GBTL v3.0: Primitives for Optimized mxm

7 August 2020

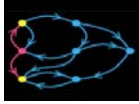
Scott McMillan

Principal Engineer

SEI Emerging Technology Center

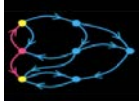
Software Engineering Institute

Carnegie Mellon University



This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

DM20-0632



GraphBLAS Base Operations

(C API Version 1.3)

Operation	Math	Out	Inputs
mxm	$\mathbf{C}\langle\neg/s\mathbf{M},z\rangle = \mathbf{C} \odot (\mathbf{A}^{\mathbf{T}} \oplus.\otimes \mathbf{B}^{\mathbf{T}})$	**C**	¬, s, **M**, z, ⊙, **A**, **T**, ⊕.⊗, **B**, **T**
mxv (vxm)	$\mathbf{c}\langle\neg/s\mathbf{m},z\rangle = \mathbf{c} \odot (\mathbf{A}^{\mathbf{T}} \oplus.\otimes \mathbf{b})$	**c**	¬, s, **m**, z, ⊙, **A**, **T**, ⊕.⊗, **b**
eWiseMult	$\mathbf{C}\langle\neg/s\mathbf{M},z\rangle = \mathbf{C} \odot (\mathbf{A}^{\mathbf{T}} .\otimes \mathbf{B}^{\mathbf{T}})$	**C**	¬, s, **M**, z, ⊙, **A**, **T**, ⊗, **B**, **T**
eWiseAdd	$\mathbf{C}\langle\neg/s\mathbf{M},z\rangle = \mathbf{C} \odot (\mathbf{A}^{\mathbf{T}} .\oplus \mathbf{B}^{\mathbf{T}})$	**C**	¬, s, **M**, z, ⊙, **A**, **T**, ⊕, **B**, **T**
reduce (row)	$\mathbf{c}\langle\neg/s\mathbf{m},z\rangle = \mathbf{c} \odot [\oplus_j \mathbf{A}^{\mathbf{T}}(:,j)]$	**c**	¬, s, **m**, z, ⊙, **A**, **T**, ⊕
apply	$\mathbf{C}\langle\neg/s\mathbf{M},z\rangle = \mathbf{C} \odot f(\mathbf{A}^{\mathbf{T}})$	**C**	¬, s, **M**, z, ⊙, **A**, **T**, *f*()
transpose	$\mathbf{C}\langle\neg/s\mathbf{M},z\rangle = \mathbf{C} \odot \mathbf{A}^{\mathbf{T}}$	**C**	¬, s, **M**, z, ⊙, **A** (**T**)
extract	$\mathbf{C}\langle\neg/s\mathbf{M},z\rangle = \mathbf{C} \odot \mathbf{A}^{\mathbf{T}}(\mathbf{i},\mathbf{j})$	**C**	¬, s, **M**, z, ⊙, **A**, **T**, **i**, **j**
assign	$\mathbf{C}\langle\neg/s\mathbf{M},z\rangle (\mathbf{i},\mathbf{j}) = \mathbf{C}(\mathbf{i},\mathbf{j}) \odot \mathbf{A}^{\mathbf{T}}$	**C**	¬, s, **M**, z, ⊙, **A**, **T**, **i**, **j**
build (meth.)	$\mathbf{C} = \mathbb{S}^{mxn}(\mathbf{i},\mathbf{j},\mathbf{v},\odot)$	**C**	⊙, m, n, **v**, **i**, **j**
extractTuples (meth.)	$(\mathbf{i},\mathbf{j},\mathbf{v}) = \mathbf{A}$	**i**,**j**,**v**	**A**

Notation: **i,j** – index arrays, **v** – value array, **m** – 1D mask, **c,b** – 1D vector (column), **M** – 2D mask, **A,B,C** – 2D matrix,
T – transpose, ¬ - complement, s – structure-only, z – replace
⊙ accumulate monoid/binary function, ⊕.⊗ semiring,
blue – optional parameters, red – optional modifiers (using Descriptors in the C API)

Part 3: GBTL v 3.0 Release

Changelog (https://github.com/cmu-sei/gbtl)

2020-06 Scott McMillan smcmillan@sei.cmu.edu

*** Version 3.0 Release. C++17 compiler required to build. This version is NOT backward compatible with Version 2.0** for the following reasons:

- namespace name "GraphBLAS" has been replaced with "grb".
- The boolean "replace_flag" has change types to an enum with MERGE and REPLACE (corresponding false and true respectively).
- Some of the names of the monoid and semiring template classes have been changed to more closely match C API 1.3 names (e.g. MaxSelect1stSemiring is now MaxFirstSemiring)
- Order of parameters in markov cluster has changed (swapped convergence_threshold and max_iters)

