A Tensor Algebra Compiler for Sparse Differentiation

Amir Shaikhha
School of Informatics
University of Edinburgh
Edinburgh, United Kingdom
amir.shaikhha@ed.ac.uk

Mathieu Huot
Department of Computer Science
University of Oxford
Oxford, United Kingdom
mathieu.huot@stx.ox.ac.uk

Shideh Hashemian
School of Informatics
University of Edinburgh
Edinburgh, United Kingdom
s.hashemian@sms.ed.ac.uk

2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) | 979-8-3503-9509-9/24/$31.00 ©2024 IEEE | DOI: 10.1109/CGO57630.2024.10444787

***Abstract*—Sparse tensors are prevalent in many data-intensive applications. However, existing automatic differentiation (AD) frameworks are tailored towards dense tensors, which makes it a challenge to efficiently compute gradients through sparse tensor operations. This is due to irregular sparsity patterns that can result in substantial memory and computational overheads. We propose a novel framework that enables the efficient AD of sparse tensors. The key aspects of our work include a compilation pipeline leveraging two intermediate DSLs with AD-agnostic domain-specific optimizations followed by efficient C++ code generation. We showcase the effectiveness of our framework in terms of performance and scalability through extensive experimentation, outperforming state-of-the-art alternatives across a variety of synthetic and real-world datasets.**

***Index Terms*—Sparse Tensor Algebra, Automatic Differentiation, Semi-Ring Dictionaries**

I. Introduction

Sparse tensors are essential in many scientific and engineering applications, such as natural language processing, computer vision, and graph analytics. Unlike dense tensors, which store all of their elements regardless of their value, sparse tensors only store non-zero values, resulting in significant memory savings and computational efficiency. Sparse tensors also enable efficient representation and manipulation of high-dimensional data structures, which are often encountered in modern machine learning and scientific computing, such as sparse tensors representing the frequency of words in a document or corpus in natural language processing, adjacency matrices of large and sparse graphs in network/relational analysis, or sparse user-item interaction matrices for collaborative filtering in recommender systems. This has inspired recent developments for better support of sparse tensors [1]–[3].

Automatic differentiation (AD) is a fundamental technique in machine learning and scientific computing that enables efficient computation of the gradient of a function. This is crucial for optimization, parameter estimation, risk analysis in finance, and many other applications in which gradient-based optimization methods are employed. While AD tools for dense tensors are well-established, the lack of efficient AD tools for sparse tensors hinders wider adoption and posing a significant research challenge for these techniques. Libraries such as TensorFlow, PyTorch, and JAX provide efficient and scalable implementations of gradient computation for dense tensor kernels, but their support for sparse operations is limited [4]–[6]. As a result, there have been various efforts to manually provide differentiation for particular sparse tensor kernels [7].

AD for sparse tensor algebra is more challenging than for dense tensor algebra for several reasons. Firstly, the structure of sparse tensors is more complex than that of dense tensors, and their sparsity patterns are often irregular and vary across different operations. This makes it challenging to propagate gradients through the computation graph efficiently and to identify which elements of the sparse tensor are relevant for the gradient computation. Secondly, sparse tensor operations often require specialized data structures and algorithms, such as compressed sparse row/column (CSR/CSC) formats, which are not fully supported by most AD-enabled frameworks.

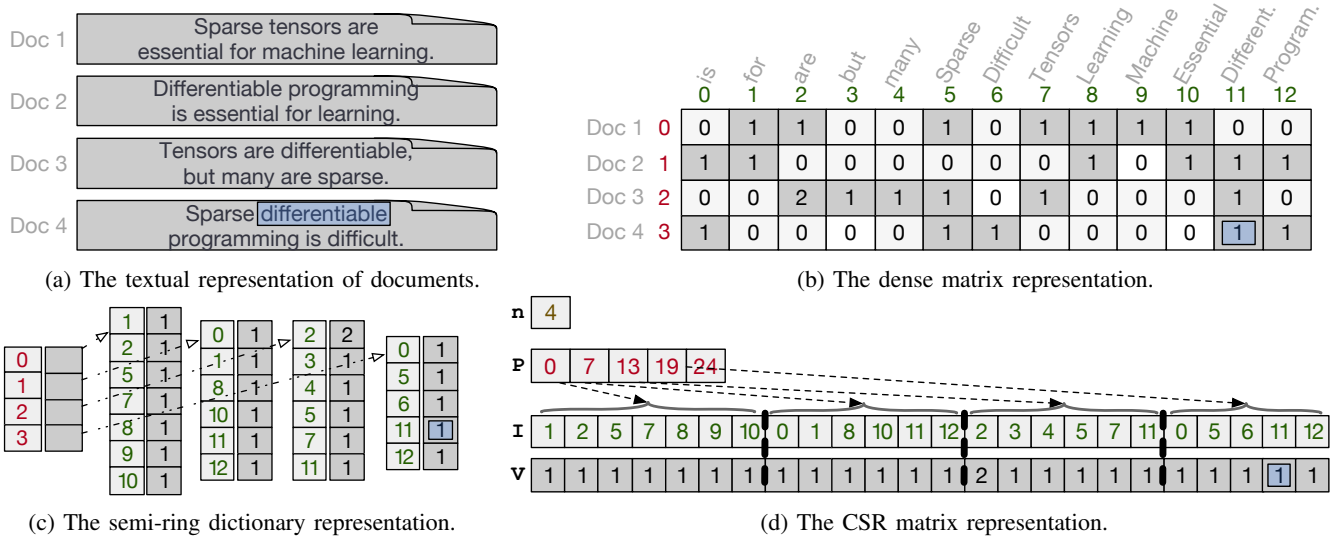
This paper presents ∇SD, the first differentiable programming framework that supports the automatic differentiation of arbitrary sparse computations. As opposed to the existing frameworks that offer AD support for a limited number of sparse kernels [7]–[10], ∇SD allows the AD of an arbitrary sparse computation expressible in tensor algebra, and user-defined expressions expressible in its intermediate language.

The key insight is to *perform differentiation over a logical representation of a sparse tensor*. This means that there is a clear separation of concerns between the semantics of differentiation over a program on the one hand, and optimizations and data layout representations on the other (cf. Figure 5).

The physical format of sparse tensors (e.g., CSR/CSC) involves multiple arrays storing a compressed representation of the matrix (cf. Figure 1d). The computations over such representations involve *imperative* while-loops over these arrays. However, our logical representation uses a nested dictionary, where sparse computations are expressed *functionally* as nested summations (reductions/foldings) over them. This representation can be later fused with a physical storage format (cf. Section V).

In more detail, the contributions of this paper are as follows:

- We present ∇SD, the first framework with systematic support for the automatic differentiation of sparse tensors. ∇SD is based on SDQLite [11], [12], an intermediate language that expresses sparse tensor workloads by *separating the tensor computations from the storage formats* (Section III).



(a) The textual representation of documents.

(b) The dense matrix representation.

(c) The semi-ring dictionary representation.

(d) The CSR matrix representation.

Fig. 1: The bag-of-words representation of documents using a matrix with documents as rows and words as columns with different dense and sparse formats. The highlighted box shows the element at the 3^{rd} row and 11^{th} column in different formats.

- We introduce *tensorized forward-mode AD* that computes the gradients in a batch (Section IV). Our AD is over the logical fragment of the language, called Logical SDQLite, without worrying about the physical storage formats.
- The differentiated program is then optimized by leveraging AD-agnostic transformations: (1) *sparsity propagation*, (2) composing with the physical storage formats, (3) algebraic rewrite rules applied in a cost-based manner using *equality saturation* [13], [14], and (4) destination-passing style to remove unnecessary tensors in nested loops (Section V).
- We experimentally validate the effectiveness of ∇SD in comparison with the state-of-the-art AD frameworks (Section VI). We show that ∇SD scales gradient computation to large sparse matrices over real-world and synthetic datasets.

II. Background

Sparse Tensors. Sparse tensors are data structures commonly used to represent high-dimensional data that have a majority of zero values. Sparse tensors have a compact representation that only stores the non-zero values and their corresponding indices, which makes them more memory-efficient than dense tensors for large-scale data. Sparse tensors are used in many domains, including natural language processing, computer vision, and scientific computing. For example, in natural language processing, sparse tensors can be used to represent text data as a bag-of-words or term frequency-inverse document frequency (TF-IDF) matrix, where the rows correspond to documents and the columns correspond to words (Figure 1).

Sparse tensors can be manipulated using a variety of specialized algorithms and data structures, such as compressed sparse row (CSR) and compressed sparse column (CSC) formats, which enable efficient matrix-vector multiplication and other operations. If the majority of the elements of the input matrices of size $m \times n$ are zeros (the number of non-zero elements, denoted by nnz, is such that $nnz << m \times n$), one can use the CSR format shown in Figure 1d. In this format, the array `P` is a compressed representation of rows, whereas `I` and `V` show the columns and values of non-zero elements. For example, `P(0)=0,P(1)=7` depicts that the indices 0-6 of `I`/`V` correspond to the column/value of the elements in the row=0 of the matrix, and 7-14 for the positions of the row=1.

