the other is dense. An example is SpMV where a sparse matrix is multiplied by a dense vector. This challenge requires an efficient way to map data in the dense structure without relying on memory-indexed instructions. In this case, a direct-mapped SPM can provide enough locality to accelerate the dense data read process [15] with minimum area overhead due the simplicity of scratchpad mapping control. Figure 3 depicts the usage of a *direct-mapped* SPM to execute the SpMV kernel on two rows from an input sparse matrix A. The first step loads the input vector into the SPM ❶. Then, we read the values and column indices from row 0; the column indices are used to map and read data from the SPM ❷. Data from the VRF and SPM is multiplied and reduced to generate the output. This output result is stored back to the SPM ❸. Then, the next row is read from memory and computed in the same way ❹. The idea of processing the index operations for the dense vector in the SPM instead of accessing the memory hierarchy reduces the overall memory traffic, thus increasing the memory bandwidth utilization to stream the input sparse matrix.

2) CAM-based SPM for Sparse-to-Sparse Operations: Challenge 2 is manifested when computing using two sparse structures (e.g. SpMA and SpMM). Current Vector ISAs do not cope well with index matching operations that are required when computing with two sparse structures. To address this issue, we propose using a specialized CAM (Content Addressable Memory) based scratchpad. CAM memories are specialized hardware structures that are particularly suitable for search and index matching algorithms. Previous state-of-the-art have shown their efficiency in sorting algorithms that execute index matching operations [33]. Our specialized CAM has simple control logic and consumes less area compared to other CAM-based processor pipeline structures such as the *issue queue* in an out-of-order processor. In our approach, allocate and delete policies are done in order, whereas in a typical out-of-order issue queue, CAM payload data is allocated and deleted in an out of order manner, thus complicating the design.

Figure 4 depicts the execution of the inner loop in Algorithm 3 for one row of matrix A and one column of matrix B respectively. The SPM only stores the indices and data from the input row. In a first step, we load the input row in the SPM ❶. Then, the row indices and values from the input column are read into the VRF ❷. Next, we use the row indices from the input column to read the SPM. This reading operation executes the index matching with the column indices from the input row ❸. The values from those indices that match are then multiplied and reduced in the FUs ❹. Finally, in a similar way to the SpMV kernel, we accumulate the output results in the SPM to reduce memory bandwidth utilization ❺.

IV. VIA: Design Implementation

In this section, we introduce the *Vector Indexed Architecture* (VIA), a hardware-software co-designed vector architecture. VIA is composed of two main building blocks: a *Smart ScratchPad Memory* (SSPM) and the *Fused Indexed Vector Unit* (FIVU). FIVU is the control interface between the Vector Functional Units (VFU) and the SSPM. Next, we provide a detailed description of the different hardware components of VIA, their operation and the software considerations to work with VIA.

A. The Smart Scratchpad Memory

The SSPM is a dedicated high bandwidth structure used to feed the VFU and it can be used in direct-mapped mode, or in CAM-based mode. For *Sparse-to-Dense* computation the SSPM is used in direct-mapped mode, while for *Sparse-to-Sparse* computations the SSPM is used in CAM-based mode. The SSPM consists of three main building blocks (see Figure 5): ① the SRAM cells to store the actual data; ② the valid bitmap to specify when an entry in the SRAM has been written before; and ③ the Index tracking logic that provides the CAM-based functionality to SSPM.

SRAM cells (SRAM): stores the values to compute, e.g. for SpMV operations, SSPM stores the vector and for both SpMM and SpMA operations, SSPM stores the sparse row data and indices of only one of the input matrices. In our implementation, SRAM is built using four byte length blocks and each block stores a single value independently on the data length.

Operation: The SRAM is used in both direct-mapped and CAM-based modes. The main difference between these two modes is where the indices come from. In the direct-mapped mode, we use directly the input index to map the entries in the SRAM (see red arrows in Figure 5). For the CAM-based, the index used to map the SRAM entries can come from either the insertion logic, or the Index table (see black arrows in Figure 5).

Valid bitmap: This structure is used in the direct-mapped mode as a written value indicator for the entries in the SRAM. It consists of a vector of bits, where each bit corresponds to an entry in the SRAM structure and determines whether an entry has been written.

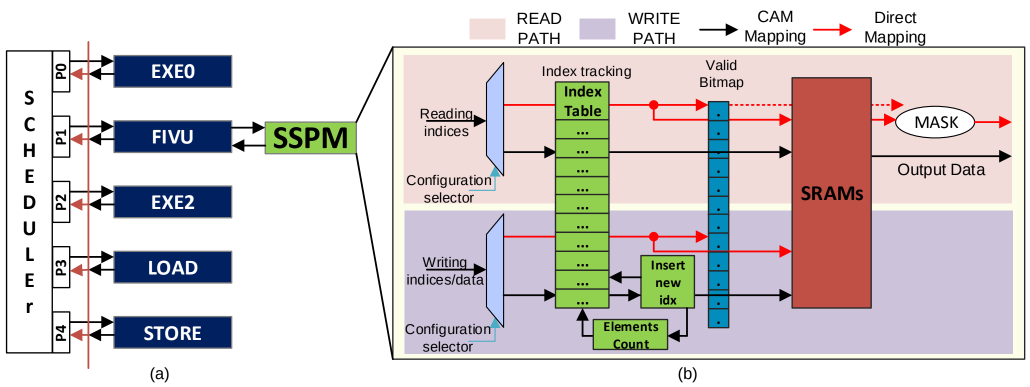


Fig. 5. VIA building blocks: (a) interconnection between FIVU, VIA and the issue logic of an out-of-order core; (b) microarchitecture of SSPM. It consists of the index tracking mechanism (Index table, Insert new Idx and Elements Count), valid bitmap, and the storage system (SRAMs). Read and write paths are depicted separately.

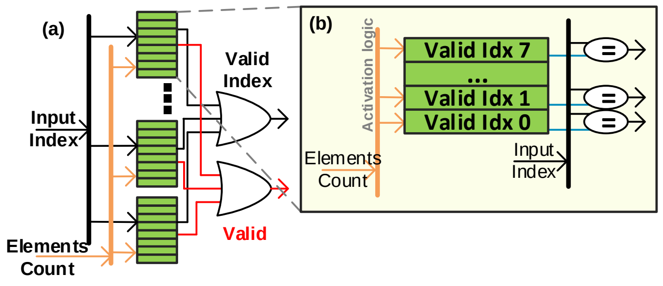


Fig. 6. Index table architecture. Our Index Table is implemented as a set of 8 entry banks, and each bank is activated based on the number of tracked elements in SSPM

Operation: ❶ *Writing in direct-mapped mode:* When a value is written to the SSPM, the corresponding bit in the valid bitmap is set. ❷ *Reading in direct-mapped mode:* When a value is read from the SSPM, the corresponding bit in the valid bitmap is checked. If it is set, the value from the SRAM is returned; otherwise a zero value is returned. ❸ *Clearing SSPM contents in direct-mapped mode:* Either the full valid bitmap or a segment of it is cleared by flash zeroing. This operation is accomplished with a single instruction (see Section IV-C).

