Sparse BLAS Working Group:
On the path to defining a standard for sparse BLAS operations

Hartwig Anzt
TU Munich & Innovative Computing Lab, University of Tennessee

Ahmad Abdelfattah, Willow Ahrens, Hartwig Anzt, Chris Armstrong, Ben Brock, Aydin Buluc, Federico Busato, Terry Cojean, Tim Davis, Jim Demmel, Grace Dinh, David Gardener, Jan Fiala, Mark Gates, Azzam Haider, Toshiyuki Imamura, Pedro Valero Lara, Jose Moreira, Sherry Li, Neil Linquist, Piotr Luszczek, Max Melichenko, Yvan Mokwinski, Riley Murray, Spencer Patty, Slaven Peles, Tobias Ribizel, Jason Riedy, Siva Rajamanickam, Piyush Sao, Manu Shantharam, Keita Teranishi, Stan Tomov, Yu-Hsiang Tsai, Heiko Weichelt

Why are we doing this? Do we need it? How is it different from dense BLAS?

Dense linear algebra

$$\begin{pmatrix} * & * & \cdots & * \\ * & * & \cdots & * \\ \vdots & \vdots & \ddots & \vdots \\ * & * & \cdots & * \end{pmatrix} \begin{pmatrix} * \\ * \\ * \\ \vdots \\ * \end{pmatrix}$$

Sparse linear algebra

$$\begin{pmatrix} * & * & \cdots & * \\ * & * & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ * & 0 & \cdots & * \end{pmatrix} \begin{pmatrix} 0 \\ * \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

Why are we doing this? Do we need it? How is it different from dense BLAS?

Dense linear algebra

$$\begin{pmatrix} * & * & \cdots & * \\ * & * & \cdots & * \\ \vdots & \vdots & \ddots & \vdots \\ * & * & \cdots & * \end{pmatrix} \begin{pmatrix} * \\ * \\ * \\ \vdots \\ * \end{pmatrix}$$

- **All entries are stored explicitly**

Sparse linear algebra

$$\begin{pmatrix} * & * & \cdots & * \\ * & * & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ * & 0 & \cdots & * \end{pmatrix} \begin{pmatrix} 0 \\ * \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$
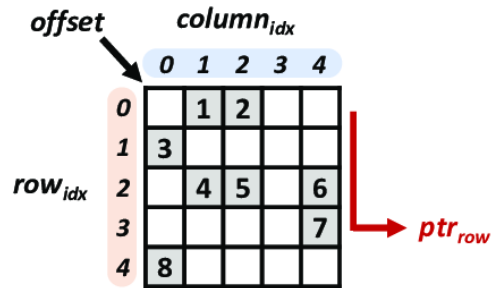
- Only nonzero values are stored explicitly
  (and sometimes also zeros for performance reasons
  or because the matrix changes over time and a zero
  becomes nonzero)

Why are we doing this? Do we need it? How is it different from dense BLAS?

Dense linear algebra

- All entries are stored explicitly
- **One standard format (excpt. for row/col major)**

Sparse linear algebra

- Only nonzero values are stored explicitly
- Different formats: COO, CSR, ELL, DIA...
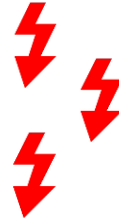

ELL Format

DIA Format

COO Format

CSR Format

Why are we doing this? Do we need it? How is it different from dense BLAS?

Dense linear algebra

$$
\begin{pmatrix}
* & * & \cdots & * \\
* & * & \cdots & * \\
\vdots & \vdots & \ddots & \vdots \\
* & * & \cdots & *
\end{pmatrix}
\begin{pmatrix}
* \\
* \\
* \\
\vdots \\
*
\end{pmatrix}
$$

- All entries are stored explicitly
- One standard format (excpt. for row/col major)
- **Elements are stored in sorted order**

Sparse linear algebra

$$
\begin{pmatrix}
* & * & \cdots & * \\
* & * & \cdots & 0 \\
\vdots & \vdots & \ddots & \vdots \\
* & 0 & \cdots & *
\end{pmatrix}
\begin{pmatrix}
0 \\
* \\
0 \\
\vdots \\
0
\end{pmatrix}
$$

- Only nonzero values are stored explicitly
- Different formats: COO, CSR, ELL, DIA…
- Not always sorted
  E.g. when elements are added in COO,
  or if diagonal elements are stored first, etc.

