

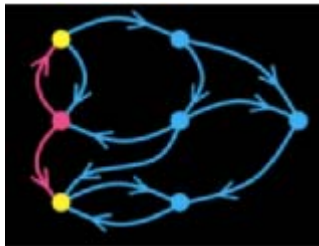
GBTL-CUDA: Graph Algorithms and Primitives for GPUs

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>



Graph BLAS Forum
<http://www.graphblas.org>



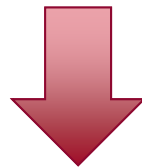
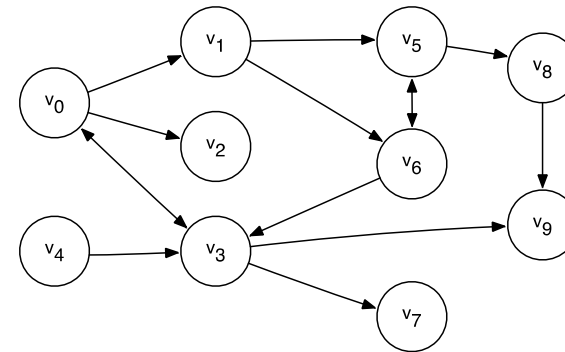
Software Engineering Institute
Carnegie Mellon University



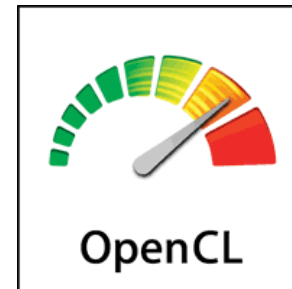
CREST
Indiana University

Graph algorithms meet hardware: love lost in translation?

Te quiero



Mitä ihmettä?!



Current Graph Algorithms

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

Implementation
Concerns

Sequential

- for loops
- contiguous memory
- ...

Data Parallel

- Parallel kernels
- PCI Memory transfer
- ...

Distributed

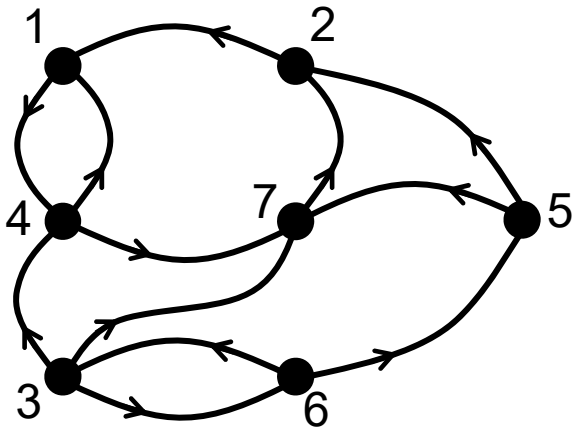
- Message passing
- Workload balancing
- ...

CPU

CUDA

MPI

It's the representation...



A^T

	<u>start vertex</u>						
	1	2	3	4	5	6	7
<u>end vertex</u>				•			
1				•			
2	•						
3				•		•	•
4	•						•
5		•					•
6			•		•		
7		•					

- Adjacency matrix representation of graphs in graph theory (König 1931)
- Matrices?!