Index tracking logic: This block implements SSPM-CAM functionality over the indexes. The index tracking logic consists of three key components: ❶ The index table, a CAM structure that stores the indices used to write data in the SRAM; ❷ The insertion logic, which inserts new indices and elements in order in the first available position in the index table and the SRAM respectively; and ❸ The element count register, which holds the number of stored indices in the index table (see green blocks in Figure 5).

Index table: It is composed of storage cells and an array of comparators (matching logic). The written indices are stored in these storage cells. In a CAM-based read operation (search process), the matching logic determines if the reading index has been written before in the index table by comparing the reading index with the content of the storage cells; if there is a match, the valid index is given as output.

Operation: ❶ *Writing in CAM-based mode:* First, the writing index searches the index table to determine if the same index has been stored before; in the case of no match, the insertion logic generates a new index to write data in both the index table and the SRAM. If the writing index matches with a position in the index table, we receive the valid index where the data is stored in the SRAM and the new value is updated. ❷ *Reading in CAM-based mode:* The reading index searches the index table for a match; if there is no a match, the output is zero. For a match case, the valid index is obtained from the index table, this index is used to read the content from the SRAM as output of the operation.

Hardware Optimizations: In order to reduce area and energy consumed by the index table we have considered two different optimizations. First, to avoid unnecessary comparisons (e.g. compare only with entries in the index table that have valid tracked indices), we split the index table structure in banks of 8 elements (see Figure 6). Then, using the value of the element count register, we apply clock gating to those banks without tracked indices to further reduce energy consumption. Second, to reduce area consumption, we customize the index table and the insertion logic to use a subset of the total storage space in the SRAM and to allocate data and new tracked indices in order. We based this decision on the fact that a great amount of real-world applications use highly sparse matrices as inputs. These highly sparse matrices only have a few elements per row [46]–[48].

B. The Fused Indexed Vector Unit

VIA introduces the *Fused Indexed Vector Unit* (FIVU) to operate over data stored in the SSPM. The FIVU works as the interface between the SSPM memory and the processor pipeline and minimally extends a generic Vector Functional Unit (VFU) with new pipeline stages to control operations to the SSPM.

The FIVU Design: The FIVU is a regular VFU that supports operations with the SSPM. We found it unnecessary to add a new VFU, which has a high cost in terms of area and power consumption. For this reason, we have extended a regular vector unit to support the accesses to the SSPM. In particular, a regular vector unit requires three new pipeline stages. Figure 7 depicts these new stages in the pipeline of a regular VFU and they are described below.

Preprocessing stage 1: It receives the decoded instruction and its vector source operands (VSRC1 and VSRC2) from the Vector Register File (VRF). This stage checks the instruction opcode to determine if it requires access to the VIA hardware. When a VIA instruction is detected, the *SSPM request logic* uses the VSRC1 as a vector of indices to generate requests to the SSPM. Depending

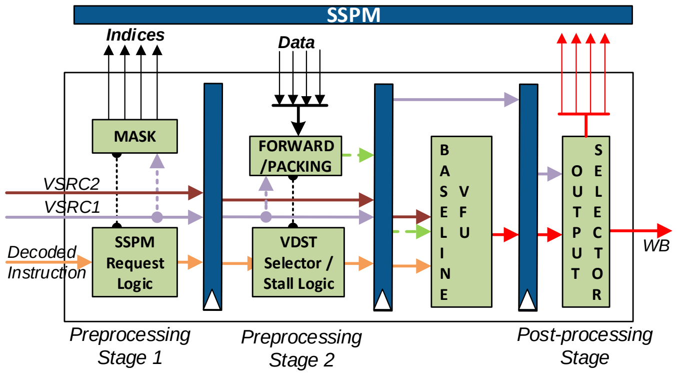


Fig. 7. FIVU: an extended regular VFU to support operations with the SSPM. Preprocessing 1 and Preprocessing 2 stages read the SSPM and the Post-processing stage determines if the operation output is stored in SSPM or directly written back to the VRF.

on the data type and vector length, more than one request may be necessary to complete the operation; these requests are executed in a nested pipeline in multiple cycles until all the requests to the SSPM have been performed. For non-VIA instructions, all the operands are bypassed to the *Baseline VFU*.

Preprocessing stage 2: It is used to: 1) receive and pack all the elements from the SSPM into a single vector operand, and 2) stall and set FIVU as busy. This stage consists of 2 modules: ❶ The *FORWARD/PACKING* module receives all the requested elements from the SSPM and generates a single vector operand. For non-VIA instructions, the VSRC1 is forwarded to the *Baseline VFU*. ❷ The *Stall Logic* marks the FIVU as busy until all the requests have been executed and packed. The FIVU is released when the control logic in this stage verifies that all the operations with the SSPM are completed.

Post-processing stage: It selects where to write back the operation results. One of two possible outcomes is selected: ❶ the output to a regular vector register, or ❷ the output to the SSPM structure. For the last option, we use again the VSRC1 as a vector of indices to write the output values in the SSPM.

C. ISA Extensions in VIA

To operate with VIA, we have added a small set of new instructions to the AVX2 SIMD ISA available in x86-64 processors [3], [37]. Our extensions are generic and can be part of other vector ISAs. These instructions are designed to store and read data from the SSPM (*vIdxmov.d*, *vIdxmov.c*, *vIdxcount*), clear the entries in the SSPM (*vclear*) and apply arithmetic operations (*vIdxAdd, vIdxSub, vIdxMult, vIdxBlkMult*).

vIdxStore.d: This instruction reads data from the VRF and stores it in the SSPM in direct-mapped mode. It has two source operands: *Data* and *Idx*. *Data* contains the values to be stored in the SSPM. *Idx* holds the indices used to map the SSPM entries.

vIdxLoad.d: Loads data from the SSPM to write it in the VRF in direct-mapped mode. It has two operands: *Idx* and *output*. *Idx* holds the indices used to map the SSPM entries and *output* is the vector register destination to store the values.

vIdxStore.c: This instruction reads data from the VRF to store it in the SSPM in CAM-based mode. It has two source operands: *Data* and *idx*. *Data* holds the values to be stored in the SSPM. *idx* has the indices to map the SSPM entries.

vIdxLoad.i: This instruction loads VL consecutive indices from index table to write them in the VRF. It has two operands: *offset* and *output*. *offset* is the index of the first entry to start reading the *index table* structure. *output* is the vector register to store the values.

vIdxcount: Reads the element count register. It has a single operand *dst*, this operand is a scalar register. *vIdxcount* is used in the SpMA algorithm to determine the number of elements from the SSPM that will be stored in memory after execution.

vclear: This instruction resets either all or a segment of the entries in the *valid bitmap*; it clears the content from the *index table* structure and it sets to 0 the element count register. It has two operands: *full.mode* and *seg*. The *full.mode* determines if the instruction resets the full content or a segment of the bitmap. *seg* is used in the clear segment mode to specify the segment to reset of the bitmap.

