GBTL-CUDA: Graph Algorithms and Primitives for GPUs

Peter Zhang, Samantha Misurda, Marcin Zalewski, Scott McMillan, Andrew Lumsdaine
What is this talk about?

• **Graph BLAS**
  – an emerging paradigm for graph computation
  – programs new graph algorithms in a highly abstract *language of linear algebra*.
  – executes in a wide variety of programming environments

• **Our implementation of Graph BLAS**
  – Graph BLAS Template Library (GBTL)
  – High-level C++ frontend
  – Switchable backends: CUDA and sequential
  – Released at: [https://github.com/cmu-sei/gbtl](https://github.com/cmu-sei/gbtl)

Graph BLAS Forum  
[http://www.graphblas.org](http://www.graphblas.org)

Software Engineering Institute  
Carnegie Mellon University

CREST  
Indiana University
Graph algorithms meet hardware: love lost in translation?

Te quiero

Mitä ihmettä?!
Current Graph Algorithms

High Level Algorithm (BFS, MIS, MST, SSSP)

Sequential
- for loops
- contiguous memory
- ...

Data Parallel
- Parallel kernels
- PCI Memory transfer
- ...

Distributed
- Message passing
- Workload balancing
- ...

Implementation Concerns

CPU
CUDA
MPI
It’s the representation…

- Adjacency matrix representation of graphs in graph theory (Kőnig 1931)
- Matrices?!
BLAS for Linear Algebra

Application

Unified BLAS Interface

X86-specific BLAS Optimization

X86 Architecture

ARM-specific BLAS Optimization

ARM Architecture

PPC-specific BLAS Optimization

PPC Architecture
Graph BLAS

Same Application Cross Platform

Unified Graph BLAS Interface

Same Interface Cross Platform

Architecture Specific Concerns

- OpenMP
- Pthreads
- Shared memory

- Asynchronous memory transfer
- Texture memory
- Dynamic Parallelism

- Memory coalescing
- Barrier control

CPU

CUDA

MPI
Graph BLAS

- A community effort to define a set of primitives used to describe graph algorithms using sparse linear algebra

- Rich data structures, algebraic abstraction
  - Sparse adjacency matrices represent graphs
  - Semirings to define specific behavior
Graph BLAS: Semiring

- The standard arithmetic semiring:
  \[ \langle D, +, \times, 0, 1 \rangle \]

- The “minPlus” semiring, for BFS with parents:
  \[ \langle D, \min, +, \infty, 0 \rangle \]
Graph BLAS: BFS example

- Breadth-first Search (BFS): can be represented by matrix-vector multiplications in linear algebra
- Wavefront: vector
- One multiplication operation results in the next wave front
Graph BLAS: BFS traversal

![Graph BLAS: BFS traversal](image)

<table>
<thead>
<tr>
<th>start vertex</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>A^T v</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>end vertex</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Visited map</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ A^T v = \]
Graph BLAS Template Library (GBTL)

• A C++ implementation of Graph BLAS
  – Allows for generic programming and metaprogramming

• A frontend-backend design
  – Uniform frontend for algorithm abstraction
    • generic semantic checks
    • simplifies the templates
  – Hardware-specific backend optimized for different architecture

• A separation of concerns: render unto hardware experts hardware-specific optimizations
GBTL: Frontend and Backend

- Graph BLAS API: boundary between the algorithms and hardware-specific implementations
- Frontend forwards calls to backend namespace via C++ templates
  - performs generic semantic checks and implementation independent operations
  - simplifies templates passed in by user for meta programming

<table>
<thead>
<tr>
<th>Graph Analytic Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graph Algorithms</td>
</tr>
<tr>
<td>GraphBLAS API (Separation of Concerns)</td>
</tr>
<tr>
<td>Graph Primitives (tuned for hardware)</td>
</tr>
<tr>
<td>Hardware Architecture</td>
</tr>
</tbody>
</table>

graphblas namespace

graphblas::backend namespace

Forward
GBTL: Algorithm Example

```plaintext
// wavefront initialized with root vertex = 0
bfs(graph, wavefront) {
  vector visited = wavefront;

  while(!wavefront.empty()) {
    // increment level in next wavefront
    wavefront = vXm(wavefront,
                   graph,
                   Add1NotZero);

    // filter out already visited vertices
    // if the vertices have values less than
    // current level in visited vector
    wavefront = eWiseMult(wavefront,
                          visited,
                          Mult);

    // update visited vector by filtered wavefront
    visited = eWiseAdd(wavefront,
                       visited,
                       throwException);
  }

  return visited;
}
```
GBTL: Frontend Matrix

- Frontend Matrix class: an opaque data structure, uniform across backends

  Frontend Matrix Object Construction:

  Matrix <double, DenseMatrixTag, DirectedMatrixTag> matrix(...);

- User can provide hints to frontend Matrix at construction time through parameter pack, backend can make decisions based on hints

