Exploring Irregular Memory Access Applications on the GPU

Mohammed Suhail Rehman
rehman@research.iiit.ac.in
200707016
Advisors: Prof. Kishore Kothapalli, Prof. P.J. Narayanan
International Institute of Information Technology
Graphics Processing Units (GPUs)

- Dedicated co-processors that handle display output.
- Increasingly programmable and powerful.
- Latest GPUs have over 1 TFLOP of peak performance.
- CUDA
  - Hardware Architecture
  - Programming Model
  - API to Program the GPU in a C-like environment
Irregular Algorithms

- Algorithms which have irregular memory access patterns
- Difficult to implement and parallelize with high performance on current generation architectures
  - Does not “play nice” with the memory hierarchy
    - Cache/Virtual Memory Issues
  - Eg: List Ranking, Tree Computations etc.
The List Ranking Problem

- Classic Irregular Algorithm
- Find the position of every element in a linked list
- Becomes Irregular when list elements are scattered in memory
  - Difficult to parallelize
- Many applications in Graph Problems
  - Tree Computations, Biconnected Components etc.
Solving the Problem

- Many Algorithms for Parallelization
  - Pointer Jumping Technique
  - Randomized Approaches
  - Sparse Ruling Set Approaches
Contributions of this Thesis

- Pointer Jumping Technique for the GPU
- Wyllie’s Algorithm and variants for the GPU
- A work-optimal, recursive algorithm for the GPU
- Applications of List Ranking
  - Euler Tour Construction and Tree Computations on the GPU
- Performance Analysis and Behavior of such Irregular Algorithms on the GPU
Introduction and Background
GPU by the Numbers (GTX280)

- 240 cores
- 4-stage pipeline on each core
- 960 threads in flight at any given time.
- Memory allocated for a set of 8 Cores (SM)
  - 64K Registers
  - 16 KB Shared Memory
  - 8K Constant Cache
  - 8K Texture Cache
GPU Architecture

Thread Execution Control Unit

Processors, Control Units, Shared Memory and Registers (on-chip area)

Multi Processors

Shared Memory + Registers

Device Memory (off-chip area)
CUDA Hardware Architecture

- SIMD Multi Processor #1
- SIMD Multi Processor #2
- SIMD Multi Processor #3

Instruction Unit

Shared Memory (16KB)

Registers (64KB)

Texture Cache (8KB)

Constant Cache (8KB)

Device Memory (~1GB)
Data-Parallel Primitives on the GPU

- Primitives on GPU to use for larger problems
  - Scan by Mark Harris et. al
  - Reduce Operations by NVIDIA
  - Efficient Sorting on GPU
- List Ranking is a primitive for lots of graph problems!
List Ranking

- Simple Problem (Sequentially Speaking)
  - Difficult to Parallelize
- Can be done on ordered or random lists:

Ordered List:
1 2 3 4 5 6 7 8 9 10

Unordered List:
1 6 2 5 7 4 9 10 3 8
## List Ranking Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time</th>
<th>Work</th>
<th>Constants</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>small</td>
<td>$c$</td>
</tr>
<tr>
<td>Wyllie[16]</td>
<td>$O\left(\frac{n \log n}{p} + \log n\right)$</td>
<td>$O(n \log n)$</td>
<td>small</td>
<td>$n + c$</td>
</tr>
<tr>
<td>Randomized[36, 37]</td>
<td>$O\left(\frac{n}{p} + \log n\right)$</td>
<td>$O(n)$</td>
<td>medium</td>
<td>$&gt; 2n$</td>
</tr>
<tr>
<td>PRAM Optimal [32, 33]</td>
<td>$O\left(\frac{n}{p} + \log n\right)$</td>
<td>$O(n)$</td>
<td>large</td>
<td>$&gt; n$</td>
</tr>
<tr>
<td>Miller-Reid[34]</td>
<td>$O\left(\frac{n}{p} + \log^2 n\right)$</td>
<td>$O(n)$</td>
<td>small</td>
<td>$5p + c$</td>
</tr>
<tr>
<td>Helman-JáJá [17]</td>
<td>$O\left(\frac{n}{p} + \log n\right)$</td>
<td>$O(n)$</td>
<td>small</td>
<td>$n + \log n + c$</td>
</tr>
</tbody>
</table>
Applications of List Ranking

- Euler Tour Technique
- Load Balancing
- Tree Contraction
- Expression Evaluation
- Planar Graph Embedding
- Etc..
Pointer Jumping and Wyllie’s Algorithm on the GPU
Pointer Jumping

- A basic technique in parallel computing.
  - Processing Data in Rooted Directed Trees
A basic technique in parallel computing.