*** API updated to implement new functionality included in GraphBLAS C API Specification version 1.3:**

- add getVersion() method (reports the C API equivalent version)
- add removeElement() method to Matrix and Vector classes.
- add resize() method to Matrix and Vector classes.
- add Exclusive-NOR (XNOR) boolean binary operator
- add bitwise OR/AND/XOR/XNOR binary operators for integer types
- add all predefined monoids and semirings.
- **add structure-only view for masks (composable with complement).**
- add kronecker matrix multiply operation.
- add apply operation that takes a binary operator and a scalar constant for one of its inputs (Note: this requires constexpr processing and not recommended for C++ due to support of lambdas and bind objects in the original variants).
- add init() and wait(obj) that are NO-OP functions

*** Modern C++ idioms from C++14 and C++17 Standards included**

- removed tag_type from Matrix, Vector and "view" classes in favor of modern traits idioms external to the classes.
- Using SFINAE techniques for template function overload resolution and also template type restriction (e.g. std::enable_if, etc) both for operators and operations.
- used tag dispatch to resolve overloaded template functions (e.g. apply and assign operations)
- removed typedefs (like result_type) from within operators to be consistent with C++17 changes to <functional> and to support use of lambdas in place of predefined operator functors. Replaced uses with decltype() and declval() where necessary.
- Replaced typedefs with using.
- range based for-loops used where possible in implementation
- structured bindings used to access elements of tuples.
- Replaced calls to std::vector's insert() and push_back() methods with emplace() and emplace_back() where possible.
- Removed GraphBLAS::BinaryOp_Bind2nd functor in favor of C++ STL's std::bind().
- Replaced std::conditional<>::type with std::conditional_t<>, std::is_same<>::value with std::is_same_v<>, etc.

* Refactored view classes: Changed design of view classes to lightweight wrappers and added support for structure-only views. The "backend" view classes have been removed. Nearly all friend declarations are no longer necessary and have been removed.

* "sequential" platform

- Numerous performance improvements applied.
- modified to replace code paths that accessed columns of the row-major storage matrix with row access only (improves performance).
- backend NoMask class removed.
- backend View classes removed.
- backend utility functions removed (refactored into classes).

*** New platform: "optimized_sequential" included that is a work in progress to improve the performance of the "sequential" platform. Currently mxm operation is implemented with more efficient code branches (72 of them).**

*** Algorithms:**

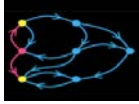
- Updated algorithms replace use of GraphBLAS::BinaryOp_Bind2nd with std::bind().
- Updated algorithms to use structure only masks where needed and removed 1-based indexing.
- Improved performance of index-of() operation for single-source parent BFS using a dense vector (of index values) and eWiseMult (with First or Second operator).
- Added single-source level BFS algorithm
- Added Louvain clustering algorithm
- Removed an erroneous implementation of maxflow algorithm (maxflow_ford_fulk2)
- Two new triangle counting algorithm variants:
  1) |L .* (L +.* U)|,
  2) |L .* (L +.* L)| (no transpose).
- Removed all experimental (FLAME-based) approaches to triangle counting.
- Switched to lambda (as a binary operator to eWiseMult) for computing random values in MIS algorithm so that random number generator objects are local variables.
- Changed how nsver is subtracted from result vector in BC algorithm (vertex_betweenness_centrality_batch_alt_trans_v2())
- Added direct translation of all algorithms in the appendix of the C API Specification v 1.3.0.

*** Test and demo programs:**

- augmented tests for operators (new and old) found in algebra.hpp
- augmented tests for mxm to cover all combinations of input transposes and mask views. Augmented other tests to cover more combinations.
- added tests for vector/matrix methods and new operations.
- removed tests for TransposeView (aka single argument transpose) and ComplementView.
- triangle_count_demo now sorts the vertices by degree (flame versions removed).
- new demo programs for appendix_algorithms, louvain clustering
- addition of mxm, mxv, and vxm timing demos to measure performance.

* Removing additional compiler warnings during build (some still exist from algebra.hpp).

GBTL C++ Signatures: MxM



$$C\langle \neg/sM, z \rangle = C \odot (A^T \oplus.\otimes B^T)$$

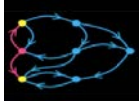
```
template<typename CMatrixT,
         typename MaskT,
         typename AccumT,
         typename SemiringT,
         typename AMatrixT,
         typename BMatrixT>
void mxm(CMatrixT       &C,
         MaskT    const &M,
         AccumT          accum,
         SemiringT       op,
         AMatrixT const &A,
         BMatrixT const &B,
         ReplaceEnum     z = MERGE);
```

Matrices:

- Templated 2-D “containers”:
  - Scalar template parameter for stored value type
  - Template parameters that *could* control type of data structures
  - Future: templatized by the index type
- Destination (C) is an in/out parameter
- Mask matrix
  - Is optional (use grb::NoMask)
  - Can be interpreted as a binary matrix or as structure only
- Input matrices: A and B

```
template <typename ScalarT, typename... StorageTagsT>
class Matrix;
```

