Design and Implementation of the GraphBLAS Template Library (GBTL) v2.0

31 January 2018

Scott McMillan

SEI Emerging Technology Center

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.

DM18-0095

Contents

- Types: GrB_Index, GrB_Index*, GrB_NULL, GrB_ALL
- Operation signatures
- Vector (and Matrix) Methods
- Algebra: Operators, Monoids, Semirings
- Descriptors

Link to GBTL repo [spiral_graph branch]: https://github.com/cmu-sei/gbtl/tree/spiral_graph

Example: mxv

```
using namespace GraphBLAS;                          // all items defined in the GraphBLAS namespace are in red

IndexArrayType      iA = { 0, 0, ...};              // typedef for std::vector<IndexType>
IndexArrayType      jA = { 0, 1, ...};
std::vector<double> vA = {12, 7, ...};

Matrix<double, DirectedMatrixTag> A(4, 3);          // Declare a 4x3 matrix of doubles. Domains are built into
                                                    // the template mechanism.  Tag gives "hint" to backend for
                                                    // selecting (opaque) data type.
A.build(iA, jA, vA);                                // There are overloadings of build that take pointers and size.

IndexArrayType      iV = {0, 2};
std::vector<double> vV = {5, 8};

Vector<double, SparseTag> v(3);                     // Same approach for vectors.
v.build(iV, vV);

Vector<double> result(4);                           // Tag can be omitted to get a default type.

mxv(result,
    NoMask(), NoAccumulate(),                       // GrB_NULL is replaced with better named Types
    ArithmeticSemiring<double>(),                   // Operator domains are tied to template mechanism
    A, v,                                           // result = A  +.* v
    true);                                          // This is the REPLACE flag.
```

Miscellany (from graphblas/{types.hpp, indices.hpp, Matrix.hpp})

```
namespace GraphBLAS
{
   typedef uint64_t                IndexType;

   typedef std::vector<IndexType>  IndexArrayType;

   class NoMask {...};

   class NoAccumulate {...};

   class AllIndices {...};
}
```

- **GraphBLAS** namespace replaces **GrB_** prefix.
- Replaces **GrB_Index**
- Replaces **GrB_Index* + size**
- Replaces **GrB_NULL** for the accumulate parameter
- Replaces **GrB_NULL** for the mask parameter
- Replaces **GrB_ALL** for the index array parameter in extract and assign

- With strong types (for the last three):
  - code paths are chosen at compile time via template specialization
  - No switch statements.

Operations signatures: mxv (from graphblas/operations.h)

```
template<typename WVectorT,
         typename MaskT,
         typename AccumT,
         typename SemiringT,
         typename AMatrixT,
         typename UVectorT>
inline void mxv(WVectorT         &w,
                MaskT      const &mask,
                AccumT           accum,
                SemiringT        op,
                AMatrixT   const &A,
                UVectorT   const &u,
                bool             replace_flag = false);
```

```
GrB_Info GrB_mxv(GrB_Vector             w,
                 GrB_Vector const       mask,
                 GrB_BinaryOp const     accum,
                 GrB_Semiring const     op,
                 GrB_Matrix const       A,
                 GrB_Vector const       u,
                 GrB_Descriptor const   desc);
```

- Goal: be “similar” to C API Spec **in use**…(*not always obvious when reading template declarations*).
- Almost all parameters *individually* templated (handles replaced with pass-by-”value”, “&”, and “const &” as needed)
  - allows arbitrary matrix backend and scalar types (different types for each operand)
  - allows replacement of GrB_NULL and GrB_ALL with more strongly typed objects for specialization purposes.
- Descriptor replaced:
  - Views for transpose and complement
  - Replace flag (needs further discussion)
- Return type replaced with exceptions (needs further discussion)

Operations signatures: eWise* (from graphblas/operations.h)

```
template <typename WScalarT,
          typename MaskT,
          typename AccumT,
          typename BinaryOpT,  //can be BinaryOp, Monoid (not Semiring)
          typename UVectorT,
          typename VVectorT,
          typename ...WTagsT>
inline void eWiseMult(Vector<WScalarT, WTagsT...> &w,
                      MaskT                 const &mask,
                      AccumT                       accum,
                      BinaryOpT                    op,
                      UVectorT              const &u,
                      VVectorT              const &v,
                      bool                         replace_flag = false);
```