BLAS for Linear Algebra

Application

Unified BLAS Interface

X86-specific BLAS
Optimization

ARM-specific BLAS
Optimization

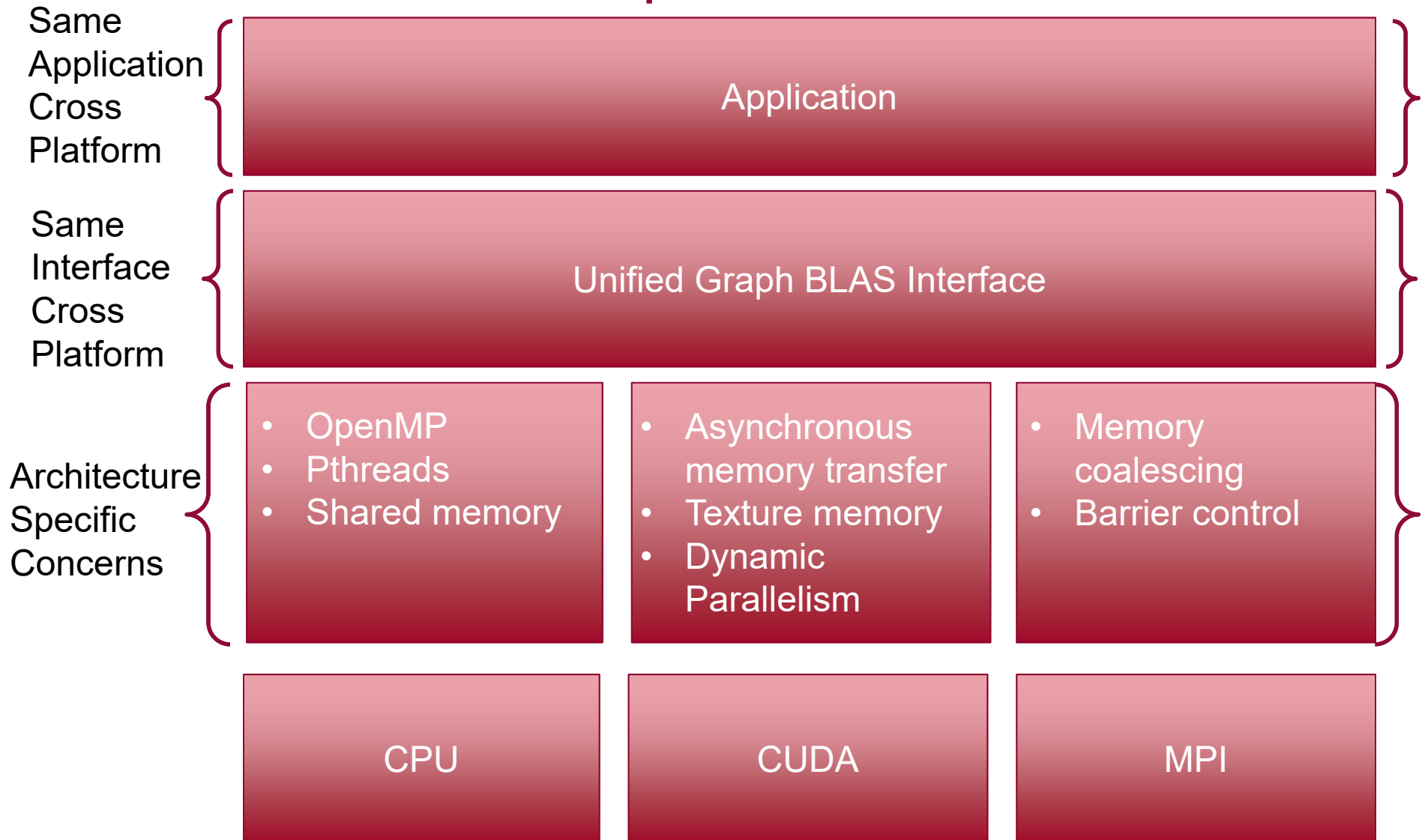
PPC-specific BLAS
Optimization

X86 Architecture

ARM Architecture

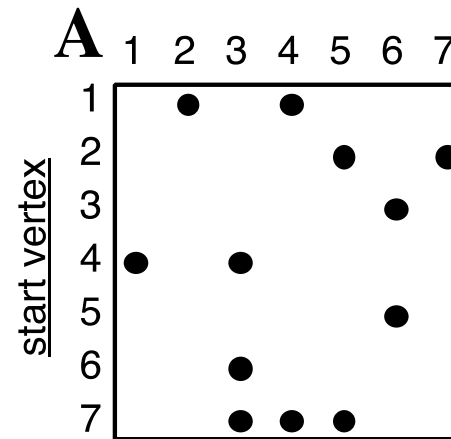
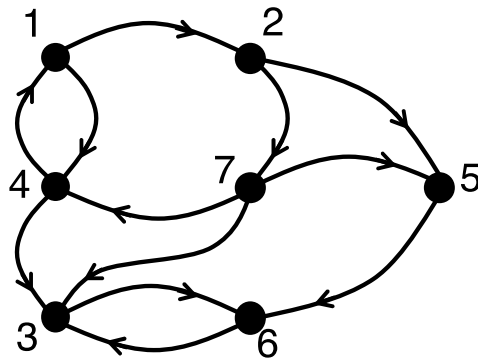
PPC Architecture