- Processing Data in Rooted Directed Trees
A basic technique in parallel computing.

- Processing Data in Rooted Directed Trees
Algorithm 2 Pointer Jumping

Input: A forest of rooted directed trees.
Output: For each vertex $i$, the root $S(i)$ of the tree containing $i$

1: for $1 \leq i \leq n$ do in parallel
2: \hspace{1em} while $S(i)$ is not end of list do
3: \hspace{2em} Set $S(i) = S(S(i))$
4: \hspace{1em} end while
5: end for
From PRAM to the GPU

- One thread per element
- Successor Array Implementation
Wyllie’s Algorithm

- Use pointer jumping repeatedly for each element of a linked list in Parallel
  - Update Rank and Successor simultaneously

**Algorithm 3 Wyllie’s Algorithm**

**Input:** An array $S$, containing successors of $n$ nodes and array $R$ with ranks initialized to 1

**Output:** Array $R$ with ranks of each element of the list with respect to the head of the list

1. for each element in $S$ do in parallel
2. while $S[i]$ and $S[S[i]]$ are not the end of list do
3. \hspace{1em} $R[i] = R[i] + R[S[i]]$
4. \hspace{1em} $S[i] = S[S[i]]$
5. end while
6. end for
Wyllie’s Algorithm on the GPU

- Implemented in CUDA
  - One Thread per list element
  - Synchronize the write operation (rank and successor fields of an element should be written to simultaneously)
  - Pack the two numbers into a 64-bit long integer
Results of Wyllie’s Algorithm

<table>
<thead>
<tr>
<th>List Size</th>
<th>Time (Milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1K</td>
<td>0.065</td>
</tr>
<tr>
<td>2K</td>
<td>0.087</td>
</tr>
<tr>
<td>4K</td>
<td>0.127</td>
</tr>
<tr>
<td>8K</td>
<td>0.22</td>
</tr>
<tr>
<td>16K</td>
<td>0.359</td>
</tr>
<tr>
<td>32K</td>
<td>0.686</td>
</tr>
<tr>
<td>64K</td>
<td>1.21</td>
</tr>
<tr>
<td>128K</td>
<td>2.747</td>
</tr>
<tr>
<td>256K</td>
<td>6.313</td>
</tr>
<tr>
<td>512K</td>
<td>12.48</td>
</tr>
<tr>
<td>1M</td>
<td>26.4</td>
</tr>
<tr>
<td>2M</td>
<td>56.84</td>
</tr>
<tr>
<td>4M</td>
<td>130.1</td>
</tr>
<tr>
<td>8M</td>
<td>279.6</td>
</tr>
<tr>
<td>16M</td>
<td>615.6</td>
</tr>
<tr>
<td>32M</td>
<td>1359</td>
</tr>
<tr>
<td>64M</td>
<td>2841</td>
</tr>
</tbody>
</table>

- **GTX280**: 0.006, 0.01, 0.018, 0.047, 0.105, 0.535, 1.357, 3.051, 6.395, 14.8, 35.09, 128.3, 324.3, 730.1, 1582, 3352, 6929
- **Core i7**: 0.006, 0.01, 0.018, 0.047, 0.105, 0.535, 1.357, 3.051, 6.395, 14.8, 35.09, 128.3, 324.3, 730.1, 1582, 3352, 6929
Optimization I – 32-bit Packing

- For lists of size $< 2^{16}$
  - Pack rank and successor in a 32-bit integer
  - Potential applications in Sparse Forests

