SuiteSparse: GraphBLAS

Graph algorithms in the language of linear algebra

Tim Davis, Texas A&M University

Talk given at SIAM CSE21, March 2021
SuiteSparse:GraphBLAS, a Parallel Implementation of the GraphBLAS API

<table>
<thead>
<tr>
<th>function name</th>
<th>description</th>
<th>GraphBLAS notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>GrB_mexm</td>
<td>matrix-matrix mult.</td>
<td>C(M) = C ⊙ AB</td>
</tr>
<tr>
<td>GrB_vxm</td>
<td>vector-matrix mult.</td>
<td>w'(m') = w' ⊙ u'A</td>
</tr>
<tr>
<td>GrB_mxv</td>
<td>matrix-vector mult.</td>
<td>w(m) = w ⊙ Au</td>
</tr>
<tr>
<td>GrB_eWiseMult</td>
<td>element-wise, set-intersection</td>
<td>C(M) = C ⊙ (A ⊙ B)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>w(m) = w ⊙ (u ⊙ v)</td>
</tr>
<tr>
<td>GrB_eWiseAdd</td>
<td>element-wise, set-union</td>
<td>C(M) = C ⊙ (A ⊕ B)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>w(m) = w ⊙ (u ⊕ v)</td>
</tr>
<tr>
<td>GrB_extract</td>
<td>extract submatrix</td>
<td>C(M) = C ⊙ A(i,j)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>w(m) = w ⊙ u(i)</td>
</tr>
<tr>
<td>GrB_assign</td>
<td>assign submatrix</td>
<td>C(M)(i,j) = C(i,j) ⊙ A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>w(m)(i) = w(i) ⊙ u</td>
</tr>
<tr>
<td>GxB_subassign</td>
<td>assign submatrix</td>
<td>C(i,j)(M) = C(i,j) ⊙ A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>w(i)(m) = w(i) ⊙ u</td>
</tr>
<tr>
<td>GrB_apply</td>
<td>apply unary op.</td>
<td>C(M) = C ⊙ f(A)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>w(m) = w ⊙ f(u)</td>
</tr>
<tr>
<td>GxB_select</td>
<td>apply select op.</td>
<td>C(M) = C ⊙ f(A,k)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>w(m) = w ⊙ f(u,k)</td>
</tr>
<tr>
<td>GrB_reduce</td>
<td>reduce to vector reduce to scalar</td>
<td>w(m) = w ⊙ [⊕jA(:,j)]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>s = s ⊙ [⊕ijA(i,j)]</td>
</tr>
<tr>
<td>GrB_transpose</td>
<td>transpose</td>
<td>C(M) = C ⊙ A'</td>
</tr>
<tr>
<td>GxB_kron</td>
<td>Kronecker product</td>
<td>C(M) = C ⊙ kron(A, B)</td>
</tr>
</tbody>
</table>

caveat: GraphBLAS notation under revision

GrB_Matrix and Vector: opaque data structures

8 internal formats: 4 formats, each held by row or column:

- **sparse**: compressed sparse vector, like MATLAB.
  a matrix is a dense vector of n sparse vectors
  good general-purpose format

- **hypersparse**: a sparse vector of sparse vectors
  can support huge graphs \( n = 2^{60} \).
  also good for extracting subgraphs.

- **bitmap**: a dense matrix for the values, with an extra dense boolean matrix to describe the pattern.
  good for vectors and tall-and-thin matrices.

- **full**: just like LAPACK, but also int, bool, user-defined...

Parallel algorithms via OpenMP (CUDA in progress):
highly specialized, depending on the data structures.

- **C<M>=A*B**: 79 different parallel algorithms, for each of 1,498 built-in semirings, plus a generic one for user-defined operators. 79*1499 = over 118 thousand C=A*B kernels.
So what can you do with these parallel kernels? Lots!