Automatic Differentiation (AD). AD is a widely used technique in machine learning and scientific computing that enables efficient computation of the gradient of a function. The gradient is a crucial quantity in many optimization, parameter estimation, and machine learning algorithms, and its computation is often a bottleneck in the training process. AD computes the gradient by breaking down a function into a series of elementary operations and applies the chain rule to compute the derivatives of each operation. The result is an exact gradient that is computed with a similar computational cost as the original function, with no need for approximate methods such as finite differences, or manual derivation.

Forward-Mode AD (FAD). One method of computing the gradient of a function is the FAD technique which involves computing the derivatives of each operation in the forward direction through the computational graph. The method starts with the input variables and propagates the values and their derivatives through the graph, until the output variable is reached. At each operation, the derivative of the output variable with respect to each input variable is computed using the chain rule, which is used to compute the function's gradient.

Example 1 - Vector Dot Product. The following function:

$$f([x_1, x_2], [y_1, y_2]) = x_1 y_1 + x_2 y_2$$

takes two pairs of input variables x_1, x_2 and y_1, y_2 and computes the dot product of the vectors $x = [x_1, x_2]$ and $y = [y_1, y_2]$. To compute the gradient of f using FAD, we start by converting the program into ANF [15]:

Construct	Description	Language
`e ::= sum(<x,y> in e) e` `| { e -> e } | { } | e(e)` `| let x=e in e | x | if e then e` `| e + e | e * e | uop(e) | e bop e` `| n | r | false | true`	*Dictionary Aggregation* *Singleton/Empty Dictionary, Lookup* *Variable Binding & Access, Conditional* *Addition, Multiplication, Unary, and Binary operators* *Numeric and Boolean Constants*	Logical SDQLite
`T ::= I | bool | D` `D ::= real | { I -> D }  I ::= int`	*Scalar and Tensor Types* *Dictionary (Tensor) Type, Index Type (Integer Only)*	
`e ::= ... | e:e | e(e:e)` *cf. [12]*	*Logical SDQLite Expressions, Range, Sub-Array, others*	Physical SDQLite
`T ::= ...  D ::= ... | int`	*Logical SDQLite Types, Tensor Types (+ Integer)*	
`I ::= ... | dense_int`	*Index Types (+ Dense Integer)*	
Definition of **zero**`[D]`: **zero**`[real]` $\triangleq$ `0` **zero**`[{I -> D}]` $\triangleq$ `{ }`	Definition of $\otimes$: **real** $\otimes$ `D` $\triangleq$ `D` `{I -> D1}` $\otimes$ `D2` $\triangleq$ `{I -> D1` $\otimes$ `D2}`	
Definition of **let**-tupling: **let** `<v1, ..., vn> = <e1, ..., en> in e` $\triangleq$ **let** `v1=e1 in ... let vn=en in e`	Definition of **tensor** `n`: **tensor** `0` $\triangleq$ **real** **tensor** `n` $\triangleq$ `{` **int** `->` **tensor** `(n-1) }`	

Fig. 2: Grammar of the languages used in ∇SD.

$$f([x_1,x_2],[y_1,y_2]) = \begin{array}{l}\textbf{let } t_1 = x_1y_1 \textbf{ in}\\ \textbf{let } t_2 = x_2y_2 \textbf{ in}\\ \textbf{let } t_3 = t_1 + t_2 \textbf{ in } t_3\end{array}$$

The forward-mode AD lifts every variable to a dual number by associating a tangent variable v' to each input and intermediate variable v. Then, each intermediate tangent variable is computed by following the chain rule. In the previous example, the function f is transformed as follows:

$$\begin{array}{l}f'([x_1,x_2],[y_1,y_2],[x_1',x_2'],[y_1',y_2']) =\\ \quad\begin{array}{ll}\textbf{let } t_1 = x_1y_1 \textbf{ in} & \textbf{let } t_1' = x_1'y_1 + x_1y_1' \textbf{ in}\\ \textbf{let } t_2 = x_2y_2 \textbf{ in} & \textbf{let } t_2' = x_2'y_2 + x_2y_2' \textbf{ in}\\ \textbf{let } t_3 = t_1 + t_2 \textbf{ in} & \textbf{let } t_3' = t_1' + t_2' \textbf{ in } t_3'\end{array}\end{array}$$

To compute the partial derivative of f with respect to each input we need to set the corresponding tangent variable to 1 and the other input tangent variables to 0. For example, the gradient of f with respect to the first vector is computed by the following partial derivative computations:

$$\begin{array}{ll}f'([a_1,a_2],[b_1,b_2],[1,0],[0,0]) \rightarrow^* & b_1 = \frac{\partial[\![f]\!]}{\partial x_1}(a_1,a_2,b_1,b_2)\\ f'([a_1,a_2],[b_1,b_2],[0,1],[0,0]) \rightarrow^* & b_2 = \frac{\partial[\![f]\!]}{\partial x_2}(a_1,a_2,b_1,b_2)\end{array}$$

Reverse-Mode AD (RAD). FAD is computationally expensive for the derivative computation of scalar-valued functions with tensor inputs, which among other use cases appear in training machine learning models by optimizing an objective function. This is due to the fact that when differentiating a program representing a function $\mathbb{R}^n \rightarrow \mathbb{R}$, one needs n runs of the program transformed by FAD to obtain the whole gradient. The RAD technique, which computes the gradient of such functions in one run, is then more appropriate and is heavily used in deep learning frameworks [8]–[10].

Example 1 (cont.). Consider a generalization of the previous function, where $\cdot$ denotes the dot product of two vectors:

$$f(V_1,V_2) = V_1 \cdot V_2$$

If each input vector has m elements, then the cost of FAD is $O(m^2)$ as it requires m forward passes, each costing $O(m)$. However, RAD can compute the gradient by one forward pass to compute the primal values and one reverse pass to compute the gradient values, resulting in an $O(m)$ overall complexity.

Vector Forward-Mode AD. The use cases that require the computation of the full Jacobian matrix [16] motivated efforts on batch computations of FAD [17], [18]. It has been demonstrated that by compiler optimizations, one can recover the asymptotic performance of RAD on vectorized FAD [18].

Sparse AD. The irregular sparsity patterns of sparse tensors and the imperative while-loops for co-iterating over the compressed arrays, pose significant challenges for AD. Existing LA frameworks do not support gradients over sparse formats and require conversion to dense formats before AD.

Example 1 (cont.). In the previous example, if the majority of the elements of the input vectors of sizxe m are zeros (the number of non-zero elements, denoted by nnz, is such that $nnz << m$), one can use the CSR format shown in Figure 1d. In this representation, the array `pos` is a compressed representation of rows, whereas `idx` and `val` show the columns and values of non-zero elements. For example, `pos(0)=0,pos(1)=7` depicts that the indices 0 to 6 of `idx`/`val` correspond to the column/value of the elements in the row=0 of the matrix, and 7 to 14 for the positions of the row=1. However, the existing linear algebra frameworks rather than computing the gradient of vector-dot product in $O(nnz)$, compute it over the dense representation in $O(m)$.

Semi-ring Dictionaries. Semi-ring dictionaries subsume sets, multisets, and dense/sparse tensors [11]. A semi-ring is a set with two binary operations that satisfy certain axioms, such as associativity, distributivity, and commutativity. For example, the set of non-negative integers with addition and multiplication forms a semi-ring, and the set of Booleans with logical $\vee$ and $\wedge$ forms another semi-ring. Semi-ring dictionaries represent sparse tensors as key-value pairs. In sparse vectors, the keys correspond to the vector indices and the values correspond to the non-zero elements. In sparse matrices, the keys correspond to the row indices and the values correspond to the sparse vector associated with that row. The semi-ring operations are

then defined in terms of the operations on the values, such as addition or multiplication. The multiplication operator for semi-ring dictionaries has a semantics of tensor outer product, as can be observed next.

Example 2 - Scalar-Vector Multiplication. Consider the scalar-vector product between a scalar value `s` and a vector value `V` represented using a semi-ring dictionary. The equivalent semi-ring dictionary representation is `s * V`, where `*` has the semantics of tensor outer product.

SDQL. SDQL [11] is a functional language for querying against semi-ring dictionaries. SDQL is expressive enough to capture database queries and linear algebra expressions; this makes it appropriate as an intermediate language for hybrid database and machine learning workloads. SDQL provides the following constructs for manipulating semi-ring dictionaries:

1) `dict(k)` accesses the value associated with the key `k` in `dict`. If the key does not exist, it returns the semi-ring's zero element (0 in the case of real and natural numbers).
2) `{k->v}` constructs a dictionary with `k,v` key-value pair.
3) **`sum`**`(<k,v> `**`in`**` dict) f(k,v)` iterates over the key-value pairs of `dict` and computes the summation of `f(k,v)`, starting from the semi-ring's zero element.