### Table

<table>
<thead>
<tr>
<th>List Size</th>
<th>32 bit</th>
<th>64 bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1K</td>
<td>0.054</td>
<td>0.065</td>
</tr>
<tr>
<td>2K</td>
<td>0.069</td>
<td>0.087</td>
</tr>
<tr>
<td>4K</td>
<td>0.104</td>
<td>0.127</td>
</tr>
<tr>
<td>8K</td>
<td>0.166</td>
<td>0.22</td>
</tr>
<tr>
<td>16K</td>
<td>0.234</td>
<td>0.359</td>
</tr>
<tr>
<td>32K</td>
<td>0.417</td>
<td>0.686</td>
</tr>
</tbody>
</table>
The original algorithm has $n$ threads working on $n$ elements, each thread doing $O(\log n)$ work in the PRAM model.

- GPU is not fully PRAM-compliant - thread execution is asynchronous.
- Force synchronous behavior by jumping once in the kernel and forcing a global synchronization.
Algorithm 4 CPU-Orchestrated Wyllie’s Algorithm

**Input:** An array of nodes $N$, each with fields $rank$, $next$ and $finished$.

**Output:** $N$ with all the node ranks updated

1: Set $done = 0$ in global memory
2: while some thread has set $done = 0$ do in parallel
3:     Set $done = 1$
4:     Load the node in $N$ designated to this thread in local register $r_1$
5:     if $r_1.f\text{inished} = 0$ then
6:         Load the successor node in local register $r_2$
7:         if $r_2.f\text{inished} = 0$ then
8:             Set $r_1.next = r_2.next$
9:             $done = 0$
10:        end if
11:     Set $r_1.rank = r_1.rank + r_2.rank$
12:     Synchronize and store the node in $N$ designated to this thread with $r_1$
13: end if
14: end while
Threads would load an element and its successor even if it's pointing to end of list
- Modify Kernel to load elements as needed.
- Block-Synchronize to force eligible threads to access memory more efficiently
## Comparison of the Methods

<table>
<thead>
<tr>
<th>List Size</th>
<th>Original</th>
<th>Stepwise</th>
<th>Optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>1K</td>
<td>0.07</td>
<td>0.387</td>
<td>0.052</td>
</tr>
<tr>
<td>2K</td>
<td>0.095</td>
<td>0.423</td>
<td>0.058</td>
</tr>
<tr>
<td>4K</td>
<td>0.153</td>
<td>0.463</td>
<td>0.059</td>
</tr>
<tr>
<td>8K</td>
<td>0.272</td>
<td>0.543</td>
<td>0.083</td>
</tr>
<tr>
<td>16K</td>
<td>0.395</td>
<td>0.68</td>
<td>0.125</td>
</tr>
<tr>
<td>32K</td>
<td>0.751</td>
<td>0.942</td>
<td>0.237</td>
</tr>
<tr>
<td>64K</td>
<td>1.242</td>
<td>1.269</td>
<td>0.419</td>
</tr>
<tr>
<td>128K</td>
<td>2.908</td>
<td>2.109</td>
<td>1.123</td>
</tr>
<tr>
<td>256K</td>
<td>6.603</td>
<td>4.313</td>
<td>2.86</td>
</tr>
<tr>
<td>512K</td>
<td>13.35</td>
<td>9.165</td>
<td>6.535</td>
</tr>
<tr>
<td>1M</td>
<td>28.85</td>
<td>20.47</td>
<td>14.49</td>
</tr>
<tr>
<td>2M</td>
<td>64.37</td>
<td>46.7</td>
<td>32.83</td>
</tr>
<tr>
<td>4M</td>
<td>148.4</td>
<td>102.8</td>
<td>73.01</td>
</tr>
<tr>
<td>8M</td>
<td>321.7</td>
<td>221.3</td>
<td>163.3</td>
</tr>
<tr>
<td>16M</td>
<td>708.4</td>
<td>462.9</td>
<td>377.6</td>
</tr>
<tr>
<td>32M</td>
<td>1565</td>
<td>1952</td>
<td>840.2</td>
</tr>
<tr>
<td>64M</td>
<td>3294</td>
<td>3094</td>
<td>1718</td>
</tr>
</tbody>
</table>

