Dedicated hardware accelerators for processing of sparse matrices and vectors: A survey

Valentin Isaac Chassande, Adrian Evans, Yves Durand, Frédéric Rousseau

Dedicated Hardware Accelerators for Processing of Sparse Matrices and Vectors: A Survey

VALENTIN ISAAC–CHASSANDE, ADRIAN EVANS, and YVES DURAND, Université Grenoble Alpes, CEA, LIST, Grenoble, France
FRÉDÉRIC ROUSSEAU, Université Grenoble Alpes, CNRS, Grenoble INP, TIMA, Saint-Martin-d'Heres, France

Performance in scientific and engineering applications such as computational physics, algebraic graph problems or Convolutional Neural Networks (CNN), is dominated by the manipulation of large sparse matrices—matrices with a large number of zero elements. Specialized software using data formats for sparse matrices has been optimized for the main kernels of interest: SpMV and SpMSpM matrix multiplications, but due to the indirect memory accesses, the performance is still limited by the memory hierarchy of conventional computers. Recent work shows that specific hardware accelerators can reduce memory traffic and improve the execution time of sparse matrix multiplication, compared to the best software implementations. The performance of these sparse hardware accelerators depends on the choice of the sparse format, *COO*, *CSR*, etc, the algorithm, *inner-product*, *outer-product*, *Gustavson*, and many hardware design choices. In this article, we propose a systematic survey which identifies the design choices of state-of-the-art accelerators for sparse matrix multiplication kernels. We introduce the necessary concepts and then present, compare, and classify the main sparse accelerators in the literature, using consistent notations. Finally, we propose a taxonomy for these accelerators to help future designers make the best choices depending on their objectives.

CCS Concepts: • **Hardware** → **Application specific processors** • **General and reference** → **Surveys and overviews**; *Performance;* • **Computing methodologies** → *Computer algebra systems;* • **Computer systems organization** → *Multicore architectures*;

Additional Key Words and Phrases: High performance computing (HPC), accelerators, sparse matrices, memory hierarchy

ACM Reference Format:
Valentin Isaac–Chassande, Adrian Evans, Yves Durand, and Frédéric Rousseau. 2024. Dedicated Hardware Accelerators for Processing of Sparse Matrices and Vectors: A Survey. *ACM Trans. Arch. Code Optim.* 21, 2, Article 27 (February 2024), 26 pages. https://doi.org/10.1145/3640542

Authors' addresses: V. Isaac–Chassande, A. Evans, and Y. Durand, Université Grenoble Alpes, CEA, LIST, 17 Avenue des Martyrs, Grenoble 38000, France; e-mails: valentin.isaacchassande@cea.fr, adrian.evans@cea.fr, yves.durand@cea.fr; F. Rousseau, Université Grenoble Alpes, CNRS, Grenoble INP, TIMA, 46 Avenue Félix Viallet, Grenoble 38000, France; e-mail: Frederic.Rousseau@univgrenoble-alpes.fr.

ACM 1544-3566/2024/02-ART27
https://doi.org/10.1145/3640542

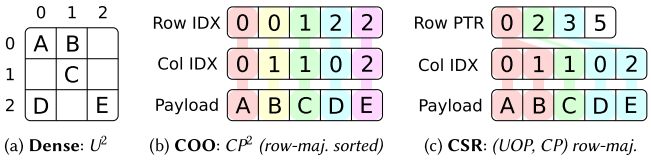
1 INTRODUCTION

The execution time of modern scientific and engineering applications is dominated by the manipulation of large sparse matrices, i.e., matrices whose number of non-zero entries NNZ is much smaller than their dimensions $M \times N$. Matrices derived from computational physics problems are often sparse and regular because they derive from a relatively small number of differential terms. Algebraic graph problems produce matrices which are still sparse but usually less regular [16]. Similarly, the weight and activation matrices used by **Convolutional Neural Networks** (**CNN**) may hold millions of terms, but most of them are explicitly set to zero (e.g., *rectified*) without loss of information [30]. It was recognized early in the 1960s [69] that working on these sparse datasets using dense array representations was inefficient. The seminal work of OSKI [83] in 2003 formalized the sparse representation alternatives and identified many optimizations. This opened the path for specialized software libraries for sparse algebra [84]. However, the single-core performance of software implementations can be deceiving, often only reaching 10% of peak performance [30].

The main matrix multiplication kernels that are of interest are **matrix-vector** (**MV**) and **matrix-matrix** (**MM**), which include SpMV and SpMSpV ($A \times b = c$), SpGEMV ($A \times b + d = c$), SpMM and SpMSpM ($A \times B = C$), and SpGEMM ($A \times B + D = C$), with *Sp* for *sparse* and *GE* for *general*. SpMV, as well as its variant SpGEMV, are by far the dominant operations of modern algebraic kernels (linear solvers, eigensolvers) which have been explicitly based on them [25, 34, 39, 44, 49, 66, 87]. Since these kernels are ubiquitous, their efficient implementation is primordial. In contrast, the SpMSpM multiplication is less frequent in standard algebraic kernels, but it is extensively used in algebraic graph manipulation [5, 14, 17, 23, 33, 46, 67], as well as in multigrid solvers [4, 13, 53, 68]. The growing importance of large graphs has motivated new research on optimizing SpMSpM applications.

The two matrix operations, SpMV and SpMSpM, pose orthogonal computational challenges: (1) access to sparse input data, (2) *reduction* operations on the intermediate result, and (3) formatting and writing back the result to memory. The different algorithms, which are detailed in Section 2, are built on different articulations of these three aspects. The sparse data storage scheme used in memory dictates the first aspect and is briefly explored in Section 2.1. The third aspect, i.e., result storage, is sometimes overlooked when the output is dense and not too large. For example, when running on a conventional CPU which is memory-bound, it becomes crucial to streamline the write accesses in order to optimize cache usage when writing a large dense vector output. The second aspect, i.e., the *reduction* operations, deserves different solutions according to the execution platform. On a conventional CPU, the bottleneck is memory throughput. Thus, most software implementations of SpMV deal with *a priori* reorganization of the input matrices, by row ordering, blocking, and loop optimization, for improving the data locality of operations. The perspective for SpMV is different when dealing with GPU platforms. The challenge shifts to finding a balanced assignment of the numerous threads and warps, and to the use of local memory resources [57]. SpMSpM is far more complex: the intermediate *reductions* introduce new NNZ values at random locations, which trigger costly memory allocations and list management operations [31]. As a consequence, this phase dominates the execution time and becomes the main memory bottleneck on a regular CPU. The issue is similar on GPUs because of the limited size of the scratchpad memory for processing large matrices, which requires complex memory management and load balancing [22, 62, 88].

The main motivation to use custom hardware accelerators is to make better use of the intermediate memory hierarchy, ensuring the input data is available near the cores and also off-loading the accumulation and *reduction* operations from the processors. The focus of this article is to review these architectures, to compare them, and to analyze their performance. Our main focus is



(a) **Dense**: U^2 (b) **COO**: CP^2 *(row-maj. sorted)* (c) **CSR**: *(UOP, CP) row-maj.*

Fig. 1. Example of a sparse matrix in *COO* and *CSR* formats.

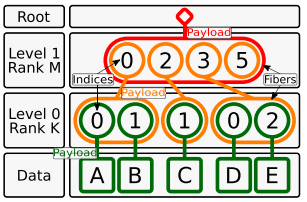


Fig. 2. Example of the *FiberTree* representation with *CSR* format.

on accelerators that eventually target custom or ASIC implementations, although we do touch on the contributions from accelerators that were prototyped on FPGAs. The reader is referred to [58] for a survey of acceleration techniques for CPUs, GPUs, and FPGAs. Section 2 reviews the storage schemes and the three basic algorithms with a focus on the data access patterns and their impact on the memory hierarchy. In Section 3, we study how the different algorithms stress the memory systems and propose an analytical model based on these insights. In Section 4, we present the operation of several state-of-the-art hardware accelerators, in the light of these hardware challenges. In Section 5, we systematically compare these accelerators. Throughout this study, we try to be consistent, and we hope that our nomenclature will be useful for architects and designers who intend to implement dedicated hardware for problems with sparse datasets.

2 SPARSE MATRIX FORMATS AND ALGORITHMS

2.1 Sparse Formats

In the absence of a compressed format, matrices are stored in fixed size two-dimensional arrays. With this *dense* format, elements can be randomly accessed ($O(1)$), and sequential accesses are highly efficient. The two dimensions are called *ranks* which correspond to the rows and columns. For matrices, there are thus two variants of the dense format, depending on which *rank* is stored sequentially in memory. In a *row-wise* (or *row-major*) storage, the elements within the rows are sequential, and conversely for *column-wise* (or *column-major*).

Sparse matrices contain a very large number of zeros, and to avoid storing these repeated zeros, there exist many compressed formats [1, 12, 18, 47, 69, 82], where certain formats are best-suited for matrices with a specific structure. The main principle of these *sparse formats* is to avoid storing the zero elements, compensating with additional *metadata* which gives the position of the non-zero elements. We can cite some of the most used sparse formats like *COO* (*COOrdinate*) that uses three tables of size NNZ with only the non-zero elements (*Payload* table in Figure 1(b)) and their row and column indices (*Row* and *Col IDX* tables), or *CSR* (*Compressed Sparse Row*) (and its *column-major* counterpart *CSC*) that stores pointers to row (or column) positions in the third table instead of indices (*Row PTR* table in Figure 1(c)).

A visual representation of compressed matrices, called *FiberTree* [81], is helpful to understand these *sparse formats*. Each *rank* of a matrix represents a level of a tree where the nodes are filled with either the *metadata* of the child nodes or the matrix data which can only appear in the leaves (see Figure 2). With this representation, we clearly identify the number of *ranks*, which define the *fibers*. A *fiber* is the generic term to describe a one-dimensional stream of data or *metadata*, which corresponds to a row or column in the case of a two-dimensional matrix. A stream of indices is a *fiber* of *metadata*. Figure 2 presents, as an example, a *FiberTree* representation of the matrix of Figure 1(a), matching the *CSR* format: each *rank* in the tree corresponds to a *fiber* stored in the table *Row PTR* or *Col IDX* of Figure 1(c).

In a recent work [85], a systematic nomenclature for *sparse matrix (and tensor) formats* was proposed, based on *FiberTree* representation. In fact, in the example of Figure 2, the *fibers* are compressed with different methods. Thus each *rank* can be characterized by a label indicating how it is compressed: ***Uncompressed*** (***U***), ***Coordinate Payload*** (***CP***), ***Uncompressed Offset Pairs*** (***UOP***). *U* defines *ranks* where all data values are stored contiguously in memory. *CP* defines *ranks* that store the indices of each payload, a payload being either a non-zero element or a sub-matrix.[1] *UOP* defines *ranks* that store pointers (offsets) to delimit sub-matrices of the next *rank*.[2] These are the three main ones as they are used in the most common *sparse formats* but many others exist or can be created.

With this nomenclature, the *sparse formats* can be classified according to their *rank* compression methods. For example, the *CSR* format is denoted as *(UOP, CP) row-major* since its *Row PTR* table stores pointers (offsets) to rows. The *CSR* and *CSC* formats are both of type *(UOP, CP)*, one being *row-major*, the other *column-major*. The *COO* format is denoted as CP^2 because there are two coordinates (row and column) for each non-zero element, as it is a *CP fiber* having two indices per data element instead of one. User defined formats can also be easily described and named. For example, a user could define a storage format with a nested *CSR*. That is, each non-zero element in the outer *CSR*, corresponds to a sub-matrix, itself stored in *CSR* format. Such a representation would be labeled *(UOP, CP, UOP, CP)*. A similar nomenclature is proposed by TACO [18]. It is more adapted to code generation applications, where the *FiberTree*-based nomenclature better describes the memory storage of the *sparse format*. For example, *U* corresponds to *Dense*, *CP* to *Singleton*, the pair *(UOP, CP)* is translated as *Compressed* while *UOP* alone corresponds to *Offset*. We chose to adopt the *FiberTree*-based nomenclature in this article since it is more suited to a hardware context.