Example 1 (cont.). The equivalent SDQL expression for $V1 \cdot V2$ can be one of the following two:

```
sum(<i, a> in V1) a * V2(i)
sum(<i, a> in V2) V1(i) * a
```

The preferred choice depends on the number of non-zero elements of `V1` and `V2`. If `V1` (resp. `V2`) has fewer non-zero elements, the left (resp. right) variant is more efficient. Otherwise, if both have the same number of non-zero elements (e.g., both are dense), both variants have the same performance.

SDQLite. SDQLite [12] is a dialect of SDQL tailored for sparse tensor processing; it restricts SDQL to the types required for sparse tensors while extending it with constructs required for different sparse storage formats (e.g., CSR, CSC).

III. Languages

In this section, we give an overview of the languages used in ∇SD. The compilation pipeline consists of two fragments of SDQLite: the smaller fragment, Logical SDQLite, on which AD will be performed, and the full fragment, Physical SDQLite, which augments the logical subset of the language with the constructs for expressing the different sparse storage formats. The grammar of these languages is shown in Figure 2.

Logical SDQLite. Initially, the program is expressed in a subset of SDQLite that is sufficient for expressing tensor programs at the logical level, i.e., without worrying about the storage format. Thus, at this stage, we do not require the support for dense arrays. Furthermore, there is no need for expressing tuples. Nevertheless, for convenience, we use tupled let-binding as a syntactic sugar for multiple let bindings.

Logical SDQLite is expressive enough to capture Einstein summations and beyond [11]. For example, `map` of function `f` over the values of a tensor `e`, cannot be expressed in frameworks such as TACO, but is expressed in SDQLite as:

```
sum(<k,v> in e) {k -> f(v)}
```

Physical SDQLite. The storage specifications require additional constructs such as:

1) `(st:en)` for building a dense array holding the range of numbers from `st` to `en` (excluding).
2) `arr(st:en)` to specify the sub-array of `arr` ranging from `st` to `en` (excluding).
3) Additional annotations for guiding rewrite rules such as **`unique`** and **`dense`** [12].

Thus, after combining the differentiated program with the storage specification, we include these constructs for the Physical SDQLite. We go back to this intermediate language in Section V. The next section focuses on Logical SDQLite and the differentiation rules over its constructs.

IV. Differentiation

In this section, we present the differentiation transformations applied to Logical SDQLite expressions. First, for exposition purposes, we present a variant of traditional forward-mode AD (FAD). Then, we show a tensorized FAD that not only subsumes the traditional FAD, but also computes gradients more efficiently. Finally, we show the high-level API exposed to the programmers.

A. Scalar Forward-Mode Transformation

Traditional FAD uses dual numbers to compute the tangent (derivative) component along with the actual (original) computation. We refer to it as scalar FAD because for each scalar expression, it stores a scalar tangent component.

Similar to other FAD frameworks, ∇SD precedes the differentiation transformation with an ANF conversion [15]. This allows for sharing sub-expressions and avoids duplication of computations for non-unary constructs such as multiplication. Logical SDQLite does not allow function definitions nor higher-order functions; all functions need to be inlined [19].

Scalar Constructs. As opposed to existing functional AD systems, ∇SD does not use explicit pair construction and projection for dealing with dual numbers. Instead, the $\mathcal{F}[\![]\!]$ construct only computes the tangent part of differentiation and refers to the expressions in the ANF transformed program for primal components (cf. the rule for let binding). This avoids the need to extend the target language of differentiation with pairing constructs. Furthermore, this makes the differentiation rules simpler. For every unary real operation `op`, we assume that the language has a unary real operation `op'` representing its derivative. Finally, the differentiation for all discrete types (**`int`** and **`bool`**) is `0`.

Example 1 (cont.). Consider again the case of the dot-product of two unrolled vectors of size two initially used in Section II. Applying differentiation over the ANF transformed program in SDQLite is as follows:

$$\mathcal{F}\left[\!\!\left[\begin{array}{l}\textbf{let } t1 = x1*y1 \textbf{ in}\\ \textbf{let } t2 = x2*y2 \textbf{ in}\\ \textbf{let } t3 = t1+t2 \textbf{ in } t3\end{array}\right]\!\!\right]$$

After applying the FAD rules, we obtain the following:

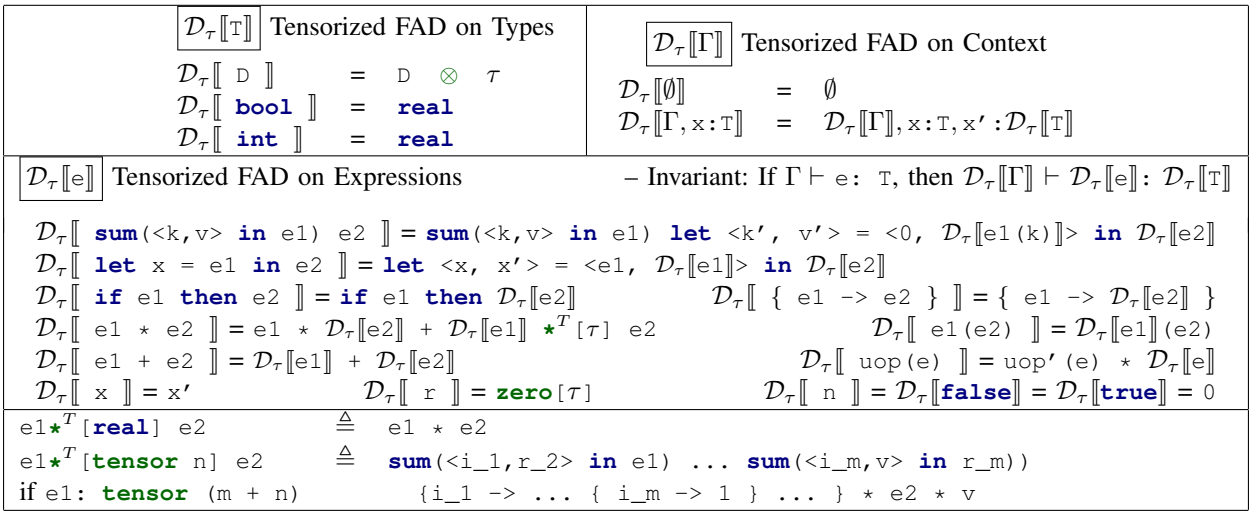


Fig. 3: Tensorized Forward-mode Automatic Differentiation (FAD) rules for SDQLite expressions. The type τ needs to be a tensor type, i.e., it follows the grammar of D. The scalar FAD is the special case of $\mathcal{F}[\![]\!]=\mathcal{D}_{\mathtt{real}}[\![]\!]$.

```
let <t1,t1'> = <x1*y1,x1*y1'+x1'*y1> in
let <t2,t2'> = <x2*y2,x2*y2'+x2'*y2> in
let <t3,t3'> = <t1+t2,t1'+t2'> in t3'
```

Note that the let-binding constructs are syntactic sugar; there is no pair created (cf. Figure 2).

Tensor Constructs. By choosing not to incorporate pairs in the language, we have eliminated the option of differentiating vectors as vectors of pairs (i.e., arrays of structs). Instead, an expression of type `tensor n` is differentiated as an expression with the same type. One of the interesting tensor-based differentiation rules is our rule for summation, where we need to access the corresponding element from the differentiated range, as we can observe in the following example.

Example 1 (cont.). Let us go back to the dot-product for two vectors V1 and V2. The differentiation transformation is expressed as follows:

$\mathcal{F}[\![\ \ \mathtt{sum(<i,a> in V2) V1(i) * a}\ \ ]\!]$

Applying differentiation rules results in the following program:

```
sum(<i,a> in V2)
  let <i',a'> = <0,V2'(i)> in
  V1(i) * a' + V1'(i) * a
```

In order to compute the gradient of this function with respect to one of its vector inputs, say V1, we need to repeatedly set V1' into a one-hot vector that is 1 at index j and 0 everywhere else, and set V2' to be the zero vector. This requires multiple rounds of running the forward-mode AD for different one-hot vectors, which is computationally expensive. Previous research [18] has shown how this can be optimized by wrapping the vector construction around the forward-mode AD and applying loop optimizations.

Example 2 (cont.). Consider the case of scalar-vector product, represented as `s * V` in SDQLite. Applying the scalar FAD rules on this program results in `s * V' + s' * V`. In the case of differentiation with respect to V, similar to dot-product, one needs to repeatedly pass all different one-hot vectors as V'. However, the differentiation with respect to s can be done by setting s' to 1 and V' to `zero[tensor 1]`.

Next, we show an alternative differentiation transformation that enables native tensorized forward-mode AD.

B. Tensorized Forward-Mode Transformation

As tensors are first-class citizens in SDQLite, one can directly express the differentiation with respect to tensor variables of type τ, represented by $\mathcal{D}_\tau[\![]\!]$. This means that the derivative of an expression of type `tensor n` with respect to a tensor of type `tensor m`, will be a `tensor (n + m)`, which is the same as the tensor product type (cf. Figure 3).

