Design Considerations for a GraphBLAS Compliant Graph Library on Clusters of GPUs

July 11, 2016

Carl Yang, University of California, Davis
Aydın Buluço, Lawrence Berkeley National Laboratory
John D. Owens, University of California, Davis
Instead of writing a graph algorithm from scratch, users can express algorithm using a small number of GraphBLAS kernels.

**Hardware-agnostic:**
Users do not have to relearn many different APIs in order to run graph algorithms on different hardware.

**Efficiency:**
Kernels are implemented with performance guarantees.
Why use GPUs for graphs?

Improving faster than traditional CPUs

P100 (2016): ~8 TFLOPS, ~1TB/s, ~3800 ALUs, 32GB mem.
CPU-to-GPU bandwidth: ~80GB/s-200GB/s (NVLink)
Xeon Phi 7250 (2016): ~6 TFLOPS, ~400GB/s, ~288 threads, 16GB mem.
CPU-to-coprocessor bandwidth: ~100GB/s (Omni-Path)
Challenges with using GPUs for graph analysis

- Hard to program
- Small memory

CPU RAM available ~100x larger

Solved by GraphBLAS API
Solved by using multi-GPU
Goal: Implement $m \times v$ on multiple GPUs following GraphBLAS standard

Same as SpMV
Same as $v \times m$, which computes $y^T = x^T A$

Input: Given adjacency matrix $A$, vector $x$

Want: Compute $y = A^T x$

Current GraphBLAS API:

```c
GrB_info GrB_mXv( GrB_Vector *y, const GrB_Semiring s, const GrB_Matrix A, const GrB_vector x )
```

Def: Think of a **semiring** as an object with a “$\times$” operator, “$+$” operator, and identity for each
Applications of SpMV
PageRank
MIS
And many more!
BFS Using SpMV (Iteration 1)

Each iteration, 1) compute $y = A^T x$
2) update BFS result
3) use BFS result to filter $y$
4) stop if BFS result is unchanged
Each iteration, 1) compute $y = A^T x$
2) update BFS result
3) use BFS result to filter $y$
4) stop if BFS result is unchanged
Each iteration, 1) compute $y = A^T x$  
2) update BFS result  
3) use BFS result to filter $y$  
4) stop if BFS result is unchanged  

new: -2
Each iteration, 1) compute \( y = A^T x \)  
2) update BFS result old: -2  
3) use BFS result to filter \( y \)  
4) stop if BFS result is unchanged
BFS Using SpMV (Iteration 2)

Each iteration, 1) compute $y = A^T x$
2) update BFS result
3) use BFS result to filter $y$
4) stop if BFS result is unchanged

<table>
<thead>
<tr>
<th>BFS</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Each iteration, 1) compute $y = A^T x$
2) update BFS result
3) use BFS result to filter $y$
4) stop if BFS result is unchanged

<table>
<thead>
<tr>
<th>BFS</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Each iteration, 1) compute $y = A^T x$
2) update BFS result
3) use BFS result to filter $y$
4) stop if BFS result is unchanged

<table>
<thead>
<tr>
<th>BFS</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
BFS Using SpMV (Iteration 3)

Each iteration, 1) compute $y = A^T x$
2) update BFS result old: 4
3) use BFS result to filter $y$
4) stop if BFS result is unchanged
BFS Using SpMV (Iteration 3)

Each iteration,
1) compute $y = A^T x$
2) update BFS result
3) use BFS result to filter $y$
4) stop if BFS result is unchanged

- Iteration 3:
  - $A^T x = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix}$
  - $x = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 0 \end{bmatrix}$
  - $y = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$

New BFS result: 8
Old BFS result: 4
Each iteration, 1) compute \( y = A^T x \)  
2) update BFS result \( new: 8 \) \( old: 8 \)  
3) use BFS result to filter \( y \)  
4) stop if BFS result is unchanged
2. How to solve multi-way merge
3. Partitioning scheme
1. Row-based SpMV

\[
\begin{array}{cccc}
0 & 0 & 1 & 0 \\
4 & 0 & 0 & 0 \\
0 & 0 & 1 & 2 \\
1 & 4 & 3 & 0 \\
0 & 5 & 2 & 0 \\
\end{array}
\]

\[
\begin{array}{c}
x \\
y \\
\end{array} = \begin{array}{c}
0 \\
0 \\
2 \\
1 \\
4 \\
\end{array}
\]

- 1) Elementwise multiply
- 2) Reduce
- Work: \( O(nnz) \)
  - Independent of vector sparsity
provide row-based SpMV GPU implementations

Why not Row-based?