2.2 Algorithms

The accelerators described in this article address sparse MV and MM. For MV we use the notation $A(M,K) \times b(K) = c(M)$ and for MM the notation $A(M,K) \times B(K,N) = C(M,N)$. The *rank K* is the *common rank* between the two inputs.

The existence of different *ranks* allows several algorithmic possibilities to compute matrix multiplication: the *inner-product* algorithm (`IP`), the *outer-product* algorithm (`OP`), and *Gustavson*'s algorithm (`Gust`) [28, 37]. In all cases, the matrix *A* is traversed sequentially. The three algorithms; however, impose different access patterns for *B* (or *b*) and *C* (or *c*). Indeed, in many cases, they have to be accessed multiple times, as discussed in the following sections.

2.2.1 Inner-Product Algorithm. The *inner-product* algorithm [28] is an intuitive way to calculate a matrix multiplication because it follows the loop order of the mathematical formulas (1) and (2) for MV and MM, respectively.

$$\forall m \in [0, M[\,, c_m = \sum_{k=0}^{K-1} A_{m,k} \times b_k \quad (1) \qquad \forall (m,n) \in [0, M[\times [0, N[\,, C_{m,n} = \sum_{k=0}^{K-1} A_{m,k} \times B_{k,n} \quad (2)$$

The *common rank* K is the inner-most loop of the algorithm (see Algorithm 1). The output *C* (or *c*) is updated sequentially, but the *B*-matrix (or *b*-vector) will be entirely read M times. In the context of sparse matrices, the reads of *B* (or *b*) are conditioned by the non-zero elements of *A*, which requires indirect accesses. If *B* (or *b*) is also sparse, the algorithm needs to perform *intersections* between the non-zero elements in the rows of *A* and the columns of *B* (or in the *b*-vector)

[1] In this section, we use the term sub-matrix but this is extended to the notion of *tiling* in Section 2.3.
[2] If some sub-matrices are empty, pointers will be repeated, hence "uncompressed".

(see Section 3). In Algorithm 1, exchanging M and N simply causes the output to be updated *column-wise*.

2.2.2 Outer-Product Algorithm. The *outer-product* algorithm [28] has the opposite *rank* loop order than the *inner-product*. It changes nothing mathematically except that the computation is separated into two phases. First, during the *multiply* phase, we compute several ***partial outputs*** (***POs***) consisting of the *partial sums* $A_{m,k} \times b_k$, then during the *merge* phase, we accumulate all the *POs* to form the final C-matrix (or c-vector). Algorithm 2 show that the *common rank* K is the outer-most loop. For MM, exchanging M and N affects whether the output C is updated by rows or by columns, but has no influence on the indirect accesses to the inputs. With *outer-product*, the B-matrix (or b-vector) is read sequentially. This algorithm necessitates the generation of K *POs*. Moreover, in the context of sparse input matrices, the *PO* element updates are conditioned by the non-zeros elements of both inputs. Thus, a *partial sum* could potentially be zero, reducing the number of *PO* element updates, and making them non-sequential. If the output C (or c) is stored in a *sparse format*, then it is necessary to repeatedly access and update each *partial sum*.

ALGORITHM 1:
MM *Inner-Product* Algorithm

```
for m ∈ [0, M[ do
    for n ∈ [0, N[ do
        for k ∈ [0, K[ do
            C_{m,n} += A_{m,k} × B_{k,n};
        end
    end
end
```

ALGORITHM 2:
MM *Outer-Product* Algorithm

```
for k ∈ [0, K[ do
    for m ∈ [0, M[ do
        for n ∈ [0, N[ do
            C_{m,n} += A_{m,k} × B_{k,n};
        end
    end
end
```

ALGORITHM 3:
MM *Gustavson's* Algorithm

```
for m ∈ [0, M[ do
    for k ∈ [0, K[ do
        for n ∈ [0, N[ do
            C_{m,n} += A_{m,k} × B_{k,n};
        end
    end
end
```

2.2.3 Gustavson's Algorithm. As the MM algorithm has three *ranks* instead of two for MV, there is a third ordering, known as *Gustavson's* algorithm [37] where the *common rank* K is the middle loop (see Algorithm 3). With this approach, the output is computed row-by-row with *PO*-rows generated and accumulated for each output C-row.

With this algorithm, updates of the C-rows are sequential, but, in the dense case, the B-matrix is entirely read M times and K *PO*-rows of size N are generated. In the context of sparse matrices, the selection of the rows in B is indirect; however, within a row the accesses are sequential. The *intersection* operations are only needed for the B-rows and not each element. The *PO*-rows can be accumulated before computing the next C-row. For this algorithm, exchanging M and N causes the three matrices to be accessed *column-wise* instead of *row-wise*.

2.3 Tiling

The *tiling* process [35] consists in matrix partitioning: a matrix may be divided into *tiles*, non-overlapping sub-matrices. In other words, this method consists of adding *ranks* to delimit the *tiles*, and thus adding loop iterations in the kernel algorithm.

Tiling methods allow to locally reduce the dimensions of data and thus to adapt the working set to the size of the local memory [51], particularly when the matrix has a structure with denser regions. Tiling can also increase parallelism opportunities. Software optimizations for sparse matrix computation, such as OSKI [83], are based on *tiling*. In previous sections, the study of the `IP`, `OP`, and `Gust` algorithms highlighted the influence of *rank* ordering on the sequentiality and data

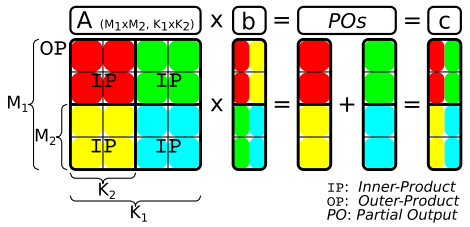


Fig. 3. An example of *tiling* on SpMV.

movement, and thus, adding intermediate *ranks* with *tiling* breaks the regularity of pure IP or OP algorithms, requiring additional hardware support (see Section 3).

For example, let us consider the *A*-matrix in Figure 3 which is divided into four tiles (one level of *tiling*). For a SpMV, two new *ranks* appear for a total of four *ranks*: M_1 and K_1 allow *tile* selection and M_2 and K_2 allow data access within a *tile*. Thus, we could decide to apply an OP on *tiles* and conversely an IP within *tiles*, mixing-up the pros and cons of these two algorithms at different *rank* levels. In the case of the example in Figure 3, the upper-left *A-tile* (red) is evaluated first, sequentially updating the upper tile of the *c*-vector in an IP manner. The lower-left *A-tile* (yellow) is then computed the same way. At this point, *intersections* have been done with only the upper *b-tile*, and *c* has been updated sequentially. The two next *A-tiles* (green, then blue) will compute *intersections* with the lower *b-tile* and update the entire *c*-vector sequentially, showing the OP characteristic of having *POs* to *merge*, two in this case.

Note that in case of *tiling*, the *A*-matrix is no longer read sequentially, which can be problematic if it is stored in a classical *sparse format* like *CSR* that is specifically made for *row-wise* accesses. Custom *sparse formats* adapted for *tiling* can be used [42], but this requires a preprocessing conversion.

3 HARDWARE CHALLENGES FOR SPARSE MATRICES AND VECTORS

In the previous section, we presented a brief background on matrix multiplication without considering the underlying hardware. Although the various *sparse formats* reduce the memory footprint, they have the disadvantage that indirect accesses are required, which creates data-dependencies and reduces the effectiveness of caches. As we have seen, with all three algorithms, at least some of the accesses are non-sequential. The lack of spatial locality for these accesses reduces the benefit of having caches. In other words, a full cache line is loaded, but only a few entries are read or modified. Moreover, the matrices of interest are extremely large and even the non-zero elements, and associated *metadata*, can not generally fit in the cache. Depending on the algorithm, successive accesses to the same element may be too far apart to benefit from temporal locality in the cache.

In an IP algorithm, the *A-metadata* needs to be read to indirectly access the *B*-matrix (or *b*-vector), which may itself be stored in a *sparse format*. In this case, the algorithm needs to check if the non-zero *A*-indices are present in *B* (or *b*). If an index appears in neither list, there is no need to perform the multiplication (*ineffectual operation*). More generally, this problem applies to *fibers* and is called the *intersection* search. As a minimum, this requires reading all the *metadata* for *A* and *B* (or *b*), and identifying those present in both.

In an OP algorithm, the inputs are accessed sequentially, but the updates of the output require indirect accesses. The pattern of the accesses, while the output is generated, is non-sequential and determined by the structure of the input. Furthermore, if these *random*[3] updates provoke cache

[3]We use the term *random* to mean accesses are irregular and difficult to predict.

Table 1. Comparison of Advantages and Challenges with Matrix Multiplication Algorithms

		Inner-product (IP)	Outer-product (OP)	Gustavson (Gust)
Access Count	**MV**	***A***: once ; ***b***: $NNZ(A)$; ***c***: once	***A***: once ; ***b***: once ; ***c***: $NNZ_r(A)$	*N/A*
	MM	***A***: N ; ***B***: $NNZ_r(A).M$; ***C***: once	***A***: once ; ***B***: $NNZ_c(A)$ ***C***: $\sim NNZ_r(A).NNZ_c(B)/K$	***A***: once ; ***B***: $NNZ_c(A)$ ***C***: $\sim NNZ_r(A).NNZ_c(B)/K$
Sequential Accesses		Accesses to A and C	Accesses to A and B	Accesses to A and C Accesses to B-rows
Input Format		Accesses to A and B benefit from alternate formats (*CSR* and *CSC*)		Allows consistent format for A and B
Parellism Opportunity		Possible with *tiling*	Generation of *POs* over *common rank* of A and B	Reading A-rows and updating C-rows
Reuse Opportunity		Potentially B, but limited by its size	An A-column or B-row while computing *POs*	A set of B-rows, based on non-zero elements of A-rows
Challenges		*Intersection* search between non-zero elements in A and B	Storage for *POs* or in-place *merge* and *reduce*	*Merge* and *reduce* of *PO rows*

misses where only one word within a cache-line is modified, the bandwidth to external memory is poorly utilized. The OP algorithm generates several *POs* that must be accumulated to obtain the final output. We can identify two strategies: *immediate update* and *deferred update*. In the former, the final output is immediately updated as each *partial sum* is generated. If the output is stored using a *sparse format*, either the index being updated does not yet exist, and the new *partial sum* must be inserted. Or, if it exists, a lookup must be done to locate it, the update performed and the result written back. The problem associated with combining the *POs* is called *reduction*.

To avoid the complexities of the indirect accesses with *immediate updates*, the *partial sums* can be stored and the accumulation can be deferred. One such technique is called *output batching* [48]. With this technique, tuples containing the index and the *partial sum* are streamed sequentially to memory. Then, during a second pass, they are read back and the accumulations are performed, an approach which results in sequential memory accesses during the first pass, with the downside that the volume of data transferred to memory is increased. In Table 1, we summarize the previously discussed advantages and hardware challenges for each algorithm, including Gust which presents a tradeoff between IP and OP.

The issues presented with IP and OP can be generalized to two concepts: *gather* and *scatter*. The *gather* problem exists when the inputs are indirectly accessed, and need to be read from memory using *metadata* to present the processor with the correct terms for computing the *partial sums*. Typically, with IP, the *gather* problem is more difficult. The *scatter* problem exists when the order in which the output matrix/vector is generated is not sequential and is typically associated with the OP algorithm. In the case of *tiling*, a hardware accelerator must handle both *gather* and *scatter*, creating a need for hardware support for both *intersection* searches and *reductions*.

3.1 SpMSpM Algorithm Analysis

To better understand how the algorithms impact the number and types of memory accesses, we developed, and presented in Figure 4, a model based on memory accesses counts. We identify five memory access patterns: (1) those where a matrix is read only once, thus there is no opportunity for reuse, (2) where the accesses are sequential, but a line is read multiple times, (3) where lines are read consecutively, but the accesses within a line are *random*, (4) where the lines are accessed *randomly*, but within a line the accesses are sequential, and (5) where the matrix elements are accessed multiple times, and with a *random* access pattern.