- Don’t have three overloadings for eWiseMult or eWiseAdd.
  - One signature supports BinaryOp and Monoid,
  - Semiring not supported.
- For Semiring: proposal to wrap Semiring in extractAdd() or extractMult() “views”

Operations signatures: extract and assign (from graphblas/operations.h)

```
template <typename WVectorT,
          typename MaskT,
          typename AccumT,
          typename UVectorT,
          typename SequenceT>
inline void extract(WVectorT           &w,
                    MaskT        const &mask,
                    AccumT              accum,
                    UVectorT     const &u,
                    SequenceT    const &indices,
                    bool                replace_flag = false);
```

- Extract (and assign) take templated “sequences” (see indices.hpp)
  - Monotonically increasing sets.
  - Supports ranges (future: strides)
  - Could contain arbitrarily complex logic or a simple list
  - **`GrB_ALL`** implemented with **`AllIndices`** class that can be passed to **`indices`**
  - Sizes (**`nindices`**, **`nrows`**, **`ncols`** parameters from C API) are “embedded” in the **`SequenceT`** objects.
  - Not currently random-access (needs more discussion about requirements)

Vector (and Matrix) methods become public member funcs (from graphblas/Vector.hpp)

```
template<typename ScalarT, typename... TagsT>
class Vector
{
public:
   Vector() = delete;
   Vector(IndexType nsize);
   Vector(Vector<ScalarT,TagsT...> const &rhs); // copy ctor

   void clear();

   IndexType size() const;
   IndexType nvals() const;

   // other overloads of build provided
   template<typename ValueT, typename BinaryOpT=Second<ScalarT>>
   void build(IndexArrayType            indices,
              std::vector<ValueT> const &vals,
              BinaryOpT                 dup = BinaryOpT());

   bool hasElement(IndexType index) const;
   void setElement(IndexType index, ScalarT const &new_val);
   ScalarT extractElement(IndexType index) const;
   /// @todo support a clearElement()?

   // other overloads of extractTuples provided
   template<typename RAIterI, typename RAIterV>
   void extractTuples(RAIterI ids, RAIterV vals);

   . . .
};
```

Signatures “close” to C API Spec.

```
        GrB_Vector_new without dimensions not allowed
4.2.2.1 GrB_Vector_new(&vec, ScalarT, nsize);
4.2.2.2 GrB_Vector_dup(&vec, rhs);

4.2.2.3 GrB_Vector_clear(vec);

4.2.2.4 GrB_Vector_size(&nsize, vec);  // C++ return val
4.2.2.5 GrB_Vector_nvals(&nvals, vec); // C++ return val

4.2.2.6 GrB_Vector_build(vec, &indices, &vals, n, dup);

        No GraphBLAS C API equivalent
4.2.2.7 GrB_Vector_setElement(vec, val, index);
4.2.2.8 GrB_Vector_extractElement(&val, vec, index);

4.2.2.9 GrB_Vector_extractTuples(&ids, &vals, n, vec);
```

Achieving Opaque Vectors and Matrices (from graphblas/Vector.hpp, graphblas/detail/param_unpack.hpp)

```
// the frontend class.
template<typename ScalarT, typename... TagsT>
class Vector
{
public:
   .
   .  // public interface here
   .
private:
   // the backend "opaque" object
   detail::vector_generator::result<
           ScalarT,
           detail::SparsenessCategoryTag,
           TagsT... ,
           detail::NullTag >::type m_vec;
};
```

Template parameters used to specify "complete" type

- ScalarT template parameter replaces Domains
  - Can be any type (not just C API predefined types)
- TagsT… is a variable number of template parameters
  - Used to provide information to the backend system to help determine the backend Vector type.
  - Examples include: DenseTag, SparseTag, UndirectedMatrixTag (for matrices), etc..
  - We could send directives like: DistributedTag, ReplicatedTag, FastColumnAccessTag, etc.
- vector_generator "parses" template parameters at compile time to compute the backend data type.
- Backend opaque type:
  - Goal: Specialized for hardware and implementation
  - Does not have to adhere to the "hints"
  - `GraphBLAS::backend::Matrix<ScalarT,SparseTag>`
  - This is a subclass of a specific type of matrix, i.e. `GraphBLAS::backend::LilSparseMatrix<ScalarT>`