**Time (Milliseconds)**
Where is the performance bottleneck?

- Wyllie’s Algorithm is not optimal in PRAM model
  - $O(n \log n)$ work for something that can be done in $O(n)$
- Expensive synchronization on GPU (64 bit packing and write operations)
- Random Reads from Global Memory
  - ~400-600 GPU cycles of read latency for each element
- There has to be a better way!
  - Is there a better algorithm?
Helman-JaJa algorithm

- Designed for SMPs – Sparse Ruling Set approach
  - Mark a number of elements as ‘splitters’
  - Traverse the list from multiple points by each thread using the splitters as a starting point
  - Store ranks w.r.t splitter position for each element
  - Stop when you hit another splitter (as marked in first step)
  - Construct new, smaller list for global ranks
  - Rank the new list and add the list prefixes to each element of the list
Step 1. Select **Splitters** at equal intervals
Step 2. **Traverse** the List until the next splitter is met and **increment** local ranks as we progress.
### Step 2.

**Traverse** the List until the next splitter is met and **increment** local ranks as we progress.
Step 2. **Traverse** the List until the next splitter is met and **increment** local ranks as we progress.
Step 3. **Stop** When all elements have been assigned a local rank.
Step 4. **Create** a new list of splitters which contains a **prefix value** that is equal to the local rank of its predecessor.
Step 4. Create a new list of splitters which contains a prefix value that is equal to the local rank of its predecessor.
Successor Array

| 2 | 4 | 8 | 1 | 9 | 3 | 7 | - | 5 | 6 |

Local Ranks

| 0 | 3 | 1 | 2 | 0 | 1 | 2 | 3 | 0 | 1 |

Step 5. Scan the global ranks array **sequentially**

New List

| 2 | - | 1 |

Successor Array

| 2 | - | 1 |

Global Ranks

| 0 | 4 | 2 |

After Ranking

| 2 | - | 1 |

| 0 | 6 | 2 |
Step 6. Add the global ranks to the corresponding local ranks to get the final rank of the list.
Step 6. Add the global ranks to the corresponding local ranks to get the final rank of the list.
Step 6. Add the global ranks to the corresponding local ranks to get the final rank of the list.
Step 6. Add the global ranks to the corresponding local ranks to get the final rank of the list.

<table>
<thead>
<tr>
<th>Successor Array</th>
<th>Local Ranks</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 4 8 1 9 3 7 - 5 6</td>
<td>0 3 1 2 0 1 2 3 0 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>New List Successor Array</th>
<th>Global Ranks</th>
<th>After Ranking</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 - 1</td>
<td>0 4 2</td>
<td>0 6 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Local Ranks</th>
<th>Final Ranks</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 3 1 2 0 1 2 3 0 1</td>
<td>0 5 1 0 0 0 0 2 0 0</td>
</tr>
</tbody>
</table>
Step 6. Add the global ranks to the corresponding local ranks to get the final rank of the list.

**New List**

<table>
<thead>
<tr>
<th>Successor Array</th>
<th>Global Ranks</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 4 8 1 9 3 7 - 5 6</td>
<td>0 4 2</td>
</tr>
</tbody>
</table>

Local Ranks  

| 0 3 1 2 0 1 2 3 0 1 | 0 5 1 4 0 0 2 0 0 0 |

After Ranking
Step 6. Add the global ranks to the corresponding local ranks to get the final rank of the list.

Successor Array

<table>
<thead>
<tr>
<th>2</th>
<th>4</th>
<th>8</th>
<th>1</th>
<th>9</th>
<th>3</th>
<th>7</th>
<th>-</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
</table>

Local Ranks

<table>
<thead>
<tr>
<th>0</th>
<th>3</th>
<th>1</th>
<th>2</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
</table>

New List

Successor Array

<table>
<thead>
<tr>
<th>2</th>
<th>-</th>
<th>1</th>
</tr>
</thead>
</table>

Global Ranks

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>2</th>
</tr>
</thead>
</table>

Local Ranks

<table>
<thead>
<tr>
<th>0</th>
<th>3</th>
<th>1</th>
<th>2</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
</table>

Final Ranks

| 0 | 5 | 1 | 4 | 0 | 3 | 2 | 0 | 0 | 0 |
Step 6. Add the global ranks to the corresponding local ranks to get the final rank of the list.
Step 6. Add the global ranks to the corresponding local ranks to get the final rank of the list.
Step 6. Add the global ranks to the corresponding local ranks to get the final rank of the list.