Graph BLAS



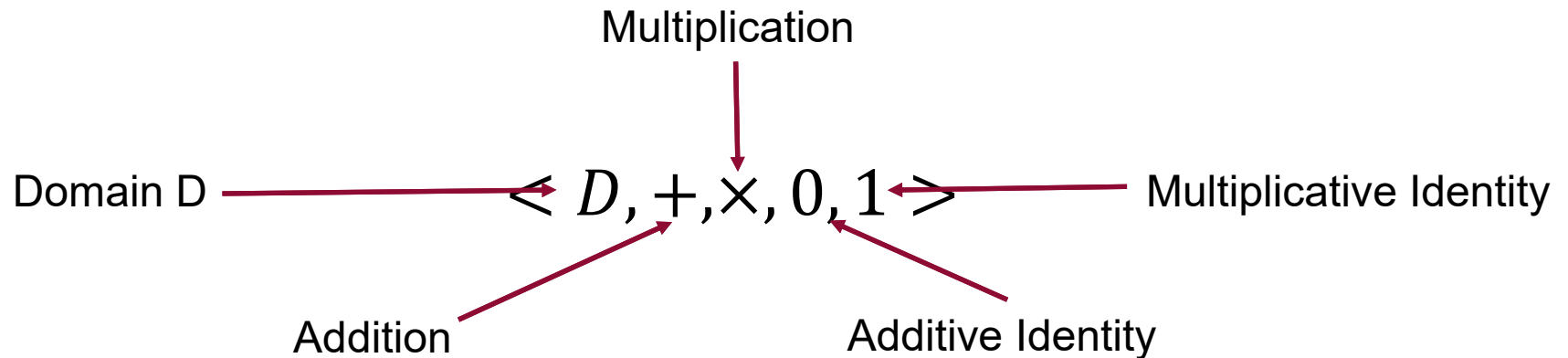
Graph BLAS

- 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

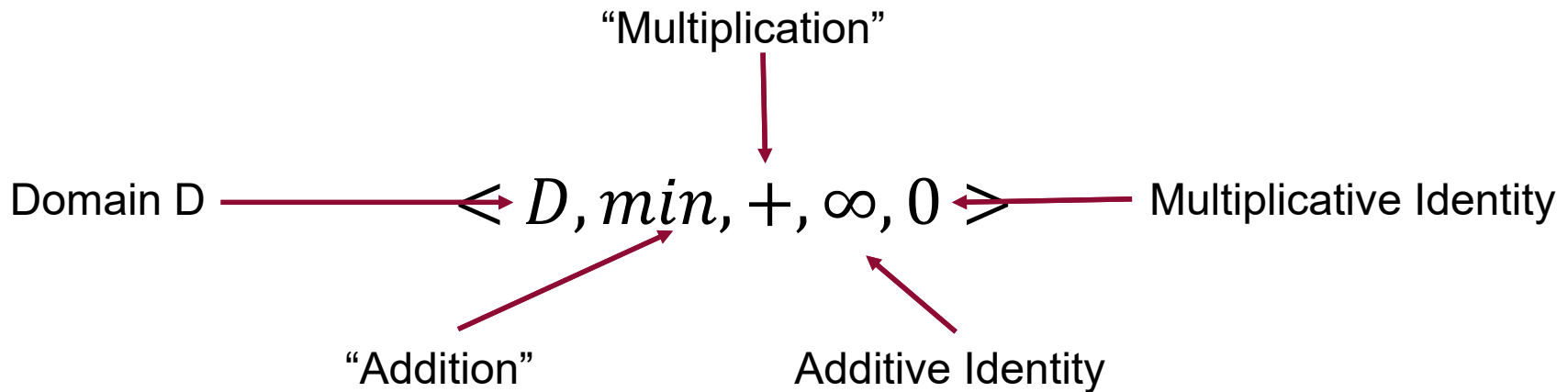


Graph BLAS: Semiring

- The standard arithmetic semiring:

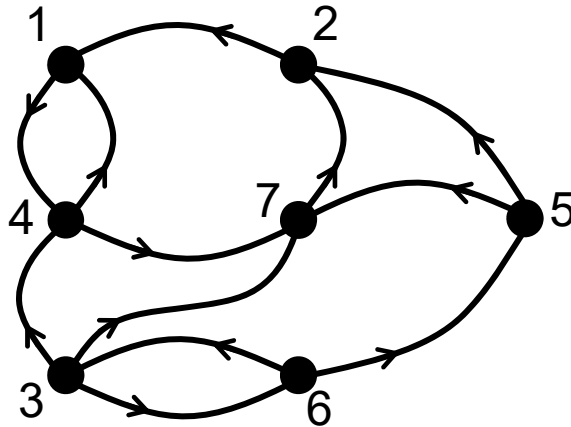


- The “*minPlus*” semiring, for BFS with parents:



