An Energy-Efficient Abstraction for Simultaneous Breadth-First Searches

Adam McLaughlin, Jason Riedy, and David A. Bader
Problem

• Data is unstructured, heterogeneous, and vast

• Serious opportunities for businesses to capitalize on insight
  – Target (pregnancy scandal)
  – Explosion of popularity in ML techniques

• Interactive, real-time analyses preferred
  – Graph analytics
  – HPC to the rescue!
“Traditional” HPC is Expensive

- Tianhe-2: 17.8 MW
- Titan: 8.2 MW

- Distributed systems are overkill
  - Too much time and energy wasted on expensive communication
  - Shared memory is large enough (~1 TB)

- Leverage the high memory bandwidth of GPUs
GPUs are Challenging to Program

• Months of domain expert programmer time required to develop/optimize code
• Efforts are typically limited to a single problem, architecture, or data set
  – Little code reuse
  – Limited number of libraries
  – Opaque performance consequences
What makes GPU Computing so Difficult?

- Parallel programming challenges
  - Deadlock, synchronization, race conditions
- Architectural/Ecosystem challenges
  - Programmer managed shared memory
  - Deep knowledge of the underlying architecture required
- Challenges unique to graph analysis
  - Data dependent memory access patterns
Solution: Abstraction

• Abstract details of parallel programming from end users
• Let social scientists, analysts, etc. focus on gathering insights
• Let domain experts focus on parallel programming, architectural details
  – Encourage modularity and code reuse
Related Abstractions

• Traversal-based
  – Ligra [Shun et al.], Gunrock [Wang et al.]
  – User applies code that operates on active vertices and provides the next frontier of vertices
  – Library efficiently transitions from one frontier to the next

• Linear-Algebraic
  – GraphBLAS [Mattson et al.], CUSP [Dalton et al.]
  – Formulate as operators on vectors and matrices

• Gather Apply Scatter (GraphLab)
The Multi-Search Abstraction

• Fits any problem requiring the simultaneous execution of many breath-first searches

1. All-Pairs Shortest Paths
2. Diameter Computations
3. Transitive Closures
4. Betweenness Centrality
What makes this abstraction different?

- Traversal based, but utilizes coarse-grained parallelism
  - Prior abstractions parallelize within the context of a single BFS
  - Our abstraction parallelizes across BFSs
Multi-Search: User Functions

• User needs to implement a small number of typically short functions:

  - `init()`: Runs prior to every search
  - `prior()`: Runs before every search iteration
  - `visitVertex()`: When an edge \((u, v)\) is traversed from source \(s\), update data in terms of \(u, v, s\)
  - `post()`: Runs after every search iteration
  - `finalize()`: Runs after the completion of every search
Multi-Search: Simple Example

• All-Pairs Shortest Paths

```c
void init(int s)
{
    for(int k=0; k<n; k++) //For each vertex
    {
        if(k == s) d[s][k] = 0;
        else d[s][k] = INT_MAX;
    }
}

void visitVertex(int s, int u, int v, int *Qnext, int Qnext_len)
{
    if(d[s][v] == INT_MAX)
    {
        d[s][v] = d[s][u] + 1;
        t = atomicAdd(&Qnext_len, 1);
        Qnext[t] = w
    }
}
```
Multi-Search: Visiting vertices

• Use a cooperative, Warp-based approach
• Warps concurrently expand adjacency lists of enqueued vertices
• Works great for vertices with high outdegree
  – Coalesced accesses to neighbor lists
• Underutilization for vertices with low outdegree
Multi-Search: Hierarchical Queues

• To resolve this underutilization, we can assign a thread to each enqueued vertex