GBTL C++ Signatures: MxM



$$C\langle \neg/sM, z \rangle = C \odot (A^T \oplus.\otimes B^T)$$

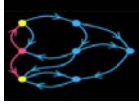
```
template<typename CMatrixT,
         typename MaskT,
         typename AccumT,
         typename SemiringT,
         typename AMatrixT,
         typename BMatrixT>
void mxm(CMatrixT       &C,
         MaskT    const &M,
         AccumT          accum,
         SemiringT       op,
         AMatrixT const &A,
         BMatrixT const &B,
         ReplaceEnum     z = MERGE);
```

```
// call site
mxm(C, NoMask(), NoAccum(), ArithmeticSemiring<float>(),
    transpose(A), transpose(B));
```

Transpose:

- A and B input matrices can be optionally transposed
- Specified at the call site with lightweight wrappers

GBTL C++ Signatures: MxM



```
template<typename CMatrixT,
         typename MaskT,
         typename AccumT,
         typename SemiringT,
         typename AMatrixT,
         typename BMatrixT>
void mxm(CMatrixT       &C,
         MaskT    const &M,
         AccumT          accum,
         SemiringT       op,
         AMatrixT const &A,
         BMatrixT const &B,
         ReplaceEnum     z = MERGE);
```

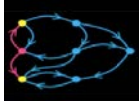
$$C\langle \neg/sM, z \rangle = C \odot (A^T \oplus.\otimes B^T)$$

Operators:

- "Semiring" ($\oplus.\otimes$) is a pair of operators templatized by two input and one output type
  - "Multiply" ($\otimes$) is a binary operator compatible templatized by two different input types and one output type
  - "Add" ($\oplus$) is a commutative monoid operator (binaryop with identity) templatized by one type for both inputs and outputs
  - E.g. Arithmetic semiring: $\oplus$ = +, $\otimes$ = x
- Accumulate ($\odot$) with the current output matrix:
  - Is optional (use grb::NoAccumulate)
  - Binary operator (e.g. std::plus<float>, grb::Plus<float, int, float>)
  - If it is not commutative and/or has no identity, unintuitive results can happen.

```
template<class D1, class D2=D1, class D3=D1>
class ArithmeticSemiring
{
   D3 add(D3 a, D3 b) const;
   D3 mult(D1 a, D2 b) const;
   D3 zero() const;
};
```

GBTL C++ Signatures: MxM



$$C\langle \neg/s M, z \rangle = C \odot (A^T \oplus . \otimes B^T)$$

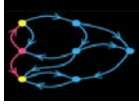
```
template<typename CMatrixT,
         typename MaskT,
         typename AccumT,
         typename SemiringT,
         typename AMatrixT,
         typename BMatrixT>
void mxm(CMatrixT       &C,
         MaskT    const &M,
         AccumT          accum,
         SemiringT       op,
         AMatrixT const &A,
         BMatrixT const &B,
         ReplaceEnum     z = MERGE);
```

```
// call site
mxm(C, complement(structure(M)), NoAccum(),
    ArithmeticSemiring<float>(),
    A, B);
```

Mask and its modifiers:

- Controls which values are written to the output matrix
  - Stucture mask (s):
    - if a value is stored at M(i,j), then C(i,j) can be written.
  - Value mask (default):
    - if a value is stored at M(i,j) *and* its evaluates to true, then C(i,j) can be written.
  - Complement (¬):
    - Inverts the logic of either of the first two
- Structure and complement specified at the call site with wrappers
  - none (value mask)
  - structure(M)
  - complement(M) – complement of value mask
  - complement(structure(M)) – complement of structure mask

GBTL C++ Signatures: MxM



$$C\langle \neg/sM, z \rangle = C \odot (A^T \oplus.\otimes B^T)$$

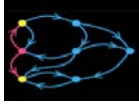
```
template<typename CMatrixT,
         typename MaskT,
         typename AccumT,
         typename SemiringT,
         typename AMatrixT,
         typename BMatrixT>
void mxm(CMatrixT       &C,
         MaskT    const &M,
         AccumT          accum,
         SemiringT       op,
         AMatrixT const &A,
         BMatrixT const &B,
         ReplaceEnum     z = MERGE);
```

```
// call site
mxm(C, M, NoAccum(), ArithmeticSemiring<float>(),
    A, B, REPLACE);
```

Replace (z) vs. Merge :

- Only when mask is present
- Controls what happens to non-masked values
  - Replace (z): The value in C will be removed (annihilated).
  - Merge (default): The location in C will be untouched (stored value or not).