Graph BLAS: BFS example

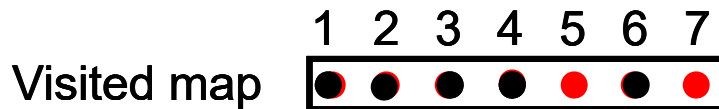
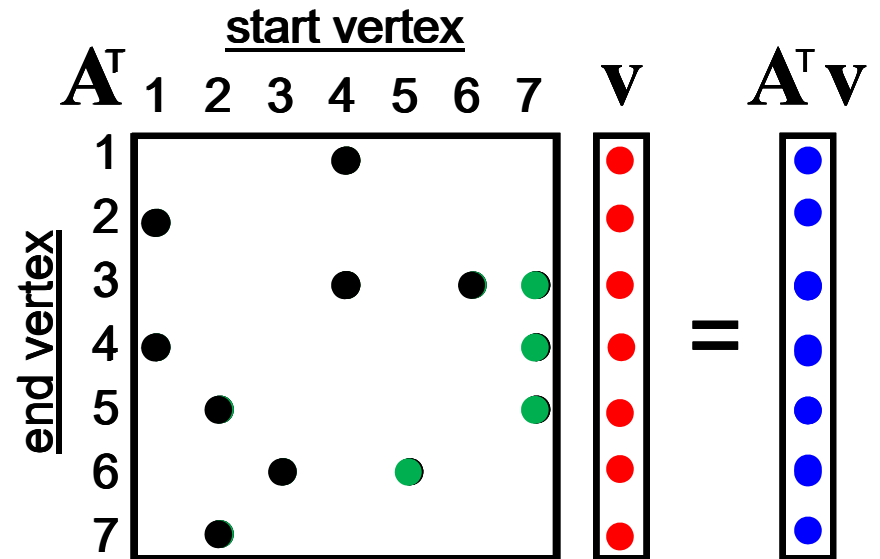
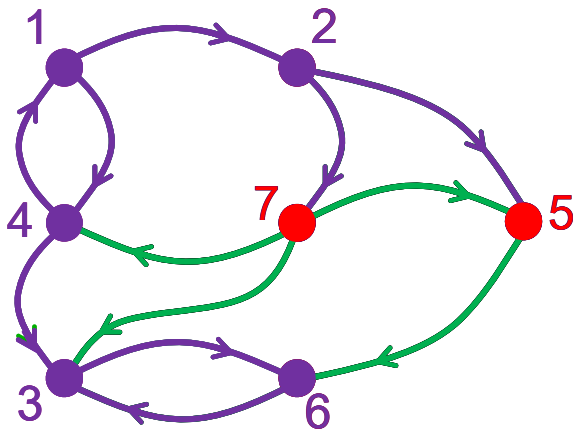
- Breadth-first Search (BFS): can be represented by matrix-vector multiplications in linear algebra
- Wavefront: vector
- One multiplication operation results in the next wave front



\mathbf{A}^T

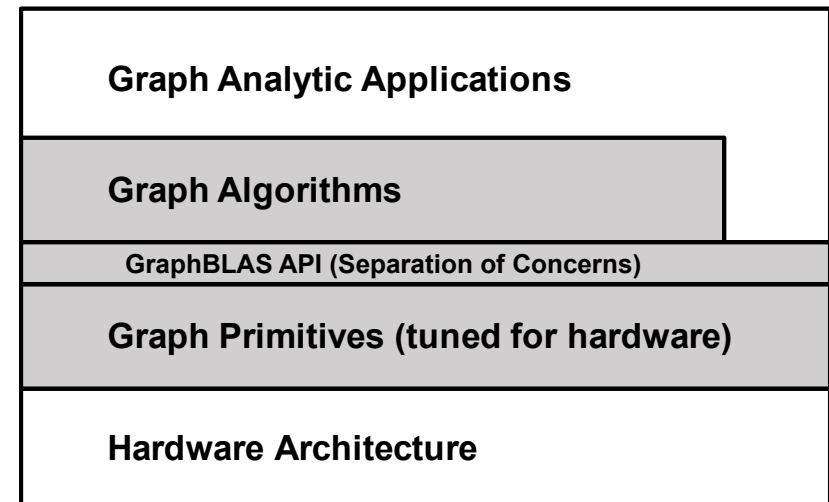
	<u>end vertex</u>						
	1	2	3	4	5	6	7
<u>start vertex</u>				•			
1							
2	•						
3				•		•	•
4	•						•
5		•					•
6			•		•		
7		•					

