Sparse Workspace

1 Motivation

General tensor assembly challenge [1] is still not fully solved in the sparse compilation theory. Sparse iteration model [2] formulates the co-iteration over any number of sparse and dense tensors. It targets the data generation problem. Static format transformation is solved by [3]. It targets the data write-back problem. However, they don't directly bridge the dynamic input and arbitrary output. Therefore, they don't solve the general tensor assembly challenge. Inspired by the workspace [4] idea, we propose ***sparse workspace*** to solve the general tensor assembly challenge. Sparse workspace aims to be a flexible sparse compilation algorithm that converts any sparse co-iteration to output with any topological order

2 Background

Tensor is an organization of points. The points are organized in modes. Each point is defined as a list of coordinates and value. The letters in parentheses are called iterators. They are the name of modes. One iterator has only one length, and its level type depends on the host tensor. As is shown in Fig. 1(a), the first mode i equals 2, the second mode j equals 1, and the value equals *green*. Sparse tensors are stored in a special mode order, which we term topological order. The sparse tensor can be accessed most efficiently in the topological order. Though the sparse tensor can also be accessed in other orders, it either requires extra memory or is asymptotically more complex. There's an exception called Knuth's format[5], which has equal cost of being accessed in any order. Fig. 1(b)(c) shows two example of topological order. Both sparse matrices' first levels are dense, and the second are compressed. However, they are different in topological order. We use $\rightarrow$ to annotate such order. We refer readers to [6] for the details of level types.

After defining sparse tensors and topological order, we can introduce expressions. One expression is composed of tensor access and arithmetic operations. The basic expression is assignment. There are two types of assignment: discordant and concordant. If both sides have the same(different) topological order, then we define them as concordant(discordant). Fig. 1(d) illustrates the concordant assignment and Fig. 1(e) illustrates discordant assignment. We

use $\forall$ to define the loop order. We consider $=$ an operator like other arithmetic operators $(+ - \times \div)$. An expression containing more than one tensor belong to the accumulation problem. The accumulation problem is composed of co-iteration and assignment. TACO's sparse iteration theory assumes that operands are accessed in the same topological order. Fig. 1(f)(g) illustrate concordant accumulation and discordant accumulation respectively.

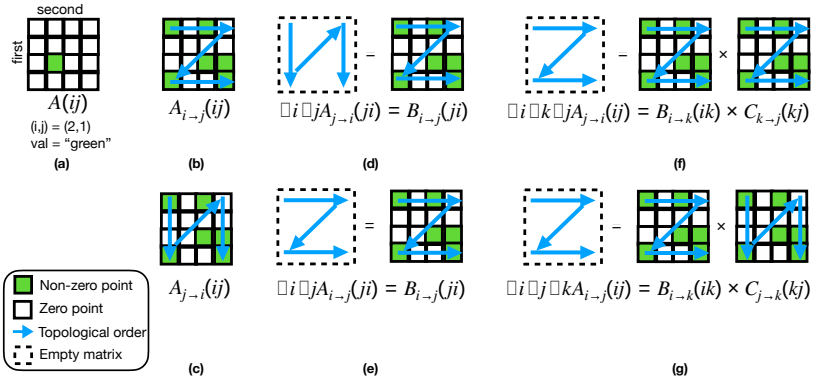

Fig. 1 Illustration of sparse tensors, concordant/discordant assignment/accumulation

Formally, concrete index notation (CIN)[1] is used to describe how an index expression is computed. However, it can't express the topological order conversion which is the key operation of discordant accumulation. Therefore, we extend the original CIN. To be specific, we change the definition of *access* and add *order* and *header*. As is shown in Fig. 2 We use *order* to annotate topological order and an extra *header* to describe conversion.