• Use a thresholding approach
  – Outdegree(v) \geq T \rightarrow \text{Warp processing}
  – Outdegree(v) < T \rightarrow \text{Thread processing}
void multi_search(int u, int v, int s) {
    parallel_for(int s=0; s<n; s++) { // Parallelize across SMs
        if (outdegree(i) > T) Q_small_curr.enqueue(s);
        else Q_curr.enqueue(s);
        init(s);
        barrier();
        while (!Q_curr.empty() && !Q_small_curr.empty()) {
            prior();
            barrier();
            // Visit vertices in the queues...
            move(Q_curr, Q_next); // Swap queues for next iteration
            move(Q_small_curr, Q_small_next);
            barrier();
            post();
            barrier();
        }
        finalize();
    }
}
Multi-Search: Under the hood

```c
while(!Q_curr.empty() && !Q_small_curr.empty()){
    prior();
    barrier();
    // Visit vertices in the queues
    for(int v=0; v<Q_curr_len; v++){
        parallel_for(int w=0; w<outdegree(v); w++){
            visitVertex(s,v,w,Q_next,Q_next_len);
        }
    }
    parallel_for(int v=0; v<Q_small_curr_len; v++){
        for(int w=0; w<outdegree(v); w++){
            visitVertex(s,v,w,Q_small_next,Q_small_next_len);
        }
    }
    move(Q_curr,Q_next); //Swap queues for next iteration
    move(Q_small_curr,Q_small_next);
    barrier();
    post();
    barrier();
}
```
Experimental Setup

• NVIDIA GTX Titan, NVIDIA Tesla K40c
  – Compute capability 3.5 ("Kepler") GPUs
  – Peak theoretical memory bandwidth: 288.4 GB/s
• Titan: 14 SMs, 6GB memory, 837MHz
• Tesla K40c: 15 SMs, 12GB memory, 745MHz, 245W TDP
• Power measurement: NVML & C++11 Futures
  – Sample every 10ms
Effect of Thresholding

- $T = 0$: Warp
- $T = \infty$: Thread
- Too small: Warp occupancy suffers
- Too large: severe workload imbalances among threads
- $T = 16$ (Half-warp)
## Benchmark Data Sets

<table>
<thead>
<tr>
<th>Graph</th>
<th>Vertices</th>
<th>Edges</th>
<th>Graph</th>
<th>Vertices</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>333SP</td>
<td>3,712,815</td>
<td>22,217,266</td>
<td>hollywood-2009</td>
<td>1,139,905</td>
<td>115,031,232</td>
</tr>
<tr>
<td>adaptive</td>
<td>6,815,744</td>
<td>27,248,640</td>
<td>kron_g500-logn19</td>
<td>524,288</td>
<td>43,561,574</td>
</tr>
<tr>
<td>as-Skitter</td>
<td>1,696,415</td>
<td>22,190,596</td>
<td>ldoor</td>
<td>952,203</td>
<td>45,570,272</td>
</tr>
<tr>
<td>auto</td>
<td>448,695</td>
<td>6,629,222</td>
<td>rgg_n_2_21_s0</td>
<td>2,097,152</td>
<td>28,975,990</td>
</tr>
<tr>
<td>delaunay_n21</td>
<td>2,097,152</td>
<td>12,582,816</td>
<td>roadNet-CA</td>
<td>1,971,281</td>
<td>5,533,214</td>
</tr>
<tr>
<td>ecology1</td>
<td>1,000,000</td>
<td>3,996,000</td>
<td>thermal2</td>
<td>1,227,087</td>
<td>7,352,268</td>
</tr>
</tbody>
</table>
### Timing Results (s)