Graph BLAS: BFS traversal



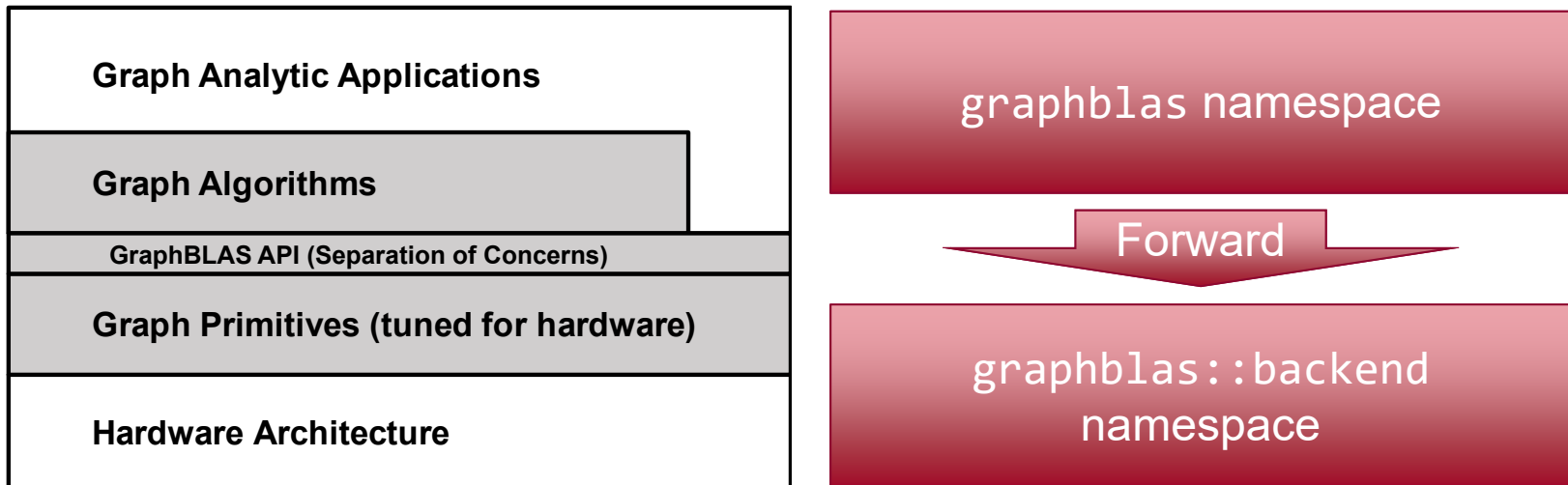
Graph BLAS Template Library (GBTL)

- 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



GBTL: Algorithm Example

```
1 // wavefront initialized with root vertex = 0
2 bfs(graph, wavefront) {
3     vector visited = wavefront;
4
5     while(!wavefront.empty()) {
6         // increment level in next wavefront
7         wavefront = vXm(wavefront,
8                         graph,
9                         Add1NotZero);
10
11         // filter out already visited vertices
12         // if the vertices have values less than
13         // current level in visited vector
14         wavefront = eWiseMult(wavefront,
15                               visited,
16                               Mult);
17
18         //update visited vector by filtered wavefront
19         visited = eWiseAdd(wavefront,
20                            visited,
21                            throwException);
22     }
23     return visited;
24 }
```

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

```
1 // TagsT template parameters provide hints
2 template <typename ScalarT, typename... TagsT>
3 class Matrix :
4     public backend::Matrix<ScalarT, TagsT...>
5 {
6 public:
7     typedef ScalarT ScalarType;
8
9     // Empty construction; fixed dimensions
10    Matrix(IndexType num_rows, IndexType num_cols);
11
12    // Other frontend matrix interface...
13
14 private:
15    // immutable dimensions:
16    IndexType const m_num_rows, m_num_cols;
17
18    // opaque backend implementation
19    backend::Matrix<
20        ScalarType,
21        detail::SparsenessCategoryTagT,
22        TagsT...>
23    m_matrix;
24 };
```