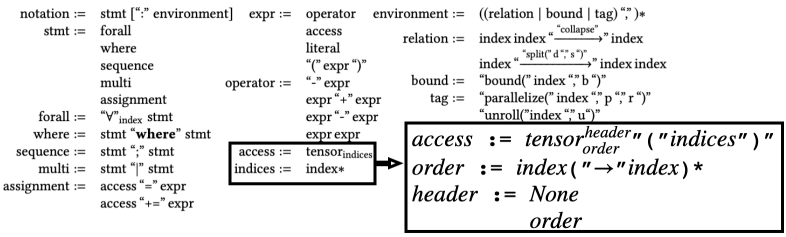

Fig. 2 Illustration of extended concrete index notation

Workspace [4] (dense workspace) can only solve the concordant accumulation problem. It detects the reduction iterator, and assumes that the outer-loop iterator can reuse the workspace. Internally, it uses a bitmap to record the generated non-zero elements. As shown in Fig. 3, the bold letter denotes the reduction iterator, and the workspace size equals the multiplication of the length of all the iterators behind the reduction iterator.

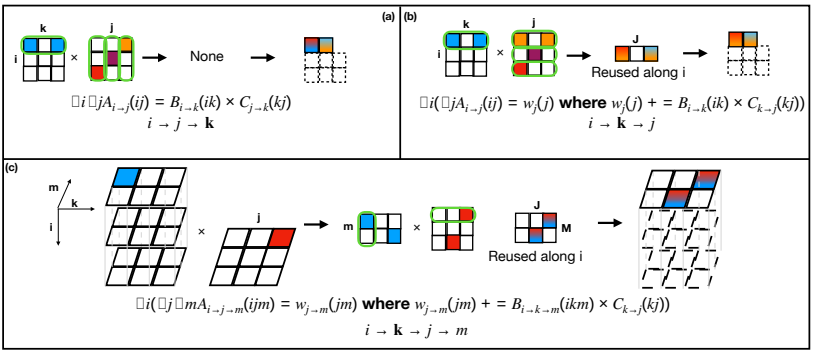

Fig. 3 Illustration of dense workspace

However, the dense workspace can't support discordant accumulation because of its assumption on workspace reuse and direct insertion to output. For example, in Fig. 4(b) current dense workspace is not enough. It requires another $I \times J$ memory to store all the intermediate data. Moreover, the size

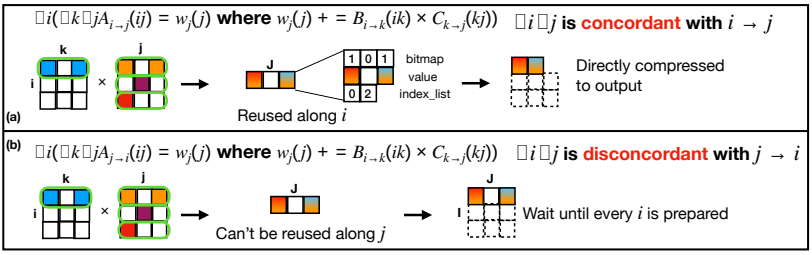

Fig. 4 Illustration of workspace reuse and direct insertion assumption

of a dense workspace is determined by data. Such variation will hurt the performance. For example, Fig. 3(c) allocates a $M \times J$ dense workspace. Such memory size is decided by input data and not controlled by users. The extra memory in Fig. 4(b) is also invisible to users. Besides, the workspace reuse assumption is so strong that it restricts the topological order of tensors. For example, $i \to j \to m$ is one of the 6 possible topological orders of A and the only one that can be handled by dense workspace.

In summary, current dense workspace has two drawbacks. First, it can't correctly convert input points to output with arbitrary topological order. Second, users have little control over the algorithm, making tuning performance challenging. These motivate us to design sparse workspace, a flexible sparse compilation algorithm that converts any sparse co-iteration to output with any topological order.