Why implement our own?
**A Motivating Example**

<table>
<thead>
<tr>
<th>$A^T$</th>
<th>$x$</th>
<th>$y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 0 1 0 0 2 2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4 0 0 0 0 1 3 0 0 2</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>0 0 1 2 1 0 0 0 7 9</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1 4 3 0 0 2 4 3 3 9</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>0 5 2 0 0 2 0 0 0 0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1 2 0 0 0 0 2 2 0 0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>3 3 2 0 0 2 0 0 0 0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>0 0 2 0 0 0 0 2 3 0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0 0 0 0 0 3 0 0 0 0</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>3 0 2 2 9 0 0 0 0 0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- Take scalar from vector $x$, and multiply it by the corresponding column from $A$
### Column-based SpMV

#### Calculation

\[
\begin{align*}
A^T \times x &= y_3 + y_4 + y_5 = y \\
\begin{bmatrix}
0 & 0 & 1 & 0 & 0 \\
4 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 2 & 1 \\
1 & 4 & 3 & 0 & 0 \\
0 & 5 & 2 & 0 & 0 \\
\end{bmatrix}
\begin{bmatrix}
x_1 \\
x_2 \\
x_3 \\
x_4 \\
x_5 \\
\end{bmatrix}
&= \\
\begin{bmatrix}
0 & 2 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
2 & 2 & 2 & 4 & 4 \\
2 & 6 & 0 & 0 & 0 \\
4 & 4 & 0 & 0 & 0 \\
\end{bmatrix}
\begin{bmatrix}
y_1 \\
y_2 \\
y_3 \\
y_4 \\
y_5 \\
\end{bmatrix}
&= \\
\begin{bmatrix}
2 \\
0 \\
2 \\
8 \\
6 \\
4 \\
\end{bmatrix}
\end{align*}
\]

#### Steps

1. Gather and elementwise multiply
2. Multiway-merge (many elementwise adds)

#### Work

\[
O \left( \sum_{x[v] \neq 0} d_v + kn \right)
\]

- Suitable for sparse vectors, due to dependence on vector sparsity
For GraphBLAS folks a column-based SpMV and automatically decide which one to use?

This is the idea behind direction-optimizing BFS
1. Column-based SpMV
2. How to solve multiway-merge
3. Partitioning scheme
Input: sorted segments, with possible duplicates

E.g. [4 5 6 4 5 6 3 4 5]

Want: unique

E.g. [4 5 6 3]
(+): Addition operator of semiring
(x): Multiplication operator of semiring

Input: Sparse Vector x
Sparse Matrix $A^T$

Gather columns from $A^T$

(+) ewiseMult

(x) Multiway-merge

Output: Sparse Vector y
Multiway-merge

Input: Sparse Vector $x$
Sparse Matrix $A^T$

Gather columns from $A^T$

$(+)$ ewiseMult

$(x)$ Multiway-merge

Output: Sparse Vector $y$

$(x)$ Multiway-merge

Radix Sort

$(x)$ Segmented Reduce
Radix sort works best

Graph showing runtime in milliseconds against edges traversed in millions. The graph compares No Sorting, Radix Sort, and Merge Sort.
Importance of multiway-merge

Multiway-merge takes ~85.6% of application runtime

Gather and ewiseMult take 11ms
For GraphBLAS folks

automatically decide which one to use?

This is the idea behind direction-optimizing BFS

Should sparse vector be allowed to contain duplicate entries?