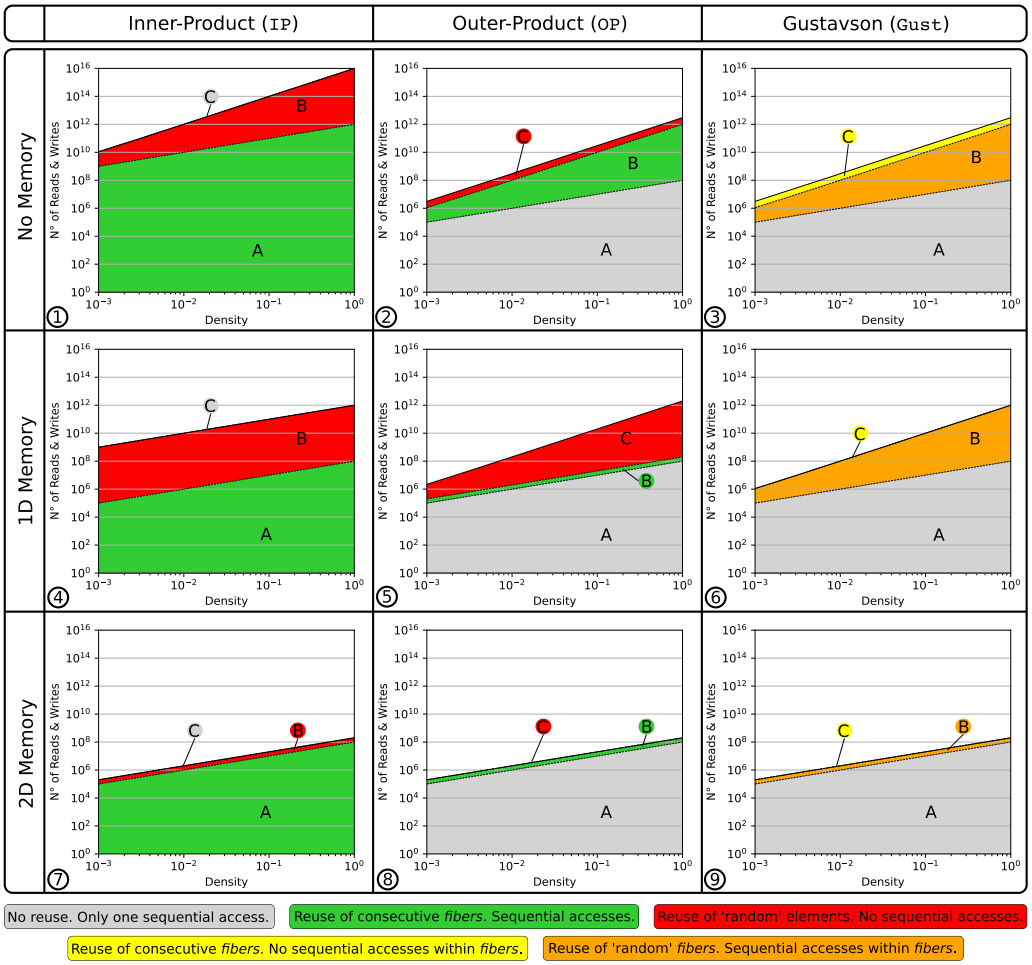


Fig. 4. Model of memory access counts depending on algorithm and local memory size.

For each of the three algorithms, we counted the number of read and write accesses, based on the density of the input matrices. We plotted the number of memory accesses in logarithmic scale for SpMSpM, based on a square matrix with $M = 10{,}000$. Indeed, the number of non-zero elements in C depends on the structure of the input matrices and in this model, we have assumed the non-zero entries in the input matrices are uniformly, randomly distributed. This corresponds to a *worst case* for sparse matrices (no spatial and temporal locality). Banded input matrices would, for example, have fewer non-zero elements in C. In each graph, we have separated the accesses into A (bottom), B (middle), and C (top). The color code shows the access pattern, based on the five categories described above.

In the top-row of Figure 4 (graphs ①, ②, and ③), we start by considering a naïve accelerator which has no local storage, thus all data comes from main memory. From these figures, we clearly see that IP requires more accesses, as the A-matrix must be read N times (graph ①). Whereas for OP and Gust, the A-matrix is read only once (thus the grey color). For OP and Gust the total number of accesses is the same; however, with OP (graph ②), the accesses to C are *random* (red). With Gust (graph ③), the accesses to B are sequential within the rows (orange) and the C-rows are generated sequentially (yellow), thus Gust avoids having any fully *random* accesses (red).

Since main memory accesses are the principal bottleneck, accelerators integrate local storage (such as buffers or caches). Thus, we consider the case of an accelerator with sufficient local memory to store one full row, per matrix. We assume this local storage is perfect (no misses) and recalculate the number of remaining main memory accesses, as shown in the middle-row of Figure 4 (graphs ④, ⑤, and ⑥). The matrices are assumed to be initially stored in main memory, thus they must be accessed at least once. As can be seen, for IP (graph ④), the number of A-accesses is reduced, because rows are reused, thus the total number of A-accesses becomes equal to that in OP and Gust. For OP (graph ⑤), the single row buffer greatly reduces the number of B-accesses, however, it does nothing for C (despite the appearance, due to the logarithmic scale). For Gust (graph ⑥), the single row buffer is sufficient to cache the *reduction* operations in C, and thus, it now has fewer overall accesses than OP. It is also interesting to note that OP and Gust scale better than IP with lower densities, as seen by the shallower slope.

Finally, we have considered the extreme case where an accelerator has a buffer of sufficient size to store two matrices, which is shown in the bottom-row of Figure 4 (graphs ⑦, ⑧, and ⑨). Although this case is not practical for large matrices, through the proper use of *tiling* to locally reduce the dimensions of a sub-matrix, this case can be achieved at the level of a *tile*. In this case, the total number of external accesses is equal for the three algorithms, as each matrix is accessed only once from main memory. For IP (graph ⑦), it is interesting to note that such a large buffer is needed to locally address the *intersection* search in B. Similarly for OP (graph ⑧), a large storage is required to address the *reductions* in C. For Gust, in fact, it is not necessary to buffer the entire B-matrix. Indeed, buffering a few rows (corresponding to $NNZ_r(A)$) is sufficient to achieve the number of accesses shown in graph ⑨. This corresponds to an intermediate design point (multiple row buffers for B) not explicitly shown in the figure (between graphs ⑥ and ⑨).

When viewed overall, Figure 4 highlights the benefits of Gust. We clearly see that Gust does not require any fully *random* accesses (red). With a single row buffer, it is possible to address the *reductions* in C. And, with a limited number of row buffers, the number of external memory accesses for B can be reduced. Based on this analysis, it is clear that Gust is the best algorithm for hardware implementation of SpMSpM.

4 HARDWARE ACCELERATORS FOR SPARSE MATRICES AND VECTORS

In this section, we describe the recent (2018–2023) hardware accelerators for sparse MM and MV, targeting ASIC-type implementation. We try to be consistent on notations and architecture schemes to simplify the comparison. Before going into the detailed overviews of the accelerators, in Table 2 we present a criteria-based classification of the designs.

A first criterion to classify the accelerators is between *standalone* accelerators that address the full kernel without the assist of a processor (e.g., OuterSPACE [64]—Section 4.1) and ones that address only a part of the kernel and need a processor to manage the computation (e.g., PHI [60]—Section 4.9). *Standalone* accelerators are directly connected to the main memory and internally manage their own custom memories. The other accelerators actually intervene at different levels of the memory hierarchy, giving a second classification criterion. Certain accelerators intervene near the processors (e.g., ASA [89]—Section 4.8), near the cache (e.g., PHI and SortCache [77]—Sections 4.9 and 4.7), or near the main memory (e.g., SPiDRE [10] and SpaceA [86]—Section 4.9).

Each accelerator is optimized for a specific algorithm (IP, OP, or Gust), our third criterion. For example, some accelerators (e.g., PHI and SortCache) address the *reduction* problem and are adapted to OP algorithms but *reductions* are also required with Gust and highly-*tiled* algorithms. ExTensor [40] (Section 4.2) is a general purpose accelerator that is demonstrated for a highly-*tiled* OP but also makes major contributions for IP.

Table 2. Algorithmic and Architectural Classification of Existing Accelerators

Name	Date	Autonomy		Position in Memory Hierarchy			Most Adapted Algorithm			Hardware Support for		Sparse Kernels Addressed
		Standalone	*Processor Assist*	*Near Processors*	*Near Cache*	*Near Main Memory*	*Inner-product*	*Outer-product*	*Gustavson*	*Gather*	*Scatter*	
OuterSPACE [64]	2018	■		*N/A*				■		■	■	MV, MM
ExTensor [40]	2019	■		*N/A*			■	[a]		■	[a]	MV[b], MM
PHI [60]	2019		■		■			■	[b]		■	MV, MM[b]
SPiDRE [10]	2019		■			■	■			■		MV, MM[b]
MatRaptor [79]	2020		■	*N/A*					■	■	■	MM
SpArch [93]	2020		■	*N/A*				■		■	■	MM
SpaceA [86]	2021	■				■	■			■	■	MV
Gamma [90]	2021	■		*N/A*					■	■	[a]	MM
InnerSP [7]	2021	■		*N/A*					■	■	■	MM
SortCache [77]	2021		■		■			■	[b]		■	MV[b], MM
ASA [89]	2022		■	■				[b]	■		■	MV[b], MM
Flexagon [59]	2023	■		*N/A*					■	■	■	MM
GSCSp [55]	2023	■		*N/A*					■	■	■	MM

Gray cell: the accelerator meets the criterion.
[a]Not main contribution but addressed in the article.
[b]Could be adapted but not discussed in the article.

From a memory perspective, the two main problems are *gather* and *scatter*, our fourth criterion. Certain accelerators offer solutions for one, the other, or both. For example, some accelerators (e.g., SpArch [93]—Section 4.3) are specialized for an OP algorithm but optimize both *gather* and *scatter*. Finally, accelerators preferably focus on sparse MM, MV or both—our final criterion.

After this first comparison, in Sections 4.1 to 4.6, we present the *standalone* accelerators, followed by the *processor assist* accelerators in Sections 4.7 to 4.8. In Section 4.9, we briefly touch on other accelerators.

4.1 OuterSPACE

OuterSPACE [64] is an OP accelerator with a focus on SpMSpM kernels, although it can also handle SpMV. It is a highly parallelized architecture which employs **Single-Program Multiple-Data (SPMD)**-style ***processing elements*** (***PEs***) that fetch data from main memory through a two-level highly-banked cache hierarchy as shown in Figure 5. The input *A*- and *B*-matrices are stored in *(UOP, CP) column-* and *row-major* formats, respectively, to match the read access patterns of the OP algorithm. The *partial* and final outputs are stored in *(UOP, CP)* format.

The computation is temporally divided into the *multiply* and *merge* phases (Figure 6). During the first phase, inputs are divided into *tiles* of several rows (*A*) and columns (*B*) and distributed through the ***processing tiles*** (***PTs***) by a *control unit*. Within each *PT*, each *PE* processes multiplications of one element of the k^{th} *A*-column with all elements of the k^{th} *B*-row and an L0 cache in the *PT* facilitates the reuse of the *B*-rows between *PEs*. This phase generates a *PO* per *PT*, corresponding to the *tile* of rows (*A*) and columns (*B*) that it was assigned.

Then, during the *merge* phase, the *POs* are combined to generate the final *C*-matrix. Each *PT* locally sorts the *POs*, and then the *PTs* work together to merge them. This algorithm was chosen to maximize data locality and parallelism, although it is less efficient. This *merge* phase only uses half

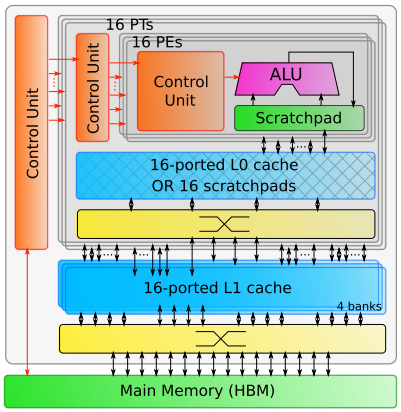


Fig. 5. Architecture of OuterSPACE.