Definition of mxm (from spec)



$\mathbf{C\langle \neg/s\mathbf{M},z\rangle = C \odot (A^T \oplus.\otimes B^T)}$

$\mathbf{T = A^T \oplus.\otimes B^T}$

$$T_{ij} = \bigoplus_{k \in \mathbf{ind}(\widetilde{\mathbf{A}}(i,:)) \cap \mathbf{ind}(\widetilde{\mathbf{B}}(:,j))} (\widetilde{\mathbf{A}}(i,k) \otimes \widetilde{\mathbf{B}}(k,j)),$$

After optional transpose

Treated as sparse matrices

$\mathbf{Z = [C \odot\ ]\ T}$

$$Z_{ij} = \widetilde{\mathbf{C}}(i,j) \odot \widetilde{\mathbf{T}}(i,j), \text{ if } (i,j) \in (\mathbf{ind}(\widetilde{\mathbf{T}}) \cap \mathbf{ind}(\widetilde{\mathbf{C}})),$$

$$Z_{ij} = \widetilde{\mathbf{C}}(i,j), \text{ if } (i,j) \in (\mathbf{ind}(\widetilde{\mathbf{C}}) - (\mathbf{ind}(\widetilde{\mathbf{T}}) \cap \mathbf{ind}(\widetilde{\mathbf{C}}))),$$

$$Z_{ij} = \widetilde{\mathbf{T}}(i,j), \text{ if } (i,j) \in (\mathbf{ind}(\widetilde{\mathbf{T}}) - (\mathbf{ind}(\widetilde{\mathbf{T}}) \cap \mathbf{ind}(\widetilde{\mathbf{C}}))),$$

Only if $\odot$ is defined

$\mathbf{C\langle \neg/s\mathbf{M},z\rangle = Z}$

After optional structure or complement

Replace → $\mathbf{L}(\mathsf{C}) = \{(i,j,Z_{ij}) : (i,j) \in (\mathbf{ind}(\widetilde{\mathbf{Z}}) \cap \mathbf{ind}(\widetilde{\mathbf{M}}))\}.$

Merge → $\mathbf{L}(\mathsf{C}) = \{(i,j,C_{ij}) : (i,j) \in (\mathbf{ind}(\mathsf{C}) \cap \mathbf{ind}(\widetilde{\neg\mathbf{M}}))\} \cup \{(i,j,Z_{ij}) : (i,j) \in (\mathbf{ind}(\widetilde{\mathbf{Z}}) \cap \mathbf{ind}(\widetilde{\mathbf{M}}))\}.$

Optimizing mxm in GBTL

$$C\langle \neg/sM, z \rangle = C \odot (A^T \oplus.\otimes B^T)$$
$$T = A^T \oplus.\otimes B^T$$
$$Z = [C \odot]\ T$$
$$C\langle \neg/sM, z \rangle = Z$$

- 72 different code paths investigated:
  - Transpose either input matrix (4 options, columns)
  - Optional write mask with optional complement or structure (5 options)
    - NoMask, Mask, StructMask, CompMask, CompStructMask
  - Optional accumulate with output matrix (2 options)
  - Merge/Replace semantics (2 options, when mask used)