(vIdxAdd, vIdxSub, vIdxMult).X: The 'X' value can be *.d* for the *Direct-map mode* configuration or *.c* for the *CAM mode*. These instructions read values from the SSPM to operate with data from the VRF. They have four operands: *Data, Idx, output* and *offset*. *Data* represents the data from the VRF to compute with the values from the SSPM. *idx* and *output* work in the same way as the previous *vldxmov* instruction. *offset* is a scalar value that is added to the values in *idx* to change the output positions when the *output* operand indicates that the instruction destination is the SSPM. These instructions always use data placed in the VRF (*Data*) to compute with values stored in the SSPM.

vIdxBlkMult.X: This instruction executes a block-based multiplication for a block-based format such as CSB. In a block based multiplication, the input-indices are split in two to represent the *cols* and *rows* relative to the block. Next, the input vector stored in the SSPM is read using the *cols* indices and is multiplied with the input data from the sparse block. Finally, the indices of the *rows* are used to accumulate the results for the output vector in the SSPM. It has four operands: *Data*, *Idx*, *Idx_offset* and *offset*. *Data* holds the input values from the sparse matrix. *Idx* are the in-block relative indices. *Idx_offset* represents the bit position where values from *Idx* operand are split into rows and columns. *offset* is the offset added to the *rows* indices to place the output values in the SSPM. This instruction has no output selection, the result is always placed in the SSPM.

D. Case Study of ISA Extensions - SpMV

To illustrate how the instructions explained in the previous section work, we detail how the SpMV computation is performed in VIA. We assume a matrix stored in CSB format with a single value to represent the row and column indices per each value.

Algorithm 4 describes the execution flow for SpMV using VIA and its ISA extensions. The first step is to clear the SSPM structure in the pre-execution step (line 1). Second, the input vector is loaded iteratively to the SSPM (lines 4-8). The vIdxmov instruction is used to move data from the VRF into SSPM. Next, we set the in-SSPM *offset* for the output vector chunk and the *idx_offset* used to split the input in-block indices into row and columns indices (lines 9-10). In the last step (lines 11-15), we read the indices and non-zero data from a sparse block from memory to the VRF, we use the *vIdxBlkMult* instruction to multiply with data in the SSPM, and accumulate the partial results back to the SSPM. The *vIdxBlkMult.d* instruction is executed in every iteration to compute the sparse block vector product for all the input data (line 14).

Algorithm 4 CSB-SpMV computation using VIA ISA extension

```
1:  vclear # clear SSPM
2:  Idx = base indices
3:  for all the blocks do
4:      for (i=0; i<block.cols; i=i+VL) do
5:          vmov &inVec[i+block_offset], Data
6:          vIdxload.d Data, Idx # load inVector
7:          increase Idx
8:      end for
9:      offset = block.cols
10:     idx_off = number of bits for columns
11:     for (i=0; i<block.size; i=i+VL) do
12:         vmov &block.Idx[i], Idx
13:         vmov &block.data[i], Data
14:         vIdxBlkMult.d Data, Idx, idx_off, offset # Mult_accum in SSPM
15:     end for
16: end for
```

E. VIA Integration In Out-of-Order Processors

The SSPM structure is direct mapped, thus VIA operands cannot be renamed. For this reason, there are two considerations to take into account to successfully integrate the VIA hardware to an out-of-order processor: ❶ to avoid SSPM pollution due to speculative execution, all the VIA instructions are executed at commit time; ❷ all the instructions detailed in Section IV-C are purely register operations, thus all their operands are registers. This allows the memory operations to be executed out-of-order between the VRF and the memory hierarchy while the VIA instructions will be executed only at commit time.

The most significant changes to the processor pipeline are concentrated in the issue, execute and commit stages. These changes are described next. Figure 8 depicts the general overview of the entire process.

Issue: VIA instructions become ready to execute when the Re-Order Buffer (ROB) notifies that they can be committed. More than one VIA instruction can be together in the same commit slot. Since VIA supports the parallel execution of these instructions, the issue logic will issue them to the FIVU unit.

Execution: VIA instructions execute the two first stages described in Section IV-B to read data from the SSPM. Then, they execute as a regular vector operation. Finally, results are stored back to the SSPM or the VRF depending on the instruction output destination.

Commit: When the VIA instructions can be committed, an *execution valid* signal is sent to the issue logic to mark the instruction as ready to be dispatched for execution. At writeback, an *execution completed* signal is sent back to the ROB and then the instruction is committed.

F. VIA Beyond Sparse Algebra

VIA hardware has been developed to maximize sparse algebra kernel performance. However, the architecture is general enough so that it can be customized to support other important kernels such as histogram generation or stencil computation. Next, we detail how VIA can be adapted to support these kernels.

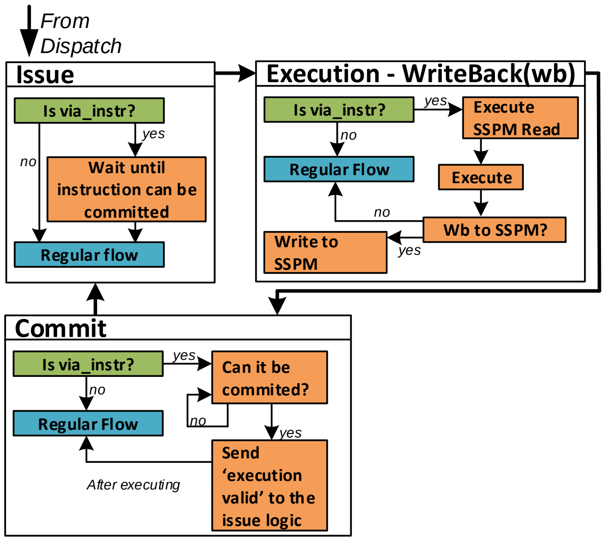


Fig. 8. The VIA hardware requirements when included in the pipeline of an out-of-order processor.

Algorithm 5 Histogram kernel using VIA

```
1: for all input data do
2:     read VL values from memory
3:     detect duplicate indices (vpconflicts)
4:     create elements count and mask
5:     use vIdxAdd.d to accumulate results at SSPM
6: end for
7: Store Histogram result at memory
```

1) Use Case 1, Histograms: Algorithm 5 describes the execution flow for a histogram kernel, implemented with a combination of the conflict detection instructions in AVX512CD extension (vpcd) [38], [39] and the VIA ISA. First, the VL values are loaded from memory to a vector register. Then, the *vpcd* instruction generates a conflict mask. Next, to merge the matching indices, a vector permutation sequence is executed [39]. Finally, the *vIdxAdd* updates and accumulates the histogram values in the SSPM. This step minimizes the store-load forwarding and reduces the memory traffic.

2) Use Case 2, Stencil Computation: Algorithm 6 describes the execution flow for a Gaussian convolution filter application, a typical stencil kernel. Compared to a VIA-oblivious implementation, our kernel reads the operand data from the SSPM instead of memory. The algorithm creates access pattern vectors to read data stored in the SSPM concerning the image and used filters. Then, the filters are stored in the SSPM to start the image processing. Next, an image's segment is read from memory and stored in the SSPM. Finally, the filter values are read from the SSPM and multiplied with the corresponding image sub-segment in all the filter iterations. The resulting vector is reduced and stored back to the SSPM, and the indices are updated to access the values in the next iteration. Finally, for the next image segment iteration, the access pattern vectors are regenerated.