Figure 3 shows the rules for tensorized FAD. They generalize the rules for scalar forward-mode AD, which one recovers by setting τ to be `real`. The key differences are in the rules for constant reals and multiplication. Rather than just returning a real-valued 0, tensorized FAD returns the zero value of type τ represented as `zero[`τ`]`. For multiplication, if τ is a tensor type with a non-zero order, the first term still computes the multiplication of e1 and the differentiation of e2. However, the second term requires re-arranging the indices of the tensors. This complication can be avoided by only allowing for the multiplication of real numbers. This is achieved by applying multiplication normalization (cf. Section V).

Example 1 (cont.). In our running example, tensorized FAD is represented as $\mathcal{D}_\tau[\![\mathtt{sum(<i,a> in V2) V1(i) * a}]\!]$. By applying the rules in Figure 3 we have:

```
sum(<i,a> in V2)
  let <i',a'> = <0,V2'(i)> in
  V1(i) * a' + V1'(i) * a
```

Although this program looks identical to the version generated by scalar FAD, the types of V1' and V2' are different. In scalar FAD, their type is the same as V1 and V2, i.e., `tensor 1`. In tensorized FAD, their type is `tensor 2`. Thus, the * operators in the last expression are now scalar-vector multiplications.

As opposed to scalar FAD, we need to assemble the zero and one-hot vectors of all iterations together; instead of passing them vector by vector, we pass them as an entire matrix in which each row represents one of the one-hot vectors. The

```
gradient e (x: τ) ≜ let <v1',..., vn'> = <ingrad v1 x,...,ingrad vn x> in D_τ⟦e⟧
            where   {v1, ..., vn} = FV(e)
  ingrad (v: D) x ≜ zero[D]                          ingrad (x: D) x ≜ onehot[D] x
   onehot[real] x ≜ 1              onehot[tensor 1] x ≜ sum(<i,_> in x) {i->{i->1}}
onehot[tensor 2] x ≜ sum(<i,v> in x) { i -> sum(<j,_> in v) {j -> {i -> {j -> 1}}}}
onehot[tensor n] x ≜ sum(<i_1,v_2> in x) {i_1 ->
          if n > 2      ... sum(<i_n,_> in v_n)) {i_n -> {i_1 -> ... {i_n -> 1}...}
```

Fig. 4: The `gradient` API exposed by ∇SD to the programmer.

definition of `onehot[tensor 1]` in Figure 4 specifies how one can build such a matrix for variable `x`.

Example 3. Consider the case of computing the trace of the matrix `M`. The tensorized FAD over this expression in SDQLite is represented as $\mathcal{D}_\tau[\![$ `sum(<i,r> in M) r(i)` $]\!]$. For each row `r` of matrix `M` at index `i`, we compute the summation of diagonal elements specified by `r(i)`. After our tensorized FAD transformation, we obtain the following program:

```
sum(<i,r> in M)
  let <i',r'> = <0,M'(i)> in r'(i)
```

To compute the one-hot input for a matrix, we need to generalize the case of a vector; for differentiation w.r.t. a vector (`tensor 1`), we passed a matrix (`tensor 2`) for the one-hot inputs. Here, for differentiation w.r.t. a matrix (`tensor 2`), we need to pass an order-4 tensor (`tensor 4`). The definition of `onehot[tensor 2]` can also be found in Figure 4.

C. Putting it All Together

Programmers who use machine learning frameworks need not concern themselves with the source-to-source transformations employed in the background. To accomplish this, ∇SD provides a high-level API through the `gradient` macro, which accepts two inputs. The first input is the SDQLite expression to be differentiated, while the second input is the free variable with respect to which we perform the differentiation.

Consider the case of computing the gradient of the expression `e` with respect to `x`. This is represented as `gradient e x`. The `gradient` macro generates let-bindings for the differentiation components of all free variables of the expression `e` (FV(`e`)). The RHS of let-binding for a free variable `v` is `ingrad v x`. The `ingrad` macro is responsible for computing the input zeros or one-hots, depending on whether variable `v` is different than `x` or is the same. In the former case, the `zero` macro is used by passing the type of `v`. In the latter case, the `onehot` macro is used. For an input variable of type `tensor n`, the `onehot` assembles a sparse `tensor (2*n)` where the `n` indices of the input variable `x` are repeated twice so that the diagonals of the hypercube are set to `1`.

Example 1 (cont.). Our running example is specified by the high-level API as follows:

```
gradient (sum(<i, a> in V1) a * V2(i)) V2
```

After macro expansion, this program is transformed as follows:

```
let V1' = {} in
let V2' = sum(<i,_> in V2) {i -> {i -> 1}} in
sum(<i, a> in V1)
  let <i', a'> = <{}, V1'(i)> in
  a * V2'(i) + a' * V2(i)
```

In the next section, we see how this expression will be optimized using general AD-agnostic transformations. Note that by simple constant propagation, the asymptotic complexity will already be $O(n)$ in this example.

V. Compilation Pipeline

In this section, we present the compilation pipeline (cf. Figure 5) and review the techniques used to improve performance.

Equality Saturation. First, ∇SD applies algebraic rewrite rules over the input program to leverage optimizations such as factorization, loop fusion, etc [12]. We use equality saturation in order not to worry about phase ordering problems and making sure that the rewrite rules are applied globally [13]. We use EGG [14], a state-of-the-art implementation of equality saturation that has been successfully used for SDQLite in the context of flexible storage specification for tensor programs [12]. We rely on the cost models and the algebraic rewrite rules specified in [12]; there is no need to specify any AD-specific cost model or rewrite rule.

ANF. Then, ∇SD applies an ANF transformation [15], which ensures that sub-expressions are simple expressions, i.e. constant values or variable references. This is achieved by using a let-binding for the sub-expressions if they are not already simple expressions. The ANF transformed program is then fed into the differentiation transformation presented in Section IV.

Sparsity Propagation. After differentiation, many intermediate highly sparse values (e.g., zero tensors) are constructed. Even though expressing local rewrite rules for simplifying them is possible, these programs are very large and optimizing them with equality saturation requires a large search space. Thus, we applied these sparsity propagation rules as a separate pass. The first group of rewrite rules in Figure 6 correspond to the sparsity propagation rules. Finally, we again pass the program to equality saturation.

Example 1 (cont.). By assuming the second variant of vector-dot product (iteration over the second vector), the differentiated program with respect to the first vector is as follows:

```
let V1' = sum(<i,a> in V1) {i->{i->1}} in
let V2' = {} in
sum(<i, a> in V2)
  let <i', a'> = <{}, V2'(i)> in
  V1(i) * a' + V1'(i) * a
```

The sparsity propagation pass propagates `V2'` and simplifies the relevant expressions (e.g., `V1(i)*a'`) as follows:

```
let V1' = sum(<i,a> in V1) {i->{i->1}} in
sum(<i, a> in V2) V1'(i) * a
```

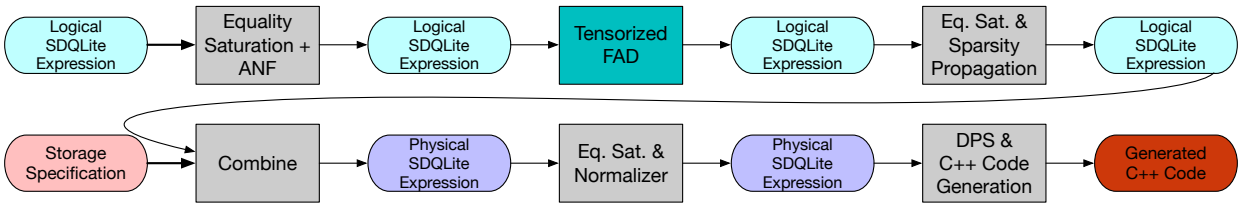


Fig. 5: The compilation pipeline in ∇SD.

Sparsity Propagation Rewrite Rules	
if e2: D2	`zero[D1] * e2` ⇝ `zero[D1 ⊗ D2]`
if e1: D1	`e1 * zero[D2]` ⇝ `zero[D1 ⊗ D2]`
if e1: D1	`e1 + zero[D1]` ⇝ `e1`
if e2: D1	`zero[D1] + e2` ⇝ `e2`
	`zero[tensor (n+1)](e2)` ⇝ `zero[tensor n]`
	`let x=zero[D] in e2` ⇝ `e2[x→zero[D]]`
	`sum(<k,v> in zero[D])e2` ⇝ `zero[D2]` if `e2:D2`

Multiplication Normalization Rewrite Rules	
if `e1:tensor n` and `n > 0`	`e1 * e2` ⇝ `sum(<i_1,v_2> in e1) ... sum(<i_n,v> in v_n)) {i_1->...{i_n-> v*e1}...}`
if `e1: real` & `e2:tensor n` & `n > 0`	`e1 * e2` ⇝ `sum(<i_1,v_2> in e2) ... sum(<i_n,v> in v_n)) {i_1->...{i_n-> e1*v}...}`

Fig. 6: The sparsity propagation and multiplication normalization transformations.