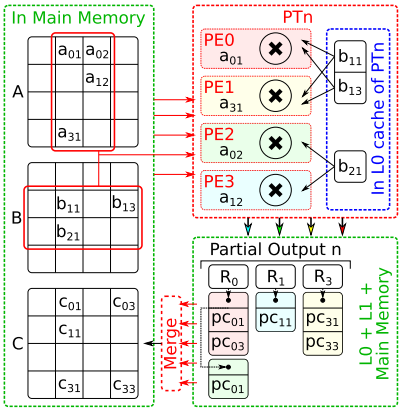


Fig. 6. OuterSPACE data flow.

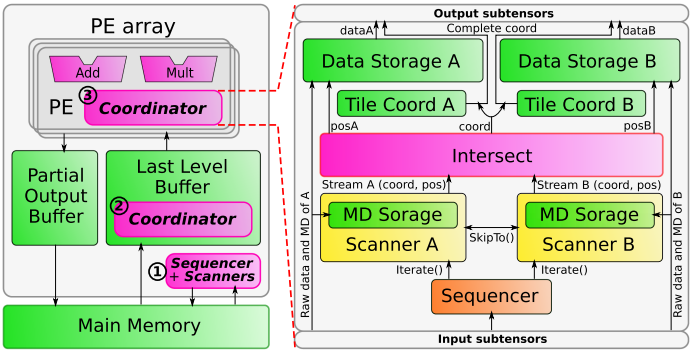


Fig. 7. Architecture of ExTensor.

of the *PEs* so the others are disabled (clock-gated) and the caches are reconfigured in a scratchpad mode to save energy.

Note, if NNZ elements in the *POs* exceed the L0 cache capacity, it overflows to the last level cache, which results in reduced performance. OuterSPACE performs best when the $NNZ_c(A)$ and $NNZ_r(B)$ are both uniform which ensures a uniform distribution of work across the large number of *PTs* and *PEs*.

4.2 ExTensor

The authors created ExTensor [40] as a general purpose sparse tensor algebra accelerator (including SpMSpM). It is an OP accelerator but due to its support for three levels of fixed *tiling* of the inputs, it optimizes the *intersection* search between non-zero elements or *tiles*.[4] By doing the *intersections* ahead of time, at different levels of the memory hierarchy (which correspond to the *tile* levels), ExTensor is able to efficiently eliminate *ineffectual operations*, thus addressing the main issue in IP or complex *tiling* algorithms. Indeed, the input matrices would typically be stored in a *(UOP, UOP, UOP, CP)* format, corresponding to three levels of *tiling*. ExTensor is designed with a flexible architecture composed of different bricks (see Figure 7). To simply aid in the understanding, the interactions between the bricks are described using function calls:

[4]Tensauraus [80] is another accelerator designed for tensors, but their focus is on SpMM.

— The *Scanner* is a storage unit that delivers a stream of coordinates in increasing order, fetching them from memory. The coordinates are delivered through the *Iterate()* operation call.
— The *Intersect* is a hardware block that co-iterates over several *Scanners* in parallel and compares the coordinate streams. This unit performs the *intersection* search required by IP type algorithms, delivering a single stream with the matching coordinates, which are the coordinates of non-zero *partial sums*. To optimize the *intersection* step, it is possible to skip a range of coordinates in a stream depending on the current coordinate of another stream. Thus, the *Intersect* sends skip messages with *SkipTo()* operations to the other *Scanners*.
— The *Coordinator* is a hardware unit that includes *Scanners* and *Intersect* units, and manages the input and output data depending on the current processed *tile*, as shown in Figure 7. Data and *metadata* are, respectively, stored in storage units and *Scanners*, while a *Sequencer* manages the *Iterate()* calls so that the *Intersect* brick delivers the stream of effective coordinates. In addition, in the *Tile Coord* brick, the offsets of the *tile* are added back, so the output of the *Coordinator* is in absolute coordinates.

These bricks can be placed at different positions in a memory hierarchy, depending on the *tiling* and selected hardware. Thus, the *Coordinator* is able to intersect at coarse-granularity with output data being *metadata* of the next *tile*, skipping an entire *tile* when appropriate, with low computation cost, and also at fine-granularity when output data are the non-zero elements that need to be multiplied.

ExTensor is designed with three levels of *tiling* that each have either input or output sequentiality. First, a *Sequencer* plus *Scanners* block ① is placed close to the main memory to determine which *tiles* to send to the next storage, the ***Last Level Buffer*** (***LLB***). Then, for each *tile*, a *Coordinator* ② in the *LLB* intersects the next level *tiles*, which are sent to several *PEs* that will process in parallel. Finally, in each *PE*, a *Coordinator* ③ intersects the non-zero elements of the computed *tile* and sends them to the arithmetic unit of the *PE* for multiplication.

In addition to these bricks which are the main contributions, ExTensor still needs to perform the *merge* phase, including the *reduction* of the *POs* which are stored in CP^2 format. The ***Partial Output Buffer*** (***POB***), is managed by *tiles*, and the content of the *tile* is stored on-chip using a **Content Addressable Memory** (**CAM**) for quick access, enabling *reductions* to be performed. On overflow, the *tiles* are flushed to main memory. To benefit from ExTensor's support for *tiling*, the input matrices need preprocessing, making it difficult to directly compare performance with other accelerators.

4.3 SpArch

SpArch [93] uses an OP algorithm for SpGEMM in order to benefit from the simplified access pattern for the inputs. To address the management of the large volume of *POs*, SpArch pipelines the *multiply* and *merge* phases and uses condensing methods to reduce the number of *POs* and scheduling methods to reduce memory traffic. Unlike other OP accelerators [64], with SpArch, both *A* and *B* are stored in *(UOP, CP) row-major* format. As a consequence of this *CSR* format, the *A*-rows can easily be condensed to the left (see Figure 8). Thus, when a condensed *A*-column is read, it will require reading all the *B*-rows corresponding to the different *A*-columns that have been combined. The *MatA Column Fetcher* reads the combined *A*-columns and the *MatB Row Prefetcher* reads the necessary *B*-rows (see Figure 9), thus masking the memory latency. The *MatB Row Prefetcher*, stores multiple *B*-rows in a cache, and it uses information about the upcoming *A*-columns (provided by the *Distance List Builder*) to ensure the needed *B*-rows are kept in the cache.

A key contribution of SpArch is the merge unit that is able to merge 64 *POs* in parallel, through a binary tree type architecture. The core of the merger is a brick which compares and merges

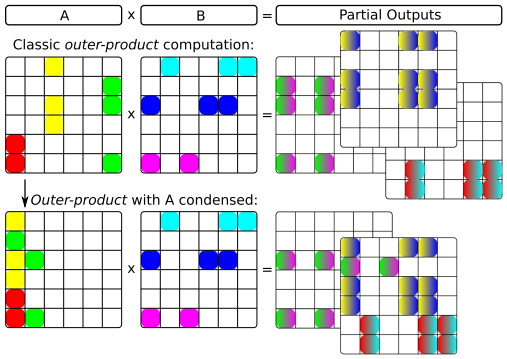


Fig. 8. *Outer-product* with a condense *A*-Matrix.

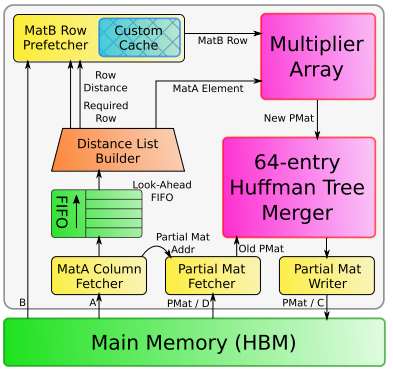


Fig. 9. Architecture of SpArch.

4×4 $(index, value)$ pairs in one cycle. These are combined to build a hierarchical merger array whose output is then written to main memory, and read back for further merging by the *Partial Matrix Fetcher*. The authors note that the order in which the *reductions* are performed is important, specifically sparser *POs* should be merged first and they propose a Huffman tree scheduler to determine the best order to minimize memory accesses. Overall, SpArch is able to compute an OP SpGEMM[5] kernel with reduced main memory traffic for the *POs* mainly by addressing the *scatter* problem with a highly parallelized merger.

4.4 MatRaptor

MatRaptor [79] is an accelerator designed to compute SpMSpM kernels. It is based on Gust algorithm,[6] which makes it possible to process multiple *A*-rows (2 to 8 for MatRaptor) in parallel. In MatRaptor (see Figure 10) a *SpAL* (*Sparse Matrix A Loader*) fetches non-zero elements of a *A*-row and sends them to a *SpBL* (*Sparse Matrix B Loader*). With the *metadata* of this element, the *SpBL* fetches non-zero elements of the corresponding *B*-rows and sends pairs of *A*- and *B*-elements to a *PE*. This *PE* computes the *partial sums* and sends the results into queues for the *merge* process. Instead of separating *multiply* and *merge* phases, several queues are used to store *partial sums* and partially merged *partial sums*, allowing the two phases to be pipelined. The *PE* merges all the *partial sums* resulting from an entire *A*-row, before sending the final output (the corresponding *C*-row) to main memory.

In MatRaptor, multiple *A*-rows are processed independently and in parallel in different *PEs* and each *PE* is assigned to a channel in an HBM memory [45]. It is important that each *PE* can read its *A*-rows without interfering with the other *PEs*. With the classic *(UOP, CP) row-major* format, the rows are consecutive in memory and are not aligned to channel boundaries. To solve this problem, the authors propose a new format called C^2SR (*(FL, UOP, CP) channel-major*) that is based on a *(UOP, CP) row-major*. For example, with two *PEs*, all the elements of the even rows are stored in one channel, and the elements of the odd rows are in another channel, thus isolating the data. With this approach, the row pointers (*UOP*) are no longer monotonic, which means that the length of a row can not be deduced by subtracting adjacent row pointers. This requires the addition of a new table (which we call *fiber length FL*), which stores the length of the non-zero elements in each row (see Figure 11). This modified storage format enables multiple *PEs* to make effective use of the HBM.

[5]Since the merger can read *POs* from memory, the *D*-matrix can be treated like an initial *PO*.
[6]Called *row-wise product* in [79].

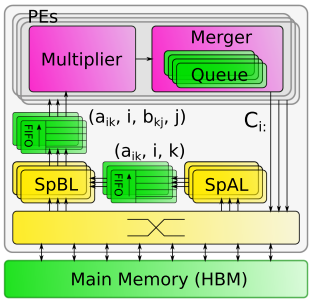


Fig. 10. Architecture of MatRaptor.

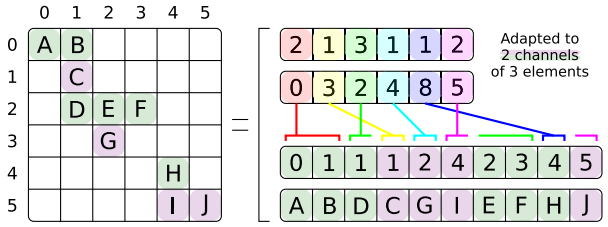


Fig. 11. *(FL, UOP, CP) channel-major* format (C^2SR).

MatRaptor is one of the first accelerators to use Gust algorithm; however, it has some limitations regarding effective reuse of B-rows, both between parallel *SpBLs* and temporally, as A is traversed. A key contribution is the proposed C^2SR input format, which makes it possible to separate rows into memory channels, to improve performance.

4.5 InnerSP

InnerSP [7] is a SpMSpM accelerator based on the Gust algorithm.[7] The authors of InnerSP justify their choice by showing quantitatively that with the OP the *POs* represent a large volume of data, and that for many matrices the *reductions* can not be done fully on-chip and thus must be buffered in main memory. Normally, with Gust algorithm, B must be read repeatedly, but InnerSP proposes a special cache for B which exploits the spatial locality.

The architecture of InnerSP is shown in Figure 12. The *A reader* reads the A-rows. Two separate caches are used for B, one that stores the row pointers and the other which stores column indices and values. The *B reader* reads the required B-rows, hopefully getting the data from the caches. Parallel multipliers perform the multiplications and then a hash-unit generates a hash based on the indices in C. A *reduction* unit based on hash tables contains parallel comparators which search for matching hash values and performs the *reductions*. When all the operations for a A-row are completed, the *C writer* writes the C-row back to main memory.

