Segments: A Reusable Abstraction for Sparse Data on Multicore CPUs and GPUs

Unifying storage, traversal, and reduction in sparse matrices, graphs, and numerical meshes in TNL

T. Oberhuber, J. Klinkovský, T. Halada, R. Fučík
Czech Technical University in Prague; Template Numerical Library – tnl-project.org

TEMPLATE NUMERICAL LIBRARY

CTU CZECH TECHNICAL UNIVERSITY IN PRAGUE

Sparse applications differ in semantics, but repeatedly process variable-length groups of elements.
TNL Segments separate the application algorithm from storage format, memory layout, and hardware-specific thread mapping.

OPEN SOURCE · C++ / CUDA / HIP
Multicore CPU · NVIDIA GPU · AMD GPU

1. Motivation and core abstraction

Many scientific structures can be viewed as differently sized groups stored in one or more global arrays.

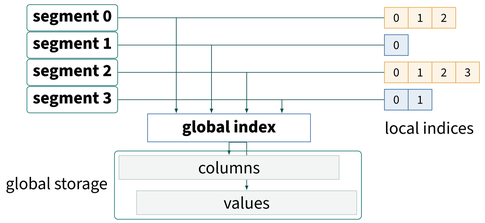
Application	Element	Group
Sparse matrix	nonzero entry	matrix row
Sparse graph	graph edge	edges incident to a vertex
Numerical mesh	mesh entity	incident/adjacent mesh entities
Particle-in-cell	particle	particles in a mesh cell

Segments separate application operations from physical storage and parallel execution.

Efficient storage
Compact representation of variable-length groups.

Portable execution
Efficient mapping to multicore CPUs and GPUs.

Common interface
Traversal and reduction independent of storage and hardware.



Segments provide the mapping
(segmentIdx, localIdx) → globalIdx,
where the global index addresses application-specific arrays.
They store no domain-specific data.

2. Irregular segment sizes require adaptable storage and execution

The segment-length distribution determines both memory layout and GPU work mapping.

nearly uniform



Regular layouts and fixed mappings work well.

variable



Sliced layouts reduce padding.

highly skewed



Cooperative or adaptive scheduling balances work.

Compile-time segment layout
- CSR and ELLPACK-family formats,
- optional row permutation,
- row-wise or column-wise ordering.

Runtime execution policy
- the number of threads per segment,
- fixed or adaptive scheduling.

3. A common interface for traversal and reduction

Element traversal:
Apply a user-defined lambda to the elements of selected segments.
- *Assembly, updates, graph-edge and mesh-entity processing.*

Per-segment reduction:
Combine element values and store one result per selected segment.
- *SpMV, row norms, graph degrees and aggregation.*

Selectable execution domain:
All segments · contiguous range · explicit index set or frontier · predicate.

User lambdas define the computation; Segments provide indexing, selection, scheduling, and parallel execution.

4. Positioning and novelty

Performance portability
Kokkos, RAJA
policies and iteration spaces

Sparse compilers
TACO, MLIR, SparseTIR, UniSparse
formats and code generation

data-structure-aware · **explicit C++ API**

Segments
reusable abstraction for **variable-length groups**

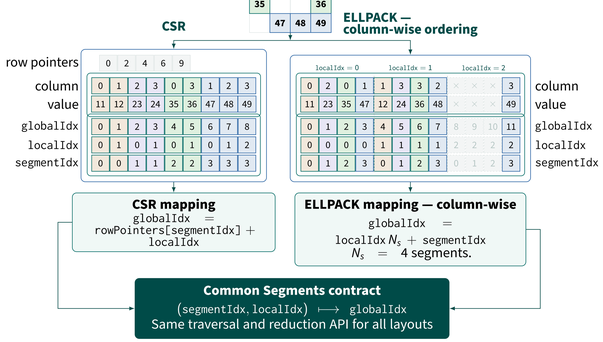
higher-level abstraction · **cross-domain reuse**

Low-level primitives
CUB
segmented GPU operations

Domain-specific libraries
Ginkgo, Gunrock
sparse-matrix or graph algorithms

The novelty is not another sparse format, but a reusable abstraction boundary between irregular-data semantics, physical representation, and parallel execution.

5. One segment abstraction, multiple sparse formats



Matrix rows form the logical segments: CSR stores them contiguously, while column-wise ELLPACK interleaves equal local positions and pads shorter rows.

6. One algorithm, multiple storage and execution strategies

`SparseMatrix`
values + column indices
↓
`Segments::Format`
storage + traversal + reduction
↓
CPU / CUDA / HIP kernels

```
// SpMV with segment reduction
auto fetch = [=] (Index globalIdx) {
   return values[globalIdx] * x[column[globalIdx]]; };
auto store = [=] (Index rowIdx, Real result) {
   out[rowIdx] = result; };
reduceAllSegments( segments, fetch, TNL::Plus{}, store );
```

The application code remains unchanged across segment formats and CPU, CUDA, or HIP backends.

7. Matrix specializations share one API

GENERAL MATRIX
column indices 0 1 3 0 2
values a b c d e
explicit values and indices

BINARY MATRIX
column indices 0 1 3 0 2
values implied as 1
indices only; values implied as 1

SYMMETRIC MATRIX

store one triangle; mirror contributions

The traversal and reduction API remains unchanged; only storage and kernels differ.

8. Performance evidence on more than 5,000 matrices

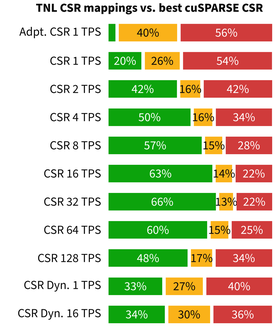
Benchmarking methodology: 5,000+ matrices from SuiteSparse Matrix Collection, 32-bit indices, FP64 SpMV, NVIDIA A100, CUDA 12.4. Median CUDA-event time after warm-up.