<table>
<thead>
<tr>
<th>Graph</th>
<th>Oracle</th>
<th>Warp-based</th>
<th>Hybrid</th>
<th>Graph</th>
<th>Oracle</th>
<th>Warp-based</th>
<th>Hybrid</th>
</tr>
</thead>
<tbody>
<tr>
<td>333SP</td>
<td>47.8</td>
<td>68.0</td>
<td>32.1</td>
<td>hollywood-2009</td>
<td>81.6</td>
<td>21.3</td>
<td>20.4</td>
</tr>
<tr>
<td>adaptive</td>
<td>54.8</td>
<td>183.6</td>
<td>42.8</td>
<td>kron_g500-logn19</td>
<td>35.4</td>
<td>16.4</td>
<td>17.1</td>
</tr>
<tr>
<td>as-Skitter</td>
<td>27.8</td>
<td>12.9</td>
<td>9.7</td>
<td>ldoor</td>
<td>35.4</td>
<td>35.7</td>
<td>36.5</td>
</tr>
<tr>
<td>auto</td>
<td>15.6</td>
<td>5.48</td>
<td>4.8</td>
<td>rgg_n_2_21_s0</td>
<td>67.0</td>
<td>37.0</td>
<td>23.7</td>
</tr>
<tr>
<td>delaunay_n21</td>
<td>23.3</td>
<td>25.1</td>
<td>15.0</td>
<td>roadNet-CA</td>
<td>12.3</td>
<td>15.0</td>
<td>9.2</td>
</tr>
<tr>
<td>ecology1</td>
<td>8.9</td>
<td>29.1</td>
<td>6.5</td>
<td>thermal2</td>
<td>11.7</td>
<td>19.4</td>
<td>7.7</td>
</tr>
</tbody>
</table>

- Savings of 42% time on average vs. Oracle
### Energy Results (J)

<table>
<thead>
<tr>
<th>Graph</th>
<th>Oracle</th>
<th>Warp-based</th>
<th>Hybrid</th>
<th>Graph</th>
<th>Oracle</th>
<th>Warp-based</th>
<th>Hybrid</th>
</tr>
</thead>
<tbody>
<tr>
<td>333SP</td>
<td>2,220</td>
<td>2,880</td>
<td>1,083</td>
<td>hollywood-2009</td>
<td>11,546</td>
<td>1,355</td>
<td>3,342</td>
</tr>
<tr>
<td>adaptive</td>
<td>8,551</td>
<td>6,790</td>
<td>1,433</td>
<td>kron_g500-logn19</td>
<td>1,180</td>
<td>1,195</td>
<td>570</td>
</tr>
<tr>
<td>as-Skitter</td>
<td>4,688</td>
<td>1,053</td>
<td>1,606</td>
<td>ldoor</td>
<td>1,829</td>
<td>1,855</td>
<td>1,210</td>
</tr>
<tr>
<td>auto</td>
<td>635</td>
<td>539</td>
<td>159</td>
<td>rgg_n_2_21_s0</td>
<td>2,851</td>
<td>1,908</td>
<td>759</td>
</tr>
<tr>
<td>delaunay_n21</td>
<td>1,425</td>
<td>1,495</td>
<td>508</td>
<td>roadNet-CA</td>
<td>2,050</td>
<td>1,144</td>
<td>1,539</td>
</tr>
<tr>
<td>ecology1</td>
<td>856</td>
<td>1,642</td>
<td>219</td>
<td>thermal2</td>
<td>996</td>
<td>1,311</td>
<td>259</td>
</tr>
</tbody>
</table>

- **Savings of 62% energy on average vs. Oracle**
Conclusions

• Abstraction is paramount for high-performance, reusable applications
  – Prior methods of abstraction miss out on coarse-grained parallelism

• If the amount of parallelism changes over time, the method of parallelism should change too

• Maximizing performance also tends to help save energy
  – Saved 42% time and 62% energy compared to prior work
Acknowledgment of Support
“To raise new questions, new possibilities, to regard old problems from a new angle, requires creative imagination and marks real advance in science.” – Albert Einstein

https://github.com/Adam27X/graph-utils
Backup
“Traditional” HPC is Expensive

• Graphs are no longer processed by supercomputers alone
  – Embedded systems
    • Computer vision
  – Mobile devices
    • Spam detection

• Systems are becoming constrained by power and energy
  – High demand for work-efficient implementations
  – Goal: Maximize performance per Watt using GPUs
Motivation

• Real world graphs are challenging to process
  – Enormous
    • Networks cannot be manually inspected
  – Varying structural properties
    • Small-world, scale-free, meshes, road networks
      – Not a one-size fits all problem
  – Unpredictable
    • Rapidly change over time
    • Data dependent memory access patterns