The architecture is efficient, as long as the *Hash Reduction Unit* does not overflow. When overflows occur, the extra entries are temporarily stored in main memory, but there is a significant performance hit. To minimize the chance of overflows, InnerSP uses a pre-scanner to estimate the number of non-zero entries in the upcoming C-rows. The estimate primarily considers the NNZ in the A-rows. If the estimated number of non-zero entries in C exceeds the capacity of the *Hash Reduction Unit*, then the row is split and processed in two passes. Conversely, if the number of non-zero entries in C is small, two A-rows can be processed simultaneously.

To improve the performance of the B-cache, the authors exploit a technique from [8]. The non-zero elements in the A-rows provide direct information about which B-rows are required. The column index table of A is already pre-fetched by the *A reader*, and when a row must be evicted from the B-cache, it chooses the ones which will not be required based on the upcoming entries in the column index table of A. This technique (also used by SpArch [93]) maximizes the reuse of B-rows.

InnerSP is an accelerator based on Gust algorithm which achieves high reuse of the B-rows, through a look-ahead in the *A-metadata*.

[7] The authors call this *row-wise inner-product*, or confusingly sometimes *inner-product*.

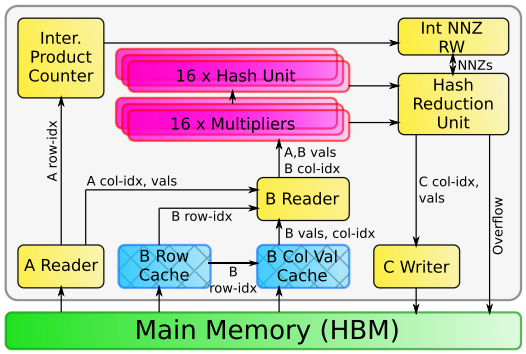


Fig. 12. Architecture of InnerSP.

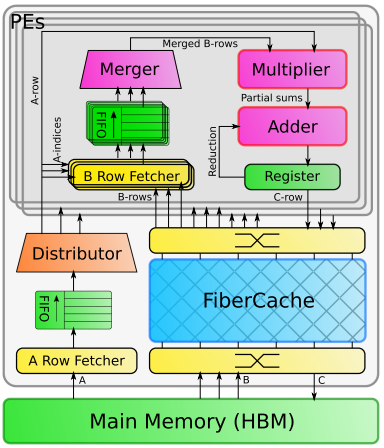


Fig. 13. Architecture of Gamma.

4.6 Gamma

Gamma [90] is a dedicated accelerator for SpMSpM kernels, which uses `Gust` algorithm. The two input matrices are both stored in *(UOP, CP) row-major* format and the output is generated in the same format, providing consistency.

The Gamma architecture, presented in Figure 13, is composed of a fetcher for the A-rows that contains a *Distributor* to distribute them to 32 *PEs*. Each *PE* processes an entire A-row and computes the corresponding C-row. The *PE* fetches the required B-rows and merges and sorts them in a 64-wide *merge* block, before multiplying with the A-elements. As a result, the output C-row is generated in order. The *reduction* step is, thus, highly simplified, because any duplicate indices are adjacent. Normally, the C-row can be written back to main memory. However, if the A-row contains more than 64 non-zero elements, then it must be processed with multiple passes. After each pass, the partial C-row is stored temporarily. Finally, the temporary C-rows are read-back and merged by the *PEs*. Of course, the B-rows must be accessed repeatedly by the different *PEs*, and Gamma proposes *FiberCache* which is a highly banked cache which fetches the B-rows in advance, based on the column index table of A. It keeps track of which rows of B are most needed by the *PEs*, and when an eviction is necessary, the victim, is the row which is least needed.

Gamma encounters some difficulties when the A-matrix has rows that are too dense, requiring multiple iterations through the *PEs*, which creates a high-load on the cache. The authors propose a software preprocessing technique which splits dense A-rows and rearranges them to make best use of the *FiberCache*.

4.7 SortCache

The authors of SortCache [77] note that modern processors are able to load an entire cache line in parallel but in many sparse applications less than 50% of the data within a line is used, even with hand-tuned code. They propose a light SpGEMM accelerator which offloads the *reduction* step, based on the ***Vectorized Binary Search Tree*** **(VBST)**, a data-structure from their earlier work [76, 78]. The VBST is a standard binary search tree where the size of node is a multiple of the cache block size and contains a sorted vector of K $(key, value)$ pairs. Associated with each node is a *pivot*; the pivots, with left and right child pointers, form a binary tree, as illustraded in Figure 14.

Using the VBST, SortCache focuses on accelerating the *scatter* updates to the output matrix, which are performed directly in the cache. The authors augment the processor with a single new

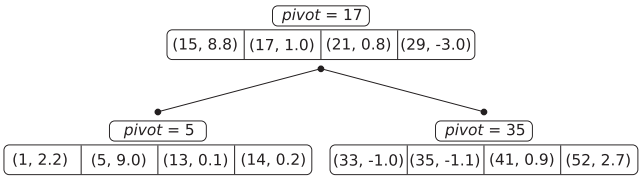


Fig. 14. VBST data structure used by SortCache.

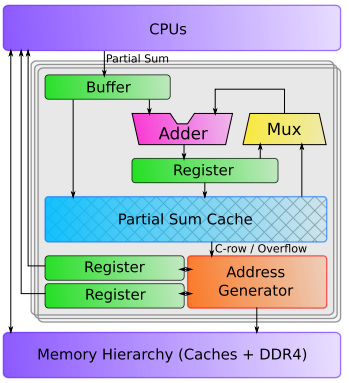


Fig. 15. Architecture of ASA.

instruction, *RED Key, Value*. After K updates have been issued, the hardware launches a *reduction* in the SortCache, where the K $(key, value)$ pairs are inserted into the VBST. This is done by traversing the tree from the root to the leaves. When duplicate keys are found, the *reduction* operation is performed in the cache. When new keys are found, they are inserted, which can result in a temporary, new node with up to $2K$ $(key, value)$ pairs. This temporary node is partitioned, with one half having exactly K entries. The insertion procedure may require visiting each level of the tree at least once ($O(log(n))$ and it does not ensure the uniqueness of the keys, so at the end, an additional full traversal of the tree is required to remove duplicates.

The nodes closest to the root of the tree are accessed most frequently and are thus mapped to the caches closest to the processor. The authors propose to re-purpose cache tracking hardware (e.g., TAG values, LRU counters) to store the additional metadata (pivots, pointers). One of the difficulties with SortCache is that, in some cases, a large fraction of the VBST nodes are underutilized.

4.8 ASA

In the scope of graph analytic problems, the authors of ASA [89] observed that `Gust` algorithm achieves high performance for SpGEMM kernels but note that it still requires acceleration for the *reduction* step.[8] The software state-of-the-art for *reduction* uses hash tables [15]. Some hardware accelerators, such as HTA [92], enhance performance for hash operations, but those solutions are not optimized for SpGEMM kernels. ASA is a low area, near-core extension for a general purpose processor that specifically accelerates the *reduction* step for `Gust` algorithm.

ASA accelerates *reduction* operations received from software via custom instructions. As presented in Figure 15, the *partial sums* from these instructions are stored in a waiting buffer as $(index, value)$ pairs with a key that is a hash of the index, computed in software to maintain flexibility. The cache line, accessed with the key, is filled with the indices and the *partial sums*. If a corresponding *partial sum* is already stored, ASA accumulates it and writes back the result. If not, the *partial sum* is simply stored into the cache.

To minimize die area, the cache is not dimensioned for an entire C-row[9] but ASA handles cache overflows in hardware. When a *partial sum* needs to be stored in a fully filled cache line, a victim is selected and evicted, based on a LRU policy. The evicted entry is written into a pre-allocated FIFO queue in the memory hierarchy. ASA uses an address generator to manage evicted data and

[8]The authors refer to the *reduction* step as SPA (Sparse Accumulation).
[9]The authors present a *column-wise* `Gust` implementation. This choice is arbitrary and, for consistency, we stick with a *row-wise* presentation.

updates head and tail registers for the list. Software then traverses the list to merge duplicate keys to build the final output C-row. To avoid overflows, the authors advocate *tiling* methods to efficiently distribute work through one or several ASA cores.

ASA is an accelerator which optimizes the *scatter* aspect of SpGEMM kernel with low area overhead, while leaving high software flexibility. Unlike other Gust accelerators, there is no hardware *gather* support for reusing or caching the B-rows.

4.9 Other Accelerators

In the above sections, we have described a selection of the main sparse accelerators that have appeared in the literature, covering different matrix multiplication algorithms and architectures. This list is not exhaustive, and in the following paragraphs, we present several other accelerators which we will cover in less detail, due to space constraints.

4.9.1 SPiDRE. The authors of SPiDRE (Sparse Data Rearrange Engine) [10] propose a solution for the poor utilization of the memory hierarchy in sparse applications. They propose a near-memory data rearrangement engine. This engine can preprocess data, near memory, to create new data-structures that are dense and benefit from the memory hierarchy. For example, with a SpMV kernel, the input vector could be rearranged through a user defined function to a dense format where all non-zero elements are in the right order of computation.

4.9.2 PHI. The authors of PHI [60] note that many large graph algorithms favor pull (each node reads inputs from its neighbors) compared to push implementations (each node propagates results to its neighbors), as push implementations require read-modify-writes which require accessing complete cache lines, even though a small unit of data is modified. The authors include OP SpMV (a push algorithm) in their evaluation.

PHI is a modified cache optimized to efficiently process commutative *scatter* updates. When an update is received but the line is not in the cache, the line is not fetched but rather marked as being in *update mode*, where it simply stores the update. If subsequent updates hit the line, they are stored, or combined via the commutative operator. When a cache line is evicted, if it contains multiple updates, PHI assumes that the coalescing has been effective, and it processes the operations via a read-modify-write from main memory. If a cache contains few updates when evicted, PHI batches these updates as $(address, value)$ tuples, allocating a new cache line to assemble the batch. Groups of updates are organized by *bins* corresponding to a small memory region. The batches are streamed to the main memory. Later, the reading of the batches is also efficient, as they are contiguous.

In multi-core systems, private caches act as local buffers, thus coalescing the updates reduces coherency protocol traffic. This combination of in-cache coalescing and update buffering enables PHI to take advantage of locality when possible (like Coup [91]), but with the addition of *update batching*, PHI improves memory utilization, even when the data being updated is too large to fit in the cache.

4.9.3 GSCSp. GSCSp [55] is an FPGA-based accelerator based on Gust algorithm. In other Gust accelerators, a *PE* processes a full A-row but the authors note that when there is variability in the number of elements in the A-rows, *PEs* may be blocked, waiting for another *PE* to complete. Thus the authors propose to distribute the elements in an A-row to different *PEs* (*element-wise* parallelism). Initially, each *PE* buffers its output C-row, and when a *PE* advances to the next row, it pushes its partial C-row to a final merger. The authors also propose to use separate caches for the *metadata* for the B-rows and the values, and report that combined with the *element-wise* parallelism, the approach is effective for reducing the memory traffic for accessing B-rows.

4.9.4 Flexagon. Flexagon [59] is a SpMSpM accelerator for DNN applications. The authors claim that the optimal algorithm (IP, OP, and Gust) depends on the matrix structure, potentially varying between DNN models and layers within a DNN, although in the majority of the cases, their data shows that Gust is the most efficient. Flexagon is configurable: it can implement all three algorithms. To achieve that, the authors designed a ***Merger-Reduction Network*** (***MRN***) that is able to perform multiplications (for *partial sum* generation), accumulations (for *reductions* operations) and comparisons (for *intersection* searches and *merge* phases) via a binary tree structure. Flexagon addresses the memory access issues of each algorithm via three custom memories: a FIFO for the *A*-matrix that is always read sequentially, a cache for the *B*-matrix that is subject to high temporal and spatial reuse in each of the three algorithms, and a PSRAM to store *POs* for OP and Gust algorithms.