ELLPACK-Family Gains Beyond Best System-Wide Baseline CSR

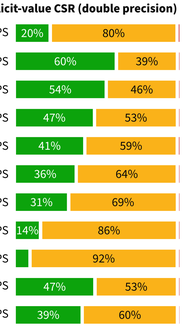
	>1.00×	>1.05×	>1.10×	>1.15×	>1.20×	>1.25×	>1.30×	>1.35×
Ellpack	123	73	23	6	2	0	0	0
RowMajor SlicedEllpack	1130	49	3	1	0	0	0	0
ColumnMajor SlicedEllpack	801	131	45	19	11	5	1	0
BiEllpack	30	6	1	0	0	0	0	0
ChunkedEllpack	72	19	8	1	0	0	0	0

Counts of matrices where the best configuration of each ELLPACK-family layout exceeds the fastest CSR result across TNL, cuSPARSE, and CPU. Thresholds are cumulative.

TNL CSR mappings vs. best cuSPARSE CSR

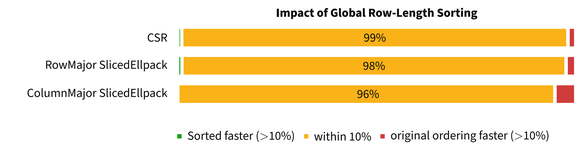


Binary vs. explicit-value CSR (double precision)



■ CSR faster (>10%) ■ within 10% ■ cuSPARSE faster (>10%)
■ Binary CSR faster (>10%) ■ within 10% ■ explicit-value CSR faster (>10%)
TPS = Threads Per Segment

Impact of Global Row-Length Sorting

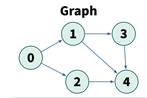


■ Sorted faster (>10%) ■ within 10% ■ original ordering faster (>10%)

Segments are globally sorted by length. SpMV remains within 10% of the original ordering on 96–99% of matrices.

On an A100, execution mapping and binary semantics deliver broad gains; alternative layouts provide niche wins, while global sorting is mostly neutral.

9. Graphs reuse sparse matrices and segment operations



Graph
vertices + edge semantics
↓
`SparseMatrix`
adjacency matrix
↓
`Segments::Format`
row traversal + reduction

vertex ↔ row ↔ segment
edge ↔ nonzero ↔ segment element

Matrix representation
- Unweighted graphs use a **binary sparse matrix** with implicit values.
- The storage format remains interchangeable through `Segments::Format`.

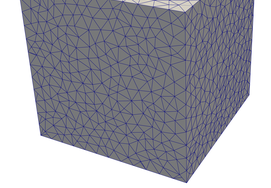
Processing selected vertices
A traversal may operate on all vertices, a contiguous range, an explicit frontier, or a predicate-selected set. Specialized kernels avoid launching work for inactive vertices.

Subgraphs without materialization
Vertex and edge predicates restrict traversal to a selected subgraph without materialization or renumbering.

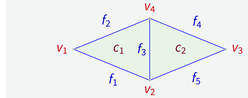
Graph algorithms such as BFS, SSSP, and connected components reuse the same segment traversal and reduction infrastructure.

10. Unstructured meshes: topology as sparse binary matrices

Mesh topology is represented by sparse binary incidence and adjacency matrices.



Topological relations as sparse binary matrices.



$$I_{0,1} = \begin{pmatrix} & f_1\ f_2\ f_3\ f_4\ f_5 \\ v_1 & 1\ 1\ \cdot\ \cdot\ \cdot \\ v_2 & 1\ \cdot\ 1\ \cdot\ 1 \\ v_3 & \cdot\ \cdot\ \cdot\ 1\ 1 \\ v_4 & \cdot\ 1\ 1\ 1\ \cdot \end{pmatrix} \qquad I_{0,2} = \begin{pmatrix} & c_1\ c_2 \\ v_1 & 1\ \cdot \\ v_2 & 1\ 1 \\ v_3 & \cdot\ 1 \\ v_4 & 1\ 1 \end{pmatrix}.$$

$I_{p,q}(i,j) = 1$ iff the p-dimensional entity i is incident to the q-dimensional entity j.
Examples: vertex–edge, vertex–cell, and cell–face incidence.

Mesh representation
- **Geometry:** vertex coordinates
- **Topology:** vertices, edges, faces, cells, and their relations

These relations reuse the same segment storage and traversal infrastructure.

11. Current evidence and next steps

Validated
- one traversal and reduction API across CSR and ELLPACK layouts,
- fixed and adaptive GPU execution mappings,
- SpMV evaluation on more than 5,000 matrices.

Next steps
- automatic format and kernel selection,
- unified filtering and post-processing of reductions,
- stronger graph/frontier benchmarks and quantitative API-reuse analysis.

Sparse-matrix results validate efficient execution; graph and mesh mappings demonstrate broader reuse.

J. Klinkovský et al., *ACM TOMS 48(3), 2022*.
T. Halada et al., *Computer Physics Communications 326, 110174, 2026*.

tnl-project.org

Manuscript and software context
T. Oberhuber et al., *Segments: Data abstraction for parallel computations on sparse data*, current manuscript.
TNL is an open-source C++ framework for multicore CPUs, CUDA GPUs, and HIP GPUs, published under the MIT license.

AI assistance: OpenAI ChatGPT-5.6 was used to draft and edit text throughout the poster. Anthropic Claude Sonnet 5.0 assisted with the development of data-analysis and plotting scripts. The authors verified all text, code, analyses, and figures and take full responsibility for the scientific content.

Take-home message:
One sparse-data abstraction; many layouts, execution policies, and hardware backends.