¬/sM	⊙	z	A*B	A*B^T	A^T*B	A^T*B^T
NoMask	NoAccum	X	C := A*B	C := A*B′	C := A'*B	C := A'*B'
NoMask	Accum	X	C := C + (A*B)	C := C + (A*B′)	C := C + (A'*B)	C := C + (A'*B′)
Mask	NoAccum	REPLACE	C := M .* (A*B)	C := M .* (A*B′)	C := M .* (A'*B)	C := M .* (A'*B′)
Mask	NoAccum	MERGE	C := [¬M .* C] U {M .* (A*B)}	C := [¬M .* C] U {M .* (A*B′)}	C := [¬M .* C] U {M .* (A'*B)}	C := [¬M .* C] U {M .* (A'*B′)}
Mask	Accum	REPLACE	C := [M .* C] + {M .* (A*B)}	C := [M .* C] + {M.* (A*B′)}	C := [M .* C] + {M.* (A'*B)}	C := [M .* C] + {M.* (A'*B′)}
Mask	Accum	MERGE	C := [¬M .* C] U {[M .* C] + M .* (A*B)}	C := [¬M .* C] U {[M .* C] + M.* (A*B′)}	C := [¬M .* C] U {[M .* C] + M.* (A'*B)}	C := [¬M .* C] U {[M .* C] + M.* (A'*B′)}
StructMask	NoAccum	REPLACE	C := s(M) .* (A*B)	C := s(M) .* (A*B′)	C := s(M) .* (A'*B)	C := s(M) .* (A'*B′)
StructMask	NoAccum	MERGE	C := [¬s(M) .* C] U {s(M) .* (A*B)}	C := [¬s(M) .* C] U {s(M) .* (A*B′)}	C := [¬s(M) .* C] U {s(M) .* (A'*B)}	C := [¬s(M) .* C] U {s(M) .* (A'*B′)}
StructMask	Accum	REPLACE	C := [s(M) .* C] + {s(M) .* (A*B)}	C := [s(M) .* C] + {s(M).* (A*B′)}	C := [s(M) .* C] + {s(M).* (A'*B)}	C := [s(M) .* C] + {s(M).* (A'*B′)}
StructMask	Accum	MERGE	C := [¬s(M) .* C] U {[s(M) .* C] + s(M) .* (A*B)}	C := [¬s(M) .* C] U {[s(M) .* C] + s(M).* (A*B′)}	C := [¬s(M) .* C] U {[s(M) .* C] + s(M).* (A'*B)}	C := [¬s(M) .* C] U {[s(M) .* C] + s(M).* (A'*B′)}
CompMask	NoAccum	REPLACE	C := ¬M .* (A*B)	C := ¬M .* (A*B′)	C := ¬M .* (A'*B)	C := ¬M .* (A'*B′)
CompMask	NoAccum	MERGE	C := [M .* C] U¬M .* (A*B)	C := [M .* C] U {¬M .* (A*B′)}	C := [M .* C] U {¬M .* (A'*B)}	C := [M .* C] U {¬M .* (A'*B′)}
CompMask	Accum	REPLACE	C := (¬M .* C) + ¬M .* (A*B)	C := {(¬M .* C) + ¬M .* (A*B′)}	C := {(¬M .* C) + ¬M .* (A'*B)}	C := {(¬M .* C) + ¬M .* (A'*B′)}
CompMask	Accum	MERGE	C := [M .* C] U {(¬M .* C) + ¬M .* (A*B)}	C := [M .* C] U {(¬M .* C) + ¬M .* (A*B′)}	C := [M .* C] U {(¬M .* C) + ¬M .* (A'*B)}	C := [M .* C] U {(¬M .* C) + ¬M .* (A'*B′)}
CompStructMask	NoAccum	REPLACE	C := ¬s(M) .* (A*B)	C := ¬s(M) .* (A*B′)	C := ¬s(M) .* (A'*B)	C := ¬s(M) .* (A'*B′)
CompStructMask	NoAccum	MERGE	C := [s(M) .* C] U¬s(M) .* (A*B)	C := [s(M) .* C] U {¬s(M) .* (A*B′)}	C := [s(M) .* C] U {¬s(M) .* (A'*B)}	C := [s(M) .* C] U {¬s(M) .* (A'*B′)}
CompStructMask	Accum	REPLACE	C := (¬s(M) .* C) + ¬s(M) .* (A*B)	C := {(¬s(M) .* C) + ¬s(M) .* (A*B′)}	C := {(¬s(M) .* C) + ¬s(M) .* (A'*B)}	C := {(¬s(M) .* C) + ¬s(M) .* (A'*B′)}
CompStructMask	Accum	MERGE	C := [s(M) .* C] U {(¬s(M) .* C) + ¬s(M) .* (A*B)}	C := [s(M) .* C] U {(¬s(M) .* C) + ¬s(M) .* (A*B′)}	C := [s(M) .* C] U {(¬s(M) .* C) + ¬s(M) .* (A'*B)}	C := [s(M) .* C] U {(¬s(M) .* C) + ¬s(M) .* (A'*B′)}

Optimizing mxm in GBTL

$$C\langle M\rangle = C \odot (A \oplus.\otimes B)$$
$$T = A \oplus.\otimes B$$
$$Z = [C \odot]\ T$$
$$C\langle M\rangle = Z$$

- 72 different code paths investigated:
  - Transpose either input matrix (4 options, columns)
  - Optional write mask with optional complement or structure (5 options)
    - NoMask, Mask, StructMask, CompMask, CompStructMask
  - Optional accumulate with output matrix (2 options)
  - Merge/Replace semantics (2 options, when mask used)