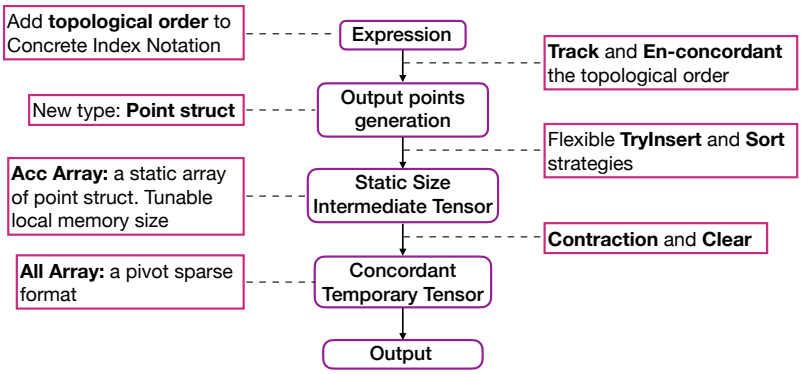

Fig. 5 Work ow of sparse workspace

3 Algorithm Design

Fig. 5 summarizes the whole workflow of sparse workspace. In the front end we extend CIN to include topological order. In the middle end, we represent point with Point struct. As the co-iteration goes, we track and en-concordant the order. This is the key operation that solves the discordant problem and en-concordant input streaming points and the output. Then we create a static size intermediate tensor called Acc Array. This provides users with control over the size of intermediate results. Next, we design a concordant temporary tensor called All Array. It is stored in a pivot sparse format. Using All Array, we transform the discordant accumulation problem to the static format conversion problem, which has been solved by [3]. In the back end, we insert definitions for TryInsert, Sort, Contraction, and Clear functions. They are exposed as low level intermediate representation(LLIR)[2] in the middle end.

Acc is short for accumulation. It is a temporary container that supports appending and iteration in order. From the memory view, Acc is a fixed-size array of Points. The point represents a temporary point in the coordinate space. In the memory, an N-order point is a struct of the whole coordinate path with N coordinates and a value. Acc has user-defined management strategy and size. For example, Acc can be managed as a hash table or a coordinate list. They have different memory control logic. All is a struct of arrays $\{crds[N][capacity], val[capacity]\}$. The arrays store the full coordinate path of each output coordinate point.

The design of sparse workspace also follows the core idea of decoupling data structures from computation [7]. Alg. 1 shows the whole algorithm. The algorithm has five important parts: *TryInsert*, *Clear*, *Sort*, *Contraction*, and *Pack*. All of these five functions except Pack depend on the management strategy of Acc. For example, Alg. 2 and Alg. 3 show the TryInsert for hash table and coordinate list, respectively. TryInsert tries to insert a newly generated point into Acc. Clear function clears the Acc after being contracted. Sort sorts Acc in the lexicographic order to prepare it for contraction. It can be in the ascend or descending order. Contraction contracts points in Acc into All. Pack packs All

to the output tensor, using current TACO's format conversion code. Though the sort and contraction can be fused as merge-sort to reduce the complexity, we separate them into two stages here as a general description. Moreover, All Array is just a logical data structure; the real implementation can be several arrays. This will prevent memory copying from the vertical to the horizontal storage.

Algorithm 1 Sparse Workspace

Require: Value arrays Val_t of TACO tensor t. Accumulator Acc. All output non-zero All. Final output tensor Out

$Allocate(Acc)$
while there's still non-zero **do**
  Iterate on one level: track and en-concordant $crds$
  if reach the last level **then**
    $TryInsert(crds, Expression(\{Val_t\}), Acc)$
    if $Acc.full$ **then**
      $Contraction(Acc, All)$
      $Clear(Acc)$
      $TryInsert(crds, Expression(\{Val_t\}), Acc)$
    end if
  end if
end while
if $Acc.notEmpty$ **then**
  $Sort(Acc)$
  $Contraction(Acc, All)$
  $Clear(Acc)$
end if
$pack(All, Out)$