New List
Successor Array
Global Ranks
Local Ranks
Final Ranks
Step 6. Add the global ranks to the corresponding local ranks to get the final rank of the list.

New List
Successor Array
Global Ranks
Local Ranks
Final Ranks
Step 6. Add the global ranks to the corresponding local ranks to get the final rank of the list.
Assume you select $n/\log n$ splitters (for large number of threads)

The resulting list itself will be very large

Will slow down the computation as that has to be computed sequentially

Amdahl’s Law comes into effect

- Large portion of the time is spent in serial section of the Code
Solution: Recursive H-J

- Recursively apply the algorithm in parallel on the newly generated list
  - Stop at a list that’s small enough to process very quickly by one thread
- We continue to extract parallelism until the resultant list becomes small
Make step 5 **recursive** to allow the GPU to continue processing the list in parallel.
Make step 5 **recursive** to allow the GPU to continue processing the list in parallel.

Process this list again using the algorithm and reduce it further.
Each phase is coded as separate GPU kernel

Since each step requires global synchronization.

- Splitter Selection
  - Each thread chooses a splitter

- Local Ranking
  - Each thread traverses its corresponding sublist and get the global ranks

- Recursive Step

- Recombination Step
  - Each thread adds the global and local ranks for each element
When do we stop?

- Convergence can be met until list size is 1
- We also have the option to send a small list to CPU or Wyllie’s algorithm so that it can be processed faster than on this algorithm.
- May save about 1% time
Choosing the right amount of splitters

- Notice that choosing splitters in a random list yields uneven sublists.
- We can attempt to load balance the algorithm by varying the no. of splitters we choose.
- \( n/\log n \) works for small lists, \( n/2 \log^2 n \) works well for lists \( > 8 \text{ M} \).
Results – Runtime w.r.t Wyllie’s

<table>
<thead>
<tr>
<th>List Size</th>
<th>Wyllie (RO)</th>
<th>RHJ (log n)</th>
<th>RHJ (2log2n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1K</td>
<td>0.052</td>
<td>0.297</td>
<td>0.259</td>
</tr>
<tr>
<td>2K</td>
<td>0.058</td>
<td>0.27</td>
<td>0.493</td>
</tr>
<tr>
<td>4K</td>
<td>0.059</td>
<td>0.351</td>
<td>0.501</td>
</tr>
<tr>
<td>8K</td>
<td>0.083</td>
<td>0.378</td>
<td>1.089</td>
</tr>
<tr>
<td>16K</td>
<td>0.125</td>
<td>0.501</td>
<td>1.444</td>
</tr>
<tr>
<td>32K</td>
<td>0.237</td>
<td>0.538</td>
<td>2.369</td>
</tr>
<tr>
<td>64K</td>
<td>0.419</td>
<td>0.692</td>
<td>2.914</td>
</tr>
<tr>
<td>128K</td>
<td>1.123</td>
<td>0.926</td>
<td>3.731</td>
</tr>
<tr>
<td>256K</td>
<td>2.86</td>
<td>1.576</td>
<td>4.992</td>
</tr>
<tr>
<td>512K</td>
<td>6.535</td>
<td>3.001</td>
<td>8.557</td>
</tr>
<tr>
<td>1M</td>
<td>14.49</td>
<td>6.188</td>
<td>16.32</td>
</tr>
<tr>
<td>2M</td>
<td>32.83</td>
<td>13.04</td>
<td>30.09</td>
</tr>
<tr>
<td>4M</td>
<td>73.01</td>
<td>27.64</td>
<td>49.75</td>
</tr>
<tr>
<td>8M</td>
<td>163.3</td>
<td>57.52</td>
<td>99.08</td>
</tr>
<tr>
<td>16M</td>
<td>377.6</td>
<td>117.8</td>
<td>200.7</td>
</tr>
<tr>
<td>32M</td>
<td>840.2</td>
<td>239.7</td>
<td>388.2</td>
</tr>
<tr>
<td>64M</td>
<td>1718</td>
<td>488.9</td>
<td></td>
</tr>
</tbody>
</table>
Two factors that contribute to performance
- Random lists
- Load Balancing among threads