¬/s**M**	⊙	z	A*B	A*B^T	A^T*B	A^T*B^T
NoMask	NoAccum	X	C := A*B	C := A*B′	C := A'*B	C := A'*B'
NoMask	Accum	X	C := C + (A*B)	C := C + (A*B′)	C := C + (A'*B)	C := C + (A'*B′)
Mask	NoAccum	REPLACE	C := M .* (A*B)	C := M .* (A*B′)	C := M .* (A'*B)	C := M .* (A'*B′)
Mask	NoAccum	MERGE	C := [¬M .* C] U {M .* (A*B)}	C := [¬M .* C] U {M .* (A*B′)}	C := [¬M .* C] U {M .* (A'*B)}	C := [¬M .* C] U {M .* (A'*B′)}
Mask	Accum	REPLACE	C := [M .* C] + {M .* (A*B)}	C := [M .* C] + {M.* (A*B′)}	C := [M .* C] + {M.* (A'*B)}	C := [M .* C] + {M.* (A'*B′)}
Mask	Accum	MERGE	C := [¬M .* C] U {[M .* C] + M .* (A*B)}	C := [¬M .* C] U {[M .* C] + M.* (A*B′)}	C := [¬M .* C] U {[M .* C] + M.* (A'*B)}	C := [¬M .* C] U {[M .* C] + M.* (A'*B′)}
StructMask	NoAccum	REPLACE	C := s(M) .* (A*B)	C := s(M) .* (A*B′)	C := s(M) .* (A'*B)	C := s(M) .* (A'*B′)
StructMask	NoAccum	MERGE	C := [¬s(M) .* C] U {s(M) .* (A*B)}	C := [¬s(M) .* C] U {s(M) .* (A*B′)}	C := [¬s(M) .* C] U {s(M) .* (A'*B)}	C := [¬s(M) .* C] U {s(M) .* (A'*B′)}
StructMask	Accum	REPLACE	C := [s(M) .* C] + {s(M) .* (A*B)}	C := [s(M) .* C] + {s(M).* (A*B′)}	C := [s(M) .* C] + {s(M).* (A'*B)}	C := [s(M) .* C] + {s(M).* (A'*B′)}
StructMask	Accum	MERGE	C := [¬s(M) .* C] U {[s(M) .* C] + s(M) .* (A*B)}	C := [¬s(M) .* C] U {[s(M) .* C] + s(M).* (A*B′)}	C := [¬s(M) .* C] U {[s(M) .* C] + s(M).* (A'*B)}	C := [¬s(M) .* C] U {[s(M) .* C] + s(M).* (A'*B′)}
CompMask	NoAccum	REPLACE	C := ¬M .* (A*B)	C := ¬M .* (A*B′)	C := ¬M .* (A'*B)	C := ¬M .* (A'*B′)
CompMask	NoAccum	MERGE	C := [M .* C] U¬M .* (A*B)	C := [M .* C] U {¬M .* (A*B′)}	C := [M .* C] U {¬M .* (A'*B)}	C := [M .* C] U {¬M .* (A'*B′)}
CompMask	Accum	REPLACE	C := (¬M .* C) + ¬M .* (A*B)	C := {(¬M .* C) + ¬M .* (A*B′)}	C := {(¬M .* C) + ¬M .* (A'*B)}	C := {(¬M .* C) + ¬M .* (A'*B′)}
CompMask	Accum	MERGE	C := [M .* C] U {(¬M .* C) + ¬M .* (A*B)}	C := [M .* C] U {(¬M .* C) + ¬M .* (A*B′)}	C := [M .* C] U {(¬M .* C) + ¬M .* (A'*B)}	C := [M .* C] U {(¬M .* C) + ¬M .* (A'*B′)}
CompStructMask	NoAccum	REPLACE	C := ¬s(M) .* (A*B)	C := ¬s(M) .* (A*B′)	C := ¬s(M) .* (A'*B)	C := ¬s(M) .* (A'*B′)
CompStructMask	NoAccum	MERGE	C := [s(M) .* C] U¬s(M) .* (A*B)	C := [s(M) .* C] U {¬s(M) .* (A*B′)}	C := [s(M) .* C] U {¬s(M) .* (A'*B)}	C := [s(M) .* C] U {¬s(M) .* (A'*B′)}
CompStructMask	Accum	REPLACE	C := (¬s(M) .* C) + ¬s(M) .* (A*B)	C := {(¬s(M) .* C) + ¬s(M) .* (A*B′)}	C := {(¬s(M) .* C) + ¬s(M) .* (A'*B)}	C := {(¬s(M) .* C) + ¬s(M) .* (A'*B′)}
CompStructMask	Accum	MERGE	C := [s(M) .* C] U {(¬s(M) .* C) + ¬s(M) .* (A*B)}	C := [s(M) .* C] U {(¬s(M) .* C) + ¬s(M) .* (A*B′)}	C := [s(M) .* C] U {(¬s(M) .* C) + ¬s(M) .* (A'*B)}	C := [s(M) .* C] U {(¬s(M) .* C) + ¬s(M) .* (A'*B′)}

