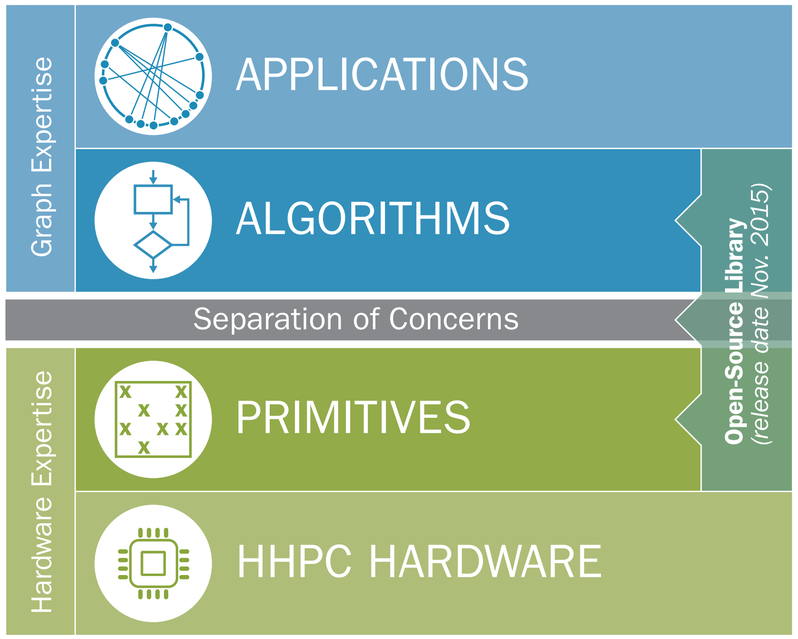
Graph Algorithms on Future Architectures

Fast, efficient graph analysis is important and pervasive. However, achieving high levels of performance is exceedingly difficult especially in the era of complex heterogeneous high-performance computing (HHPC) architectures. By defining a set of graph primitives and operations, we are able to separate the concerns between the graph expertise needed to develop advanced graph analytics and the hardware expertise needed to achieve high levels of performance on the ever-increasing complexity of the underlying hardware.



The software architecture: the abstraction layer (gray) separates the concerns between the expertise needed to develop graph algorithms and applications (above), and intimate knowledge of the hardware needed for high performance (below).

Overview. For the last two years, the members of the Emerging Technology Center at the SEI have been collaborating with graph analytics experts at Indiana University to identify and implement a set of primitives and operations to separate the concerns between graph analytics development and the increasing complexity of programming for the underlying hardware. During that time, we have joined with other leading experts from industry, academia and government to create an application programming interface (API) standard, now called the GraphBLAS (Graph Basic Linear Algebra Subprograms), that will capture this separation of concerns (http://graphblas.org).

Graph Algorithms: Simplified by GraphBLAS API

Algorithms Implemented with Less Code. We are developing a library of graph algorithms that are implemented in terms of the new operations and data primitives currently defined by the GraphBLAS API. Classes of algorithms include

- Metrics: e.g., degree, diameter, centrality, triangle counting
- Traversals: Breadth-First Search (BFS)
- Shortest Path/Cost Minimization
- Community Detection/Clustering
- Connected Components
- (Minimum) Spanning Tree
- Maximum Flow
- PageRank

Separation of Concerns: GraphBLAS API Spec

Standardization In Progress. Researchers from the SEI, industry, academia, and the U.S. government are developing the API specification:

- The mathematical properties are defined by semi-ring algebra.
- Nine operations are specified currently (see right).
- The key primitive type is the sparse matrix.
- We are exploring extensions to this set of operations that can offer greater expressivity and greater opportunities for tuning.

Tuning the sparse matrix multiplies (MxM, MxV, VxM) is key to achieving performance on underlying hardware. Many different sparse formats already exist, and the "best" format depends on both the underlying hardware architecture and operation performed.