Test the effects on four configurations
- Ordered List
- Ordered Unbalanced List
- Random Balanced List
- Random
## Runtime Performance Analysis

### Graph:
- **X-axis:** List Size
- **Y-axis:** Time (Milliseconds)
- Colors:
  - Purple: Ordered List
  - Green: Ordered/Unbalanced
  - Blue: Random/Balanced
  - Red: Random List

### Table:
<table>
<thead>
<tr>
<th>List Size</th>
<th>Ordered List</th>
<th>Ordered/Unbalanced</th>
<th>Random/Balanced</th>
<th>Random List</th>
</tr>
</thead>
<tbody>
<tr>
<td>1M</td>
<td>2.719</td>
<td>5.897</td>
<td>5.649</td>
<td>6.188</td>
</tr>
<tr>
<td>2M</td>
<td>4.956</td>
<td>12.111</td>
<td>12.211</td>
<td>13.038</td>
</tr>
<tr>
<td>4M</td>
<td>9.56</td>
<td>24.523</td>
<td>26.164</td>
<td>27.638</td>
</tr>
<tr>
<td>8M</td>
<td>18.615</td>
<td>49.585</td>
<td>54.992</td>
<td>57.516</td>
</tr>
<tr>
<td>16M</td>
<td>36.432</td>
<td>99.716</td>
<td>112.59</td>
<td>117.771</td>
</tr>
</tbody>
</table>
Results – Ordered Lists

- Ordered Lists are Lists with successor of element \( i = i + 1 \)
- List Ranking on Ordered List is equivalent to scan

<table>
<thead>
<tr>
<th>List Size</th>
<th>CPU</th>
<th>GPU RHJ</th>
<th>CUDPP</th>
</tr>
</thead>
<tbody>
<tr>
<td>512K</td>
<td>3.55</td>
<td>1.61</td>
<td>0.32</td>
</tr>
<tr>
<td>1M</td>
<td>7.12</td>
<td>2.72</td>
<td>0.59</td>
</tr>
<tr>
<td>2M</td>
<td>14.27</td>
<td>4.96</td>
<td>1.13</td>
</tr>
<tr>
<td>4M</td>
<td>28.55</td>
<td>9.56</td>
<td>2.21</td>
</tr>
<tr>
<td>8M</td>
<td>56.62</td>
<td>18.62</td>
<td>4.37</td>
</tr>
<tr>
<td>16M</td>
<td>113.96</td>
<td>36.43</td>
<td>8.68</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Time (Milliseconds)</th>
<th>List Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.10</td>
<td>512K</td>
</tr>
<tr>
<td>1.00</td>
<td>1M</td>
</tr>
<tr>
<td>10.00</td>
<td>2M</td>
</tr>
<tr>
<td>100.00</td>
<td>4M</td>
</tr>
<tr>
<td>1000.00</td>
<td>8M</td>
</tr>
<tr>
<td>10000.00</td>
<td>16M</td>
</tr>
</tbody>
</table>
Comparison with recent Results

Fastest single-chip List Ranking implementation (at time of publishing)
Applications of List Ranking
Euler Tour Technique

- Parallel Processing of Graphs
  - Utilizes List Ranking
- Can be used for a variety of Tree Computations
  - Parent Finding
  - Traversal
  - Vertex Levels
  - Subtree Size
Representing a Tree

- Use two linked lists:
  - Adjacency List (Contains pointers from each vertex to the set of edges it’s adjacent to)
  - Edge List (List of edges in the tree)
Creating an Eulerian Graph from T

Each edge is replaced with a set of anti-parallel arcs, and an additional pointer for each edge pointing to their anti-parallel sibling.
Once the Eulerian graph of Tree $T$ is created, an Euler path can be created:
- Number the edges in some order and create a linked list.

Can be done for all edges in parallel:

$$EP[e] = (EL[e].sib).next$$
Once an Euler Path is constructed, it is ranked using the List Ranking algorithm.