- Backend Matrix classes: specialized for hardware and implementation
GBTL: Frontend Matrix Class

```cpp
// TagsT template parameters provide hints
template <typename ScalarT, typename... TagsT>
class Matrix {
public:
    // Empty construction; fixed dimensions
    Matrix((IndexType num_rows, IndexType num_cols);

    // Other frontend matrix interface...

private:
    // immutable dimensions:
    IndexType const m_num_rows, m_num_cols;

    // opaque backend implementation
    backend::Matrix<
        ScalarType,
        detail::SparsenessCategoryTagT,
        TagsT...
    > m_matrix;
};
```
GBTL: Algorithm Example

```c
// wavefront initialized with root vertex = 0
bfs(graph, wavefront) {
    vector visited = wavefront;

    while(!wavefront.empty()) {
        // increment level in next wavefront
        wavefront = vXm(wavefront, graph, Add1NotZero);

        // filter out already visited vertices
        // if the vertices have values less than
        // current level in visited vector
        wavefront = eWiseMult(wavefront, visited, Mult);

        // update visited vector by filtered wavefront
        visited = eWiseAdd(wavefront, visited, throwException);
    }

    return visited;
}
```
**GBTL: vXm, semiring overloading**

```cpp
template<typename AVectorT,
         typename BMATRIXT,
         typename CVectorT,
         typename SemiringT = graphblas::ArithmeticSemiring<T>,
         typename AccumT = graphblas::math::Assign<T> >
inline void vxm(AVectorT const &a,
    BMATRIXT const &b,
    CVectorT &c,
    SemiringT s = SemiringT(),
    AccumT accum = AccumT())
{
    vector_multiply_dimension_check(a, b.get_shape());
    backend::vxm(a.m_vec, b.m_mat, c.m_vec, s, accum);
}
```

**“Add1NotZero” Semiring:**

- **Domain D:**
  - Addition: $D = \langle D, \oplus, \odot \rangle$
  - Multiplication: $\odot$
  - Additive Identity: $0$
  - Multiplicative Identity: $1$

```cpp
T SelectAdd1(T first, T second) {
    return first == 0? 0: ++first;
}
```
GBTL: Algorithm Example

```c
// wavefront initialized with root vertex = 0
bfs(graph, wavefront) {
  vector visited = wavefront;
  while(!wavefront) {
    // increment wavefront
    wavefront = wavefront ⊗ visited;
    // filter out already visited vertices
    // if the vertices have values less than
    // current wavefront
    visited = wavefront ⊕ visited;
    // update visited vector by filtered wavefront
    visited = eWiseAdd(wavefront, visited, throwException);
  }
  return visited;
}
```
GBTL: GPU Backend

- Implemented using CUSP: parallel algorithms and data structures for sparse linear algebra
- CUSP: built on top of the Thrust, a C++ library with GPU programming primitives
- Generalization meets performance
GPU Backend Data Type: Sparse Matrix

- We use sparse matrices to improve storage efficiency
- Sparse matrices have unstored elements called \textit{structural zeros}
- Different sparse matrix formats: Compressed Sparse Row (CSR), Coordinate (COO), List Of Lists (LIL)
- Backend makes decision on format-of-choice, based on hardware layout
Sparse Matrix Example: COO
A tale of three vectors

- COO matrices: enables easy stream processing
  - Regularity in data layout

\[
\begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 7 \\
2 & 4 & 5 & 7 & 6 & 1 & 3 & 4 & 5 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1
\end{array}
\]
GPU Backend: mXm Scaling

mXm scaling in Millions of Floating-point Operations Per Second (MFLOPS)

- Erdős–Rényi graphs
- Average of 16 runs
GPU Backend: BFS Performance

- We test runtime of our BFS algorithm on several real world graphs

<table>
<thead>
<tr>
<th>Graph Name</th>
<th># Vertices</th>
<th># Edges</th>
<th>Runtime(ms)</th>
<th>MTEPS(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Journals</td>
<td>124</td>
<td>12,068</td>
<td>5.76</td>
<td>2.1</td>
</tr>
<tr>
<td>G43</td>
<td>1,000</td>
<td>19,980</td>
<td>14.61</td>
<td>1.4</td>
</tr>
<tr>
<td>ship_003</td>
<td>121,728</td>
<td>3,777,036</td>
<td>558.95</td>
<td>6.8</td>
</tr>
<tr>
<td>belgium_osm</td>
<td>1,441,295</td>
<td>3,099,940</td>
<td>10502.4</td>
<td>0.3</td>
</tr>
<tr>
<td>roadNet-CA</td>
<td>1,971,281</td>
<td>5,533,214</td>
<td>4726.21</td>
<td>1.2</td>
</tr>
<tr>
<td>delaunay_n24</td>
<td>16,777,216</td>
<td>100,663,202</td>
<td>65507.7</td>
<td>1.5</td>
</tr>
</tbody>
</table>

*MTEPS = Millions of Traversed Edges Per Second*
GBTL: API Overhead

Environment:
• NVIDIA GPU
• Overhead of API call compared against direct CUSP call

Methodology:
• Average of 16 runs on Erdős–Rényi graphs generated using the same dimension and sparsity
Recap and Future Plans

• **Graph BLAS:**
  – Graph algorithms on sparse linear algebra primitives

• **Graph BLAS Template Library (GBTL):**
  – Extensibility meets performance
  – Abstraction layer: translator with low overhead penalty
  – Proof-of-concept: it works well!

• **Future Plans**
  – Multi-GPU backend
  – Distributed CPU/GPU backend
  – Participate in community discussion on future specifications
Thank You

Questions?