Algorithm 6 Convolution Filter kernel using VIA

```
 1: Input: 2D image
 2: Create filter(s) access pattern vector (VR1)
 3: create data access pattern vector (VR2)
 4: Store filter(s) in SSPM
 5: for all image/segment_lenght segments do
 6:     Store a segment from input image in SSPM
 7:     for all values/(filter size) in SSPM do
 8:         use VR1 to read filter(s) values
 9:         multiply filter(s) values with data in VR2 positions
10:         reduce and accumulate results in SSPM
11:         Increase VR1 and VR2
12:     end for
13:     reset VR1 and VR2
14: end for
```

TABLE I
SIMULATED SYSTEM SETUP

CPU 2.0 GHz, skylake-like [21] [66] 4-wide superscalar OoO; 192-entriy ROB; 72-entry LQ; 42-entry SQ; Vector support: AVX2 and AVX512;	
L1 inst. + Data cache 32KB, 8-way assoc., 2-cycles; 64B line; 10 MSHR; Stride prefetcher	
L2 Cache 256KB, 4-way assoc., 12-cycles; 64B line; 16 MSHR; stride prefetcher	
DRAM 2GB DDR4	
VIA Hardware Configurations	
16_4p	Ports: 4, SSPM: 16KB, valid bitmap: 512B, Index Table: 4KB
16_2p	Ports: 2, SSPM: 16KB, valid bitmap: 512B, Index Table: 4KB
4_4p	Ports: 4, SSPM: 4KB, valid bitmap: 128B, Index Table: 1KB
4_2p	Ports: 2, SSPM: 4KB, valid bitmap: 128B, Index Table: 1KB

V. EXPERIMENTAL SETUP

A. Full-System Simulation Infrastructure

We model and evaluate VIA using *Gem5* [7], [12] to simulate an x86 full-system running an Ubuntu 16.04 OS with a 4.9.4 Linux Kernel. We simulate a single core processor using the out-of-order CPU and memory models, extended with the proposed processing pipeline changes to support VIA ISA extensions and hardware. Table I summarizes the main simulation parameters, including the different VIA hardware configurations evaluated in Section VI. As explained in Section IV, the SSPM structure requires multiple cycles to process an instruction when the VL is larger than the number of ports. We model these latency in detail in Gem5.

Power consumption is evaluated with McPAT [49] for 22nm technology, a voltage of 0.8V and the default clock gating scheme. We incorporate the changes suggested by Xi et al. [68] to improve the accuracy of the models. The VIA structures are modeled in CACTI 6.5 [55], adding the appropriate counters in gem5 to measure the extra power. In addition, for a more accurate evaluation of the area, the VIA design is implemented in RTL and synthesized on a commercial standard cell library in 22nm technology. All configurations from Table I are explored and the results for the area and leakage power are discussed in Section VI.

B. Benchmarks

We evaluate VIA efficiency using three representative sparse matrix kernels: Sparse Matrix Vector multiplication (SpMV), Sparse Matrix Addition (SpMA) and Sparse Matrix Matrix multiplication (SpMM). There has been a good deal of research work on compressed formats for the SpMV operation. We evaluate SpMV using four different compressed formats representations (all of them are supported in VIA): CSR (from Eigen library) [14], [23], [31], SPC5 [9], Sell-c-σ [44] and CSB [10], [11]. The block size of the CSB format has been tuned to match half of the SSPM storage capacity. For the other three formats, we make use of the implementations and best performance configurations provided by the developers of CSR, SPC5 and Sell-c-σ. For the SpMA and SpMM kernels, we make use of the CSR representation for each kernel and compare against a CSR vectorized implementation from the Eigen library [23], [31] a highly optimized state-of-the-art C++ library.

As input dataset, we use 1,024 sparse matrices from 56 different application domains of the *University of Florida Sparse Matrix Collection* [17]. From all the available matrices, we select the ones that consist of real numbers and are square matrices with row count lower or equal to 20,000 (due to simulation time constraints). The selected sparse matrices have between 0.01% and 2,6% non-zero elements.

VI. DESIGN SPACE EXPLORATION

In this section, we evaluate the VIA configurations from Table I to size VIA hardware. As explained in Section III, VIA is composed of two main building blocks, SSPM and FIVU. From these two, SSPM is the more critical component. The *SSPM* structure has two key simulation-configurable blocks: number of ports and memory size (which includes the *SRAMs* and *Index table* structures). Both blocks impact the performance, area and leakage power of VIA. We compare the CSB VIA-SpMV implementation that depicts the best performance results in Section VII-A, and the CSR VIA-SpMA and VIA-SpMM implementations. We evaluated the VIA configurations from Table I.

A. SSPM size and number of ports impact

We start analyzing the impact on performance of the number of ports and size of SSPM. On one hand, the number of ports determines the number of cycles a VIA instruction takes to access data in the SSPM. On the other hand, the SSPM size determines the maximum number of non-zero values that can be computed in each step. Bigger SSPM sizes reduce the store-load forwarding because data is accessed and reordered in the SSPM. Figure 9 shows the performance speedup obtained when the SSPM number of ports and memory sizes vary. All kernels are evaluated separately, and each kernel is normalized to its own 4_2p configurations.

VIA SpMV algorithm (figure 9 SpMV columns) provides on average 2% better performance when using a 4_4p configuration, 26% using a 16_2p and 33% using 16_4p. For VIA SpMA algorithm (Figure 9 SpMA columns), the 16_4p configuration has the highest performance improvement of 20% on average, while configurations 16_2p and 4_4p present speedups of 16% and 4% respectively. The VIA SpMM kernel is constrained by the average number of non-zero elements per row in one of the input matrices;

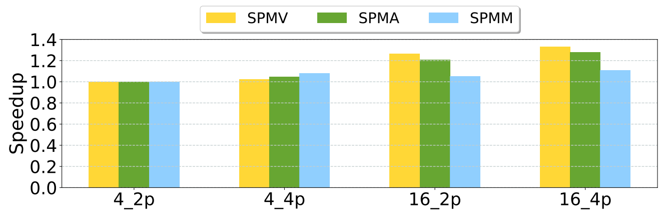


Fig. 9. SSPM size configuration speedup comparison for the SpMV, SpMA and SpMM kernels. Each kernel results are normalized to its own 4_2p configuration performance.

TABLE II
AREA AND POWER CONSUMPTION FOR 4 DIFFERENT SSPM CONFIGURATION (22NM TECHNOLOGY)

16_4p area: 0.827 mm^2; Leakage: 0.69 mW;
16_2p area: 0.515 mm^2; Leakage: 0.5 mW;
4_4p area: 0.180 mm^2; Leakage: 0.22 mW;
4_2p area: 0.118 mm^2; Leakage: 0.14 mW;

and this kernel is mostly executed using the *Index table* structure. The input dataset is highly sparse, thus the number of elements per row is really low. For most of the input matrices the smaller SSPM-(*Index table*) configuration is enough to store almost all the longest rows from each input matrix. For this reason, performance results of this kernel varies more when the number of ports is increased (See Figure 9 SpMM columns). Results show that the 4_2p configuration is outperformed on average by 11%, 5% and 8% when comparing with the 16_4p, 16_2p and 4_4p respectively. In the three algorithms, the 16_4p configuration provides the best performance. However, the 16_2p configuration performs almost as good as 16_4p with a modest reduction in performance while using half of the ports.