Optimizing mxm in GBTL

Mask+Accum+Merge+NoTranspose, assuming rowMajor storage

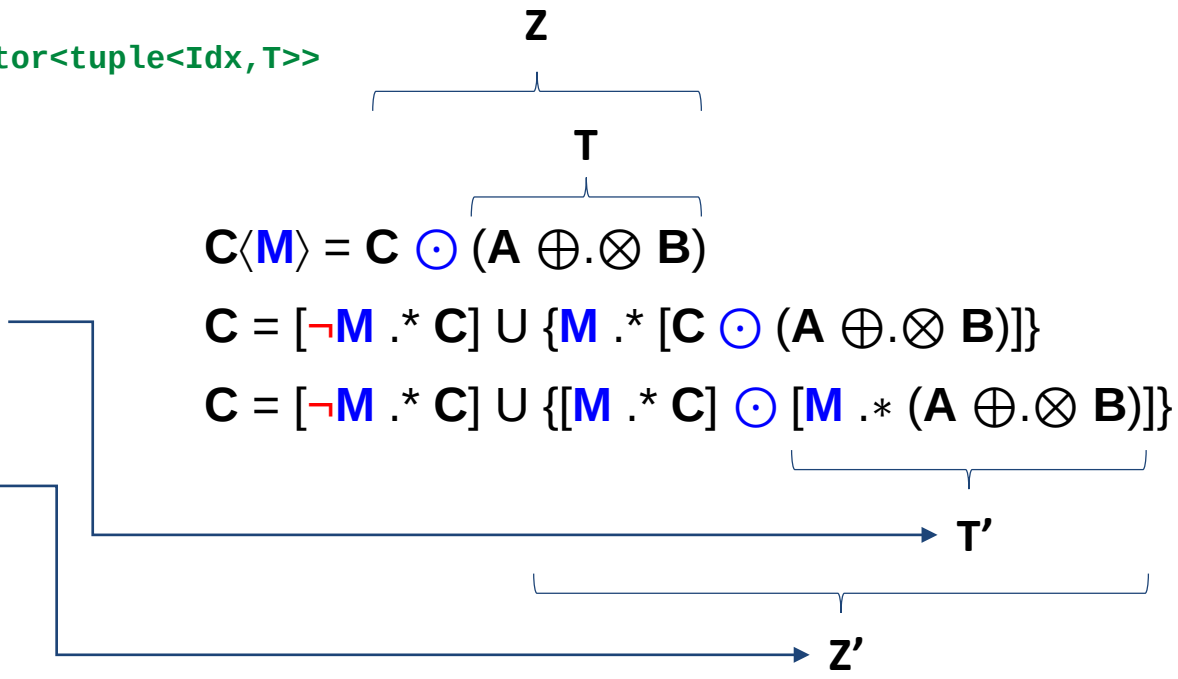
```
SparseRow<T> t, z, c; // like vector<tuple<Idx,T>>

for i=0..A.rows {
  t.clear()

  for (k, a_ik) in A[i] {
    // masked_axpy
    t = t ⊕ [M[i].*(a_ik ⊗ B[k])]
  }

  // masked_accum
  z = M[i].*C[i] ⊙ t

  // masked_merge
  c = (!M[i].*C[i]) U z
  C[i] = c  // set_row
}
```