Why are we doing this? Do we need it? How is it different from dense BLAS?

Dense linear algebra

- All entries are stored explicitly
- One standard format (excpt. for row/col major)
- Elements are stored in sorted order
- **Vectors and matrices are conceptually the same**

Sparse linear algebra

- Only nonzero values are stored explicitly
- Different formats: COO, CSR, ELL, DIA...
- Not always sorted
- Vectors use a different storage format

Sparse Vector

0	12	0	24	0	0	9

Index   0   1   2   3   4   5   6

↓

Dense Vector | 1 | 12 | → | 3 | 24 | → | 6 | 9 |

Index   Value

Why are we doing this? Do we need it? How is it different from dense BLAS?

Dense linear algebra

- All entries are stored explicitly
- One standard format (excpt. for row/col major)
- Elements are stored in sorted order
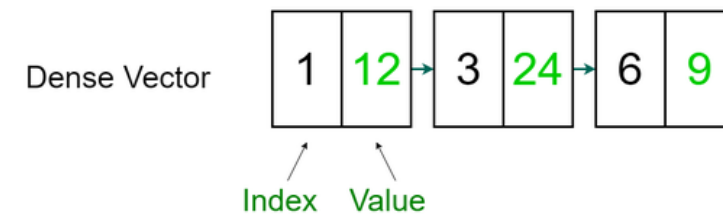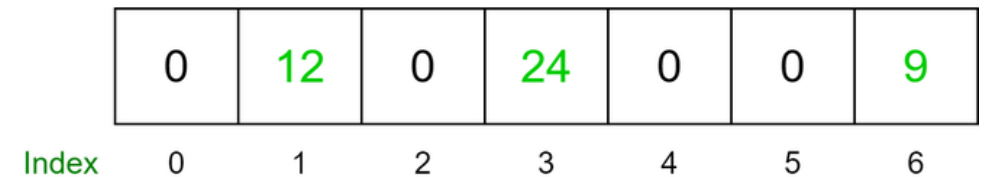- Vectors and matrices are conceptually the same
- **Size of output is known upfront -> single kernel**

Sparse linear algebra

- Only nonzero values are stored explicitly
- Different formats: COO, CSR, ELL, DIA...
- Not always sorted
- Vectors use a different storage format
- Generally, the memory requirement is not known before (e.g. sparse matrix multiply)
  1) symbolic phase
  2) memory allocation
  3) numeric phase

Why are we doing this? Do we need it? How is it different from dense BLAS?

$$\begin{pmatrix} * & * & \cdots & * \\ * & * & \cdots & * \\ \vdots & \vdots & \ddots & \vdots \\ * & * & \cdots & * \end{pmatrix} \begin{pmatrix} * \\ * \\ * \\ \vdots \\ * \end{pmatrix}$$

$$\begin{pmatrix} * & * & \cdots & * \\ * & * & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ * & 0 & \cdots & * \end{pmatrix} \begin{pmatrix} 0 \\ * \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

Dense linear algebra

- All entries are stored explicitly
- One standard format (excpt. for row/col major)
- Elements are stored in sorted order
- Vectors and matrices are conceptually the same
- Size of output is known upfront -> single kernel
- **Fortran/C interface adopted by most vendors**

Sparse linear algebra

- Only nonzero values are stored explicitly
- Different formats: COO, CSR, ELL, DIA…
- Not always sorted
- Vectors use a different storage format
- Generally, the memory requirement is not known before
- Vendors increasingly use a C++ interface, no agreement

Plethora of options makes a C-interface "impossible"

Why are we doing this? Do we need it? How is it different from dense BLAS?