Breadth-first-search (initialization):
q = {source} ; parent = [ size n, all zero ]
parent (source) = source

Traditional BFS:
while (q not empty)
  for each i in frontier q
    for each edge (i,j)
      if (j not yet seen)
        add j to next q
        parent (j) = i
        flag j as seen

GraphBLAS BFS: using the \textbf{ANY–SECONDI} semiring
while (q not empty)
  q<\text{parent}> = A’ * q \quad \text{masked parallel matvec}
  parent<q> = q \quad \text{masked parallel assignment}

\textbf{SECONDI} multiplier: \( z = A(i,k)*q(k) = k \), the parent node id
\textbf{ANY} additive operator: \textbf{ANY} function: any(x,y) = x or y, arbitrary choice

So what can you do with these parallel kernels? Lots!
Parallel matrix-matrix multiply

- masked dot product: $C<M>=A' \times B$
- unmasked dot product: $C=A' \times B$ or $C<\neg M>A' \times B$
- saxpy-style, $C=A \times B$, $C<M>=A \times B$, or $C<\neg M>=A \times B$. Mix of 4 kinds of tasks:
  - coarse Gustavson: $C(:,j1:j2) = A \times B(:,j1:j2)$ with $O(n)$ workspace
  - fine Gustavson: $C(:,j) = A \times B(:,j)$ with many threads, atomics and shared $O(n)$ workspace
  - coarse Hash: $C(:,j1:j2) = A \times B(:,j1:j2)$ with $O(f)$ workspace, $f = \text{max "flops" for any } C(:,j)$
  - fine Hash: $C(:,j) = A \times B(:,j)$ with many threads, uses atomics and shared $O(f)$ hash space
  - all four tasks in any $C=A \times B$

$A \times B$ all variants: total 9K lines of code, not including 320K lines of generated code for 1,499 semirings

References:
Fine Hash/Gustavson task with mask $C<\mathbf{M}> = A \times B$:

- Each thread given a range $i_1:i_2$ of rows of $B$:
  
  $$C(:,j) += A \times B(i_1:i_2,j)$$

- Each hash entry contains a row index $i$ and 2-bit atomic state.

Fine Hash tasks:
- Phase1: scatter $M$ into hash
- Phase2: numerical work
- Phase3/4: count $C(:,1:m)$
- Phase5: gather from hash
Fine Hash/Gustavson task: with negated mask M

\[ M(i, j) = 1 \]
\[ M(i, j) = 1 \]

- Phase 2: ignore
- Scatter mask
- Initial state
- Lock
- Unlock
- C(ij) seen
- \( X(i) \) initialized
- \( M(i, j) = 0 \)
Parallel assignment: \( C_{<M>}(i,j) = A \quad C_{<M>}(i,j) += A \)

- A blizzard of combinations:
  - mask: present or not, complemented or not, structural or not
  - replace option: true or false
  - accumulator: present or not
  - A: matrix or scalar
  - S: constructed or not
  - C, M, A: sparse/hypersparse(bitmap/full), by row/col

- Algorithms:
  - some use \( S = C(i,j) \), symbolic extraction. Given \( C(I,J) = A \) where \( I \) and \( J \) are vectors of indices.
  - \( C(I(2),J(3)) = A(2,3) \), then \( S(2,3) = \) position of \( C(I(2),J(3)) \) in the data structure for \( C \).
  - Allows for \( C [ S(x,y) ] = A(x,y) \) assignment for some row \( x \) and column \( y \).
  - some algorithms do not use \( S \) and thus do not construct it.
Parallel assignment: \( C^<M>(I,J)=A \), using \( S \)

About 40 different algorithms. Most are 2-pass. For example: \( C^<M>(I,J)=A \), with \( S \):

- sort \( I \) and \( J \) index lists, if needed, and remove duplicates; permute \( A \) if changed
- \( S = C(I,J) \), a parallel structural extraction, does not use the mask \( M \).
- Symbolic analysis: construct parallel tasks for 1st and 2nd passes
- First pass: Iterate through all of set union of \( (A,S) \), like \( A+S \).
  - For each entry found in set union \( A+S \), lookup \( M \). If false, skip it. Otherwise:
  - if both \( A \) and \( S \) present: assign \( C[S(i,j)] = A(i,j) \), updating the existing value
  - if \( A \) present but not \( S \): \( C[S(i,j)] = A(i,j) \) must be added to \( C \) as a new entry: pending tuple (count them)
  - if \( S \) present but not \( A \): \( C[S(i,j)] \) must be deleted: mark it for deletion (a zombie)
- Middle pass: cumulative sum of all pending tuple counts, for all tasks
- Second pass: repeat the algorithm, but only insert pending tuples into the pile
## MATLAB: native sparse matrices vs @GrB objects