4 Complexity analysis

Though current TACO can't generate code for discordant accumulation, it has provided basic modules that can be assembled to solve some discordant accumulation problems. We claim that there's no asymptotic difference between such an assembly solution and our sparse workspace. Theoretically, however, sparse workspace can be tuned to outperform the assembly solution. For sparse tensor algebra, asymptotic improvement is brought by the empirical observation: $nnz \ll shape$ [1]. For example, an Amazon product review tensor can have 8 billion zeros for every nonzero [8]. In other words, $\frac{nnz}{shape} \sim 1.25 \times 10^{-10}$ which is extremely small! Because both sparse workspace and assembly solution use the same sparse iteration model to explore all the opportunities for

Algorithm 2 Hash table TryInsert

Require: The expression result of all tensors $val = Expression(\{Val_t\})$. The full coordinate path of the result $Q[:] = getFullCoord(\{t\})$. The accumulator Acc

```
hashkey ←Hashfunc(Q[:])
bitmap.empty()
while bitmap.has(hashkey) do
    if !Acc.conflict(hashkey) then
        Acc[hashkey] ← {Q[:], val}
        Acc.size++
        Acc.full ← False
        return
    else
        if Acc[hashkey] == {Q[:], val} then
            Acc[hashkey].val+ = val
            Acc.full ← False
            return
        else
            bitmap.insert(hashkey)
            hashkey ←Probe(hashkey, Acc)
        end if
    end if
end while
Acc.full ← Ture
Sort(Acc)
Acc.size ← Acc.capacity
```

asymptotic improvement, they have the same asymptotic complexity [9]. However, our analysis shows that sparse workspace can be tuned to outperform assembly solution.

Fig. 6 shows an example of complexity analysis. The CIN of assembly solution is: $\textsf{sequence}\,(\forall i \forall j A_{j\,!\,i}(ij) = T_{i\,!\,j}(ij), \forall i(\forall k \forall j T_{j\,!\,i}(ij) = w_j(j)\,\textsf{where}\,w_j(j)+ = B_{i\,!\,k}(ik) \times C_{k\,!\,j}(kj)))$. The CIN of sparse workspace is: $\forall i(\forall k \forall j T_{j\,!\,i}(ij) = w^{j\,!\,i}_{i\,!\,j}(ij)\,\textsf{where}\,w^{j\,!\,i}_{i\,!\,j}(ij)+ = B_{i\,!\,k}(ik) \times C_{k\,!\,j}(kj))$. Since the asymptotic complexity is the same, we don't consider arithmetic operations. We count two types of instruction: Store(St), and Load and Store(LaS). We use the uppercase of an iterator to denote its length. Therefore, A is $I \times K$, B is $K \times J$, and C is $I \times J$. We also omit the memory allocation cost because it is nearly the same. Block 1 of (a)and(b) have the same number of loop iterations, which we denote as L. We denote the length of Acc Array as a. We assume value and index are both 4 bytes. Therefore, $InsertFail$ happens about $T = \lceil L/a \rceil$ times, and we use l to denote the l-th appearance of $InsertFail$. $nnz(L)$ denotes the total times of bitmap being zero, which happens at the inner-loop of Block a1. $nnz(C)$ denotes the number of non-zero output elements. Fig. 7 lists all the costs of assembly solution and

Algorithm 3 Coordinate list TryInsert

Require: The expression result of all tensors $val = Expression(\{Val_t\})$. The full coordinate path of the result $Q[:] = getFullCoord(\{t\})$. The accumulator Acc

1: **if** $Acc.size == Acc.capacity$ **then**
2: $Acc.full \Leftarrow Ture$
3: $Sort(Acc)$
4: **else**
5: $Acc[Acc.size] \Leftarrow \{Q[:], val\}$
6: $Acc.size++$
7: $Acc.full \Leftarrow False$
8: **end if**

sparse workspace. We assume the *sort* of array with n elements is $nlog_2(n)$. The total cost of *sort* in the inner-loop of Block a1 is $\sum_{i=0}^{I} {}^{1} r_i log_2(r_i)$, where r_i denotes the number of non-zeros in the i-th row of C. To simply our notation, we use $\bar{r}$ that $\sum_{i=0}^{I} {}^{1} r_i log_2(r_i) = \bar{r} log_2 \bar{r}$.