After applying equality saturation, if `V1` is a dense vector, the program is optimized and reduces to simply `V2`, a slightly less optimized version of which is:

```
sum(<i, a> in V2) { i -> a }
```

Storage Composition. After optimizing the differentiated expression, ∇SD uses the storage specifications as the definition for each input tensor. Figure 7 shows the specification of several storage formats in SDQLite. ∇SD generates let-bindings that bind every input tensor to the expression specifying its physical storage format. However, this makes the performance even worse, due to unnecessary intermediate tensors. Luckily, previous research [12] showed how equality saturation can not only remove these intermediate tensors but also recover well-known algorithms for sparse tensor computations.

Example 1 (cont.). In the previous example, if we assume that `V1` is a dense vector and `V2` is a sparse vector using the COO representation, combining the storage specification with the differentiated program results in:

```
let V1 = sum(<_,i> in 0:V1_n)
  {unique(i)->V1_V(i)} in
let V2 = sum(<_,i> in 0:V2_n)
  {unique(V2_R(i))->V2_V(i)} in
sum(<i, a> in V2) { i -> a }
```

By applying equality saturation, ∇SD returns the following program that does not introduce unnecessary intermediate tensors, as expected from [12]:

```
sum(<_,i> in 0:V2_n) { V2_R(i) -> V2_V(i) }
```

Normalization. ∇SD performs additional lower-level transformations on the optimized storage-format-aware program. First, multiplication normalization rewrites tensor outer products into summation expressions with scalar multiplications. The rewrite rules for multiplication normalization are shown in the second group of rules of Figure 6. Once combined with loop fusion rules, multiplication normalization removes unnecessary intermediate tensors. The second normalization involves ANF transformations which have been previously cancelled by equality saturation and other transformations.

Example 5 - BATAX. The BATAX kernel [20] is represented as $\beta A^T Ax$ where β is a scalar value, A a matrix, and x a vector. This kernel is expressed in SDQLite as follows:

```
sum(<i,r>in A) sum(<j,v1>in r)sum(<k,v2>in r)
 { j -> ((beta * v1) * v2) * (x(k)) }
```

After differentiating with respect to `x`, post-differentiation optimizations, ∇SD produces:

```
beta * (sum(<i, r> in A) r * r)
```

Considering a CSR representation for `A` and a dense representation for `x`, after applying storage composition and equality saturation we derive:

```
beta * sum(<_, i> in (0:n))
 let r =
  sum(<p,j> in I(P(i):P(i+1))) { j->V(p) } in
 r * r
```

The multiplication of `beta` with the result of summation is a scalar-vector product. Also, the last expression `r * r` corresponds to a vector-outer product. The multiplication normalization rewrites both expressions as follows:

```
sum(<_, i> in (0:n))
 let r =
  sum(<p,j> in I(P(i):P(i+1))) { j->V(p) } in
```

Storage Format (Inputs)	**Physical SDQLite Definition**
Vector COO `(n,R,V)`	`sum(<_, i> in 0:n) { unique(R(i)) -> V(i) }`
Vector Dense `(n,V)`	`sum(<_, i> in 0:n) { unique(i) -> V(i) }`
Matrix CSR `(n,P,I,V)`	`sum(<_,i> in 0:n) { unique(i)-> sum(<p,j> in I(P(i):P(i+1))) { unique(j) -> V(p) }}`
Matrix Dense Row Major `(n,m,M)`	`sum(<_,i> in 0:n) { unique(i)-> sum(<_, j> in 0:m) { unique(j) -> M(i*m+j) }}`

Fig. 7: The specification of various sparse/dense formats in Physical SDQLite. Matrix CSC and Dense Column Major are expressed similarly to Matrix CSR and Dense Row Major.

```
sum(<i1, v1> in r) { i1 ->
 sum(<i2, v2> in r) { i2 -> beta*v1*v2 } }
```

Discussion. The dominating time for compilation is equality saturation due to its search-based nature. The main reason for separating optimizations such as sparsity propagation from equality saturation is the scalability issues of this approach [12], which is a topic of active research [21].

Code Generation. As the final step, ∇SD generates C++ code. The conversion of types is summarized in the table below:

`int`	⇝	`size_t`
`real`	⇝	`double`
`{int->real}`	⇝	`dict_t<size_t,double>`
`{dense_int->real}`	⇝	`arr_t<double>`

Nested dictionaries are recursively translated. We use the robinhood dictionary [22] for the C++ runtime.

The C++ code generation for most constructs of SDQLite is straightforward. The construct `dict(key)` is translated into `dict[key]`, which corresponds to a dictionary or array lookup in C++. The addition and multiplication constructs are converted to the same primitives in C++. This requires the support for `+` and `*` over dictionary types, which is provided by our C++ runtime. The `sum` construct is translated into a for-loop. Similarly, nested summations are translated into nested for-loops. If the range expression of a summation is the range construction `(st:en)` or sub-array `arr(st:en)`, the translation produces a standard for-loop. However, if it is a dictionary, the translation generates a for-each construct over the dictionary. Another interesting case is the code generation for the singleton dictionary construct. This construct is mostly used inside a summation. The following SDQL code:

```
sum(<k,v> in dict) { f(k) -> g(v) }
```

is translated to a dictionary update as follows:

```
dict_t<size_t, double> res;
for(auto& kv : dict)
  res[f(kv.first)] += g(kv.second);
```

Note that the above translation works if `g(v)` outputs a dictionary. This is because `+=` is also overloaded by dictionaries.

Example 1 (cont.). The generated C++ code for the differentiated program is as follows:

```
  for(size_t i = 0; i < V2_n; i++)
    result[V2_R[i]] += V2_V[i];
```

Destination-Passing Style. In order to generate efficient C++, ∇SD leverages the destination-passing style (DPS) technique [23] in two ways. First, the generated function is provided with the destination object to store the final results [24]. Second, The DPS transformation removes intermediate tensors created in inner loops. This is achieved by pushing all the singleton dictionary constructions into the inner loops to avoid unnecessary intermediate dictionary constructions [25].

Example 4 (cont.). In the BATAX kernel, the C++ code generation produces the following program:

```
for(size_t i = 0; i < n; i++) {
  dict_t<size_t, double> r;
  for(size_t p=P[i]; p<P[i+1]; p++)
  { size_t j = I[p]; r[j] += V[p]; }
  dict_t<size_t,dict_t<size_t,double>> tmp1;
  for(auto& iv1 : r) {
    dict_t<size_t, double> tmp2;
    for(auto& iv2 : r)
      tmp2[iv2.first] +=
        beta_S * (iv1.second * iv2.second);
    tmp1[iv1.first] += tmp2;
  }
  result += tmp1;
}
```

Note that the last inner loop (highlighted) requires constructing two intermediate dictionaries `tmp1` and `tmp2`. One can use the associativity of semi-ring dictionaries in order to reorder the insertions so that all the insertions happen in the most-inner loop directly into `result`:

```
for(auto& iv1 : r)
  for(auto& iv2 : r)
    result[iv1.first][iv2.first] +=
      beta_S * (iv1.second * iv2.second);
```

This removes the allocation of the intermediate dictionaries, the impact of which is shown in the next section.

TABLE I: Real-world matrices used in the experiments.

Matrix	Dimensions	Density	# NNZ	Description
bcspwr10	5.3K × 5.3K	$2^{-9.3}$	22K	Power Network
nopoly	10K × 10K	$2^{-9.5}$	70K	Undir. Weighted Graph
pdb1HYS	36K × 36K	$2^{-8.2}$	2.19M	Undir. Weighted Graph
rma10	46K × 46K	$2^{-9.8}$	2.37M	Comput. Fluid Dynamics
cant	62K × 62K	$2^{-9.9}$	2.03M	Finite Element Method
consph	83K × 83K	$2^{-10.1}$	3.05M	Finite Element Method
cop20k_A	121K × 121K	$2^{-12.4}$	1.36M	Finite Element Method

VI. Experimental Results

In this section, we see the effectiveness of ∇SD in practice by using real-world and synthetic tensors. We answer the following research questions:

- How does ∇SD perform for tensor kernels over real-world sparse datasets in comparison with the state-of-the-art AD frameworks? Does ∇SD scale to large sparse matrices?
- What is the impact of different physical storage formats?
- What is the impact of different optimizations?
- For which densities does it make sense to use sparse representations and compute the gradients using ∇SD?

A. Experimental Setup

The experiments were conducted on a MacBook Pro featuring a dual-core Intel Core i7 CPU clocked at 3.5 GHz with 16 GB of LPDDR3 RAM (2133MHz), running macOS Big Sur 11.5.2. We employed CLang 1205.0.22.11 to compile the C++ code and Python 3.8.1 for executing the Python code. We used TensorFlow 2.11.0 (XLA-enabled) and PyTorch 1.13.1.