<table>
<thead>
<tr>
<th>@GrB vs MATLAB syntax</th>
<th>@GrB advantages / limitations</th>
<th>@GrB speedup relative to MATLAB native on 20 cores</th>
</tr>
</thead>
</table>
| \( C = A \times B \)   | @GrB: any semiring, any mask  
MATLAB: just plus-times  
caveat: Tim D wrote both | Up to 30x |
| \( C(I, J) = A \)      | Same syntax, more types: sparse int8, int16, ..., single complex, ... | 2x to 1000x |
| \( C = \text{sparse} \ (2^{60}, 2^{60}) \)  
\( C = \text{GrB} \ (2^{60}, 2^{60}) \) | MATLAB: too big  
@GrB: no problem; hypersparse |  |
| \( C(M) = A \)        | MATLAB mask: same syntax, @GrB much faster | MATLAB: > one week +  
GraphBLAS: 7 seconds, 100,000x speedup |

Intel® Xeon® E5-2698 v4 CPU with 20 cores and 40 threads, with gcc 5.4.0. Caveat: MATLAB R2018a, I need to upgrade.
### Performance of BFS:

Time in seconds, NVIDIA DGX Station (Intel Xeon, 20 hardware cores, 40 threads)

<table>
<thead>
<tr>
<th></th>
<th>Urand</th>
<th>Kron</th>
<th>Twitter</th>
<th>Web</th>
<th>Road</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAP</td>
<td>0.58</td>
<td>0.31</td>
<td>0.22</td>
<td>0.34</td>
<td>0.25</td>
</tr>
<tr>
<td>SuiteSparse</td>
<td>1.22</td>
<td>0.52</td>
<td>0.33</td>
<td>0.66</td>
<td>3.32</td>
</tr>
</tbody>
</table>

### Performance of Betweenness Centrality:

<table>
<thead>
<tr>
<th></th>
<th>Urand</th>
<th>Kron</th>
<th>Twitter</th>
<th>Web</th>
<th>Road</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAP</td>
<td>46.4</td>
<td>31.5</td>
<td>10.8</td>
<td>3.0</td>
<td>1.5</td>
</tr>
<tr>
<td>SuiteSparse</td>
<td>32.7</td>
<td>23.6</td>
<td>9.25</td>
<td>8.2</td>
<td>34.4</td>
</tr>
</tbody>
</table>

GAP, by Scott Beamer: 6 parallel kernels, fastest method in most cases; but difficult code to write, not a user library.

SuiteSparse:GraphBLAS: also parallel, simple to write, sometimes faster; easy code to write, able to write “any” algorithm
### Performance of PageRank:

time in seconds, NVIDIA DGX Station (Intel Xeon, 20 hardware cores, 40 threads)

<table>
<thead>
<tr>
<th></th>
<th>Urand</th>
<th>Kron</th>
<th>Twitter</th>
<th>Web</th>
<th>Road</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAP</td>
<td>25.3</td>
<td>19.8</td>
<td>15.2</td>
<td>5.1</td>
<td>1.0</td>
</tr>
<tr>
<td>SuiteSparse</td>
<td>22.2</td>
<td>22.2</td>
<td>17.2</td>
<td>9.3</td>
<td>1.3</td>
</tr>
</tbody>
</table>

### Performance of Triangle Counting:

<table>
<thead>
<tr>
<th></th>
<th>Urand</th>
<th>Kron</th>
<th>Twitter</th>
<th>Web</th>
<th>Road</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAP</td>
<td>21.8</td>
<td>374.1</td>
<td>79.6</td>
<td>22.2</td>
<td>0.03</td>
</tr>
<tr>
<td>SuiteSparse</td>
<td>34.0</td>
<td>918.0</td>
<td>239.6</td>
<td>34.7</td>
<td>0.23</td>
</tr>
</tbody>
</table>