4.9.5 Other Accelerators. In GraphR [75], the authors propose to use ReRAM technology and a cross-bar to perform graph operations directly in hardware, but this approach is not currently scalable to large problems. The ITS accelerator [70] focuses on an OP implementation of SpMV and they apply data-compression to the *metadata*. The authors of SPU [20] propose a more general hardware solution to input data dependencies and focus on graph applications. The focus of Alrescha [3] is an innovative and adaptive preprocessing of the matrix in hardware to simplify the accesses to the *b*-vector. The authors of CoSparse [29] focus on a software layer that selects the hardware acceleration that is best adapted to the given problem.

The SSSR accelerator [73] for RISC-V processors goes further by addressing *intersection* and *union* while streaming data to and from memory and it can be used for multiple kernels; however, the RISC-V performance can not easily be compared with the other accelerators presented in this article. In passing we note three other accelerators EIE [38], SCNN [65], and CANDLES [36] which have hardware support for sparsity but which are highly focused on CNNs, besides SDMA [32] that focuses on GNNs.

The SpaceA [86] accelerator exploits near-memory computing inside a 3D DRAM memory to implement SpMV using an IP algorithm. With preprocessing, the rows of the matrix are assigned to banks within specific DRAM dies, while optimizing spatial locality. A separate die is used for storing the input and output vectors. Associated with each DRAM bank which stores the matrix, there is a *PE* which reads the non-zero values of the matrix, fetches the required vector entries and performs the computations. Bringing the compute near the DRAM and exploiting emerging 3D technology, is an orthogonal approach to reduce memory access overheads.

4.9.6 Other FPGA-based Accelerators. Many sparse accelerators have been prototyped on FPGAs [58]. Indeed, AMD/Xilinx provides a turnkey library for SpMV [19, 56]. In [24], the authors propose a higher-performance SpMV using a modified *CSR* format where the data is packed for efficient, streaming access in HBM.[10] They also propose a highly-banked on-chip memory for the vector updates, and a hazard resolution strategy that allows efficient pipelining.

5 COMPARISON OF EXISTING ACCELERATORS

In this section, we analyze and further compare the accelerators that were described in Section 4. We start with Table 3 where we summarize the accelerators, chronologically, showing the research group where they were developed. We see that MIT is very active in the field, as they were involved in developing four different accelerators (e.g., ExTensor, PHI, SpArch, and Gamma).

5.1 Benchmarks, Evaluation, and Performance

Up to this point, we have discussed the architecture of the various accelerators, without reporting the performance they achieve. Most of the matrices used for benchmarking come from the

[10]GraphLily [41] is a precursor of HiSparse, also from Cornell University.

Table 3. Chronological Summary of Sparse Accelerators

Name	Date	Institute/University	Summary
OuterSPACE [64]	2018	Univ. of Michigan/Arizona State Univ.	Highly parallelized SPMD, two levels cache – OP SpMSpM
ExTensor [40]	2019	Univ. of Illinois/NVIDIA / MIT	General purpose tensor algebra – With *tiled* SpMSpM
PHI [60]	2019	MIT (CSAIL)/Carnegie Mellon Univ.	In cache, hash-based accelerator for *reductions*
SPiDRE [10]	2019	UPC, BSC (Spain)/Arm Research	Near main memory data reorganization for *intersections*
MatRaptor [79]	2020	Cornell	Row-parallel with HBM-aware custom format – Gust SpMSpM
SpArch [93]	2020	MIT/Stanford/NVIDIA	Compression method to reduce *POs*, *multiply/merge* pipe – OP SpGEMM
SpaceA [86]	2021	UC Santa Barbara	In memory accelerator for 3D DRAM – IP SpMV
Gamma [90]	2021	MIT (CSAIL)	Parallelized architecture with *FiberCache* for B – Gust SpMSpM
InnerSP [7]	2021	KAIST/DGIST (S. Korea)	A-look-ahead, caches for B, *Hash Reduction Unit* for C – Gust SpMSpM
SortCache [77]	2021	Georgia Tech/Zettaflops/Sandia Labs	In cache accelerator for *reductions*, based on the VBST
ASA [89]	2022	Lehigh Univ./Lawrence Berkeley	Custom cache accelerator for *reductions* – Gust SpGEMM
Flexagon [59]	2023	Univ. of Murcia/Georgia Tech/NVIDIA	Configurable accelerator supporting all three algorithms – SpMSpM
GSCSp [55]	2023	Nanyang Technological Univ.	Element-wise, FPGA-based accelerator – Gust SpMSpM

SuiteSparse Matrix Collection [50] that contains a large number of matrices from diverse, real applications. Domain specific matrix libraries are also used [6, 27, 52, 74], as well as some synthetic matrix generators [2, 11]. The densities are in general less than 1% and can be as low as 10^{-5}%. ExTensor, SortCache, and Flexagon also benchmark with denser matrices. In all cases, large matrices with several million NNZ, exceeding the size of a normal cache, have been evaluated.

In Table 4, we note any required preprocessing, assuming, arbitrarily, that matrices are initially stored in *CSC* format.[11] We see that a few accelerators require a conversion to their specific format (e.g., ExTensor and MatRaptor). PHI, SortCache, and ASA are not concerned by preprocessing since they only address *scatter*. In any case, fair benchmarking must take into account the cost of specialized preprocessing.

Not all accelerators have reached the same level of design maturity, and in Table 4, we have shown how each design was analyzed or simulated. All designs report that they have been simulated at a cycle-accurate level. Most of the accelerators use custom simulators or tools like *gem5* [26], *ZSim* [71], and STONNE [61], but unfortunately, there is no common approach, making it difficult to fairly compare the reported performance. RTL descriptions are often only generated for key parts of the design. To estimate power and area, authors used tools such as CACTI [9] and McPAT [54].[12] In the last column of the table, we show the size of the local memory storage used in simulation by each accelerator. Except for SPiDRE and SpaceA which work directly in main memory, the local memories are in the range of 0.3 MiB to 38 MiB, corresponding to 27 k to 2.8 M entries. Thus, in the model presented in Section 3.1, most of the accelerators are situated in the middle-row of Figure 4, with a local memory corresponding to a few matrix rows.

The accelerators' performances are reported as a speedup versus a baseline, which may differ according to their hardware and software platforms: typically the Intel MKL library [43] and the Armadillo library [72] for CPUs, or cuSPARSE [63] and CUSP [21] for GPUs. Since different baselines with different numbers of threads and cores are used, these speedups need to be interpreted with care. The speedups are in general higher for the *standalone* accelerators as they provide optimized hardware for the entire kernel, while *processor assist* accelerators have less impressive speedups but aim to be general purpose and have lower area overhead.

[11] This choice is based on the SuiteSparse Matrix Collection where matrices are stored *column-major*.
[12] We have not attempted to compare power/energy as the data in the literature is heterogeneous and can not be fairly compared.

Table 4. Qualitative Analysis and Evaluation of Sparse Accelerators

Name	Preprocessing[1]	Functional Evaluation (Tool/Evaluation Specifics/Host[2])	Local Memory Size
OuterSPACE [64]	$A \rightarrow$ *row-major*	*gem5*/Model only key elements; Skip memory allocation; Ignore start-up time/∅	528 KiB (33.8 k entries)
ExTensor [40]	$A, B \rightarrow$ highly *tiled* custom format	Custom Python simulator/Only the *Coordinator* is evaluated; Remainder modeled analytically/∅	~38 MiB (2.5 M entries)
PHI [60]	*N/A*	*ZSim*/Entirely evaluated/16-cores system with 3 level cache	32.3 MiB (2.8 M entries)
SPiDRE [10]	Not needed	*gem5*/*Unknown*/Single ARM core	∅ (main memory size)
MatRaptor [79]	$A, B \rightarrow C^2SR$	*gem5*/Entirely evaluated/∅	320 KiB (27.3 k entries)
SpArch [93]	$A, B \rightarrow$ *row-major*	Custom C++ simulator/Entirely evaluated/∅	940 KiB (62.5 k entries)
SpaceA [86]	$A \rightarrow$ row-aligned to DRAM banks	Custom simulator/Simulation tracking values stored in 3D-DRAM/∅	4 GiB (268.4 M entries)
Gamma [90]	$A, B \rightarrow$ *row-major*	Custom simulator/Entirely evaluated/∅	3 MiB (262.1 k entries)
InnerSP [7]	Not needed	Custom simulator/Entirely evaluated/∅	≤976.5 KiB (119.9 k entries)
SortCache [77]	*N/A*	Custom simulator/Entirely evaluated/Single threaded, in-order core with 3 level cache	16.3 MiB (1.4 M entries)
ASA [89]	*N/A*	Modified *ZSim*/Entirely evaluated, one ASA per core/8-cores with 3 level cache	8×8 KiB (8×1 k entries)
Flexagon [59]	Not needed	Custom simulator based on STONNE/Entirely evaluated/∅	1.3 MiB (327.7 k entries)
GSCSp [55]	$A, B \rightarrow$ *row-major*	HLS/FPGA Prototype/∅	2.1 MiB (178.2 k entries)

[1]We assume matrices are initially stored in *(UOP, CP) column-major* format (*CSC*).
[2]Only for *processor assist* accelerators.

The only accelerators that have a comparable baseline are the *standalone* ones addressing SpM-SpM. They all compare themselves to OuterSPACE with the same benchmark matrices, so we can sort them by their speedup: Gamma (6.6×),[13] InnerSP (4.57×), SpArch (4×), MatRaptor (1.8×), and OuterSPACE (1×). Three of the five *standalone* accelerators use the `Gust` algorithm (Gamma, InnerSP, and MatRaptor). SpArch, an `OP` accelerator employing a complex architecture, outperforms MatRaptor, but does not achieve the same performance as InnerSP or Gamma, which suggests that `Gust` is most adapted to high-performance hardware accelerators.

These measurements are consistent with our model presented in Section 3.1 (Figure 4). OuterSPACE, the baseline, corresponds to graph ⑤. MatRaptor, the first `Gust` accelerator, was able to achieve higher performance by reducing the number of *random* accesses, and creating better regularity (red to yellow), as seen in graph ⑥.

SpArch, is an `OP` accelerator with a much larger local memory, which is used to address the red region in graph ⑤. Their *A*-matrix condensing method, allows them to reduce the number of *POs*, and approach the performance achievable in graph ⑧, although, *in fine*, it is impossible to store all the *POs* in local memory. InnerSP and Gamma correspond to optimizations of the `Gust` approach in graph ⑥, to approach the performance in graph ⑨. This is achieved by custom caches for the *B*-matrix to address the orange region. With the `Gust` algorithm, it is only necessary to store a limited number of *B*-rows, which is achievable with the large local memories in these accelerators. The model presented in Section 3.1 is helpful in understanding the local memory requirements for SpMSpM accelerators.

[13]This speedup increases to 7.7× with their proposed preprocessing.

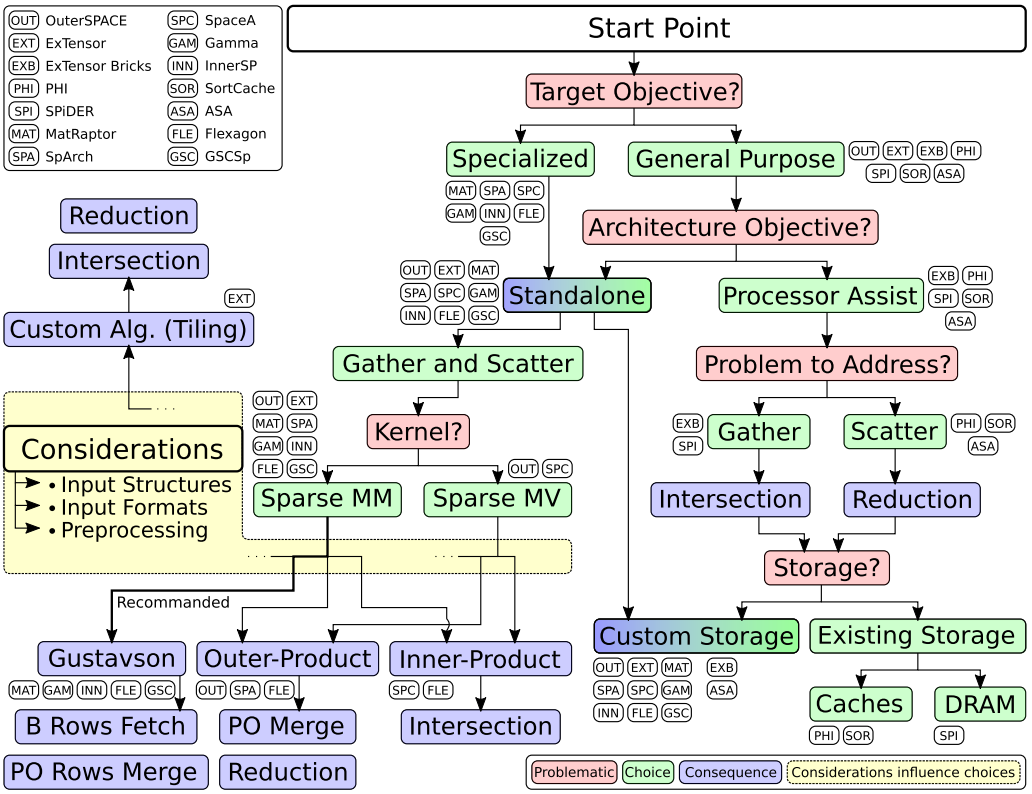


Fig. 16. Flow diagram for architecture selection.

