Multipredicate Join Algorithms for Accelerating Relational Graph Processing on GPUs

Haicheng Wu\textsuperscript{1}, Daniel Zinn\textsuperscript{2}, Molham Aref\textsuperscript{2}, Sudhakar Yalamanchili\textsuperscript{1}

1. Georgia Institute of Technology
2. LogicBlox Inc.
System Diversity Today

Amazon EC2 GPU Instances

Mobile Platforms (DSP, GPUs)

Hardware Diversity is Mainstream

Keeneland System (GPUs)

Cray Titan (GPUs)
GPU and CUDA

- GPU is a many core co-processor
  - 1000s of cores
  - 1000s of concurrent threads
  - Higher memory bandwidth
  - Smaller memory capacity
  - CUDA and OpenCL are the dominant programming models

- Well suited for data parallel apps
  - Molecular Dynamics, Options Pricing, Ray Tracing, etc.

- Commodity: led by NVIDIA, AMD, and Intel
Relational Queries and Data Analytics

- The Opportunity
  - Significant potential data parallelism

- The Problem
  - Need to process 1-50 TBs of data
    - Small Mem Capacity & Small PCIe bandwidth
  - Irregularity
    - Fine grained computation
    - Data dependent
    - Low locality

---

The Challenge

Relational Computations Over Massive Unstructured Data Sets: Sustain 10X – 100X throughput over multicore
Multipredicate Join

- **Goal**: Implementation of Leapfrog Triejoin (LFTJ) on GPU
  - A worst-case optimal multi-predicate join algorithm
  - Details (e.g., complexity analysis) in T. L. Veldhuizen, *ICDT 2014*

- **Benefits**
  - Smaller memory footprint and data movement
  - No data reorganization (e.g. sorting or rebuilding hash table) after changing join key

- **Approach**
  - CPU version
  - CPU-Friendly GPU version
  - Customized GPU version
An Important Example – Graph Problems

- Finding cliques
  - $\text{triangle}(x,y,z) \leftarrow E(x,y), E(y,z), E(x,z), x < y < z.$
  - $\text{4cl}(x,y,z,w) \leftarrow E(x,y), E(x,z), E(x,w), E(y,z), E(y,w), E(z,w), x < y < z < w.$

![Diagram of a graph with edges labeled]

<table>
<thead>
<tr>
<th>Edge</th>
<th>From</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
Leapfrog Join (LFJ)

- LFJ is the base of LFTJ
- Essentially multi-way-intersections
- Basic primitives: \textit{seek()}, \textit{next()}

\textit{Courtesy: T. L. Veldhuizen, ICDT 2014}
Trie Data Structure

LFTJ works on Trie Data Structure
LFTJ Algorithm – *join 3 tries*

E(x,y)

E(y,z)

E(x,z)

x

y

z
LFTJ Algorithm – \textit{open()} level \( x \)

\[ E(x,y) \]

\[ E(y,z) \]

\[ E(x,z) \]
LFTJ Algorithm – \textit{seek}(0) in $E(x,z)$ level $x$
LFTJ Algorithm – open() level $y$

$E(x,y)$

Root

$E(y,z)$

$E(x,z)$
LFTJ Algorithm – \textit{seek}(1) in $E(y,z)$ level $y$
LFTJ Algorithm – open() level z

E(x,y)
Root
x
0 1 2 3
y
1 2 3 3 4 5

E(y,z)
Root
E(x,z)
Root
x
0 1 2 3
y
1 2 3 3 4 5

z
1 2 3

E(x,z)
Root
0 1 2 3
1 2 3 4 5
LFTJ Algorithm – seek(2) in $E(x,z)$ level $z$ and failed
LFTJ Algorithm – *up() to level y*

```
x  0  1  2  3
y  1  2  3  3  4  5
    E(x,y)
```

```
y  0  1  2  3  4  5
z  1  2  3  3  4  5
    E(y,z)
    E(x,z)
```

```
x  0  1  2  3
y  1  2  3  3  4  5
    E(x,y)
    E(y,z)
    E(x,z)
```
LFTJ Algorithm – \textit{up()} to level \( x \)
LFTJ Algorithm – **seek(1) in E(x,z) level x**