Then we can sum the costs in the Fig. 7. The cost of assembly solution is:

$$(L + nnz(L))[St] + (I\bar{r}log(\bar{r}) + 2nnz(L) + 4J + 3nnz(C)[LaS] \quad (1)$$

The cost of sparse workspace is:

$$3L[St] + (J + nnz(C) + 3Llog_2(3a) + 3(\frac{1}{2}T^2 + \frac{3}{2}T))[LaS] \quad (2)$$

From Equation 1and 2, we can find that both input and output sparse patterns determine their costs. Therefore, sparse workspace can be tuned to outperform assembly solution.

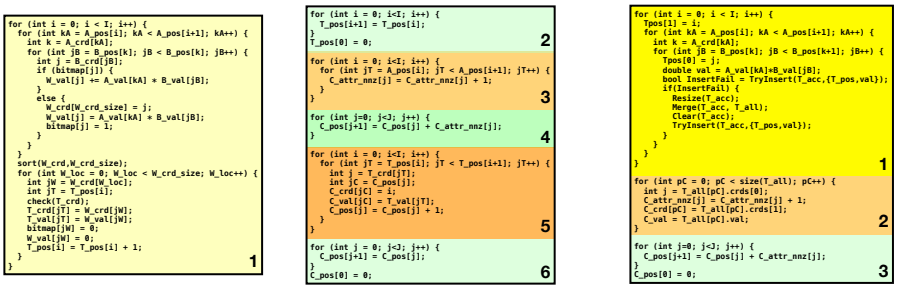

Fig. 6 Comparison of codes between assembly solution and sparse workspace. The color of Block a1 is darker than that of Block b1, meaning they have the same steps of outer-loop iteration, but Block a2 costs more than a1. The same rule is applied to coloring all the blocks.

Block	Description	Cost
-	TryInsert	$3[St]$
-	Sort	$3alog_2(3a)[LaS]$
-	Merge	$3a(l+1)[LaS]$
a1	**L** * Write to W_val + **nnz(L)** * Write to W_crd + **I** *sort + **nnz(L)** * Write to (T_crd + T_val)	$L[St] + nnz(L)[St] + I\bar{r}log(\bar{r})[LaS] + 2nnz(L)[LaS]$
a2,a6,b3	-	$J[LaS]$
a3	-	$nnz(C)[LaS]$
a4	-	$2J[LaS]$
a5	**nnz(C)** * Write to (C_crd + C_val)	$2nnz(C)[LaS]$
b1	**L** * TryInsert + **L/a** * (Resize+Sort+Merge)	$3L[St] + 3Llog_2(3a)[LaS] + 3(\frac{1}{2}T^2 + \frac{3}{2}T)[LaS]$
b2	**nnz(T)** * Load and Store + In-place Zero Cost Copy	$nnz(C)[LaS]$

Fig. 7 Cost analysis of assembly solution and sparse workspace. The dash in **Block** means the entry isn't bound with a block. The dash in **Description** means the entry's functionality is apparent and does not need more explanation. \X * Write to Y" means 4X element stored to array Y. The cost of \Resize" is omitted as mentioned before. \In-place Zero Cost Copy" means that elements stored in the logical array of structs can be transformed to logical arrays with zero cost.