Graph Primitives: Tuned for GPU Architectures

Collaboration with Indiana University. Researchers including Andrew Lumsdaine from the Center for Research in Exascale Technologies have been collaborating with the SEI on this project to explore efficient implementations of graph primitives. The graph at the right shows the performance of our BFS algorithm (orange) using a compressed, sparse row matrix format on the newest generation of GPU cards using dynamic parallelism. It is compared to best-in-class implementations reported in the literature.

Future Work.

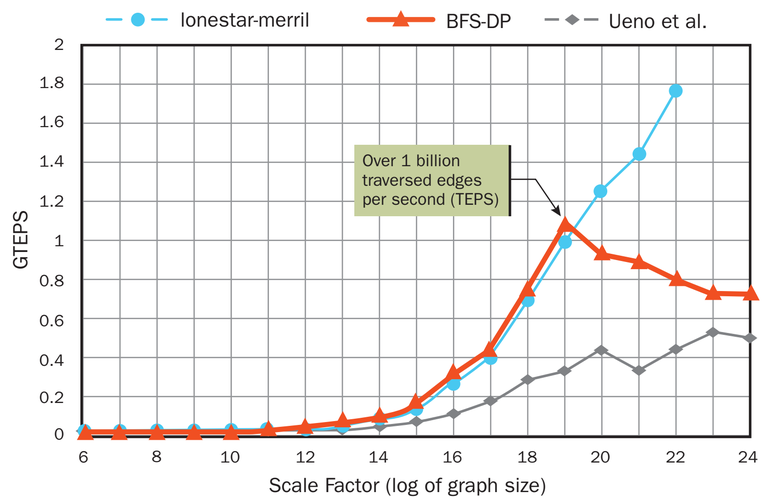
- Continued participation in the GraphBLAS standardization effort
- Addressing scaling issues for larger graphs
- Developing distributed primitives to support multiple GPU nodes.
- Tuning for a variety of different sparse matrix formats that will be required for high performance across a wide range of algorithms
- Future versions that include sparse solvers to support other important algorithms (e.g., PCA, graph partitioning)

```
void bfs(SparseMatrix<bool> const &graph,
         Vector<bool>              wavefront,
         Vector<int>              &level)
{
   visited = wavefront;
   level_val = 0;

   while (!wavefront.empty()) {
      // traverse one level from current wavefront
      wavefront   = VxM(wavefront, graph, LogicalSemiring);

      // compute which nodes have NOT been visited before
      not_visited = Apply(visited, LogicalNot);
      wavefront   = EWiseMult(not_visited, wavefront, LogicalAnd);
      visited     = EwiseAdd(visited, wavefront, LogicalOr);

      // Assign the level to all newly visited vertices
      level_val++;
      level      += EwiseMult(wavefront, level_val, Mutiply);
   }
}
```

The BFS algorithm implemented using only five GraphBLAS operations. With the masking extension proposed for matrix multiplies, BFS could be implemented with only three operations.

Operation Name	Description
BuildMatrix	Build a sparse matrix from row, column, value tuples
ExtractTuples	Extract the row, column, value tuples from a sparse matrix
MxM, MxV, VxM	Perform sparse matrix *multiplication* (e.g., BFS traversal)
Extract	Extract a sub-matrix from a larger matrix (e.g., sub-graph selection)
Assign	Assign to a sub-matrix of a larger matrix (e.g., sub-graph assignment)
EwiseAdd, EwiseMult	Element-wise *addition* and *multiplication* of matrices (e.g., graph union, intersection)
Apply	Apply *unary function* to each element of matrix (e.g., edge weight modification)
Reduce	*Reduce* along columns or rows of matrices (vertex degree)
Transpose	Swaps the rows and columns of a sparse matrix (e.g., reverse directed edges)

The principle GraphBLAS operations (as of 9/17/2015).



BFS performance reported by P. Zhang, et al.,, "Dynamic Parallelism for Simple and Efficient GPU Graph Algorithms," submitted to 5th IEEE Workshop on Irregular Applications: Architectures and Algorithms, Nov 2015.

Contact: Scott McMillan smcmillan@sei.cmu.edu