We consider the following sparse matrix kernels, where the first two produce a matrix, and the last one produces a vector:

- **BATAX.** The BATAX kernel computed using the following formula [20]: $f(j) = \Sigma_{i,k}\beta \cdot A(i,j) \cdot A(i,k) \cdot X(k)$. We consider its gradient with respect to the vector: $\frac{\partial f}{\partial X}$.
- **SMMM.** The summation of elements of matrix-matrix-multiplication [12]: $f = \Sigma_{i,j,k} A(i,k) \cdot B(k,j)$. We consider its gradient with respect to the second matrix: $\frac{\partial f}{\partial B}$.
- **SMVM.** The summation of the elements of a matrix-vector-multiplication [26]: $f = \Sigma_{i,j} A(i,j) \cdot X(j)$. We consider its gradient with respect to the vector: $\frac{\partial f}{\partial X}$.

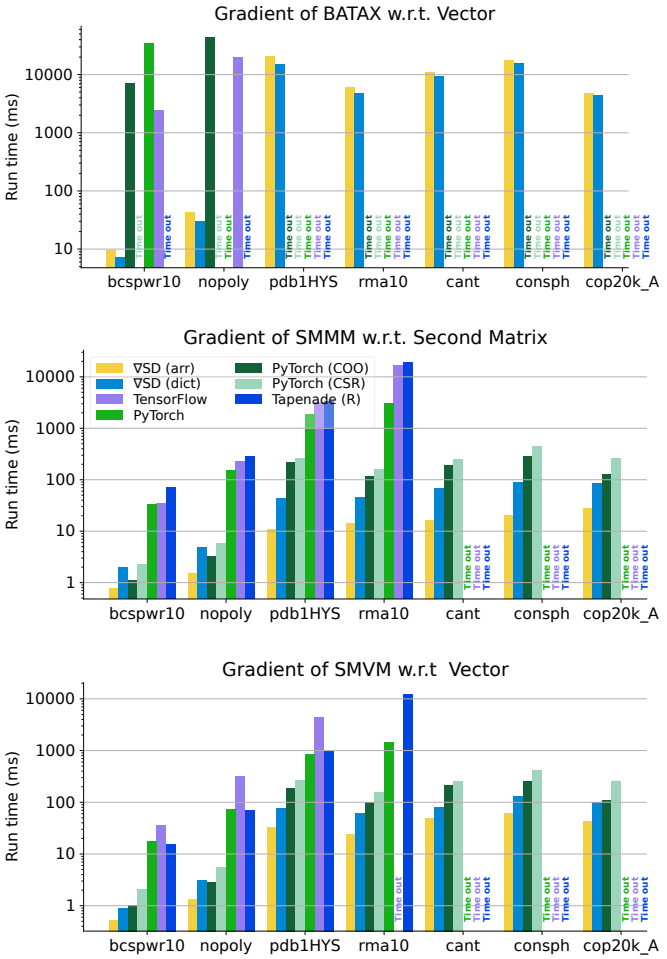


Fig. 8: Performance results for differentiation of kernels over real-world sparse matrices.

We also consider the following vector kernels from dF̃ [18], where the first kernel produces a matrix result, while the other two produce a vector:

- **VVA.** The addition of two vectors: $f(i) = V_1(i) + V_2(i)$. We consider its gradient with respect to the first vector: $\frac{\partial f}{\partial V_1}$.
- **VVD.** The dot product of two vectors: $f = \Sigma_i V_1(i) \cdot V_2(i)$. We consider its gradient with respect to the first vector: $\frac{\partial f}{\partial V_1}$.
- **VSM.** The vector-scalar multiplication: $f(i) = V(i) \cdot s \cdot s$. The scalar value is multiplied by itself. We consider its gradient with respect to the scalar value: $\frac{\partial f}{\partial s}$.

Most sparse tensor frameworks (e.g., TACO [2]) do not support AD. TensorFlow fails to perform AD for all our sparse kernels and PyTorch fails for two sparse vector kernels. As competitors for the matrix kernels, we consider:

- **∇SD (dict/arr)**: Generated C++ code by ∇SD from tensorized forward-mode AD that uses a nested dictionary/the physical storage specified by the sparse representations.
- **TensorFlow**: The reverse-mode AD of the TensorFlow framework, using its `gradient` and `jacobian` APIs.
- **PyTorch (/COO/CSR)**: The reverse-mode AD of PyTorch, using `jacobian` API with dense, COO, and CSR formats.

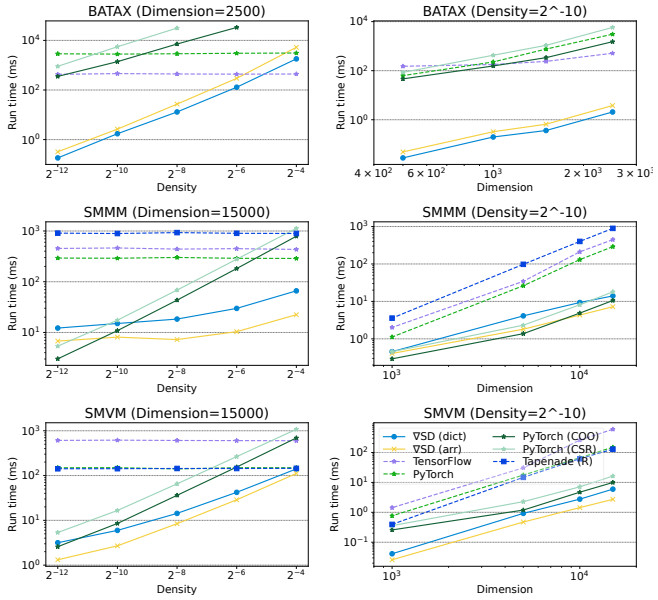


Fig. 9: Performance results for differentiation of different sparse matrix kernels by varying sparsity and dimensions.

- **Tapenade (R)**: Transformed C code by providing TACO-generated dense code to Tapenade using reverse-mode AD. Tapenade failed for TACO-generated sparse code.

For the vector kernels, in addition to the above competitors, we consider the following systems:

- **Tapenade (F)**: Generated C code by Tapenade with FAD.
- **dF̃/ dF̃ + DPS**: Generated C code by dF̃ using its vectorized forward-mode AD without/with the DPS optimization.

We used both real-world and synthetic datasets in our study. To obtain the former, we gathered seven sparse matrices from the SuiteSparse Matrix Collection [27]. Table I summarizes these datasets. For synthetic data, we created random vectors/-matrices with different density and dimension configurations.

In all these benchmarks, we consider the gradient with respect to a dense variable. All experiments use a single-core. We take the average time of running five runs.

B. Benchmarks over Real-world Datasets

We first consider real-world sparse matrices. We compare ∇SD with TensorFlow and PyTorch. Both systems support sparse operations, but none support AD over sparse matrices.

Figure 8 shows the results for the real-world sparse matrices. We make the following observations. First, for all kernels ∇SD outperforms all the dense competitors, thanks to leveraging sparse representations. Second, the generated code by ∇SD that leverages the array-based physical representation can run faster than the version that uses nested dictionaries in SMVM and SMMM. Finally, *∇SD scales to large sparse matrices*, as opposed to TensorFlow and the dense variant of PyTorch, which do not manage to process such matrices due to storing the entire matrix, including the zero elements. In addition, ∇SD performs better than the COO/CSR variants of PyTorch.

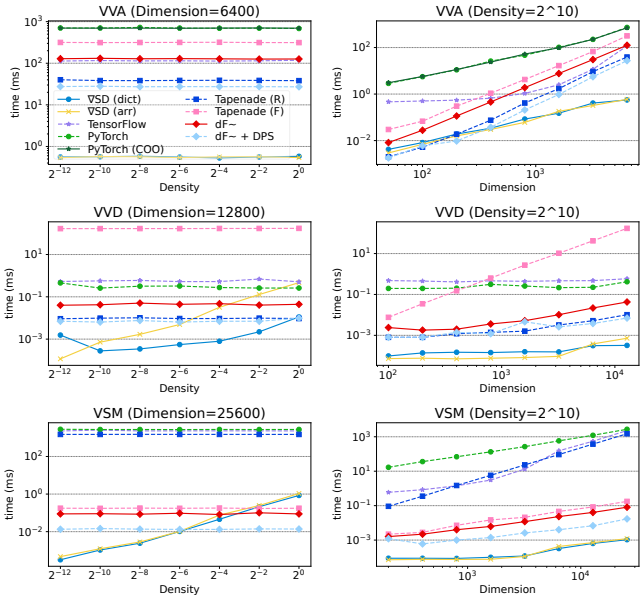


Fig. 10: Performance results for differentiation of different sparse vector kernels by varying sparsity and dimensions.

