Hornet: An Efficient Data Structure for Dynamic Sparse Graphs and Matrices

Oded Green
Hornet

- A scalable and dynamic data structure for
  - Sparse data
  - Graph algorithms
  - Linear algebra based problems
- Formerly known as cuSTINGER
  - Hornet initialization is hundreds of times faster
  - Hornet updates are 4X-10X faster
  - The Hornet data structure offers is more robust and scalable than cuSTINGER.
- Essentially a dynamic CSR data structure
- Easy to use
“Separation of powers”

• Dynamic graph data structure and dynamic graph algorithms are in two different repositories
  – Easy to integrate with external library
  – Can also be used with matrices

• This talk focuses on the data structure
Graph Primitives – Upfront summary

• Great performance for static and dynamic graph algorithms

• Scalable

• Simple to use
  • Will discuss algorithm framework later today
    – 1:00pm
    – Same room as this talk
Hornet – Upfront Summary

• Can support over **150 million** updates per second
• Can easily scale to graphs with billions of vertices
• CSR comparison
  – Initializing is also relatively in-expensive – usually less than 3X slower
  – Hornet requires 30% more storage
  – **Identical** performance
• COO (edge-list) comparison
  – Hornet requires 20% less storage
  – Hornet has better locality
Big Data problems need Graph Analysis

Communication networks:
- World-wide connectivity
- High velocity changes
- Different types of extracted data:
  - Physical communication network.
  - Person-to-person communication network.

Financial networks:
- Transactions between players.
- Different transactions types (property graph)

Health-Care networks:
- Various players.
- Pattern matching and epidemic monitoring.
- Problem sizes have doubled in last 5 years.
Hornet Properties

✓ A Simple programming model
  ✓ Enable algorithm designers to implement dynamic & streaming graph algorithms with ease.

✓ Can easily grows 1000X initial size (no restart needed)

✓ Millions of updates per second to graph
  ✓ Updates are not bottlenecks for analytics.

✓ Automated data management
  ✓ Transfers data between host and device automatically
  ✓ Reduces fragmentation
  ✓ Supports memory reclamation

• Scalable data structure

cuSTINGER paper: [Green&Bader; HPEC, 2016]:

cuSTINGER: Supporting dynamic graph algorithms for GPUs
Definitions

• Dynamic graphs
  - Graph can change over time.
  - Changes can be to topology, edges, or vertices.
    - For example new edges between two vertices.
  - Changes to edge or vertex weights

• Streaming graphs:
  - Graphs changing at high rates.
  - 100s of thousands of updates per second.

• Dynamic matrices
  - Adding a perturbation to the matrix
Dynamic graph example

- Only a subset of the entire graph...
- Dynamic:
  - At time $t$:
    - $v$ and $w$ become friends.
    - $\text{insert\_edge}(v, w)$
  - At time $\hat{t}$:
    - $u$ and $v$ no longer friends
    - $\text{delete\_edge}(u, v)$
- Additional operations include vertex insertions & deletions
### Widely used graph data structures

<table>
<thead>
<tr>
<th>Names</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dense Adjacency Matrix</td>
<td>• Supports updates</td>
<td>• Poor locality</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Massive storage requirements</td>
</tr>
<tr>
<td>Linked lists</td>
<td>• Flexible</td>
<td>• Poor locality</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Limited parallelism</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Allocation time is costly</td>
</tr>
<tr>
<td>COO (Edge list) - unsorted</td>
<td>• Has some flexibility</td>
<td>• Poor locality</td>
</tr>
<tr>
<td></td>
<td>• Updates are simple</td>
<td>• Stores both the source and destination</td>
</tr>
<tr>
<td></td>
<td>• Lots of parallelism</td>
<td></td>
</tr>
<tr>
<td>CSR</td>
<td>• Uses exact amount of memory</td>
<td>• Inflexible</td>
</tr>
<tr>
<td></td>
<td>• Good locality</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Lots of parallelism</td>
<td></td>
</tr>
</tbody>
</table>

These data structures don’t cut it
Compressed Sparse Row (CSR)

Pros:
• Uses precise storage requirements
• Great locality
  – Good for GPUs
• Handful of arrays
  – Simple to use and manage