This is the idea behind Gunrock's idempotent discovery that lets them get away without solving multiway-merge completely

Some duplicates are allowed
1. Column-based SpMV
2. How to solve multiway-merge
3. Partitioning scheme
Each iteration:
1) GPU $i$ computes $y^{(i)} = A_i x_i$
2) Computes histogram of send counts
3) Send to other GPUs using MPI_Alltoall with send counts
4) Send to other GPUs using MPI_Alltoallv
5) Each GPU multiway-merges piece they are responsible for
3. Partitioning scheme

Each iteration:
1) GPU $i$ computes $y^{(i)} = A_i x_i$
2) Computes histogram of send counts
3) Send to other GPUs using MPI_Alltoall with send counts
4) Send to other GPUs using MPI_Alltoallv
5) Each GPU multiway-merges piece they are responsible for
3. Partitioning scheme

Each iteration:
1) GPU $i$ computes $y^{(i)} = A_i x_i$
2) Computes histogram of send counts
3) Send to other GPUs using MPI_Alltoall with send counts
4) Send to other GPUs using MPI_Alltoallv
5) Each GPU multiway-merges piece they are responsible for
3. Partitioning scheme

Each iteration:
1) GPU $i$ computes $y^{(i)} = A_i x_i$
2) Computes histogram of send counts
3) Send to other GPUs using MPI_Alltoall with send counts
4) Send to other GPUs using MPI_Alltoallv
5) Each GPU multiway-merges piece they are responsible for
3. Partitioning scheme

Each iteration:
1) GPU $i$ computes $y^{(i)} = A_i x_i$
2) Computes histogram of send counts
3) Send to other GPUs using MPI_Alltoall with send counts
4) Send to other GPUs using MPI_Alltoallv
5) Each GPU multiway-merges piece they are responsible for
On GPU 1, 2, ..., p:

Block Matrix (Multi-GPU)

Input: Sparse Vector $x_i$
Sparse Matrix $A_i$

$mXv$ (Single GPU)

Histogram

MPI_Alltoall

MPI_Alltoallv

Multiway-merge

Output: Sparse Vector $y_i$

$mXv$ (Single GPU)

Gather columns from $A_i$

ewiseMult

Multiway-merge

Why 2 multiway-merges?
Experimental Setup

**CPU**
- 64x 16-core AMD Opteron 6274 CPU
- 32 GB of main memory

**GPU**
- 64x NVIDIA K20X GPUs
- 14 SMs, 192 ALUs per SM, 732 MHz, 6 GB on-board memory

**Interconnect**
- Gemini interconnect between nodes
- 10.4 GB/s (HyperTransport 3.0)
Strong Scaling

Billions of Edges Traversed (GTEPS)

Ideal Scaling
Actual Scaling

Number of Processors

1 2 4 8 16 32 64
Weak Scaling (Vertex)

Billions of Edges Traversed per Processor (GTEPS)

Number of Processors

Ideal Scaling

Actual Scaling

1  2  4  8  16  32
For 4 GPUs, everything looks fine

- Majority of runtime is compute
  - Two biggest iterations brought down from 121ms to 40ms
Compute time > Communication

MPI calls comprise a sliver of runtime
At 16 GPUs, picture becomes bleak

MPI calls take up ~50% runtime now
Not overlapping compute with communication.

GPUs sit idle to wait for other GPUs to finish compute.
Communication vs. Compute

How to bridge this gap?
Work in progress

1. Overlap compute with communication
   Use multiple cudaStreams to make communication asynchronous

2. Try other partitioning schemes
   Variants of 1D column-based partitioning
   2D partitioning

3. Partition graph better
   Reverse Cuthill–McKee
Both have their uses

Should sparse vector tolerate duplicate entries if user does not care about intermediate sparse vectors?
Acknowledgements

This work is funded by:

DARPA XDATA program
US Army award W911QX-12-C-0059

DARPA STTR award D15PC00010

Department of Energy, Office of Science, ASCR Contract No. DE-AC02-05CH11231
Aydın Buluç, Lawrence Berkeley National Laboratory

Questions?

abuluc@lbl.gov

John D. Owens, University of Davis, California

jowens@ece.ucdavis.edu
Applications of Interest

- Bioinformatics
- Social network analysis
- Computer vision
- Fraud detection
- Urban planning
Challenges with Graph Analysis

Software complexity
Structural properties vary

Small-world:
Difficult to partition
Impacts scalability

Mesh:
Limits scalability of synchronous graph algorithms
1. Row-based mXv (SpMV)

- GPU threads compute each row simultaneously
- Work: $O(nnz)$
  - Suitable for multiplication by dense vector, because work is independent of vector sparsity
  - Direction-optimizing or bottom-up BFS is a special case of this
Dominance of Big Iterations in BFS of Scale-free Graphs

Two biggest iterations take 91% of runtime
- Other six iterations take 6ms combined