Dense linear algebra

- All entries are stored explicitly
- One standard format (excpt. for row/col major)
- Elements are stored in sorted order
- Vectors and matrices are conceptually the same
- Size of output is known upfront -> single kernel
- Fortran/C interface adopted by most vendors
- **Exceptions are clearly handled**

Sparse linear algebra

- Only nonzero values are stored explicitly
- Different formats: COO, CSR, ELL, DIA…
- Not always sorted
- Vectors use a different storage format
- Generally, the memory requirement is not known before
- Vendors increasingly use a C++ interface, no agreement
- No consistent handling of *NaN/Inf*:
  What if an implicit matrix zero is multiplied with a *NaN*?
  Checking for *NaN* in the vector introduces overhead

The working group

Cross-Institutional working group, loose collaboration, not everyone attends every meeting, workshops, SC 2024 BoF;
(Vendor) library developers, Application specialists, standard enthusiasts;
Weekly virtual meetings, Sparse BLAS design proposal is working document;

1st Sparse BLAS workshop, 2023

2nd Sparse BLAS workshop, 2024

Intel Corporation
NVIDIA
AMD
Arm
MathWorks
University of California, Berkeley
Intel Labs
ORNL,
Karlsruhe Institute of Technology
LLNL
Sandia National Laboratories
MIT

The working group

Cross-Institutional working group, loose collaboration, not everyone [...]
(Vendor) library developers, Application specialists, standard enthus[...]
Weekly virtual meetings, Sparse BLAS design proposal is working do[...]

1st Sparse BLAS workshop, 2023

2nd Sparse BLAS work[...]

Interface for Sparse Linear Algebra Operations

September 19, 2024

Contents

But there already is a sparse BLAS standard?

Sparse extensions to the FORTRAN Basic Linear Algebra Subprograms

Authors: David S. Dodson, Roger G. Grimes, John G. Lewis | Authors Info & Claims

ACM Transactions on Mathematical Software (TOMS), Volume 17, Issue 2 • Pages 253 - 263 • https://doi.org/10.1145/108556.108577

Published: 01 June 1991 Publication History

Check for updates

An overview of the sparse basic linear algebra subprograms: The new standard from the BLAS technical forum

Authors: Iain S. Duff, Michael A. Heroux, Roldan Pozo | Authors Info & Claims

ACM Transactions on Mathematical Software (TOMS), Volume 28, Issue 2 • Pages 239 - 267 • https://doi.org/10.1145/567806.567810

Published: 01 June 2002 Publica

RESEARCH-ARTICLE

X in 🔴 f ✉

Object-Oriented Techniques for Sparse Matrix Computations in Fortran 2003

Authors: Salvatore Filippone, Alfredo Buttari | Authors Info & Claims

ACM Transactions on Mathematical Software (TOMS), Volume 38, Issue 4 • Article No.: 23, Pages 1 - 20
https://doi.org/10.1145/2331130.2331131

Published: 01 August 2012 Publication History

Check for updates

- Not only one…
- Designed in a pre-GPU era
- Not adopted by GPU libraries
- AI needs other functionality
- New C++ standard features

Should libraries own the sparse objects or a non-owning interface?

- Owning allows for more optimization as the opaque object can use any storage format

- Owning makes it impossible for the user to easily access the data or modify the data

- Packing/Unpacking routines introduce overhead – maybe error prone?

Owning or non-owning?

- Owning allows for more optimization as the opaque object can use any storage format

- Owning makes it impossible for the user to easily access the data or modify the data

- Packing/Unpacking routines introduce overhead – maybe error prone?

Most use cases want to apply operation to application-owned data.

--> We decide to go for light-weight "views" of the data

Light-weight views with additional data?

- Owning allows for more optimization as the opaque object can use any storage format

- Owning makes it impossible for the user to easily access the data or modify the data

- Packing/Unpacking routines introduce overhead – maybe error prone?

Most use cases want to apply operation to application-owned data.

        --> We decide to go for light-weight "views" of the data
            - Very limited optimization potential
            - Where can we store additional matrix information?
            - Create a copy?

            --> We add a "bag" to the "view" for additional data

Light-weight views with additional data?

- Owning allows for more optimization as the opaque object can use any storage format

- Owning makes it impossible for the user to easily access the data or modify the data

- Packing/Unpacking routines introduce overhead – maybe error prone?

Most use cases want to apply operation to application-owned data.

       --> We decide to go for light-weight "views" of the data
          - Very limited optimization potential
          - Where can we store additional matrix information?
          - Create a copy?

           --> We add a "bag" to the "view" for additional data
                – `matrixHandle`
              - How should the space for additional data be allocated?