**Tree Computations**

- **Parent Finding**
  - Mark incoming and outgoing edges according to ranks
  - If $R[xy]<R[yx]$ then $\text{Par}[y] = x$

- **Subtree Size**
  - $\text{Subtree}[i] = (\text{Rank}[ij] - \text{Rank}[ji] + 1)/2$ where $j = \text{Par}[i]$
Tree Computations

- **Preorder Traversal**
  - Assign 1 to forward edge, 0 to reverse edge
  - Scan Weights, preorder numbering is obtained

- **Vertex Levels**
  - Assign 1 to forward edge, -1 to reverse edge
  - Scan weights, level of each vertex is obtained.
On the GPU

- Load Tree
  - Construct Euler Path $EP$
    - Perform List Rank on $EP$
      - Find Parent of each Node
        - Rooted Tree

- Subtree Size Calculation
- Assign Weights (1,-1)
  - CUDPP Scan
    - Vertex Level
- Assign Weights (0,1)
  - CUDPP Scan
    - Traversal
Results for GPU

Tree Rooting and Sublist Size Calculation (Random Binary Trees)

<table>
<thead>
<tr>
<th>No. of Vertices</th>
<th>GPU ETT</th>
<th>GPU BFS</th>
<th>CPU BFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1M</td>
<td>7.50</td>
<td>17.29</td>
<td>66.59</td>
</tr>
<tr>
<td>2M</td>
<td>14.71</td>
<td>32.14</td>
<td>142.29</td>
</tr>
<tr>
<td>4M</td>
<td>28.11</td>
<td>63.94</td>
<td>319.59</td>
</tr>
<tr>
<td>8M</td>
<td>56.70</td>
<td>125.90</td>
<td>578.73</td>
</tr>
<tr>
<td>16M</td>
<td>111.28</td>
<td>253.42</td>
<td>1174.66</td>
</tr>
</tbody>
</table>

Time (Milliseconds)
Timing Breakup

### Parent Finding

<table>
<thead>
<tr>
<th>No. of Vertices</th>
<th>FindParent</th>
<th>List Ranking</th>
<th>EP Construction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1M</td>
<td>0.77</td>
<td>5.21</td>
<td>1.52</td>
</tr>
<tr>
<td>2M</td>
<td>1.49</td>
<td>10.26</td>
<td>2.96</td>
</tr>
<tr>
<td>4M</td>
<td>2.93</td>
<td>19.28</td>
<td>5.90</td>
</tr>
<tr>
<td>8M</td>
<td>5.88</td>
<td>39.17</td>
<td>11.65</td>
</tr>
<tr>
<td>16M</td>
<td>11.70</td>
<td>76.35</td>
<td>23.23</td>
</tr>
</tbody>
</table>

### Preorder Traversal

<table>
<thead>
<tr>
<th>No. of Vertices</th>
<th>Last Kernel</th>
<th>CUDPP Scan</th>
<th>FindParent</th>
<th>List Ranking</th>
<th>EP Construction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1M</td>
<td>0.66</td>
<td>0.55</td>
<td>1.12</td>
<td>5.21</td>
<td>1.52</td>
</tr>
<tr>
<td>2M</td>
<td>1.30</td>
<td>0.98</td>
<td>2.16</td>
<td>10.26</td>
<td>2.96</td>
</tr>
<tr>
<td>4M</td>
<td>2.60</td>
<td>1.72</td>
<td>4.24</td>
<td>19.28</td>
<td>5.90</td>
</tr>
<tr>
<td>8M</td>
<td>5.17</td>
<td>3.31</td>
<td>8.40</td>
<td>39.17</td>
<td>11.65</td>
</tr>
<tr>
<td>16M</td>
<td>10.36</td>
<td>6.50</td>
<td>16.90</td>
<td>76.35</td>
<td>23.23</td>
</tr>
</tbody>
</table>
Performance Prediction for Irregular Applications on the GPU
A Performance prediction model that lets you estimate the runtime of a GPU program
- Inputs from PRAM, BSP, QRQW models
- Developed by Kothapalli et. al.
- Based on Cycle Estimation

\[ C(K) = N_B(K) \times N_w(K) \times N_t(K) \times C_T(K) \times \frac{1}{N_C \times D} \] cycles

\[ T(K) = \frac{C(K)}{R} \] seconds
Kernel Cycles consists of
- Data Instructions
- Compute Instructions
  - Simple to compute