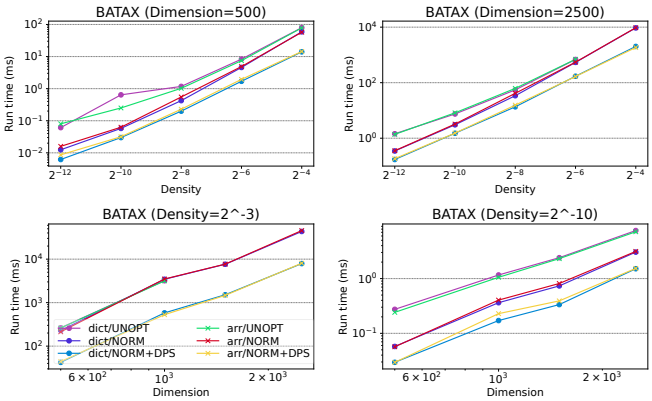


Fig. 11: Impact of the multiplication normalization (NORM) and DPS over the performance of ∇SD on the BATAX kernel.

C. Benchmarks over Synthetic Datasets

In this section, we use synthetic datasets to better analyse the impact of the sparsity and dimensions. We consider two scenarios: (1) fixing the dimension and varying the sparsity, and (2) fixing the sparsity and varying the dimension.

Matrix Kernels. Figure 9 shows the results for matrix kernels by varying the sparsity for small and large matrices. We make the following observations. First, for low densities, which are in the same range as the real-world datasets, we see a clear advantage for ∇SD over TensorFlow and PyTorch. Second, the performance of TensorFlow and PyTorch is not dependent on the sparsity, as expected. As the matrices get denser, the gap between these two frameworks and ∇SD becomes smaller. Finally, for the BATAX kernel, there is an advantage for the dictionary-based representation over the array-based one. However, for the other two kernels, especially for larger matrices, the array-based representation performs better.

Figure 9 additionally shows the results by varying the dimension for two different sparsities. Matrices with high densities show a better performance for TensorFlow and PyTorch over ∇SD. The gap even widens for larger dimensions. However, we observe the opposite impact for a lower density.

Vector Kernels. Figure 10 show the results for vector kernels by varying the sparsity and density. For smaller dimensions, there is no clear advantage for ∇SD; for VVA $\widetilde{\mathrm{dF}}$ performs better than ∇SD. However, for larger dimensions, we observe a clear advantage for ∇SD, especially for matrices with lower densities. For VSM, we observe a clear advantage for forward-mode-based systems over reverse-mode-based ones; for lower densities ∇SD outperforms the rest, whereas for higher densities $\widetilde{\mathrm{dF}}$ is the most performant system.

Impact of Optimizations. Finally, we investigate the impact of optimizations on the performance of ∇SD. For each data representation, we consider the following alternatives for the generated code: (1) without low-level transformations (UNOPT), (2) with multiplication normalization (NORM), and (3) with multiplication normalization and Destination-Passing Style for removing unnecessary intermediate tensors in nested loops (NORM+DPS).

Figure 11 shows the results for the BATAX kernel. Similar to the previous experiments, we see an advantage for the dictionary-based representation over the array-based one, due to the intermediate dictionary created in the array-based version (cf. Section V). Furthermore, the optimizations have a positive impact on performance, especially for lower densities.

VII. Related Work

Automatic Differentiation. There are several existing automatic differentiation (AD) frameworks and libraries for imperative and functional programming languages. ADIFOR [28] and Tapenade [29] perform AD for Fortran and C programs respectively, while Adept [30] and ADIC [31] perform AD for C++ using expression templates. ForwardDiff [32] uses vector forward-mode AD for differentiating Julia programs, while DiffSharp [33] is an AD library implemented in F# that provides both forward-mode and reverse-mode. Stalingrad [34] is an optimizing compiler for a dialect of Scheme with a first-class AD operator and supports both forward mode and reverse mode of AD. Similarly, Karczmarczuk [35] presents a Haskell implementation for both forward and reverse mode AD, and Elliott [36] provides a generalization of AD based on category theory for implementing both forward and reverse-mode AD. There has been recent efforts on providing correct and asymptotically efficient reverse-mode AD for functional languages [26], [37], [38], the ideas of which are implemented in JAX [10], [39] and Dex [19].

Machine learning libraries like Tensorflow and PyTorch are implemented based on tensor abstractions. These systems come with a predefined set of efficient kernels for manipulating tensors and can use compilation backends for further optimization.

Lantern [40] uses multi-stage programming to perform reverse-mode AD. However, the mentioned frameworks either lack the AD support for sparse tensors, or their support is limited [4]–[6]. There have been efforts on statically incorporating sparsities, however, this requires manually specifying the sparsity patterns by the programmers [41] and do not scale to large sparse matrices with arbitrary patterns [42].

Sparse Tensor Algebra. Sparse tensor algebra has been the focus of much research and development in recent years, leading to the emergence of several frameworks and systems. TACO [2], [43] is a system capable of handling both sparse and dense computations over tensor algebra. Another noteworthy framework is the sparse polyhedral framework [1], which extends the capabilities of polyhedral compilation to support sparse tensor algebra. In addition, packages such as SciPy [44], TensorFlow, PyTorch, and the MATLAB Tensor Toolbox [45] offer support for various sparse matrix representations, enabling efficient computation on sparse tensors.

Despite the progress made in this area, AD for sparse tensors is still not widely supported. For example, TACO [2] compiles high-level tensor algebra expressions to low-level C code without AD support. One of the few recent efforts [7] provides manual gradients for a limited set of kernels. The primary challenge in differentiating sparse tensors is the irregular data representation, making differentiation a complex process. To address this issue, we propose separating the logical sparse representation from its physical storage format, allowing for more efficient and effective differentiation.

VIII. Conclusion and Outlook

In this paper, we present ∇SD, the first AD framework for sparse tensors. Our main insight is to separate the logical concerns from the physical data storage representations. We provide a tensorized forward-mode transformation over the logical fragment of SDQLite, a language that fuses the physical storage of sparse tensors with the logical specification of kernels. We improve the performance by globally applying algebraic optimizations using equality saturation. We empirically show that our framework outperforms the state-of-the-art AD frameworks over both real-world and synthetic datasets.

Our tensorized forward-mode AD recovers the performance of reverse-mode by combining batching with loop optimizations [18]. This frees us from dealing with the challenges of reverse-mode AD (e.g., mutable states). For the future, we plan to add support for reverse-mode AD, using ideas similar to [38], [46]. We also plan to add the support for scheduling transforms to add the optimizations similar to TACO [2] and GPU backend [47]. This way we can use our framework for training deep, yet sparse learning models such as Graph Neural Networks (GNNs) [7]. Finally, we plan to add parallelism inspired by earlier work on semi-ring dictionaries [48].

Acknowledgements

The first author thanks Huawei for their support of the distributed data management and processing laboratory at the University of Edinburgh. The second author is supported by a Royal Society University Research Fellowship.

References

[1] M. M. Strout, M. W. Hall, and C. Olschanowsky, "The sparse polyhedral framework: Composing compiler-generated inspector-executor code," *Proc. IEEE*, vol. 106, no. 11, pp. 1921–1934, 2018. [Online]. Available: https://doi.org/10.1109/JPROC.2018.2857721

[2] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 77:1–77:29, Oct. 2017. [Online]. Available: http://doi.acm.org/10.1145/3133901

[3] X. Tang, T. Schneider, S. Kamil, A. Panda, J. Li, and D. Panozzo, "EGGS: sparsity-specific code generation," *Comput. Graph. Forum*, vol. 39, no. 5, pp. 209–219, 2020. [Online]. Available: https://doi.org/10.1111/cgf.14080

[4] TensorFlow, "Github repository issue #43497," https://github.com/tensorflow/tensorflow/issues/43497, 2023, accessed: 2023-01-30.

[5] JAX, "Github repository issue #13118," https://github.com/google/jax/issues/13118, 2023, accessed: 2023-01-30.

[6] PyTorch, "Github repository issue #12498," https://github.com/pytorch/pytorch/issues/12498, 2023, accessed: 2023-01-30.

[7] N. Nytko, A. Taghibakhshi, T. U. Zaman, S. MacLachlan, L. N. Olson, and M. West, "Optimized sparse matrix operations for reverse mode automatic differentiation," *arXiv preprint arXiv:2212.05159*, 2022.

[8] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[9] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035.

[10] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, "JAX: composable transformations of Python+NumPy programs," 2018. [Online]. Available: http://github.com/google/jax

[11] A. Shaikhha, M. Huot, J. Smith, and D. Olteanu, "Functional collection programming with semi-ring dictionaries," *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA1, pp. 1–33, 2022. [Online]. Available: https://doi.org/10.1145/3527333

[12] M. Schleich, A. Shaikhha, and D. Suciu, "Optimizing tensor programs on flexible storage," *Proc. ACM Manag. Data*, vol. 1, no. 1, pp. 37:1–37:27, 2023. [Online]. Available: https://doi.org/10.1145/3588717

[13] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner, "Equality saturation: a new approach to optimization," in *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2009, pp. 264–276.