B. Area and Power

Next, we measure the area and leakage power for the different configurations to facilitate performance comparisons. We synthesized all the configurations of SSPM using the Genus tool from Cadence [13] using a 22nm technology and a target frequency of 2GHz. To increase the number of ports in SSPM memory, we modeled it using the *Live Value Table* technique [1], [45], which coordinates multiple banks of memory into a true multi-ported memory. This technique increases the area but allows the system to use faster SRAM structures for the storage. Table II summarizes the results for the area and leakage power.

As expected, 4_Xp configurations feature the smallest leakage power, energy and area results, while the 16_Xp configurations get the biggest. Nevertheless, the performance for the 16_Xp configurations outperforms the 4_Xp ones by an average of 20%. Comparing the area of 16_Xp configurations with a single 22nm Intel Haswell core [32], VIA increases the area by 5% for 16_4p and 3% for 16_2p. The increase in the whole chip area is 1.5% and 1% respectively. We synthesized two extra VIA configurations: 8_4p (8KB, 4 ports and CAM:2KB) and 8_2p (8KB, 2 ports and CAM:2KB). Results indicate area results of 0.43 mm^2 and 0.29 mm^2 and leakage power of 0.39mW and 0.28mW respectively.

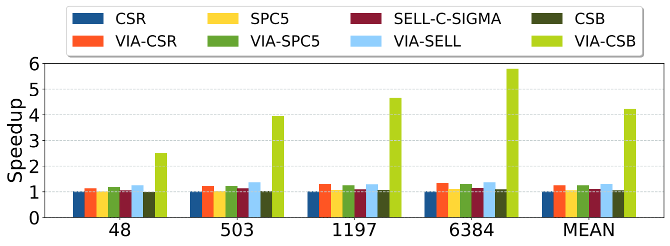


Fig. 10. Speedup for VIA SpMV kernel. Results are normalized to the CSR implementation for every category.

Area and leakage power benefits from reducing the number of ports while maintaining the memory size are significant compared with the small performance reduction overhead that this change entails. Thus, the 16 KB size and 2 ports configuration present the best combination of performance and resource consumption. Thus, results in the next Section are obtained using this configuration.

VII. EVALUATION

In this section, we evaluate the performance of VIA-SpMV, VIA-SpMA and VIA-SpMM using the baseline explained at Section V-B and the hardware configuration of 16_2p.

A. SpMV Performance

Figure 10 depicts performance results of the different VIA-SpMV implementations for all the input dataset. The most noteworthy results are presented by the CSB version. All the evaluated matrices were sorted by the CSB block density and evenly split among 4 categories. The x-Axis at Figure 10 shows the median non-zero values per block among each category. VIA SpMV achieves on average speedup of 4.22× when executing using CSB; and average speedups of 1.25×, 1.24× and 1.31× over the CSR, SPC5 and Sell-C-σ implementations repectively. The CSB format increases the locality of the input and output vectors, thus a chunk of the input vector needs to be placed in SSPM only once to compute with a single block. For the other formats, the indices to map the input or output vectors of two consecutive matrix values can be really sparse, thus the efficiency of VIA is limited to work as an accumulator for the output vector. Nevertheless, even with this limitation, VIA improves performance over the other formats by 1.26× on average.

For the best usage case (executing with CSB VIA-SpMV), VIA: (1) reduces the total energy consumption (leakage + dynamic) by a factor of 3.8×. (2) Increases the memory bandwidth by 2.5×. We make two observations. First, using VIA with CSB allows to work with bigger block ranges; when the range of the blocks is increased, the number of non-zero elements per block will increase too. Second, with more useful elements per block, the time used to preload into SSPM the input vector chunk for each block is negligible compared with the time of reading the block non-zero values. The reading bandwidth increases, thus the performance.

B. SpMA Performance

Figure 11 shows the performance of the SpMA kernel (VIA-CSR-SPMA column). In a similar manner to SpMV, results from SpMA were sorted and evenly split into 4 categories. As we use CSR format in the SpMA construction, we used the non-zero elements

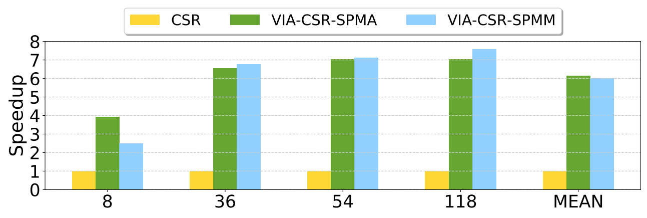


Fig. 11. Speedup for VIA SpMA and VIA SpMM. Both Kernels are normalized to their base CSR implementation.

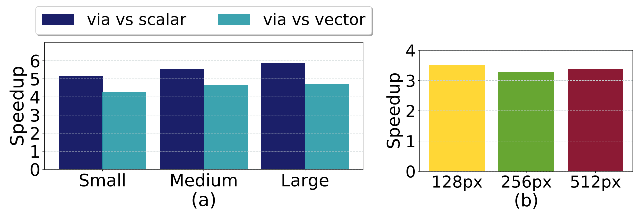


Fig. 12. Speedup for VIA histogram(a) and Gaussian convolution filter(b) kernels

per row as the criteria to sort the entire input dataset. In figure 11, the x-Axis presents the median non-zero elements per row among each category.

On average, VIA achieves 6.14× speedup on the input dataset (Figure 11). In terms of energy, VIA SpMA reduces on average 5.6× of the total energy and increases the memory bandwidth by a factor of 2.1×. We make one observation. The index matching operations regarding to the SpMA are difficult to vectorize with conventional VFUs. The components of VIA allow to vectorize the index matching computation with a single vector instruction without any extra software comparisons. This capability helps to reduce the memory traffic, reduce the store-load forwarding and to increase the efficiency of the VFU over this kernel.

C. *SpMM Performance*

Figure 11 shows the performance of the SpMM kernel (VIA-CSR-SPMM column). VIA outperforms the baseline by an average speedup of 6×. The benefits of VIA come from the *CAM* mapping hardware support. VIA speeds up the index matching operation by transforming it to an internal SSPM operation. VIA can execute the full process of index matching, multiplication and output accumulation with a single vIdxBlkMult instruction. Matrices from real world applications have few non-zero elements per row, thus VIA manages to improve the efficiency of the VFU over this kernel. In terms of energy reduction and memory bandwidth, VIA outperforms the conventional CSR by 5.1× and 3.2× respectively.