5 Implementation

We create a new subclass of Format called *SpFormat*. It stores the two roles of Acc to the content of TensorVar. It also transports the user-defined information of Acc, such as management strategy and size from the user interface to the lower process. We use the *where* statement in CIN to express sparse workspace. Therefore, tensorVar that holds the sparse workspace must be the lhs of the producer. In lower process, tensorVars will be further transformed to iterators per level. The level-type of tensorVar decides the iterator's level attribute and level function. Except the Pack function, the workspace algorithm is executed in the producer side. In TACO, consumer is lowered before producer. Since the sparse workspace acts like a COO tensor in the consumer side, its relevant iterators will be recreated using the COO format. After the consumer is lowered, they will be recreated using the dense format which is the illusion for producer side. *crds* is tracked during the lowering of forall. In each level, *crds* is appended with the last of current coordinates. In the lowering of ForallBody, compute statement will be undefined if it contains sparse workspace tensor variable because relevant variable of sparse workspace are not registered globally. Such design may seem unsafe, but it actually turns to be effective. First, it can isolate relevant variables from other variables to reduce the risk of bug. Second, it serves as a label, which means that TCC functions can be done when the compute statement if undefined.

6 Compatibility with Other Schedules

These techniques are compatible with other schedules in TACO, except parallelize.

Fuse. This schedule transforms the iterator from coordinate space to position space. Though the sparse workspace tracks the iterator in the coordinate space, iterators all have functions from position space back to coordinate space.

Where. Multiple sparse workspaces can co-exist. Because the iterators are also tensor dependent, changing the role of one sparse workspace will not influence others.

Split. The split schedule creates two new index variables for one index variable. Because the size of All is decided after every generated point is inserted, sparse workspace for child index variables is feasible.

7 Related works

Many optimizations for sparse tensor algebra have been proposed on CPU and GPU. Some representative works on SpGEMM(sparse matrix-sparse matrix multiplication) are listed in Table 1. The criteria are (1) clear explanation of the workspace optimization, (2) code is available or clearly explained by other materials. Various domain specific architectures have been developed to accelerate sparse tensor algebra [10, 11]. Take SpGEMM $C(ij) = A(ik) \times B(kj)$ as an example. [12, 13] accelerate outer-product e.g. $C_{i\,!\,j}(ij) = A_{k\,!\,i}(ik) \times B_{k\,!\,j}(kj)$, [14] accelerates Gustavson's algorithm e.g. $C_{i\,!\,j}(ij) = A_{i\,!\,k}(ik) \times B_{k\,!\,j}(kj)$, and [15] accelerates inner-product algorithm e.g. $C_{i\,!\,j}(ij) = A_{i\,!\,k}(ik) \times B_{j\,!\,k}(kj)$. The most similar one is MeNDA [16]. It uses COO as the intermediate storage and accelerates the sparse matrix transposition. The All array with 2 dimensions is exactly COO format. That shows the potential of sparse workspace being accelerated by novel architecture.

Table 1 Summary of existing sort-based and hash-based accumulators

Paper	Year	Platform	Method
CUSP[17]	2015	Nvidia Tesla/CPU	ESC+COO+COO
AC-SpGEMM[18]	2019	Nvidia Tesla and Titan	ESC+HiCOO
cuSPARSE[19]	2012	Nvidia Tesla	Hash+CSR+CSR
nSPARSE[20]	2017	Nvidia Pascal	Hash+CSR+CSR
bhSPARSE[21]	2014	Nvidia Titan/CPU	Hash+Heap+Merge

References

[1] Kjolstad, F.: Sparse tensor algebra compilation. Ph.d. thesis, Massachusetts Institute of Technology, Cambridge, MA (Feb 2020). http://tensor-compiler.org/files/kjolstad-phd-thesis-taco-compiler.pdf

[2] Kjolstad, F., Kamil, S., Chou, S., Lugato, D., Amarasinghe, S.: The tensor algebra compiler. Proc. ACM Program. Lang. **1**(OOPSLA), 77–17729 (2017). https://doi.org/10.1145/3133901