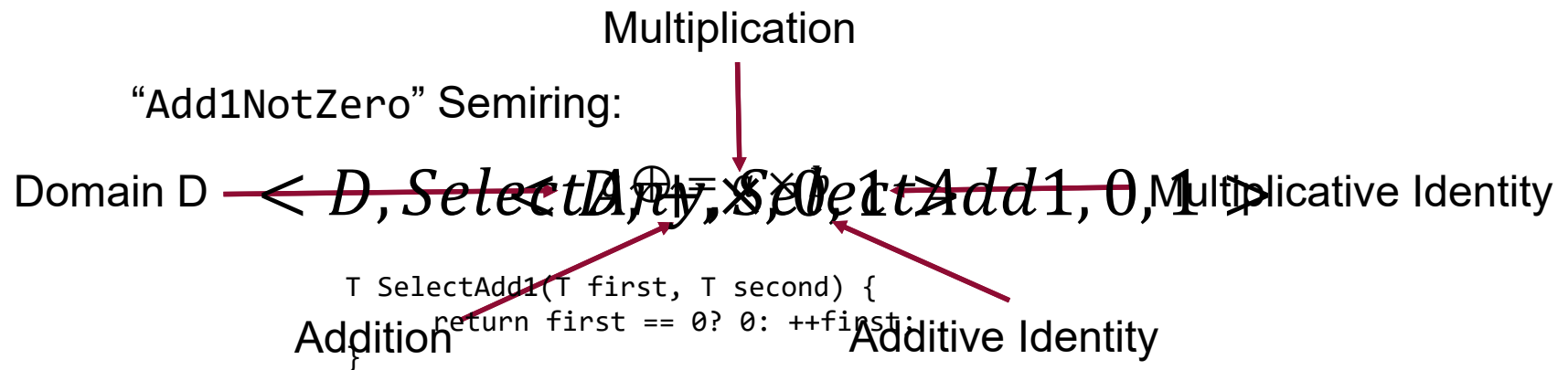

GBTL: Algorithm Example

```
1 // wavefront initialized with root vertex = 0
2 bfs(graph, wavefront) {
3     vector visited = wavefront;
4
5     while(!wavefront.empty()) {
6         // increment level in next wavefront
7         wavefront = vXm(wavefront,
8                         graph,
9                         Add1NotZero);
10
11        // filter out already visited vertices
12        // if the vertices have values less than
13        // current level in visited vector
14        wavefront = eWiseMult(wavefront,
15                              visited,
16                              Mult);
17
18        //update visited vector by filtered wavefront
19        visited = eWiseAdd(wavefront,
20                           visited,
21                           throwException);
22    }
23    return visited;
24 }
```

GBTL: vXm, semiring overloading

```

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);
}
    
```



GBTL: Algorithm Example

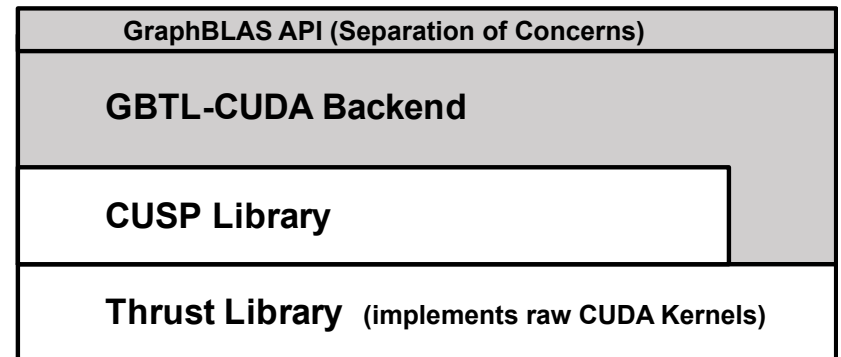
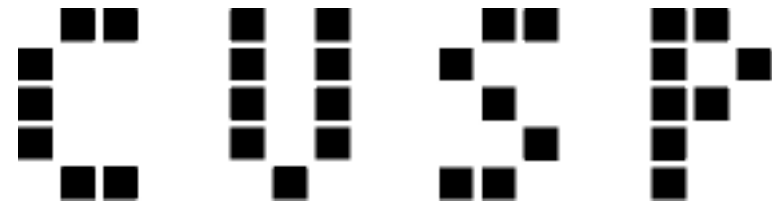
```
1 // wavefront initialized with root vertex = 0
2 bfs(graph, wavefront) {
3     vector visited;
4
5     while(!wavefront.empty()) {
6         // increment wavefront
7         wavefront++;
8
9         Add(visited, wavefront);
10
11        // filter out already visited vertices
12        // if the vertices have values less than
13        // current wavefront value
14        wavefront = wavefront & visited;
15
16        //update visited vector by filtered wavefront
17        visited = eWiseAdd(wavefront,
18                           visited,
19                           true);
20    }
21    return visited;
22 }
23 }
24 }
```

wavefront =
wavefront \otimes visited

visited =
wavefront \oplus visited

GBTL: GPU Backend

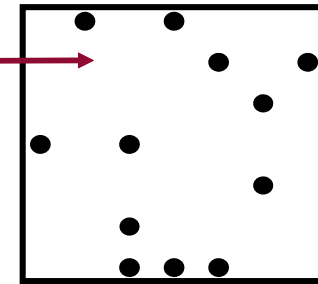
- Implemented using CUSP: parallel algorithms and data structures for sparse linear algebra
- CUSP: built on top of the Thrust, a C++ library with GPU programming primitives
- Generalization meets performance



GPU Backend Data Type: Sparse Matrix

- We use sparse matrices to improve storage efficiency
- Sparse matrices have unstored elements called *structural zeros*