D. *Histograms and Stencil Applications*

Figure 12(a) depicts the performance of VIA histogram kernel (via_hist) presented in Section IV-F1. We evaluated via_hist using the scalar and vectorized histogram implementations from the intel-optimization manual [39]. Each algorithm is evaluated with 3 random uniform distributed datasets, *small*, *medium* and *big*, which contain 51,200, 512,000 and 5,120,000 elements respectively. A *Small* dataset is able to reside in the L2 cache while the latter two increase in size by one and two orders of magnitude respectively. This helps to identify trends in performance with different input sizes and also to pinpoint any effect due to the cache. On average, via_hist outperforms intel scalar and vector by 5.49× and 4.51× respectively. This algorithm is constrained by memory bandwidth, and reducing the traffic as well as store-load forwarding provides significant speedups.

Gaussian filtering [51] is a very important application for image processing and computer vision. We employ a classic implementation of a 4x4 Gaussian filter as a representative application for the broad class of stencil computation. We evaluate using 3 images with different sizes, 128x128pixels (128px), 256x256pixels (256px) and 512x512pixels (512px) respectively. Figure 12.b depicts speedup results for VIA Gaussian filter. VIA outperforms the baseline by 3.39×. Improvements in this algorithm come from the efficient access of neighbor values directly from SSPM, also reducing the kernel execution time.

VIII. Related Work

A great deal of software solutions has been published on accelerating sparse algebra kernels, mostly for SpMV [4], [9]–[11], [24], [25], [29], [36], [50], [53], [63], [64], but also for SpMM [35], [67], [70]. Most of these works are based on format and data transformations, where **block-based** sparse matrix representations have received most attention for two main reasons: *1)* sparse matrices in real applications generally have a block sub-structure, and *2)* on-chip memory requests may be decreased when using block relative indices instead of directly using row/column ones.

Of the most prominent software approaches, Liu et al. [50] propose CSR5, which is an extension to the CSR format. CSR5 represents the matrix using 2D blocks named tiles, adding new structures to facilitate in-tile fast data search. Their results show average performance improvements of 17.6%, 28.5%, 173.0% and 293% over the baselines of dual-socket Intel CPUs, Nvidia GPU, AMD GPU and Intel Xeon Phi, respectively. Bramas et al. [9] propose SPC5, which is another block-based format without zero padding, optimized for the AVX-512 engines. SPC5 presents 2.5× average speedups over CSR5.

Compared to the software approaches that are mainly focused on general solutions among different platforms, the hardware works have focused on application specific accelerators. The work proposed by Pal et al. [57] and Sadi et al. [59], implement hardware-software co-designed accelerators for sparse algebra kernels. VIA employs a similar concept for a software-oriented hardware implementation. However, both previous works are only applicable for specific algorithms, like SpMV or SpMM, and they leverage dedicated hardware for their operations, like HBM2 technology, which by itself dramatically increases the available bandwidth. Our architecture has been implemented and attached to a commodity out-of-order processor architecture and compared with the same architecture. Also, VIA aims to present a more general solution that can be applied to a wider set of sparse algebra kernels and compressed representations. Zhang et al. [72] proposed SpArch, a Sparse Matrix Multiplication accelerator. Similar to [59] and [57], SpArch is only applicable to SpMM and it is a standalone accelerator instead of being integrated in a general purpose core. Although it is only designed for SpMM, SpArch consumes 14x more area than VIA when compared using the same technology node (22nm). More recently, Kannellopoulos et al. [41] propose SMASH, a software-hardware co-designed scheme within the CPU where the compression is software-based whereas indexing is accelerated in hardware. SMASH only achieves 1.38× average speedups for both SpMV and SpMM kernels using its own compressed representation. In comparison, VIA accelerates different compressed representations with noteworthy results when using CSB format with an average speedup of 4.22×. Finally, the NEC SX-ACE vector computer [22], [54] uses a highly ported memory named ADB (Assignable Data Buffer) as last level cache, directly connected to the scalar and vector processing units. The ADB ports provide enough bandwidth to all the lanes. Our SSPM follows a similar principle by adding a multi-ported storage structure to accelerate kernels with indirect memory access or index matching operations. Although SX-ACE has a large chip area of 570mm^2, it is a traditional vector architecture and thus it does not feature any hardware optimized for sparse matrix operations.

IX. Conclusions

In this paper, we introduce VIA, a specialized vector architecture that significantly improves performance over sparse linear algebra computations. The main goal of VIA is (1) to reduce the memory traffic incurred by memory indexed operations, and (2) to improve the efficiency of vector architectures over index matching operations. To this end, we develop a smart scratchpad memory specifically designed to tackle both issues mentioned previously. This scratchpad memory makes use of two different content mapping techniques for the two execution scenarios of sparse-dense and sparse-sparse computations. As a result, VIA greatly reduces the performance overheads of memory indexed operations and index matching operations. The rich set of new VIA instructions provides with a simple and general interface to program the hardware, facilitating its adaptation to any SIMD ISA in the market. Our evaluation over a diverse set of 1,024 matrices from real applications demonstrates that VIA significantly improves the performance of SpMV, SpMA, and SPMM, compared to different state-of-the-art solutions. In addition, we demonstrate the generality and applicability of VIA to other important kernels that share characteristics with sparse matrices. Our evaluation with histogram and stencil computations demonstrates the effectiveness of VIA in applications with irregular memory access patterns. Moreover, we believe that VIA is applicable to other application domains such as graph computing and bioinformatics.

X. acknowledgements

The authors would like to thank the anonymous shepherd who meticulously reviewed and commented the text leading to a considerably improved paper. We would also like to thank all of our anonymous reviewers for their valuable feedback. This work has been supported by the European HiPEAC Network of Excellence, by the Spanish Ministry of Science and Innovation (contract TIN2015-65316-P), by the Generalitat de Catalunya (contracts 2017-SGR-1414 and 2017-SGR-1328), and by the DRAC project, which is co-financed by the European Union Regional Development Fund within the framework of the ERDF Operational Program of Catalonia 2014-2020 with a grant of 50% of total eligible cost. A. Barredo has been supported by the Spanish Government under Formación del Personal Investigador fellowship number BES-2017-080635. M. Moreto has been partially supported by the Spanish Ministry of Economy, Industry and Competitiveness under Ramon y Cajal fellowship number RYC-2016-21104.

References

[1] A. M. Abdelhadi and G. G. Lemieux, "Modular multi-ported sram-based memories," in *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, 2014, pp. 35–44.

[2] L. Alvarez, L. Vilanova, M. Moreto, M. Casas, M. Gonzàlez, X. Martorell, N. Navarro, E. Ayguadé, and M. Valero, "Coherence protocol for transparent management of scratchpad memories in shared memory manycore architectures," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2015, pp. 720–732.

[3] AMD Corporation, "AMD64 Architecture Programmer's Manual Volume 4:128-Bit and 256-Bit Media Instructions," 2019.

[4] H. Anzt, S. Tomov, P. Luszczek, W. Sawyer, and J. Dongarra, "Acceleration of gpu-based krylov solvers via data transfer reduction," *The International Journal of High Performance Computing Applications*, vol. 29, no. 3, pp. 366–383, 2015.

[5] K. Asanoviĉ, "Vector Microprocessors," Ph.D. dissertation, University of California, Berkeley, 1998.

[6] J. C. Beard, "The Sparse Data Reduction Engine (SPiDRE): Chopping Sparse Data One Byte at a Time," in *Proceedings of the Second International Symposium on Memory Systems*, 2017.