Algebra: Type, UnaryOp, BinaryOp (from src/graphblas/algebra.hpp)

```
// Unary Operator
template <typename D1, typename D2 = D1>
struct AdditiveInverse
{
   typedef D2 result_type;
   inline D2 operator()(D1 input) { return -input; }
};

typedef GraphBLAS::AdditiveInverse<bool>      GrB_AINV_BOOL;
typedef GraphBLAS::AdditiveInverse<int8_t>    GrB_AINV_INT8;
typedef GraphBLAS::AdditiveInverse<uint8_t>   GrB_AINV_UINT8;
...
```

*_new functions not necessary (Type, Unary, Binary, etc)

GBTL supports domains through template parameters (same order as std::unary_function

Typedefs make it look more like the C API. I am not sure that these are necessary.

```
// Binary Operator
template<typename D1, typename D2 = D1, typename D3 = bool>
struct GreaterThan
{
   typedef D3 result_type;
   inline D3 operator()(D1 lhs, D2 rhs) { return lhs > rhs; }
};
```

`GrB_GreaterThan_<D1>`, returns bool by default but can be changed.

- NOTE: there is no penalty for user-defined operators when defined as functors like these.
- We need to investigate the use of lambda functions

Algebra: BinaryOp → UnaryOp, (from src/graphblas/algebra.hpp)

```
// Turn Binary Operator into Unary Operator by binding rhs
template <typename ConstT, typename BinaryOpT>
struct BinaryOp_Bind2nd
{
   typedef typename BinaryOpT::result_type result_type;

   BinaryOp_Bind2nd(ConstT const &value,
                    BinaryOpT     operation = BinaryOpT() )
      : n(value), op(operation)
   {}

   result_type operator()(result_type const &value)
   {
      return op(value, n);
   }

   ConstT n;
   BinaryOpT op;
};
```

You can turn a BinaryOp into a UnaryOp by binding an input operand (the second one in this example) to a constant “value”.

The UnaryOp interface

```
BinaryOp_Bind2nd<unsigned int,
                 Minus<unsigned int>> subtract_1(1);
```

Example: `subtract_1`: A UnaryOp that subtracts one from any element.

NOTE: same type of mechanism can be used to turn Semirings into Monoids or BinaryOps.

Algebra: Monoids (from src/graphblas/algebra.hpp)

```
// Monoids
#define GEN_GRAPHBLAS_MONOID(M_NAME, BINARYOP, IDENTITY)\
template <typename ScalarT>                              \
struct M_NAME                                            \
{                                                        \
public:                                                  \
   typedef ScalarT ScalarType;                           \
   typedef ScalarT result_type;                          \
                                                         \
   ScalarT identity() const {                            \
      return static_cast<ScalarT>(IDENTITY);             \
   }                                                     \
                                                         \
   ScalarT operator()(ScalarT lhs, ScalarT rhs) const { \
      return BINARYOP<ScalarT,                           \
                      ScalarT,                           \
                      ScalarT>()(lhs, rhs);              \
   }                                                     \
};


// Building a few common Monoids
GEN_GRAPHBLAS_MONOID(PlusMonoid, Plus, 0)
GEN_GRAPHBLAS_MONOID(TimesMonoid, Times, 1)
GEN_GRAPHBLAS_MONOID(MinMonoid, Min, numeric_limits<ScalarT>::max())
GEN_GRAPHBLAS_MONOID(MaxMonoid, Max, 0)
GEN_GRAPHBLAS_MONOID(LogicalOrMonoid, LogicalOr, false)
```

- We can generate monoid templates from any BinaryOp using this macro
- Using this generator is not necessary. Any class with this interface will work.
- NOTE: Monoid Interface satisfies the BinaryOp requirements: Monoids can be passed where BinaryOps are required.
- Enforcing one domain on the BINARYOP by specifying ScalarT three times.