5.2 Analysis for Designers

There is no single "best overall" accelerator. A designer must consider the required performance, the available area, and the class of problems to be addressed. In Figure 16, we present a flow diagram which can guide a designer towards an architecture based on their requirements. Starting from the top-right of the figure, the designer must first decide between a *specialized accelerator* for a specific kernel or a *general purpose accelerator*. The former is preferable if performance must be maximized, requiring a *standalone* accelerator addressing both *gather* and *scatter* (e.g., MatRaptor, SpArch, Gamma, and InnerSP).

A *general purpose accelerator* can either take the form of a highly configurable *standalone* block (e.g., OuterSPACE and ExTensor) or a *processor assist* that boosts performance for a specific task. With a *processor assist* (rightmost branch in the figure), the processor still performs important parts of the kernel, typically the multiplications, but off-loads the *scatter* or *gather* tasks to the accelerator, as these are the tasks which are the bottleneck with a traditional cache hierarchy. For example, for an `IP` algorithm, the difficult part of the *gather* task is performing the *intersection* (e.g., the ExTensor bricks and SPiDRE). Conversely, with `OP` or `Gust`, the challenge with the *scatter* is to efficiently perform *reductions* (e.g., PHI, SortCache, and ASA). Finally, the designer must decide whether the storage is done by reusing existing memory resources (e.g., PHI, SPiDRE, and SortCache) or through custom caches/memories.

Going back to the case of a *standalone* accelerator, the algorithms differ according to the supported kernels. For sparse MV, the accelerators studied here have opted for an `OP` algorithm. For sparse MM, the designers of three of the five highest performing accelerators have chosen `Gust`

algorithm, as this algorithm provides a good tradeoff, simplifying the *gather* by reusing input *B*-rows and the row-by-row generation of the output, simplifying the *scatter* aspect. The fact that the outputs are generated row-by-row, facilitates parallel implementations. `Gust` works well with *(UOP, CP)*, providing consistent input and output formats. Beyond the basic algorithm, *tiling* approaches, as proposed by ExTensor, can be explored.

Orthogonal to previous choices, other design considerations are important. First, the starting point needs to be defined: some accelerators require software to perform *tiling* (e.g., ExTensor) or to rearrange the rows to improve performance (e.g., Gamma). The cost of such preprocessing can be high and is usually only justified if the same matrix is reused multiple times.

6 CONCLUSION

The size of the datasets considered in computational physics, machine learning, and graph analytics is continuing to grow and over the last six years, we note the emergence of many hardware accelerators designed to improve upon the performance of CPUs and GPUs. In this article, we have presented the most important designs and established a comparison, while highlighting the underlying hardware challenges associated with sparse multiplication kernels.

As we have seen, with this class of problem, the challenge is to efficiently use the memory system despite the dereferencing required to access the *metadata* associated with the *sparse data formats*. Depending on the selected MM algorithm, the major challenge is either with the *gather* and *intersection* or with the *scatter* updates. We have identified that accelerators based on *Gustavson's* algorithm achieve a tradeoff, where, on the *gather* side, a limited number of *B*-rows need to be accessed, and on the output side, the result is generated row-by-row, simplifying the *scatter* updates. The best *Gustavson* accelerators have implemented further optimizations such as smart cache eviction strategies where the victim is selected by looking ahead at the *A*-indices. Several of the smaller *processor assist* accelerators reuse existing cache memories and simply modify the behavior of the control logic (directories, tags, eviction policies) to optimize the operation for sparse matrices. On the other hand, the highest performance accelerators propose dedicated, massively parallel *reduction/mergers*, to maximize parallelism.

Given the number and diversity of existing designs, it is clear that further hardware optimizations and improvements are possible. As stated earlier, there may never be a single "best" sparse accelerator but rather designs that are best suited to specific classes of problems. We believe that the structure of the matrices is critical in terms of the sizing of caches and other resources and that in the coming years, new accelerators will be increasingly tailored based on the structure of the matrices for specific classes of problems.

REFERENCES

[1] Richard Barrett, Michael Berry, Tony F. Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk van der Vorst. 1994. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. Society for Industrial.

[2] James Alfred Ang, Brian W. Barrett, Kyle Bruce Wheeler, and Richard C. Murphy. 2010. *Introducing the graph 500*. Office of Scientific, United States.

[3] Bahar Asgari, Ramyad Hadidi, Tushar Krishna, Hyesoon Kim, and Sudhakar Yalamanchili. 2020. ALRESCHA: A lightweight reconfigurable sparse-computation accelerator. In *Proceedings of the 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 249–260.

[4] Venkitesh Ayyar, Evan Weinberg, Richard C. Brower, M .A. Clark, and Mathias Wagner. 2023. Optimizing staggered multigrid for exascale performance. In *Proceedings of the 39th International Symposium on Lattice Field Theory — PoS (LATTICE2022)*. 335.

[5] Ariful Azad, Aydin Buluç, and John Gilbert. 2015. Parallel triangle counting and enumeration using matrix algebra. In *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. 804–811.

[6] Ariful Azad, Georgios A. Pavlopoulos, Christos A. Ouzounis, Nikos C. Kyrpides, and Aydin Buluç. 2018. HipMCL: a high-performance parallel implementation of the Markov clustering algorithm for large-scale networks. *Nucleic Acids Research* 46, 6 (January 2018), e33–e33.

[7] Daehyeon Baek, Soojin Hwang, Taekyung Heo, Daehoon Kim, and Jaehyuk Huh. 2021. InnerSP: A memory efficient sparse matrix multiplication accelerator with locality-aware inner product processing. In *Proceedings of the 2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 116–128.

[8] Vignesh Balaji, Neal Crago, Aamer Jaleel, and Brandon Lucia. 2021. P-OPT: Practical optimal cache replacement for graph analytics. In *Proceedings of the 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 668–681.

[9] Rajeev Balasubramonian, Andrew B. Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New tools for interconnect exploration in innovative off-chip memories. *ACM Transactions on Architecture and Code Optimization* 14, 2, Article 14 (2017), 25 pages.

[10] Adrián Barredo, Jonathan C. Beard, and Miquel Moretó. 2019. POSTER: SPiDRE: Accelerating sparse memory access patterns. In *Proceedings of the 2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. Institute of Electrical and Electronics Engineers, New York, NY, USA, 483–484.

[11] Scott Beamer, Krste Asanović, and David Patterson. 2017. The GAP Benchmark Suite. (2017).

[12] Nathan Bell and Michael Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC'09)*. Association for Computing Machinery, New York, NY, USA, Article 18, 11 pages.

[13] Achi Brandt and Dorit Ron. 2003. *Multigrid Solvers and Multilevel Optimization Strategies*. Springer US, Boston, MA.

[14] Sergey Brin and Lawrence Page. 1998. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems* 30, 1 (1998), 107–117.

[15] Benjamin Brock, Aydın Buluç, Timothy Mattson, Scott McMillan, and José Moreira. 2021. *The GraphBLAS C API Specification*. Technical Report.

[16] Aydın Buluç, John Gilbert, and Viral B. Shah. 2011. 13. Implementing Sparse Matrices for Graph Algorithms. In *Graph Algorithms in the Language of Linear Algebra*. Society for Industrial, 287–313.

[17] Timothy M. Chan. 2007. More algorithms for all-pairs shortest paths in weighted graphs. In *Proceedings of the 39th Annual ACM Symposium on Theory of Computing (STOC'07)*. Association for Computing Machinery, New York, NY, USA, 590–598.

[18] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2018. Format abstraction for sparse tensor algebra compilers. *Proceedings of the ACM on Programming Languages* 2, OOPSLA, Article 123 (2018), 30 pages.

[19] Xilinx Corporation. 2021. *Vitis Sparse Library*. Technical Report. Retrieved from https://xilinx.github.io/Vitis_Libraries/sparse/2021.1/index.html

[20] Vidushi Dadu, Jian Weng, Sihao Liu, and Tony Nowatzki. 2019. Towards general purpose acceleration by exploiting common data-dependence forms. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'52)*. Association for Computing Machinery, New York, NY, USA, 924–939.

[21] Steven Dalton, Nathan Bell, Luke Olson, and Michael Garland. 2014. *Cusp: Generic Parallel Algorithms for Sparse Matrix and Graph Computations*. Retrieved from http://cusplibrary.github.io/

[22] Mehmet Deveci, Simon David Hammond, Michael M. Wolf, and Sivasankaran Rajamanickam. 2018. *Sparse Matrix-Matrix Multiplication on Multilevel Memory Architectures: Algorithms and Experiments*. Office of Scientific, United States.

[23] Stijn Dongen. 2000. *Graph Clustering by Flow Simulation*. PhD Thesis. Center for Math and Computer Science.

[24] Yixiao Du, Yuwei Hu, Zhongchun Zhou, and Zhiru Zhang. 2022. High-performance sparse linear algebra on hbm-equipped fpgas using HLS: A case study on SpMV. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'22)*. Association for Computing Machinery, New York, NY, USA, 54–64.

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

[26] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jeronimo Castrillon, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Carlos Escuin, Marjan Fariborz, Amin Farmahini-Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Anthony Gutierrez, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Miquel Moreto, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur,

Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, William Wang, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Éder F. Zulian. 2020. *The gem5 Simulator: Version 20.0+*. arXiv.

[27] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. 2020. MLPerf inference benchmark. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 446–459.

[28] D. K. Faddeev and V. N. Faddeeva. 1981. Computational methods of linear algebra. *Journal of Soviet Mathematics* 15, 5 (1981), 531–650.

[29] Siying Feng, Jiawen Sun, Subhankar Pal, Xin He, Kuba Kaszyk, Dong-hyeon Park, Magnus Morton, Trevor Mudge, Murray Cole, Michael O'Boyle, Chaitali Chakrabarti, and Ronald Dreslinski. 2021. CoSPARSE: A software and hardware reconfigurable SpMV framework for graph analytics. In *Proceedings of the 2021 58th ACM/IEEE Design Automation Conference (DAC)*. 949–954.

[30] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. 2020. Sparse GPU kernels for deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'20)*. IEEE, 14 pages.

[31] Jianhua Gao, Weixing Ji, Fangli Chang, Shiyu Han, Bingxin Wei, Zeming Liu, and Yizhuo Wang. 2023. A systematic survey of general sparse matrix-matrix multiplication. *ACM Computing Surveys* 55, 12, Article 244 (2023), 36 pages.

[32] Yingxue Gao, Lei Gong, Chao Wang, Teng Wang, Xi Li, and Xuehai Zhou. 2023. Algorithm/hardware co-optimization for sparsity-aware SpMM acceleration of GNNs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42, 12 (2023), 4763–4776.

[33] John R. Gilbert, Steve Reinhardt, and Viral B. Shah. 2006. High-performance graph algorithms from parallel sparse matrices (PARA'06). Springer-Verlag, Berlin.

[34] John R. Gilbert, Steve Reinhardt, and Viral B. Shah. 2008. A unified framework for numerical and combinatorial computing. *Computing in Science and Engineering* 10, 2 (2008), 20–25.