[7] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib Bin Altaf, N. Vaish, M. Hill, and D. Wood, "The gem5 simulator," *SIGARCH Computer Architecture News*, vol. 39, pp. 1–7, 08 2011.

[8] B. Bramas and P. Kus, "Computing the sparse matrix vector product using block-based kernels without zero padding on processors with AVX-512 instructions," *CoRR*, vol. abs/1801.01134, 2018. [Online]. Available: http://arxiv.org/abs/1801.01134

[9] B. Bramas and P. Kus, "Computing the sparse matrix vector product using block-based kernels without zero padding on processors with avx-512 instructions," *PeerJ Computer Science*, vol. 4, p. e151, 2018.

[10] 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 *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*. ACM, 2009, pp. 233–244.

[11] A. Buluc, S. Williams, L. Oliker, and J. Demmel, "Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication," in *2011 IEEE international Parallel & distributed processing symposium (IPDPS)*. IEEE, 2011, pp. 721–733.

[12] A. Butko, R. Garibotti, L. Ost, and G. Sassatelli, "Accuracy evaluation of gem5 simulator system," in *7th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*. IEEE, 2012, pp. 1–7.

[13] Cadence, "Genus Synthesis Solution," Available at https://www.cadence.com/content/cadence-www/global/en_US/home/tools/digital-design-and-signoff/synthesis.html. [Online]. Available: https://www.cadence.com/content/cadence-www/global/en_US/home/tools/digital-design-and-signoff/synthesis/genus-synthesis-solution.html

[14] C. Chan, "Optimizations to the orthogonal matching pursuit algorithm for sparse basis representations of photometric redshift pdfs," *University of Illinois at Urbana-Champaign*, 2016.

[15] C. H. Chou, A. Severance, A. D. Brant, Z. Liu, S. Sant, and G. G. Lemieux, "Vegas: Soft vector processor with scratchpad memory," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, 2011, pp. 15–24.

[16] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Optimization and performance modeling of stencil computations on modern microprocessors," *SIAM review*, vol. 51, no. 1, pp. 129–159, 2009.

[17] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, p. 1, 2011.

[18] K. Dawson-Howe, *A practical introduction to computer vision with opencv*. John Wiley & Sons, 2014.

[19] E. F. D'Azevedo, M. R. Fahey, and R. T. Mills, "Vectorized sparse matrix multiply for compressed row storage format," in *Computational Science – ICCS 2005*, V. S. Sunderam, G. D. van Albada, P. M. A. Sloot, and J. J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 99–106.

[20] J. Dongarra, M. A. Heroux, and P. Luszczek, "High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems," *The International Journal of High Performance Computing Applications*, vol. 30, no. 1, pp. 3–10, 2016.

[21] J. Doweck, W.-F. Kao, A. K.-y. Lu, J. Mandelblat, A. Rahatekar, L. Rappoport, E. Rotem, A. Yasin, and A. Yoaz, "Inside 6th-generation intel core: New microarchitecture code-named skylake," *IEEE Micro*, vol. 37, no. 2, pp. 52–62, 2017.

[22] R. Egawa, K. Komatsu, S. Momose, Y. Isobe, A. Musa, H. Takizawa, and H. Kobayashi, "Potential of a modern vector supercomputer for practical applications: performance evaluation of sx-ace," *The Journal of Supercomputing*, vol. 73, no. 9, pp. 3948–3976, 2017.

[23] Eigen, "The matrix class, dense matrix and array manipulation," https://www.eigen.tuxfamily.org/dox/group_TutorialMatrixClass.html, accessed: 2017-10-01.

[24] A. Elafrou, G. Goumas, and N. Koziris, "Basmat: Bottleneck-aware sparse matrix-vector multiplication auto-tuning on gpgpus," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '19. New York, NY, USA: ACM, 2019, pp. 423–424. [Online]. Available: http://doi.acm.org/10.1145/3293883.3301490

[25] A. Elafrou, G. Goumas, and N. Koziris, "Conflict-free symmetric sparse matrix-vector multiplication on multicore architectures," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: ACM, 2019, pp. 48:1–48:15. [Online]. Available: http://doi.acm.org/10.1145/3295500.3356148

[26] R. Espasa, M. Valero, and J. E. Smith, "Vector Architectures: Past, Present and Future," in *Proceedings of the 12th International Conference on Supercomputing (ICS)*, 1998, pp. 425–432. [Online]. Available: http://doi.acm.org/10.1145/277830.277935

[27] T. Evgeniou and M. Pontil, "Support vector machines: Theory and applications," in *Advanced Course on Artificial Intelligence*. Springer, 1999, pp. 249–257.

[28] O. Fuhrer, T. Chadha, T. Hoefler, G. Kwasniewski, X. Lapillonne, D. Leutwyler, D. Lüthi, C. Osuna, C. Schär, T. C. Schulthess *et al.*, "Near-global climate simulation at 1 km resolution: establishing a performance baseline on 4888 gpus with cosmo 5.0," *Geoscientific Model Development*, vol. 11, no. 4, pp. 1665–1681, 2018.

[29] D. Fujiki, N. Chatterjee, D. Lee, and M. O'Connor, "Near-memory data transformation for efficient sparse matrix multi-vector multiplication," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: ACM, 2019, pp. 55:1–55:17. [Online]. Available: http://doi.acm.org/10.1145/3295500.3356154

[30] J. R. Gilbert, S. Reinhardt, and V. B. Shah, "A unified framework for numerical and combinatorial computing," *Computing in Science & Engineering*, vol. 10, no. 2, pp. 20–25, 2008.

[31] G. Guennebaud, B. Jacob *et al.*, "Eigen," *URl: http://eigen. tuxfamily. org*, 2010.

[32] P. Hammarlund, A. J. Martinez, A. A. Bajwa, D. L. Hill, E. Hallnor, H. Jiang, M. Dixon, M. Derr, M. Hunsaker, R. Kumar *et al.*, "Haswell: The fourth-generation intel core processor," *IEEE Micro*, vol. 34, no. 2, pp. 6–20, 2014.

[33] T. Hayes, O. Palomar, O. Unsal, A. Cristal, and M. Valero, "Vsr sort: A novel vectorised sorting algorithm & architecture extensions for future microprocessors," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2015, pp. 26–38.

[34] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Sixth Edition: A Quantitative Approach*, 6th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017.

[35] C. Hong, A. Sukumaran-Rajam, I. Nisa, K. Singh, and P. Sadayappan, "Adaptive sparse tiling for sparse matrix multiplication," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '19. New York, NY, USA: ACM, 2019, pp. 300–314. [Online]. Available: http://doi.acm.org/10.1145/3293883.3295712

[36] K. Hou, W.-c. Feng, and S. Che, "Auto-tuning strategies for parallelizing sparse matrix-vector (spmv) multiplication on multi-and many-core processors," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2017, pp. 713–722.

[37] Intel Corporation, "Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A: Instruction Set Reference." 2015.