GAP: about tied with GraphBLAS for PageRank. About 3x faster than GraphBLAS for TC.
SuiteSparse: not yet fully exploiting non-blocking mode, so `L=tril(A); C<L>=L'*L ; nt=sum(C)` constructs C then sums it up.
Performance of Connected Components:

time in seconds, NVIDIA DGX Station (Intel Xeon, 20 hardware cores, 40 threads)

<table>
<thead>
<tr>
<th></th>
<th>Urand</th>
<th>Kron</th>
<th>Twitter</th>
<th>Web</th>
<th>Road</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAP</td>
<td>1.7</td>
<td>0.53</td>
<td>0.23</td>
<td>0.22</td>
<td>0.05</td>
</tr>
<tr>
<td>SuiteSparse</td>
<td>4.5</td>
<td>3.4</td>
<td>1.5</td>
<td>2.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Performance of Single-Source Shortest Paths:

<table>
<thead>
<tr>
<th></th>
<th>Urand</th>
<th>Kron</th>
<th>Twitter</th>
<th>Web</th>
<th>Road</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAP</td>
<td>7.2</td>
<td>4.9</td>
<td>2.0</td>
<td>0.81</td>
<td>0.21</td>
</tr>
<tr>
<td>SuiteSparse</td>
<td>25.5</td>
<td>17.4</td>
<td>8.5</td>
<td>9.6</td>
<td>46.8</td>
</tr>
</tbody>
</table>

SuiteSparse: parallel code, easy to write, but typically 3x to 4x slower than the GAP, still worse for the Road graph, for Connected Components and Single-Source-Shortest-Paths.
GraphBLAS non-blocking mode

```c
GxB_select (&L, ... GxB_TRIL, A, ...); // L=tril(A,-1)
GxB_select (&U, ... GxB_TRIU, A, ...); // U=triu(A,1)
GrB_mxm (C, L, NULL, GxB_PLUSPAIR_INT64, L, U, GrB_DESC_ST1); // C<L>=L*U'
GrB_reduce (s, NULL, GrB_PLUS_INT64_MONOID, C, NULL); // s=sum(C) as GrB_Scalar
GrB_free (&C); GrB_free (&L); GrB_free (&U); // C, L, U now known to be temporary
GrB_extractElement (&ntriangles, s); // ntriangles as int64_t
```

- non-blocking API allows intermediate matrices to not be instantiated
- allows for depenency DAG with fusion and lazy evaluation
- no need to form L, U, and C
- not yet exploited in SuiteSparse:GraphBLAS. In progress.
Matrix-based API vs Vertex-centered API

It's not **either-or**, it's **both-and**

See Roger Pearce's talk, this session
In summary: GraphBLAS strengths & limitations

Strengths:

- avoids “for all j in Adj(i) ...” loops; akin to triply-nested loops vs C=A*B
- simple high-level API; bulk operations give lots of power to underlying implementation
- typically simple algorithms; most parallel graph algorithms can be expressed in linear algebra
- non-blocking mode in API: can fuse kernels and skip instantiating intermediate results
- SuiteSparse:GraphBLAS:
  - some asynchronous features can be expressed (ANY monoid)
  - no loss of performance in Python vs C API; nearly same in MATLAB
  - parallel performance can rival or even beat highly-tuned graph libraries

Limitations:

- no “for all j in Adj(i) ...” loops, but can work side-by-side with vertex-centered libraries
- some algorithms hard to express (Depth-First-Search, Afforest CC, ...)
- SuiteSparse:GraphBLAS: non-blocking mode: just zombies, pending tuples, & lazy sort so far
- fully asynchronous methods hard to express (PageRank with Gauss-Seidel, for example) ... but might be possible to extend the API
SuiteSparse: GraphBLAS

Graph algorithms in the language of linear algebra

Tim Davis, Texas A&M University