Library-internal data allocation

- Some applications want to control all GPU memory with their `allocator` (Matlab, etc.)

- We need the possibility to pass this `allocator` to the library

- If no `allocator` is passed to the library, the standard allocator is used

```
using namespace sparseblas;
csr_view<float> A_view(values, rowptr, colind, shape, nnz);
// create a handle to hold some internal stuff more than view
matrix_handle A(A_view, allocator);
```

Which data format?

- We focus on the most universal and popular formats: CSR, CSC, COO

- We handle dense matrices/vectors as C++20 mdspan

Type	Name	Description
index_type	nrows	Number of rows of the matrix
index_type	ncols	Number of columns of the matrix
offset_type	nnz	Structural number of non-zeros
offset_type array	rowptr	Row pointer array (length: nrows+1)
index_type array	colindx	Column indices array (length: nnz)
scalar_type array	values	Structural values (length: nnz)
base_type	index_base	Base indexing

Table 2: CSR input parameters

Type	Name	Description
index_type	nrows	Number of rows of the matrix
index_type	ncols	Number of columns of the matrix
offset_type	nnz	Structural number of non-zeros
index_type array	rowindx	Row index array (length: nnz)
index_type array	colindx	Column index array (length: nnz)
scalar_type array	values	Structural values (length: nnz)
base_type	index_base	Base indexing

Table 4: COO input parameters

Which operations?

Operation	Notation
Scaling	$A := \alpha A$
Transpose and Conjugate Transpose	$A := A^T, A := A^H$
Infinity Matrix Norm	$\alpha := \|A\|_{\text{inf}}$
Frobenius Matrix Norm	$\alpha := \|A\|_F$
Element-wise Multiplication	$C := A \,.{*}\, B$
Sparse Matrix – Sparse Matrix Addition	$C := A + B$
Sparse Matrix – Sparse Matrix Multiplication	$C := A \cdot B + D$
Sparse Matrix – Dense Matrix Multiplication	$Y := A \cdot X + Y$
Sparse Matrix – Dense Vector Multiplication	$y = A \cdot x$
Triangular Solve	Solve $A \cdot x = y$ for x
Sparse Matrix Format Conversion	$B = \text{sparse}(A)$
Predicate Selection	$B = A(\text{predicate})$
Sampled dense dense matrix multiplication	$C\langle\text{mask}\rangle = A \cdot B$

Table 1: List of the Sparse BLAS functionalities. Uppercase letters represent matrices; lowercase represent vectors. Letters from the start of the alphabet are used for sparse matrices/vectors; letters from the end are used for dense matrices/vectors. α is a scalar value. Note that scaling may be applied to any input matrix/vector, and the transpose/conjugate transpose operation may be applied to any input matrix.

What if an operation needs temporary buffers?

- An operation takes a `state` object *t*hat can contain temporary buffers

- Some applications want to control all GPU memory with their `allocator` (Matlab, etc.)

- We need the possibility to pass this `allocator` to the library

- If no `allocator` is passed to the library, the standard allocator is used – also in the state object

```
using namespace sparseblas;
csr_view<float> A(values, rowptr, colind, shape, nnz);

// scale() overwrites the values of A
scale_state_t state(allocator);
scale(policy, state, 2.3, A);
```

Listing 1: Scaling, $A := \alpha A$

What is the execution policy?

- *Handles the execution*

  (like a ROCm/CUDA stream, sycl queue)

- *Sequential / parallel*

- *Synchronous / asynchronous*

Preserving explicit zeros?

Storing zeros explicitly can have advantages:

- Block-sparse matrix: sparse matrix with dense blocks (excluded for now?)

- Matrix entries change over the application time, zeros become nonzero

- Better performance through padding (cache line, coalesced access…)

- What if we convert from one format to another? Do we preserve the explicit zeros?

    - Current discussion: yes, conversion preserves zeros, except for conversion from dense.