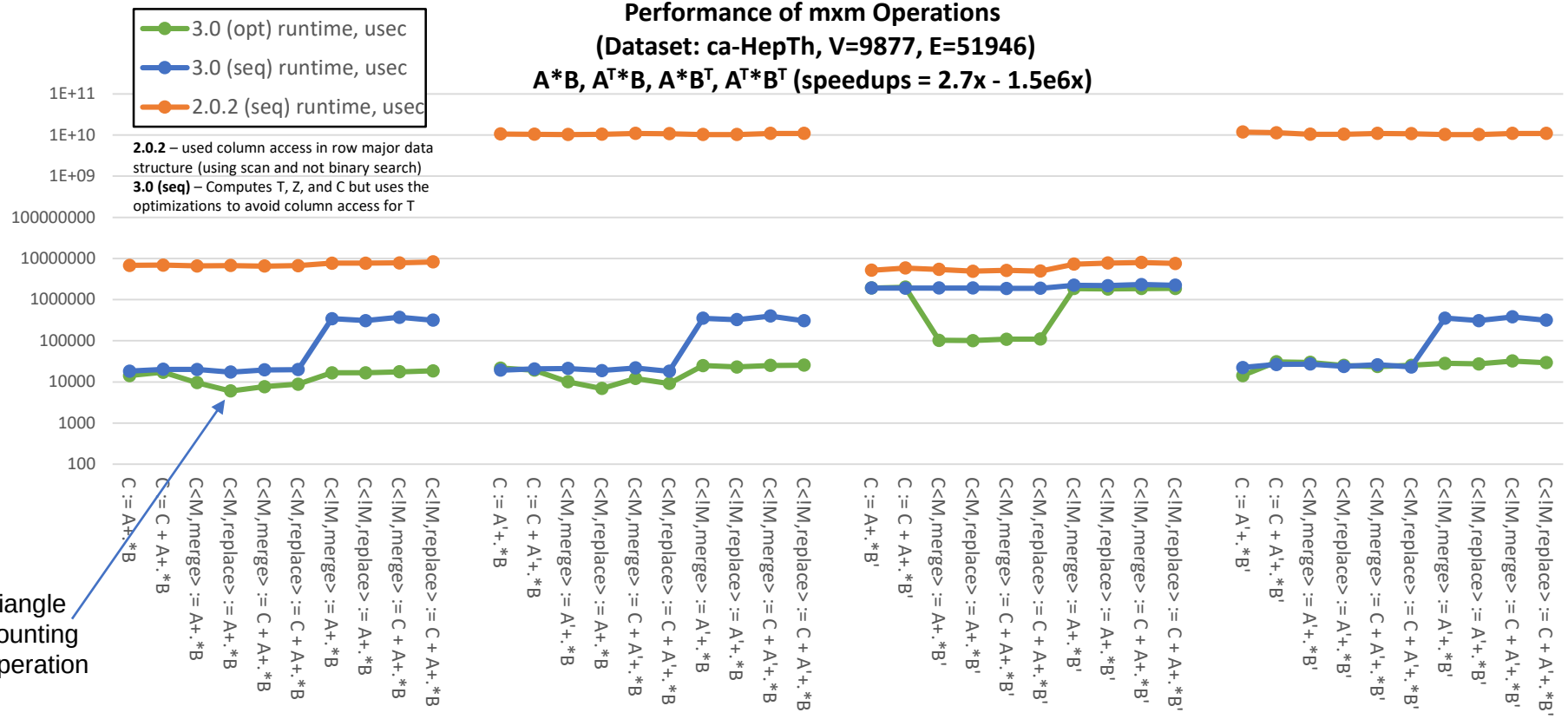


Recurring Sparse Primitives

- Sparse “axpy” using a semiring :
  - Sparse set union with “plus” operation, $\oplus$, where elements overlap, one set is scaled.
  - $\mathbf{t} = \mathbf{t} \oplus (a_{ik} \otimes \mathbf{B}[k])$
- Masked sparse “axpy” using a semiring:
  - Like “sparse axpy” where elements are filtered by the binary mask, M:
  - $\mathbf{t} = \mathbf{M}[i] .* [\mathbf{t} \oplus (a_{ik} \otimes \mathbf{B}[k])]$
- Sparse accumulate:
  - Sparse set union applying an accumulate operation, $\odot$, where elements intersect
  - $\mathbf{C}[i] = \mathbf{C}[i] \odot \mathbf{t}$
- Masked sparse accumulate:
  - $\mathbf{C}[i] = (\mathbf{M}[i] .* \mathbf{C}[i]) \odot \mathbf{t}$
  - where **t** has already been masked by **M**[i] (as in masked sparse axpy).

Backups

GBTL 3.0 Performance Comparison (mxm, 40 paths)



triangle counting operation

Math	C := A (+.*) B	C := C (+) A (+.*) B	C := M .* [A (+.*) B]	C := [!M.*C] U {M.*[A(+.*)B]}	C := [M.*C] (+) {M.*[A(+.*)B]}	C := [!M.*C] U {[M.*C] (+) M.*[A(+.*)B
Func name	**NoMask_NoAccum**	**NoMask_Accum**	**Mask_NoAccum, REPLACE**	**Mask_NoAccum, MERGE**	**Mask_Accum, REPLACE**	**Mask_Accum, MERGE**
Inputs	A, B, (+.*)	C, A, B, (+.*) (+)	M, A, B, (+.*)	M, A, B, (+.*)	C, M, A, B, (+.*), (+)	C, M, A, B, (+.*), (+)
Outputs	C	C	C	C	C	C
early exits	if (A | B are empty)	if (A | B are empty)	if (A|B|M are empty)	if M.empty	if M.empty	if M.empty
	clear C	return	clear C	return	clear C	return
axpy kernel	for i=0..A.rows	for i=0..A.rows	for i=0..A.rows	for i=0..A.rows	for i=0..A.rows	for i=0..A.rows
					if M[i] empty skip i	if M[i] empty skip i
	T.clear	T.clear	T.clear	T.clear	T.clear	T.clear
			if M[i] not empty	if M[i] not empty		
	for a_ik in A[i]	for a_ik in A[i]	for a_ik in A[i]	for a_ik in A[i]	for a_ik in A[i]	for a_ik in A[i]
	if B[k] not empty	if B[k] not empty				
	//axpy	//axpy	//masked_axpy(Mi)	//masked_axpy(Mi)	//masked_axpy(Mi)	//masked_axpy(Mi)
Compute T	T += (a_ik*B[k])	T += (a_ik*B[k])	T += M[i] .* a_ik*B[k]	T += M[i] .* a_ik*B[k]	T += M[i] .* a_ik*B[k]	T += M[i] .* a_ik*B[k]
					// maskedAccum	// masked_accum
Compute Z					Z = M[i].*C[i] + T	Z = M[i].*C[i] + T
				//maskedMerge		//masked_merge
		if T not empty		Z = (!M[i].*C[i]) U T		Z = (!M[i].*C[i]) U Z
Compute C	C[i] = T //setRow	C[i] += T //mergeRow	C[i] = T //setRow	C[i] = Z // setRow	C[i] = Z // setRow	C[i] = Z // setRow