Difficult to estimate time taken for a data instruction to be serviced
- Global memory can take around 300 cycles compared to 4 cycles for compute
- Performance is Memory bound
Global Memory Coalescing

- Global Memory can fetch up to 128 bytes in a single transaction
  - If all threads of a half-warp access elements in 128 byte segment at the same time – then bandwidth improves
- Unlikely to happen with irregular memory applications
- What’s the effect of un-coalesced memory accesses?
Effect on Bandwidth

Copy with Offset

Copy with Stride
How do you model Irregular Applications?

- Estimate how much coalescing is happening
  - “Instantaneous Degree of Locality”
- Estimate cycles per warp
  - Regard all memory transactions are uncoalesced for irregular applications.
  - Memory instructions will overshadow compute in number of cycles
Modeling List Ranking

Calculations

\[
\frac{N}{512 \times 30} \times 16 \times \frac{32}{8 \times 4} \times 4 \log N \times 3 \times 500 \times \frac{1}{1.3 \times 10^9} \text{ sec}
\]

<table>
<thead>
<tr>
<th>List Size</th>
<th>No of Blocks</th>
<th>Blocks per SM</th>
<th>Compute Cycles</th>
<th>Memory Cycles</th>
<th>Expected Runtime</th>
<th>Actual Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>256K</td>
<td>28</td>
<td>1</td>
<td>288</td>
<td>108000</td>
<td>1.333</td>
<td>0.773</td>
</tr>
<tr>
<td>512K</td>
<td>54</td>
<td>2</td>
<td>304</td>
<td>114000</td>
<td>2.814</td>
<td>1.909</td>
</tr>
<tr>
<td>1M</td>
<td>102</td>
<td>3</td>
<td>320</td>
<td>120000</td>
<td>4.443</td>
<td>4.404</td>
</tr>
<tr>
<td>2M</td>
<td>195</td>
<td>7</td>
<td>336</td>
<td>126000</td>
<td>10.884</td>
<td>9.588</td>
</tr>
<tr>
<td>4M</td>
<td>372</td>
<td>13</td>
<td>352</td>
<td>132000</td>
<td>21.176</td>
<td>19.958</td>
</tr>
<tr>
<td>8M</td>
<td>712</td>
<td>25</td>
<td>368</td>
<td>138000</td>
<td>42.575</td>
<td>40.806</td>
</tr>
<tr>
<td>16M</td>
<td>1365</td>
<td>47</td>
<td>384</td>
<td>144000</td>
<td>83.521</td>
<td>82.542</td>
</tr>
</tbody>
</table>
Accuracy of the Model

![Graph showing the relationship between time (milliseconds) and list size. The graph compares expected runtime (blue line) and actual runtime (red line) across various list sizes: 256K, 512K, 1M, 2M, 4M, 8M, and 16M.]
Strategies for Implementing Irregular Algorithms

- Exploiting massive parallelism of the GPU
  - Enough threads to offset memory access latencies
  - Small input data which fits on CPU cache will be faster on CPU
- Avoid over-reliance on expensive synchronization
- Load Balancing through probabilistic means of dividing work
Conclusions and Future Work

- Pointer Jumping for GPUs
  - Implemented with some optimization techniques for problems that require it.
- List Ranking for GPUs
  - Our implementation was the fastest single-chip implementation at time of publishing
  - Applied LR for Tree Computations with good results
- Strategies for Irregular Algorithms on the GPU
2007 – 2008 Course Work at IIIT-H
- Parallel Algorithms, Parallel Programming, Multicore Architectures, Graph Theory
August 2008 – Work on List Ranking Started
Jan 2009 – Work accepted at ICS 2009
August 2009 – Started work on Tree Computations
Current Work – Book Chapter in GPU Computing Gems, Tree Computations on GPU and Larger Applications
April 2010: Wei and JaJa’s paper at IPDPS 2010 improves upon our work, 30% faster timings optimizing thread count and load balancing with appropriate splitter selection.
Thank You!