Algebra: Semirings (from src/graphblas/algebra.hpp)

```
// Semirings
#define GEN_GRAPHBLAS_SEMIRING(SRNAME,ADD_MONOID,MULT_BINARYOP)\
template <typename D1, typename D2=D1, typename D3=D1>          \
class SRNAME                                                   \
{                                                              \
public:                                                        \
   typedef D3 ScalarType;                                      \
   typedef D3 result_type;                                     \
                                                               \
   D3 add(D3 a, D3 b) const                                    \
   { return ADD_MONOID<D3>()(a, b); }                          \
                                                               \
   D3 mult(D1 a, D2 b) const                                   \
   { return MULT_BINARYOP<D1,D2,D3>()(a, b); }                 \
                                                               \
   ScalarType zero() const                                     \
   { return ADD_MONOID<D3>().identity(); }                     \
};


// Building a few common Semirings
GEN_GRAPHBLAS_SEMIRING(ArithmeticSemiring, PlusMonoid, Times)
GEN_GRAPHBLAS_SEMIRING(LogicalSemiring, LogicalOrMonoid, LogicalAnd)
...
```

We can generate semiring templates from any Monoid and BinaryOp using this macro.

The MULT_BINARYOP could also be a monoid.

Using this generator is not necessary. Any class with this interface will work.

Descriptors (from graphblas/{operations.hpp, TransposeView.hpp, ComplementView.hpp})

```
template<typename MatrixT>
class TransposeView
{
public:
   typedef typename backend::TransposeView<
      typename MatrixT::BackendType> BackendType;

   TransposeView(BackendType backend_mat)
      : m_mat(backend_mat) {}

   // Implements entire Matrix interface passes to m_mat

private:
   BackendType m_mat;
};
```

Descriptors are replaced with the following

- GrB_TRAN: TransposeView “wraps” input matrices
- GrB_SCMP: ComplementView “wraps” mask
- GrB_REPLACE: bool `replace_flag` parameter

```
template<typename MatrixT>
inline TransposeView<MatrixT> transpose(MatrixT const &A)
{
   return TransposeView<MatrixT>(
      backend::transpose(A.m_mat));
}
```

Wrapper function used to instantiate the class properly. Not to be confused with transpose operation.

Mask structural complement done the same way with a “complement” wrapper function.

Example: mxv revisited

```
using namespace GraphBLAS;

IndexArrayType      iA = { 0, 0, ...};
IndexArrayType      jA = { 0, 1, ...};
std::vector<double> vA = {12, 7, ...};

Matrix<double, DirectedMatrixTag> A(4, 4);      // Declare a 4x4 matrix of doubles.

A.build(iA, jA, vA);                            // There are overloadings of build that take pointers.

IndexArrayType      iV = {0, 2};
std::vector<double> vV = {5, 8};

Vector<double, SparseTag> v(4);                 // Same approach for vectors.
v.build(iV, vV);

Vector<double> result(4);                       // Tag can be omitted to get a default type.

mxv(result,
    complement(v), NoAccumulate(),              // Use structural complement of v as a mask
    ArithmeticSemiring<double>(),               // Use transpose of A matrix
    transpose(A), v,                            // result<~v,REPLACE> = A' +.* v
    true);                                      // the REPLACE flag.
```

BFS algorithm: Appendix B.2 soup to nuts (from src/algorithms/bfs.hpp)

```
using namespace GraphBLAS;

template <typename MatrixT, typename WavefrontT, typename LevelListT>
void bfs_level(MatrixT const  &graph,
               WavefrontT      wavefront, // roots, copy made
               LevelListT     &levels)
{
   using T = typename MatrixT::ScalarType;

   // Assert graph is square/have a compatible shape with wavefront
   IndexType grows(graph.nrows());
   IndexType gcols(graph.ncols());
   IndexType wsize(wavefront.size());

   if ((grows != gcols) || (wsize != grows)) throw DimensionException();

   IndexType depth = 0;
   while (wavefront.nvals() > 0) {
      ++depth; // Increment the level

      // Apply the level to all newly visited nodes
      BinaryOp_Bind2nd<IndexType, Times<IndexType>> apply_depth(depth);

      apply(levels, NoMask(), Plus<unsigned int>(), apply_depth, wavefront, true);

      // Advance the wavefront and mask out nodes already assigned levels
      mxv(wavefront, complement(levels), NoAccumulate(),
          LogicalSemiring<IndexType>(),
          transpose(graph), wavefront, true);
   }
}
```

BFS algorithm to compute depth at which each node is reached. Root is level 1.

Bind Times BinaryOp with a constant to create UnaryOp for apply

mxv used to illustrate transpose wrapper.

Questions?