Blank space in matrix

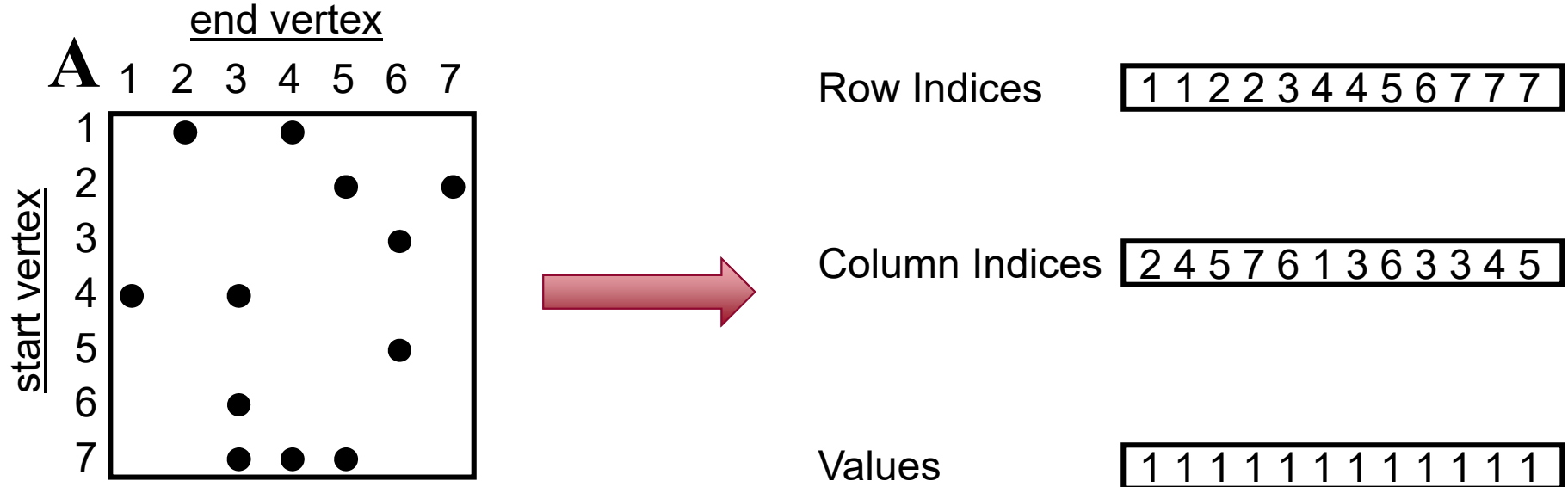


- Different sparse matrix formats: Compressed Sparse Row (CSR), Coordinate (COO), List Of Lists (LIL)
- Backend makes decision on format-of-choice, based on hardware layout