[35] Branko Grünbaum and Geoffrey C. Shephard. 1987. *Tilings and Patterns*. W. H. Freeman and Co., New York.

[36] Sumanth Gudaparthi, Sarabjeet Singh, Surya Narayanan, Rajeev Balasubramonian, and Visvesh Sathe. 2022. CANDLES: Channel-aware novel dataflow-microarchitecture co-design for low energy sparse neural network acceleration. In *Proceedings of the 2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 876–891.

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

[38] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient inference engine on compressed deep neural network. *SIGARCH Computer Architecture News* 44, 3 (2016), 243–254.

[39] Václav Hapla, David Horák, and Michal Merta. 2012. Use of direct solvers in TFETI massively parallel implementation. In *Proceedings of the 11th International Conference on Applied Parallel and Scientific Computing (PARA'12)*. Springer-Verlag, 192–205.

[40] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W. Fletcher. 2019. ExTensor: An accelerator for sparse tensor algebra. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'52)*. Association for Computing Machinery, New York, NY, USA, 319–333.

[41] Yuwei Hu, Yixiao Du, Ecenur Ustun, and Zhiru Zhang. 2021. GraphLily: Accelerating graph linear algebra on HBM-equipped FPGAs. In *Proceedings of the 2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 1–9.

[42] Eun-Jin Im, Katherine Yelick, and Richard Vuduc. 2004. Sparsity: Optimization framework for sparse matrix kernels. *The International Journal of High Performance Computing Applications* 18, 1 (2004), 135–158.

[43] Intel. 2003. Intel Math Kernel Library. Retrieved from https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html#gs.ylo2j7

[44] Satoshi Itoh, Pablo Ordejón, and Richard M. Martin. 1995. Order-N tight-binding molecular dynamics on parallel computers. *Computer Physics Communications* 88, 2 (1995), 173–185.

[45] JESD235A 2016. *JEDEC Updates Groundbreaking High Bandwidth Memory (HBM) Standard*. Technical Report. JEDEC. Retrieved from https://www.jedec.org/news/pressreleases/jedec-updates-groundbreaking-high-bandwidth-memory-hbm-standard

[46] Jeremy Kepner, David Bader, Aydın Buluç, John Gilbert, Timothy Mattson, and Henning Meyerhenke. 2015. Graphs, Matrices, and the GraphBLAS: Seven Good Reasons. *Procedia Computer Science, International Conference On Computational Science (ICCS'15)*, Vol. 51, 2453–2462.

[47] David R. Kincaid, Thomas C. Oppe, and David M. Young. 1989. *ITPACKV 2D user's guide*. Office of Scientific, United States.

[48] Vladimir Kiriansky, Yunming Zhang, and Saman Amarasinghe. 2016. Optimizing indirect memory references with milk. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT'16)*. Association for Computing Machinery, New York, NY, USA, 299–312.

[49] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages* 1, OOPSLA, Article 77 (2017), 29 pages.

[50] Scott P. Kolodziej, Mohsen Aznaveh, Matthew Bullock, Jarrett David, Timothy A. Davis, Matthew Henderson, Yifan Hu, and Read Sandstrom. 2019. The SuiteSparse matrix collection website interface. *Journal of Open Source Software* 4, 35 (2019), 1244.

[51] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. 1991. The cache performance and optimizations of blocked algorithms. *SIGPLAN Notices* 26, 4 (1991), 63–74.

[52] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford large network dataset collection. Retrieved from http://snap.stanford.edu/data

[53] Ruipeng Li, Björn Sjögreen, and Ulrike Meier Yang. 2021. A new class of amg interpolation methods based on matrix-matrix multiplications. *SIAM Journal on Scientific Computing* 43, 5 (2021), S540–S564.

[54] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Institute of Electrical and Electronics Engineers and Association for Computing Machinery, New York, NY, USA, 469–480.

[55] Shiqing Li, Shuo Huai, and Weichen Liu. 2023. An efficient gustavson-based sparse matrix.matrix multiplication Accelerator on Embedded FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42, 12 (December 2023), 4671–4680.

[56] Shiqing Li, Shuo Huai, and Weichen Liu. 2023. An efficient gustavson-based sparse matrix–matrix multiplication accelerator on embedded FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42, 12 (2023), 4671–4680.

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

[58] Thaha Mohammed and Rashid Mehmood. 2022. Performance enhancement strategies for sparse matrix-vector multiplication (SpMV) and iterative linear solvers. (2022).

[59] Francisco Muñoz Martínez, Raveesh Garg, Michael Pellauer, José L. Abellán, Manuel E. Acacio, and Tushar Krishna. 2023. Flexagon: A multi-dataflow sparse-sparse matrix multiplication accelerator for efficient DNN processing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 252–265.

[60] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. 2019. PHI: Architectural support for synchronization- and bandwidth-efficient commutative scatter updates. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'52)*. Association for Computing Machinery, New York, NY, USA, 1009–1022.

[61] Francisco Muñoz-Martínez, José L. Abellán, Manuel E. Acacio, and Tushar Krishna. 2021. STONNE: Enabling cycle-level microarchitectural simulation for dnn inference accelerators. In *Proceedings of the 2021 IEEE International Symposium on Workload Characterization (IISWC)*. 201–213.

[62] Yusuke Nagasaka, Satoshi Matsuoka, Ariful Azad, and Aydın Buluç. 2018. High-performance sparse matrix-matrix products on intel knl and multicore architectures. In *Workshop Proceedings of the 47th International Conference on Parallel Processing (ICPP Workshops'18)*. Association for Computing Machinery, New York, NY, USA, Article 34, 10 pages.

[63] NVIDIA. 2014. cuSPARSE Library. Retrieved from https://developer.nvidia.com/cusparse

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

[65] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally. 2017. SCNN: An accelerator for compressed-sparse convolutional neural networks. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, IEEE Computer Society, Los Alamitos, CA, USA, 27–40.

[66] Gerald Penn. 2006. Efficient transitive closure of sparse matrices over closed semirings. *Theoretical Computer Science* 354, 1 (2006), 72–81.

[67] Michael O. Rabin and Vijay V. Vazirani. 1989. Maximum matchings in general graphs through randomization. *Journal of Algorithms* 10, 4 (1989), 557–567.

[68] Oleg I. Ryabkov. 2022. Implementation of the algebraic multigrid solver designed for graphics processing units based on the AMGCL framework. In *Proceedings of the Parallel Computational Technologies*. Leonid Sokolinsky and Mikhail Zymbler (Eds.), Vol. 1618, Springer International Publishing, Cham, 131–142.

[69] Yousef Saad. 2003. *Iterative Methods for Sparse Linear Systems* (2nd. ed.). Society for Industrial and Applied Mathematics.

[70] Fazle Sadi, Joe Sweeney, Tze Meng Low, James C. Hoe, Larry Pileggi, and Franz Franchetti. 2019. Efficient SpMV operation for large and highly sparse matrices using scalable multi-way merge parallelization. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'52)*. Association for Computing Machinery, New York, NY, USA, 347–358.

[71] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and accurate microarchitectural simulation of thousand-core systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*. Association for Computing Machinery, New York, NY, USA, 475–486.

[72] Conrad Sanderson and Ryan Curtin. 2016. Armadillo: a template-based C++ library for linear algebra. *Journal of Open Source Software* 1, 2 (2016), 26.

[73] P. Scheffler, F. Zaruba, F. Schuiki, T. Hoefler, and L. Benini. 2023. Sparse stream semantic registers: A lightweight ISA extension accelerating general sparse linear algebra. *IEEE Transactions on Parallel and Distributed Systems* 34, 12 (December 2023), 3147–3161.

[74] Shaden Smith, Jee W. Choi, Jiajia Li, Richard Vuduc, Jongsoo Park, Xing Liu, and George Karypis. 2017. FROSTT: The formidable repository of open sparse tensors and tools. Retrieved from http://frostt.io/

[75] Linghao Song, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. 2018. GraphR: Accelerating graph processing using ReRAM. In *Proceedings of the 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 531–543.

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

[77] Sriseshan Srikanth, Anirudh Jain, Thomas M. Conte, Erik P. Debenedictis, and Jeanine Cook. 2021. SortCache: Intelligent cache management for accelerating sparse data workloads. 18, 4, Article 56 (2021), 24 pages.

[78] Sriseshan Srikanth, Anirudh Jain, Joseph M. Lennon, Thomas M. Conte, Erik Debenedictis, and Jeanine Cook. 2019. MetaStrider: Architectures for scalable memory-centric reduction of sparse data streams. *ACM Trans. Archit. Code Optim.* 16, 4 (October 2019), 1–26.

[79] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. 2020. MatRaptor: A sparse-sparse matrix multiplication accelerator based on row-wise product. In *Proceedings of the 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Institute of Electrical and Electronics Engineers, New York, NY, USA, 766–780.

[80] Nitish Srivastava, Hanchen Jin, Shaden Smith, Hongbo Rong, David Albonesi, and Zhiru Zhang. 2020. Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations. In *Proceedings of the 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 689–702.

[81] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel Emer. 2020. *Efficient Processing of Deep Neural Networks* (1st. ed.). Springer Cham.

[82] Reginald P. Tewarson. 1973. *Sparse Matrices*. Academic press New York, State University of New York, Stony Brook, NY.

[83] Richard Vuduc, James W. Demmel, and Katherine A. Yelick. 2005. OSKI: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series* 16, 1 (January 2005), 521.

[84] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. 2007. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *SC'07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*. 1–12.

[85] Yannan Nellie Wu, Po-An Tsai, Angshuman Parashar, Vivienne Sze, and Joel S. Emer. 2022. Sparseloop: An analytical approach to sparse tensor accelerator modeling. In *Proceedings of the 2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Institute of Electrical and Electronics Engineers and Association for Computing Machinery, New York, NY, USA, 1377–1395.

[86] Xinfeng Xie, Zheng Liang, Peng Gu, Abanti Basak, Lei Deng, Ling Liang, Xing Hu, and Yuan Xie. 2021. SpaceA: Sparse matrix vector multiplication on processing-in-memory accelerator. In *Proceedings of the 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 570–583.

[87] Ichitaro Yamazaki and Xiaoye S. Li. 2011. On techniques to improve robustness and scalability of a parallel hybrid linear solver. In *Proceedings of the High Performance Computing for Computational Science – VECPAR 2010*. José M. Laginha M. Palma, Michel Daydé, Osni Marques, and João Correia Lopes (Eds.), Springer, Berlin, 421–434.

[88] Orestis Zachariadis, Nitin Satpute, Juan Gómez-Luna, and Joaquín Olivares. 2020. Accelerating sparse matrix–matrix multiplication with GPU Tensor Cores. *Computers and Electrical Engineering* 88 (2020), 106848. https://doi.org/10.1016/j.compeleceng.2020.106848

[89] Chao Zhang, Maximilian Bremer, Cy Chan, John Shalf, and Xiaochen Guo. 2022. ASA: Accelerating sparse accumulation in column-wise SpGEMM. *ACM Transactions on Architecture and Code Optimization* 19, 4, Article 49 (2022), 24 pages.

[90] Guowei Zhang, Nithya Attaluri, Joel S. Emer, and Daniel Sanchez. 2021. Gamma: Leveraging gustavson's algorithm to accelerate sparse matrix multiplication. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*. Association for Computing Machinery, New York, NY, USA, 687–701.

[91] Guowei Zhang, Webb Horn, and Daniel Sanchez. 2015. Exploiting commutativity to reduce the cost of updates to shared data in cache-coherent systems. In *Proceedings of the 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Institute of Electrical and Electronics Engineers and Association for Computing Machinery, New York, NY, USA, 13–25.

[92] Guowei Zhang and Daniel Sanchez. 2019. Leveraging caches to accelerate hash tables and memoization. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'52)*. Association for Computing Machinery, New York, NY, USA, 440–452.

[93] Zhekai Zhang, Hanrui Wang, Song Han, and William J. Dally. 2020. SpArch: Efficient architecture for sparse matrix multiplication. In *Proceedings of the 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 261–274.

Received 31 May 2023; revised 29 September 2023; accepted 21 December 2023