How about NaN propagation?

$$\begin{pmatrix} 0 & * & \cdots & * \\ 0 & * & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & * \end{pmatrix} \begin{pmatrix} NaN \\ * \\ \vdots \\ * \end{pmatrix} = ?$$

- Checking right-hand-side for *NaN/Inf* by default is too expensive

- This leads to inconsistency:

    a format storing an explicit zero will get a different result than a format storing an implicit zero

- Plan for different execution modes:

    - Permissive (as fast as possible)

    - Consistent (check for NaN/Inf)

    - Reproducible (bitwise reproducibility)

One-stage? Two-stage? Three-stage? Any consensus?

1. *inspect phase* – (optional) prepare any potential optimizations for subsequent phases, or may do nothing

2. *compute phase* – computing the size of the sparse output structure, which typically requires significant work

3. *allocation phase* – allocating the memory for the sparse output data structure and placing it in the output matrix object to be filled

4. *fill phase* – complete execution of the operation and filling the output structure with the result.

One-stage? Two-stage? Three-stage? Any consensus?

```cpp
using namespace sparseblas;
csr_view<float> A(values, rowptr, colind, shape, nnz);

// scale() overwrites the values of A
scale_state_t state(allocator);
scale(policy, state, 2.3, A);
```

Listing 1: Scaling, $A := \alpha A$

One-stage? Two-stage? Three-stage? Any consensus?

```cpp
using namespace sparseblas;
csr_view<float> A(values, rowptr, colind, shape, nnz);
auto X = std::mdspan(raw_x.data(), k, n);
auto Y = std::mdspan(raw_y.data(), m, n);

float alpha = 1.1, beta = 1.2;

multiply_state_t state(allocator);
multiply_inspect(policy, state, scaled(alpha, A), X, beta, Y)); // optional
multiply_compute(policy, bind_info{scaled(alpha, A), info}, X, beta, Y);
multiply_fill(policy, bind_info{scaled(alpha, A), info}, X, beta, Y);
```

Listing 9: Sparse Matrix – Dense Matrix Multiplication, $Y = \alpha A \cdot X + \beta Y$

One-stage? Two-stage? Three-stage? Any consensus?

```cpp
using namespace sparseblas;
csr_view<float> A_view(values, rowptr, colind, shape, nnz);
// create a handle to hold some internal stuff more than view
matrix_handle A(A_view, allocator);
// ... likewise for B, D
csr_view<float> C(m, n);

multiply_state_t state(allocator);
multiply_inspect(policy, state, transposed(A),
                        B, C, D); // optional
// it will store the transposed A in the matrix_handle
multiply_symbolic_compute(policy, state, transposed(A),
                        B, C, D);
index_t nnz = state.get_result_nnz();
// allocate C rowptr/colind arrays and put in C
multiply_symbolic_fill(policy, state, scaled(alpha, transposed(A)),
                        B, C, scaled(beta, D));
// C structure is now able to be used

// ...
multiply_numeric_compute(policy, state, transposed(A),
                        B, C, D);
index_t nnz = state.get_result_nnz();
// allocate C values arrays and put in C  (Note: this can also be done previously
// since nnz is already known from symbolic stage)
multiply_numeric_fill(policy, state, scaled(alpha, transposed(A)),
                        B, C, scaled(beta, D));
// C structure and values are now able to be used
```

Can be combined

Listing 8: Sparse Matrix – Sparse Matrix Multiplication with matrix_handle, $C = \alpha \cdot A^T \cdot B + \beta D$ with separate symbolic and numeric stages