[3] Chou, S., Kjolstad, F., Amarasinghe, S.: Automatic generation of efficient sparse tensor format conversion routines. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI 2020, pp. 823–838. Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3385412.3385963. https://doi.org/10.1145/3385412.3385963

[4] Kjolstad, F., Ahrens, P., Kamil, S., Amarasinghe, S.: Tensor algebra compilation with workspaces. In: 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp. 180–192 (2019). IEEE

[5] Knuth, D.E.: Sorting and searching (1973)

[6] Chou, S., Kjolstad, F., Amarasinghe, S.: Format abstraction for sparse tensor algebra compilers. Proc. ACM Program. Lang. **2**(OOPSLA), 123–112330 (2018). https://doi.org/10.1145/3276493

[7] Codd, E.F.: A relational model of data for large shared data banks. Communications of the ACM **13**(6), 377–387 (1970)

[8] McAuley, J., Leskovec, J.: Hidden factors and hidden topics: understanding rating dimensions with review text. In: Proceedings of the 7th ACM Conference on Recommender Systems, pp. 165–172 (2013)

[9] Ahrens, P., Kjolstad, F., Amarasinghe, S.: Autoscheduling for sparse tensor algebra with an asymptotic cost model. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. PLDI 2022, pp. 269–285. Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3519939.3523442. https://doi.org/10.1145/3519939.3523442

[10] Wu, Y.N., Tsai, P.-A., Parashar, A., Sze, V., Emer, J.S.: Sparseloop: An analytical approach to sparse tensor accelerator modeling. arXiv preprint arXiv:2205.05826 (2022)

[11] Hsu, O., Strange, M., Won, J., Sharma, R., Olukotun, K., Emer, J., Horowitz, M., Kjolstad, F.: The sparse abstract machine. arXiv preprint arXiv:2208.14610 (2022)

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

[13] Wang, Y., Zhang, C., Xie, Z., Guo, C., Liu, Y., Leng, J.: Dual-side sparse tensor core. In: 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA), pp. 1083–1095 (2021). IEEE

[14] Srivastava, N., Jin, H., Liu, J., Albonesi, D., Zhang, Z.: Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product. In: 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 766–780 (2020). IEEE

[15] Hegde, K., Asghari-Moghaddam, H., Pellauer, M., Crago, N., Jaleel, A., Solomonik, E., Emer, J., Fletcher, C.W.: Extensor: An accelerator for sparse tensor algebra. In: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, pp. 319–333 (2019)

[16] Feng, S., He, X., Chen, K.-Y., Ke, L., Zhang, X., Blaauw, D., Mudge, T., Dreslinski, R.: Menda: a near-memory multi-way merge solution for sparse transposition and dataflows. In: Proceedings of the 49th Annual International Symposium on Computer Architecture, pp. 245–258 (2022)

[17] Dalton, S., Olson, L., Bell, N.: Optimizing sparse matrix—matrix multiplication for the gpu. ACM Transactions on Mathematical Software (TOMS) **41**(4), 1–20 (2015)

[18] Winter, M., Mlakar, D., Zayer, R., Seidel, H.-P., Steinberger, M.: Adaptive sparse matrix-matrix multiplication on the gpu. In: Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming, pp. 68–81 (2019)

[19] Demouth, J.: Sparse matrix-matrix multiplication on the gpu. In: Proceedings of the GPU Technology Conference, vol. 3 (2012)

[20] Nagasaka, Y., Nukada, A., Matsuoka, S.: High-performance and memory-saving sparse general matrix-matrix multiplication for nvidia pascal gpu. In: 2017 46th International Conference on Parallel Processing (ICPP), pp. 101–110 (2017). https://doi.org/10.1109/ICPP.2017.19

[21] Liu, W., Vinter, B.: An efficient gpu general sparse matrix-matrix multiplication for irregular data. In: 2014 IEEE 28th International Parallel and Distributed Processing Symposium, pp. 370–381 (2014). IEEE