Cons:
• Inflexible.
• Network growth unsupported
• Topology changes unsupported
• Property graphs not supported
Hornet – A High Level View

- Every vertex points at its own array
- Many edges array (blocks)
- Block size is determined by the number of neighbors (always powers of 2)
- Extra space left at the end of the block
## Hornet – Property Graph Support

### User Interface

<table>
<thead>
<tr>
<th>Vertex Id</th>
<th>Used</th>
<th>Pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

### Dest./Col.

<table>
<thead>
<tr>
<th>Weight</th>
<th>Type</th>
<th>Time 1</th>
<th>User 1</th>
<th>User 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>2</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

- Programmers can add fields per edge
- Easy to manage for static graph data structures
- Hornet manages the data movement
Hornet in Detail

**USER-INTERFACE**

<table>
<thead>
<tr>
<th>Vertex Id</th>
<th>0</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>Used (#Neighbors/nnz)</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Pointer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**MEMORY MANAGER**

- **BA₀,₁**: bsize = 1
- **BA₁,₁**: bsize = 2
- **BA₁,₂**: bsize = 2
- **BA₂,₁**: bsize = 4

- Vec-Tree Bit status
- Dest./Col. Weight

Over-allocated space for vertex insertions

Over-allocated space for power-of-two rule
Hornet Performance

• Memory Utilization
  – Independent of the GPU being used
• Initialization overhead
• Update rate
Hornet Performance Analysis

• All performance analysis is for the P100
  - 56 SMs
  - 3584 SPs
  - 16GB HBM2 memory
Inputs Graphs

- DIMACS 10 Graph Implementation Challenge
- SNAP – Stanford Network Analysis Project
- Florida Matrix Collection

The following is only a subset of these graphs:

| Name             | Type             | $|V|$    | $|E|$     | Source  |
|------------------|------------------|--------|----------|---------|
| coAuthorsDBLP    | Collaboration    | 299k   | 1.95M    | DIMACS  |
| as – skitter     | Trace route      | 1.69M  | 11.1M    | SNAP    |
| kron_21          | Random           | 2M     | 201M     | DIMACS  |
| cit – patents    | Citation         | 3.77M  | 16.5M    | SNAP    |
| cage15           | Matrix           | 5.15M  | 94M      | DIMACS  |
| uk – 2002        | Webcraw1         | 18.52M | 523M     | DIMACS  |
Memory Utilization - Overall

- BlockArrays of size $2^{16}$
- 70% average utilization of CSR
- Better utilization than: COO, cuSTINGER, AIM
  - AIM allocates all GPU memory
Initialization overhead

- Time to initialize data structure in comparison to CSR
- In most cases 2X-3X slower
  - One time penalty
- Much faster than cuSTINGER
Insertion Rates

- Supports over 150M updates per second
- Hornet
  - $4X - 10X$ faster than cuSTINGER
  - Does not have *performance dip* like cuSTINGER
- Scalable growth in update rate

**cuSTINGER**

**Hornet**
Take away

• Anything you can do with CSR you can also do with Hornet (other way is not true)
• Supports high update rates

• Scalable in both data size and in performance
• Simple and high-level programming model
  – See you at 1:00pm
• Also, look for James Fox’s talk on a cool algorithm for finding the maximal K-Truss in a graph
  – Uses dynamic triangle counting and the Hornet’s deletion...
Hornet Team (Current & Alumni)
Thank you

• Email: ogreen@gatech.edu

• Hornet:
  - https://github.com/hornet-gt/hornet

• HornetsNest:
  - https://github.com/hornet-gt/hornetsnest
Backup slides
Memory Utilization - Overall

- 70% average utilization of CSR
- Better utilization in comparison to: COO, cuSTINGER, AIMS
Part 2: HornetsNest

• Algorithm framework for Hornet data structure
  – We support CSR as well
• All algorithms are implemented using a small set of operations
  – We show that these operators are efficient for static graph algorithms and can be used for dynamic graph algorithms
• Uses features from C++11 and C++14

Oded Green, GTC-18
Sparse Matrix Vector Multiplication

- In comparison to DCSR [King et al; 2016; ISC]
  - DCSR requires customized SpMV
- Hornet uses identical algorithm code as CSR.