[14] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha, "egg: Fast and extensible equality saturation," *Proc. ACM Program. Lang.*, vol. 5, no. POPL, pp. 1–29, 2021. [Online]. Available: https://doi.org/10.1145/3434304

[15] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen, "The essence of compiling with continuations," in *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, R. Cartwright, Ed. ACM, 1993, pp. 237–247. [Online]. Available: https://doi.org/10.1145/155090.155113

[16] J. J. Moré, "The levenberg-marquardt algorithm: implementation and theory," in *Numerical Analysis: Proceedings of the Biennial Conference Held at Dundee, June 28–July 1, 1977*. Springer, 2006, pp. 105–116.

[17] K. A. Khan and P. I. Barton, "A vector forward mode of automatic differentiation for generalized derivative evaluation," *Optim. Methods Softw.*, vol. 30, no. 6, pp. 1185–1212, 2015. [Online]. Available: https://doi.org/10.1080/10556788.2015.1025400

[18] A. Shaikhha, A. W. Fitzgibbon, D. Vytiniotis, and S. P. Jones, "Efficient differentiable programming in a functional array-processing language," *Proc. ACM Program. Lang.*, vol. 3, no. ICFP, pp. 97:1–97:30, 2019. [Online]. Available: https://doi.org/10.1145/3341701

[19] A. Paszke, D. D. Johnson, D. Duvenaud, D. Vytiniotis, A. Radul, M. J. Johnson, J. Ragan-Kelley, and D. Maclaurin, "Getting to the point: index sets and parallelism-preserving autodiff for pointful array programming," *Proceedings of the ACM on Programming Languages*, vol. 5, no. ICFP, pp. 1–29, 2021.

[20] T. Nelson, G. Belter, J. G. Siek, E. R. Jessup, and B. Norris, "Reliable generation of high-performance matrix algebra," *ACM Trans. Math. Softw.*, vol. 41, no. 3, pp. 18:1–18:27, 2015. [Online]. Available: https://doi.org/10.1145/2629698

[21] T. Koehler, P. Trinder, and M. Steuwer, "Sketch-guided equality saturation: Scaling equality saturation to complex optimizations of functional programs," *arXiv preprint arXiv:2111.13040*, 2021.

[22] "Fast & memory efficient hashtable based on robin hood hashing for c++," 2023, https://github.com/martinus/robin-hood-hashing.

[23] Y. Minamide, "A functional representation of data structures with a hole," in *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1998, pp. 75–84.

[24] A. Shaikhha, A. Fitzgibbon, S. Peyton Jones, and D. Vytiniotis, "Destination-passing style for efficient memory management," in *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing*, 2017, pp. 12–23.

[25] A. Shaikhha, "Deep fusion for efficient nested recursive computations," in *Proceedings of the 21st ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, 2022, pp. 33–44.

[26] T. Smeding and M. Vákár, "Efficient dual-numbers reverse AD via well-known program transformations," *Proc. ACM Program. Lang.*, vol. 7, no. POPL, pp. 1573–1600, 2023. [Online]. Available: https://doi.org/10.1145/3571247

[27] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, 2011. [Online]. Available: https://doi.org/10.1145/2049662.2049663

[28] C. H. Bischof, A. Carle, G. F. Corliss, A. Griewank, and P. D. Hovland, "ADIFOR - generating derivative codes form fortran programs," *Sci. Program.*, vol. 1, no. 1, pp. 11–29, 1992. [Online]. Available: https://doi.org/10.1155/1992/717832

[29] L. Hascoët and V. Pascual, "The tapenade automatic differentiation tool: Principles, model, and specification," *ACM Trans. Math. Softw.*, vol. 39, no. 3, pp. 20:1–20:43, 2013. [Online]. Available: https://doi.org/10.1145/2450153.2450158

[30] R. J. Hogan, "Fast reverse-mode automatic differentiation using expression templates in C++," *ACM Trans. Math. Softw.*, vol. 40, no. 4, pp. 26:1–26:16, 2014. [Online]. Available: https://doi.org/10.1145/2560359

[31] S. H. K. Narayanan, B. Norris, and B. Winnicka, "ADIC2: development of a component source transformation system for differentiating C and C++," in *Proceedings of the International Conference on Computational Science, ICCS 2010, University of Amsterdam, The Netherlands, May 31 - June 2, 2010*, ser. Procedia Computer Science, P. M. A. Sloot, G. D. van Albada, and J. J. Dongarra, Eds., vol. 1, no. 1. Elsevier, 2010, pp. 1845–1853. [Online]. Available: https://doi.org/10.1016/j.procs.2010.04.206

[32] J. Revels, M. Lubin, and T. Papamarkou, "Forward-mode automatic differentiation in julia," *CoRR*, vol. abs/1607.07892, 2016. [Online]. Available: http://arxiv.org/abs/1607.07892

[33] A. G. Baydin, B. A. Pearlmutter, and J. M. Siskind, "Diffsharp: Automatic differentiation library," *CoRR*, vol. abs/1511.07727, 2015. [Online]. Available: http://arxiv.org/abs/1511.07727

[34] B. A. Pearlmutter and J. M. Siskind, "Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator," *ACM Trans. Program. Lang. Syst.*, vol. 30, no. 2, pp. 7:1–7:36, 2008. [Online]. Available: https://doi.org/10.1145/1330017.1330018

[35] J. Karczmarczuk, "Functional differentiation of computer programs," in *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27-29, 1998*, M. Felleisen, P. Hudak, and C. Queinnec, Eds. ACM, 1998, pp. 195–203. [Online]. Available: https://doi.org/10.1145/289423.289442

[36] C. Elliott, "The simple essence of automatic differentiation," *Proc. ACM Program. Lang.*, vol. 2, no. ICFP, pp. 70:1–70:29, 2018. [Online]. Available: https://doi.org/10.1145/3236765

[37] F. Krawiec, S. P. Jones, N. Krishnaswami, T. Ellis, R. A. Eisenberg, and A. W. Fitzgibbon, "Provably correct, asymptotically efficient, higher-order reverse-mode automatic differentiation," *Proc. ACM Program. Lang.*, vol. 6, no. POPL, pp. 1–30, 2022. [Online]. Available: https://doi.org/10.1145/3498710

[38] A. Radul, A. Paszke, R. Frostig, M. J. Johnson, and D. Maclaurin, "You only linearize once: Tangents transpose to gradients," *Proc. ACM Program. Lang.*, vol. 7, no. POPL, pp. 1246–1274, 2023. [Online]. Available: https://doi.org/10.1145/3571236

[39] R. Frostig, M. J. Johnson, and C. Leary, "Compiling machine learning programs via high-level tracing," *Systems for Machine Learning*, vol. 4, no. 9, 2018.

[40] F. Wang, D. Zheng, J. M. Decker, X. Wu, G. M. Essertel, and T. Rompf, "Demystifying differentiable programming: shift/reset the penultimate backpropagator," *Proc. ACM Program. Lang.*, vol. 3, no. ICFP, pp. 96:1–96:31, 2019. [Online]. Available: https://doi.org/10.1145/3341700

[41] M. J. Peng and C. Dubach, "Lagrad: Statically optimized differentiable programming in mlir," in *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction*, ser. CC 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 228–238. [Online]. Available: https://doi.org/10.1145/3578360.3580259

[42] M. Ghorbani, M. Huot, S. Hashemian, and A. Shaikhha, "Compiling structured tensor algebra," *arXiv preprint arXiv:2211.10482*, 2022.

[43] S. Chou, F. Kjolstad, and S. P. Amarasinghe, "Format abstraction for sparse tensor algebra compilers," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 123:1–123:30, 2018. [Online]. Available: https://doi.org/10.1145/3276493

[44] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, I. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy, "Scipy 1.0-fundamental algorithms for scientific computing in python," *CoRR*, vol. abs/1907.10121, 2019. [Online]. Available: http://arxiv.org/abs/1907.10121

[45] B. W. Bader and T. G. Kolda, "Efficient MATLAB computations with sparse and factored tensors," *SIAM J. Sci. Comput.*, vol. 30, no. 1, pp. 205–231, 2007. [Online]. Available: https://doi.org/10.1137/060676489

[46] B. v. d. Berg, T. Schrijvers, J. McKinna, and A. Vandenbroucke, "Forward- or reverse-mode automatic differentiation: What's the difference?" *arXiv preprint arXiv:2212.11088*, 2022.

[47] R. Senanayake, C. Hong, Z. Wang, A. Wilson, S. Chou, S. Kamil, S. Amarasinghe, and F. Kjolstad, "A sparse iteration space transformation framework for sparse tensor algebra," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Nov. 2020. [Online]. Available: https://doi.org/10.1145/3428226

[48] H. Shahrokhi and A. Shaikhha, "Building a compiled query engine in python," in *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction, CC 2023, Montréal, QC, Canada, February 25-26, 2023*, C. Verbrugge, O. Lhoták, and X. Shen, Eds. ACM, 2023, pp. 180–190. [Online]. Available: https://doi.org/10.1145/3578360.3580264