![Diagram of trees and nodes representing the LFTJ Algorithm](image-url)
LFTJ Algorithm – open() level \( y \)

\[
\begin{align*}
&\text{E}(x,y) \\
&\text{Root} \\
&\quad 0 \quad 1 \quad 2 \quad 3 \\
&\quad 1 \quad 2 \quad 3 \\
&\quad 1 \quad 2 \quad 3 \\
&\text{E}(y,z) \\
&\quad 0 \quad 1 \quad 2 \quad 3 \\
&\quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \\
&\quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \\
&\text{E}(x,z) \\
&\quad 0 \quad 1 \quad 2 \quad 3 \\
&\quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \\
&\quad 1 \quad 2 \quad 3 \quad 4 \quad 5
\end{align*}
\]
LFTJ Algorithm – seek(2) in $E(y,z)$ level $y$

```
\[ E(x,y) \]
```

```
\[ E(y,z) \]
```

```
\[ E(x,z) \]
```
LFTJ Algorithm – open() level z

\[ E(x,y) \]

\[ E(y,z) \]

\[ E(x,z) \]
LFTJ Algorithm – seek(3) in $E(x,z)$ level $z$
LFTJ Algorithm – next()
LFTJ Algorithm – *final result*

- $E(x,y)$
  - Root
  - $x$: 0, 1, 2, 3
  - $y$: 1, 2, 3, 3, 4, 5

- $E(y,z)$
  - Root
  - $y$: 0, 1, 2, 3
  - $z$: 1, 2, 3, 3, 4, 5

- $E(x,z)$
  - Root
  - $x$: 0, 1, 2, 3
  - $z$: 1, 2, 3, 3, 4, 5
LFTJ Algorithm – short conclusion

- Very simple set of primitives to implement
- A sequential algorithm
- Traverse the Trie in depth first order

- Two methods for applying this technique with GPUs
  - CPU algorithm per GPU thread
  - Customize data parallel application
LFTJ-GPU: First Algorithm

- Evenly map the top level of the leftmost trie to GPU threads
- Run sequential LFTJ in each GPU thread
- `seek()` is implemented as binary search
  - Data dependent control flow
  - No spacial or temporal locality

Example: mapping to 2 GPU threads
LFTJ-GPU: Optimizations and CPU Variant

- Set current level (e.g. x) as template to avoid branching
- Reduce the search scope of binary searches
  - Put a search tree (similar as B-tree) in shared memory
    - First several lookups of binary searches are run in the shared memory
    - 26% improvement in triangle; 3.2x improvement in 4-clique
- Amenable to CPU multi-thread implementation
  - Simply replace GPU threads by CPU threads
  - Referred as LFTJ-CPU

<table>
<thead>
<tr>
<th>Value</th>
<th>2</th>
<th>5</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>UpperBound</td>
<td>8</td>
<td>20</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

*Its UpperBound must between 10 and 20*
Optimized-GPU: Second Algorithm

- Optimized for
  - Load balance
  - Better Memory Access Pattern

- Change from depth first order to Breadth first order

- Divide the algorithm into three smaller problems
  - Tree node expansion
  - Parallel array intersection
  - Filtering

*Hot GPGPU research topic*

*Well-known in GPGPU*
Optimized-GPU: Two APIs from ModernGPU library

- **Vectorized Sorted Search & Load Balancing Search**

- **ModernGPU** is Designed by S. Baxter

- Based on Merge-Path* framework to balance workload between CTAs and threads

- Optimized for coalesced memory access

---

Optimized-GPU: Data Structure

- Similar as CSR

Children of the different parent may *not* be Sorted or Unique

Children of the same parent are Sorted and Unique
Optimized-GPU: Intersect level by level

\[ E(x,y) \quad E(y,z) \quad E(x,z) \]