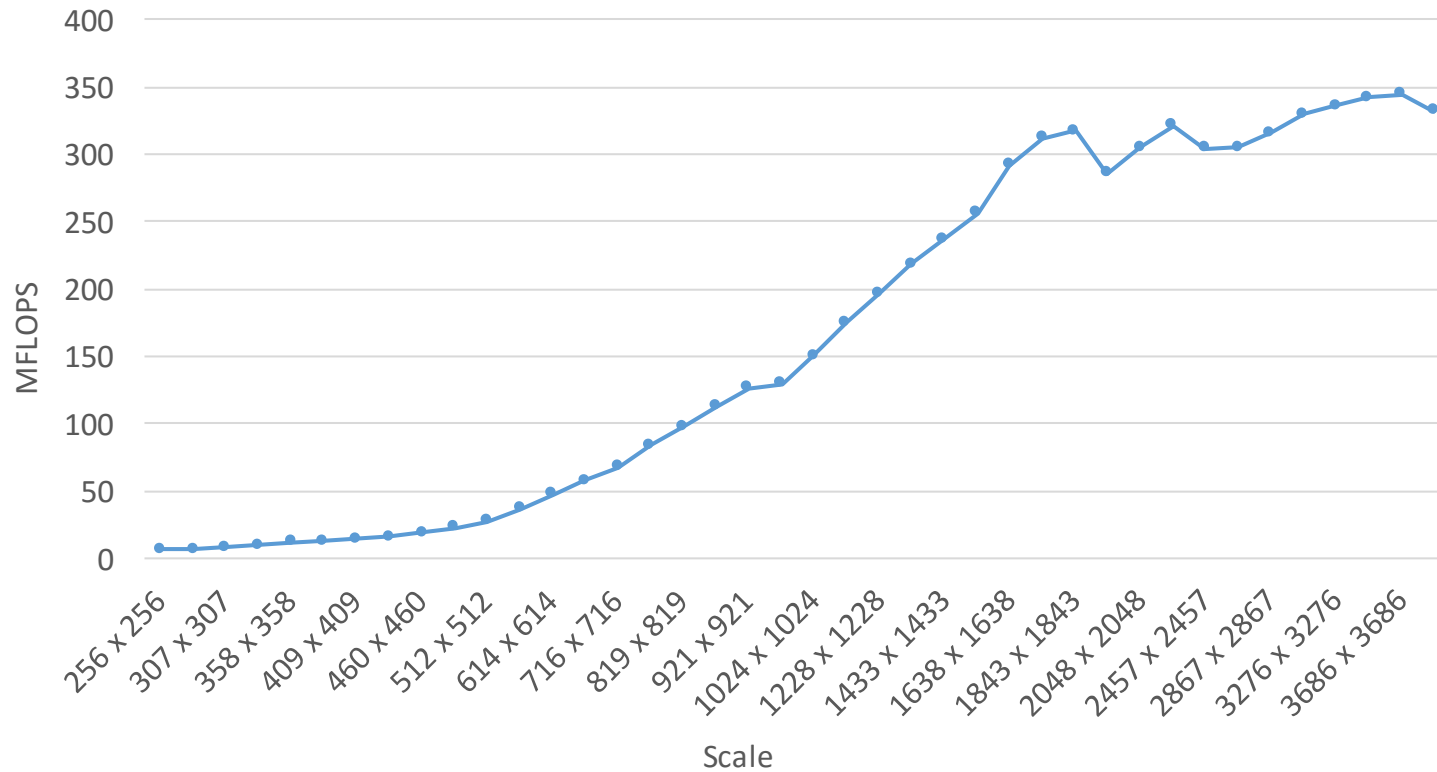
Sparse Matrix Example: COO

A tale of three vectors

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



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

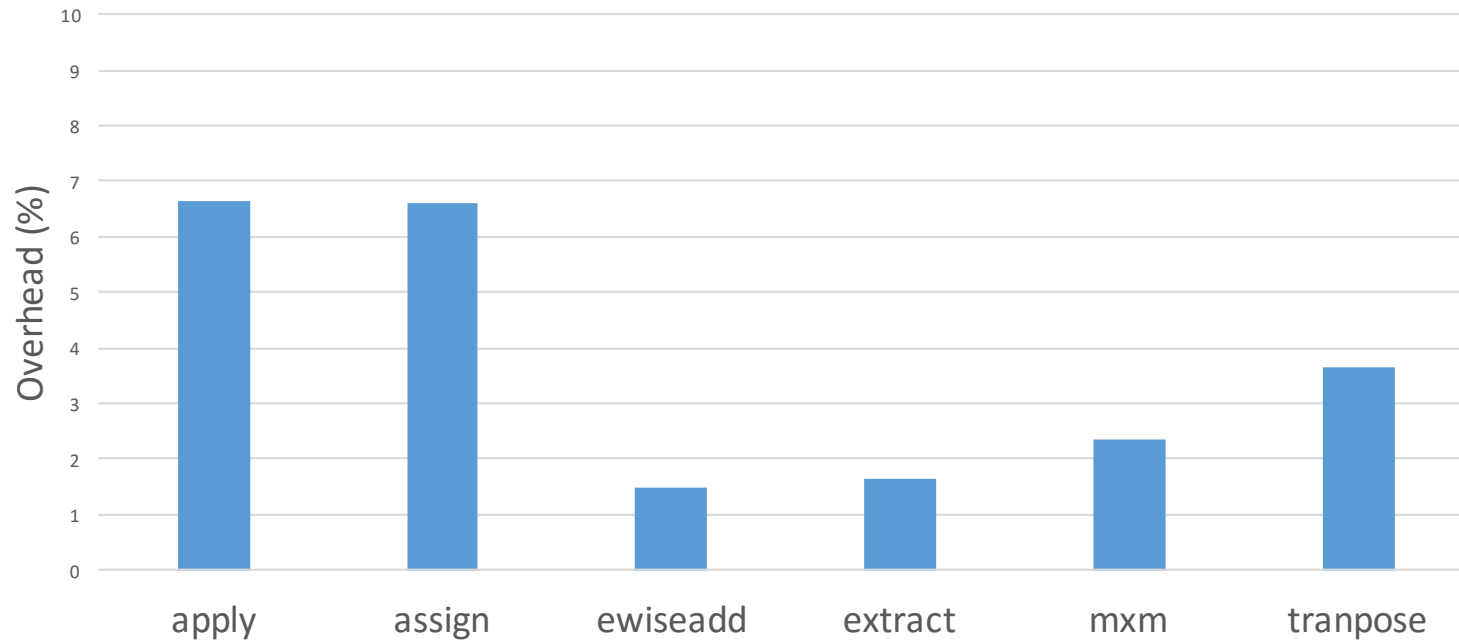
- We test runtime of our BFS algorithm on several real world graphs

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

*MTEPS = Millions of Traversed Edges Per Second

GBTL: API Overhead

GraphBLAS Template Library 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?