How about out-of-place submatrices?

```cpp
using namespace sparseblas;
csr_view<float> A_view(values, rowptr, colind, shape, nnz);
csr_view<float> B(values, rowptr, colind, shape, nnz);

auto x = std::mdspan(raw_x.data(), n);
auto b = std::mdspan(raw_b.data(), m);

auto pred = [](auto i, auto j, auto v) {
        return v > 0;
    };

auto pred = [](auto i, auto j, auto v) {
        return (i < 10 ) && (j < 10);
    };

filter_state_t state(allocator);

// matrix_handle is opaque and may contain vendor optimization details
matrix_handle A(A_view, allocator);
filter_compute(policy, state, A, B, pred);
// allocate arrays for B
filter_fill(policy, state, A, B, pred);
```
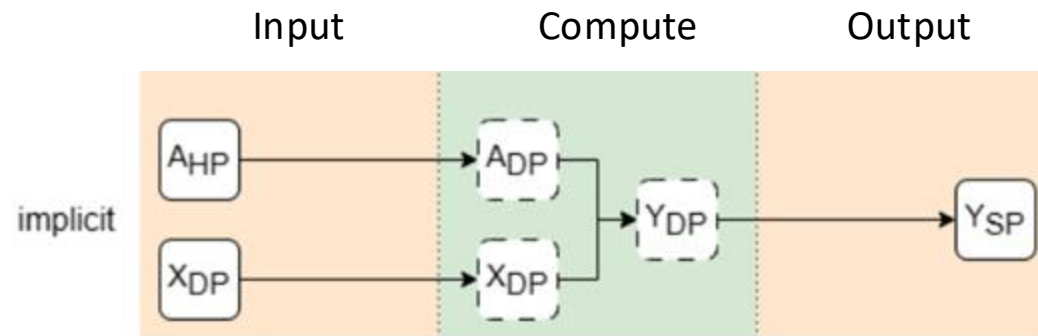
Extract submatrix of non-negative values

Extract first left-top 10x10 submatrix

Which precisions are relevant?

- Fp64, fp32, fp16, bf16 (+ complex variants)

- Mixed Precision (different inputs / outputs)

- Arithmetic precision is always the highest input/output precision (no performance loss)

- Large number of routine variants -> C++ overloading

C++ may be doable, but what about C and Fortran?

- Combinatorial complexity make a Fortran/C interface infeasible.

- Maybe an interface for the most relevant combinations?

- Vendors prefer using internally a C++ interface for their sparse routine interfaces.

Aren't the vendors already doing it anyway?

- Yes... and they are all going in similar directions – it is only a question of moderating the process...

oneMKL SYCL C++ API with an out-of-order queue (has an optional optimize stage)

```
ev_gemv = oneapi::mkl::sparse::gemv(
        queue, oneapi::mkl::transpose::nontrans,
        alpha, csrA, x, beta, y, {ev_opt});
```

hipSPARSE ROCm/CUDA C API

```
status = hipsparseDcsrmv(handle,
    HIPSPARSE_OPERATION_NON_TRANSPOSE,
        m, n, nnz, p_alpha, descrA,
        valA, rowptrA, colindA, x, p_beta, y);
```

cuSPARSE CUDA C, generic API (has multiple required stages)

```
status = cusparseSpMV(handle, CUSPARSE_OPERATION_NON_TRANSPOSE,
        p_alpha, matA, vecX, p_beta, vecY,
        CUDA_R_64F, CUSPARSE_SPMV_ALG_DEFAULT,
        externalBuffer);
```

Anyway, what is a standard?

- A standard is a standard because a significant portion of the community follows the same convention;

- We discuss how we do things in the different NLA libraries, and what the customers want;

- We try to find an interface that combines the good ideas;

- Ideally, we have vertical compatibility: efforts can adopt the interface we design w/o significant changes;

- We hope for horizontal compatibility (Fortran, C) but acknowledge this will be limited;

Anyway, what is a standard?

- A standard is a standard because a significant portion of the community follows the same convention;

- We discuss how we do things in the different NLA libraries, and what the customers want;

- We try to find an interface that combines the good ideas;

- Ideally, we have vertical compatibility: efforts can adopt the interface we design w/o significant changes;

- We hope for horizontal compatibility (Fortran, C) but acknowledge this will be limited;

Next step: present the interface design at a BoF at SC and request feedback from the broader community.

Working Toward an Interface for Sparse BLAS

Description: While sparse matrix computations are at the heart of many scientific and engineering applications, there exists no widely adopted interface standard. A reason for this may be the plethora of optimization options relevant to today's accelerator architectures. At the same time, many vendors already provide support for sparse matrix computations in proprietary libraries, but due to diverging architectural constraints, these libraries have different execution models, APIs, and formats supported. We started a cross-institutional effort involving academia and industry to define an API for sparse linear algebra operations. In the BoF, we present a blueprint and discuss considerations motivating design choices.

Event Type: Birds of a Feather

+ Add to Schedule

Time:
Thursday, 21 November 2024
12:15pm - 1:15pm EST

Location: B207

Registration Categories:
TP   XO/EX

Links:
Website