[38] Intel Corporation, "Improve Vectorization Performance with Intel AVX-512." 2016. [Online]. Available: https://software.intel.com/en-us/articles/improve-vectorization-performance-using-intel-advanced-vector-extensions-512

[39] Intel Corporation, "Intel 64 and IA-32 Architectures Optimization Reference Manual." 2019. [Online]. Available: https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-optimization-reference-manual

[40] Z. Istvan, L. Woods, and G. Alonso, "Histograms as a side effect of data movement for big data," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, 2014, pp. 1567–1578.

[41] K. Kanellopoulos, N. Vijaykumar, C. Giannoula, R. Azizi, S. Koppula, N. M. Ghiasi, T. Shahroodi, J. G. Luna, and O. Mutlu, "Smash: Co-designing software compression and hardware-accelerated indexing for efficient sparse matrix operations," in *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: ACM, 2019, pp. 600–614. [Online]. Available: http://doi.acm.org/10.1145/3352460.3358286

[42] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke *et al.*, "Mathematical foundations of the GraphBLAS," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2016, pp. 1–9.

[43] F. Kjolstad, S. Chou, D. Lugato, S. Kamil, and S. Amarasinghe, "Taco: A tool to generate tensor algebra kernels," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 943–948.

[44] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop, "A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide simd units," *SIAM Journal on Scientific Computing*, vol. 36, no. 5, pp. C401–C423, 2014.

[45] C. E. LaForest and J. G. Steffan, "Efficient multi-ported memories for fpgas," in *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, 2010, pp. 41–50.

[46] J. Leskovec, L. A. Adamic, and B. A. Huberman, "The dynamics of viral marketing," *ACM Transactions on the Web (TWEB)*, vol. 1, no. 1, pp. 5–es, 2007.

[47] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[48] J. Leskovec and R. Sosič, "Snap: A general-purpose network analysis and graph-mining library," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 8, no. 1, pp. 1–20, 2016.

[49] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," in *Proceedings of the 42nd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, Dec. 2009, pp. 469–480.

[50] W. Liu and B. Vinter, "Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication," in *Proceedings of the 29th ACM on International Conference on Supercomputing*. ACM, 2015, pp. 339–350.

[51] D. Marr and E. Hildreth, "Theory of edge detection," *Proceedings of the Royal Society of London. Series B. Biological Sciences*, vol. 207, no. 1167, pp. 187–217, 1980.

[52] G. A. McMechan, "Migration by extrapolation of time-dependent boundary values," *Geophysical prospecting*, vol. 31, no. 3, pp. 413–420, 1983.

[53] D. Merrill and M. Garland, "Merge-based sparse matrix-vector multiplication (spmv) using the csr storage format," in *ACM SIGPLAN Notices*, vol. 51, no. 8. ACM, 2016, p. 43.

[54] S. Momose, T. Hagiwara, Y. Isobe, and H. Takahara, "The brand-new vector supercomputer, sx-ace," in *Proceedings of the 29th International Conference on Supercomputing - Volume 8488*, ser. ISC 2014. New York, NY, USA: Springer-Verlag New York, Inc., 2014, pp. 199–214. [Online]. Available: http://dx.doi.org.recursos.biblioteca.upc.edu/10.1007/978-3-319-07518-1_13

[55] N. Muralimanohar and R. Balasubramonian, "CACTI 6.0: A Tool to Understand Large Caches," Available at https://github.com/HewlettPackard/cacti.

[56] S. N., "Understanding gather-scatter instructions and the -gather-scatter-unroll compiler switch," https://software.intel.com/en-us/articles/understanding-gather-scatter-instructions-and-the-gather-scatter-unroll-compiler-switch, accessed: 2017-09-21.

[57] S. Pal, J. Beaumont, D. Park, A. Amarnath, S. Feng, C. Chakrabarti, H. Kim, D. Blaauw, T. Mudge, and R. Dreslinski, "Outerspace: An outer product based sparse matrix multiplication accelerator," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2018, pp. 724–736.

[58] M. O. Rabin and V. V. Vazirani, "Maximum matchings in general graphs through randomization," *Journal of Algorithms*, vol. 10, no. 4, pp. 557–567, 1989.

[59] F. Sadi, J. Sweeney, T.-M. Low, J. C. Hoe, L. Pileggi, and F. Franchetti, "Efficient spmv operation for large and highly sparse matrices using scalable multi-way merge parallelization," in *MICRO*, 2019.

[60] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey, "Can Traditional Programming Bridge the Ninja Performance Gap for Parallel Computing Applications?" in *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA)*, 2012, pp. 440–451. [Online]. Available: http://dl.acm.org.recursos.biblioteca.upc.edu/citation.cfm?id=2337159.2337210

[61] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, A. Reid, A. Rico, and P. Walker, "The ARM Scalable Vector Extension," *IEEE Micro*, vol. 37, no. 2, pp. 26–39, 2017.

[62] H. Tan, H. Chen, S. Liu, and J. Wu, "Modeling and evaluation for gather/scatter operations in vector-simd architectures," in *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, July 2017, pp. 143–148.

[63] W. T. Tang, R. Zhao, M. Lu, Y. Liang, H. P. Huyng, X. Li, and R. S. M. Goh, "Optimizing and auto-tuning scale-free sparse matrix-vector multiplication on intel xeon phi," in *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2015, pp. 136–145.

[64] R. W. Vuduc and J. W. Demmel, *Automatic performance tuning of sparse matrix kernels*. University of California, Berkeley, 2003, vol. 1.

[65] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang, "Intel math kernel library," in *High-Performance Computing on the Intel® Xeon Phi™*. Springer, 2014, pp. 167–188.

[66] wikichip, "Skylake (client) - Microarchitectures - Intel," 2020. [Online]. Available: https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client)

[67] M. Winter, D. Mlakar, R. Zayer, H.-P. Seidel, and M. Steinberger, "Adaptive sparse matrix-matrix multiplication on the gpu," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '19. New York, NY, USA: ACM, 2019, pp. 68–81. [Online]. Available: http://doi.acm.org/10.1145/3293883.3295701

[68] S. Xi, H. Jacobson, P. Bose, G.-Y. Wei, and D. Brooks, "Quantifying sources of error in McPAT and potential impacts on architectural studies," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 577–589.

[69] X. Xie, D. Du, Q. Li, Y. Liang, W. T. Tang, Z. L. Ong, M. Lu, H. P. Huynh, and R. S. M. Goh, "Exploiting sparsity to accelerate fully connected layers of cnn-based applications on mobile socs," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 17, no. 2, pp. 1–25, 2017.

[70] Z. Xie, G. Tan, W. Liu, and N. Sun, "Ia-spgemm: An input-aware auto-tuning framework for parallel sparse matrix-matrix multiplication," in *Proceedings of the ACM International Conference on Supercomputing*, ser. ICS '19. New York, NY, USA: ACM, 2019, pp. 94–105. [Online]. Available: http://doi.acm.org/10.1145/3330345.3330354

[71] A. Yzelman, "Generalised vectorisation for sparse matrix: vector multiplication," in *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*. ACM, 2015, p. 6.

[72] Z. Zhang, H. Wang, S. Han, and W. J. Dally, "Sparch: Efficient architecture for sparse matrix multiplication," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 261–274.