Sorted intersects sorted

Sorted intersects not sorted

Not sorted intersects not sorted
Optimized-LFTJ: Algorithm

- Process layer by layer from top to bottom

- In each layer
  - Intersect all sorted arrays (simplest)
    - Simple Set Intersection
  - Intersect all not sorted arrays
    - Segmented Intersection
  - Intersect the above two results (heaviest)
    - Binary Search

avoid using binary searches
Optimized-LFTJ: Compared with Other Algorithms

- Compared with GPU-LFTJ
  - Depth first order => Breadth first order
  - GPU threads collaborate together in intersections
  - Less binary searches – at most 1 per layer
  - Larger memory footprint – tradeoff between time and space

- Compared with pair-wise joins
  - No sorting
  - No huge temporary result
### Experimental Environment

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU</strong></td>
<td>Intel i7-4771 @ 3.50GHz</td>
</tr>
<tr>
<td><strong>GPU</strong></td>
<td>Geforce GTX Titan (2688 cores, $1000 USD)</td>
</tr>
<tr>
<td><strong>PCIe</strong></td>
<td>3.0 x 16</td>
</tr>
<tr>
<td><strong>OS</strong></td>
<td>Ubuntu 12.04</td>
</tr>
<tr>
<td><strong>G++/GCC</strong></td>
<td>4.6</td>
</tr>
<tr>
<td><strong>NVCC</strong></td>
<td>6.0</td>
</tr>
<tr>
<td><strong>Thrust</strong></td>
<td>1.7</td>
</tr>
</tbody>
</table>
Evaluated Graphs

- 10K to 100M edges
  - Fits the GPU memory

- Node Id: 64-bit random number

- NumNode
  - Triangle: NumNode = NumEdge
  - 4-Clique: NumNode = NumEdge^{3/4}

- Results are very sparse
  - Found cliques are less than 4

Larger graph has larger node degree
Evaluated Algorithms

- GPU-LFTJ: First Algorithm
- Optimized-GPU: Second Algorithm
- CPU-LFTJ: CPU Variant of GPU-LFTJ
- Red Fox: Run regular pairwise sort-merge join from ModernGPU library
**Overall Performance**

- Optimized-GPU is fastest
- GPU-LFTJ can run much larger problem
Reason Behind the Performance

- **Optimized-GPU vs. LFTJ-GPU**
  - Less binary searches
    - Optimized-GPU spends 40% (triangle) or 7% (4-clique) in binary searches
  - Better load balance
    - Over 94% *warp execution efficiency*
  - Better memory access pattern
    - *ld/st replay* of Optimized-GPU is only 4.6 (triangle) or 0.6 (4-clique)
    - *ld/st replay* of binary search is 19

- **Optimized-GPU vs. Red Fox**
  - No sorting
    - Sorting time of Red Fox is more than overall time of Optimized-GPU

- **LFTJ-GPU vs. LFTJ-CPU = GPU vs. CPU**
Conclusion

- **GPU-LFTJ**
  - Simple to implement
  - Easy to integrate into existing system
  - Reasonable performance
  - Much less memory footprint

- **Optimized-GPU**
  - Better performance
  - Larger memory footprint
  - Sophisticated traditional GPGPU program
  - Room to improve
    - Better expansion, intersection algorithms
    - Fusing small CUDA kernels
    - Completely remove binary searches
Out-of-Core Support

- In-Core algorithm is the building block

- Pipeline the execution with PCIe

- Currently, throughput is smaller than PCIe bandwidth
  - Out-of-Core performance is determined by In-Core algorithm

- Ideally, push the performance to be PCIe-bounded
  - GPU computation can be completely hidden by PCIe
  - ~10GB/s throughput
The Future is Acceleration

Thank You
Results so far and Things to do

Results so far

Things to do:
Segmented Intersection

\[ E(x,y) \]
\[ E(x,z) \]
\[ E(y,z) \]
Memory Footprint

Redfox > GPU-Optimized > LFTJ-GPU

Triangle

4-Clique