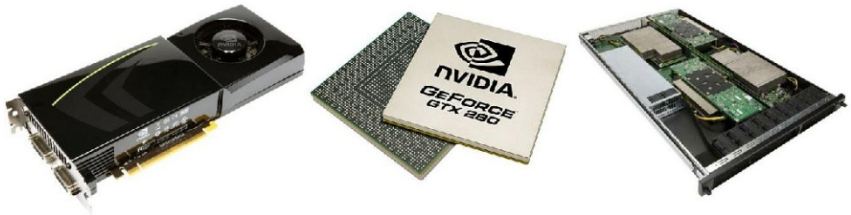
# Exploring Irregular Memory Access Applications on the GPU

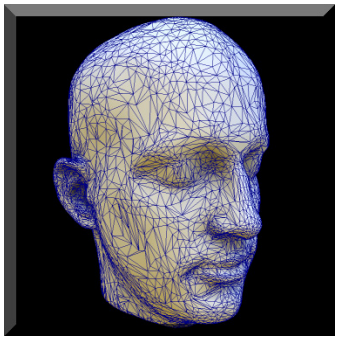
Mohammed Suhail Rehman
rehman@research.iiit.ac.in
200707016
Advisors: Prof. Kishore Kothapalli, Prof. P.J. Narayanan
International Institute of Information Technology

# 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

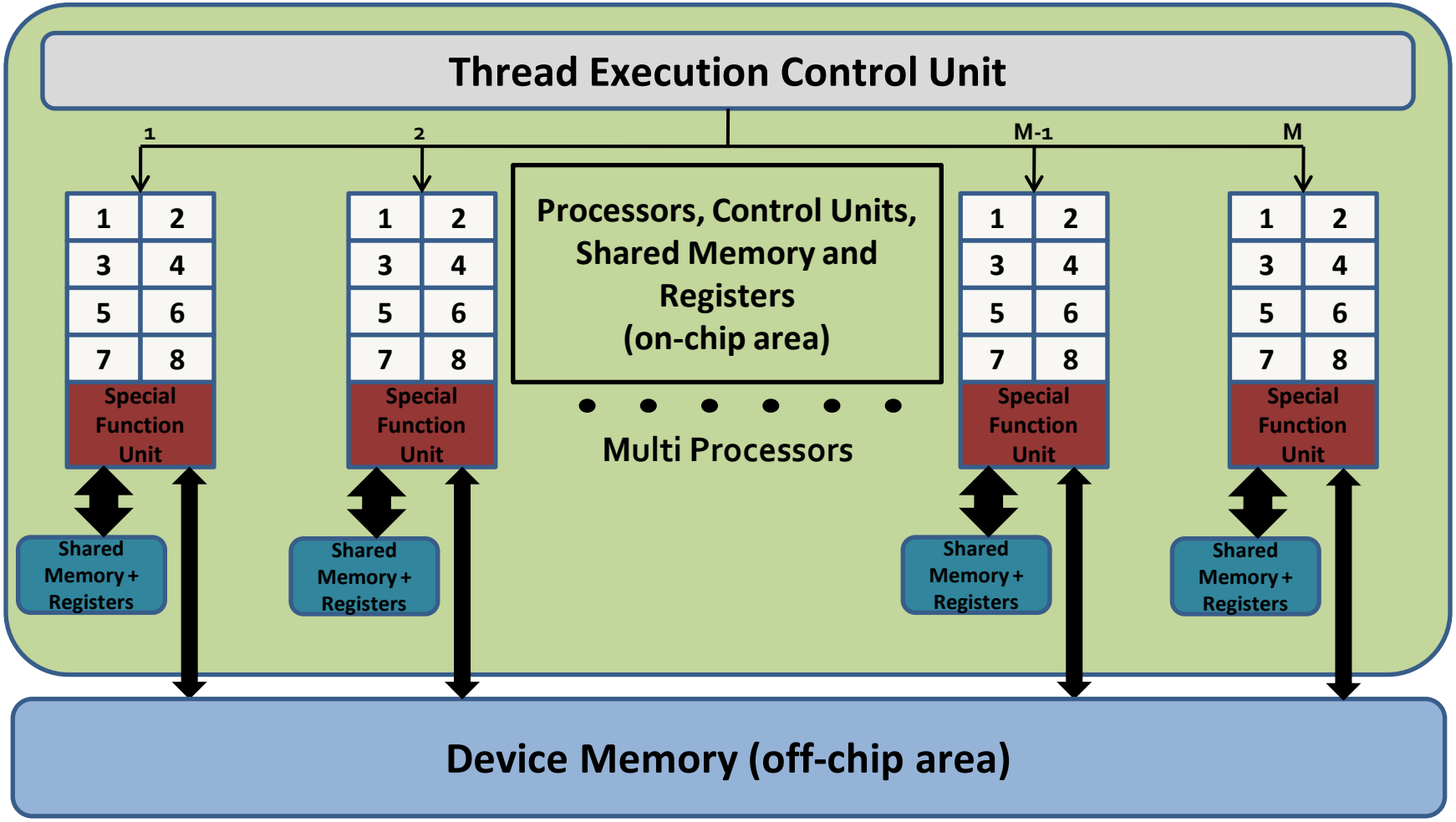
- Pointer Jumping Technique for the GPU
- Wyllie's Algorithm and variants for the GPU
- A work-optimal, recursive algorithm for the GPU
- Applications of List Ranking
  - Euler Tour Construction and Tree Computations on the GPU
- Performance Analysis and Behavior of such Irregular Algorithms on the GPU

# Introduction and Background

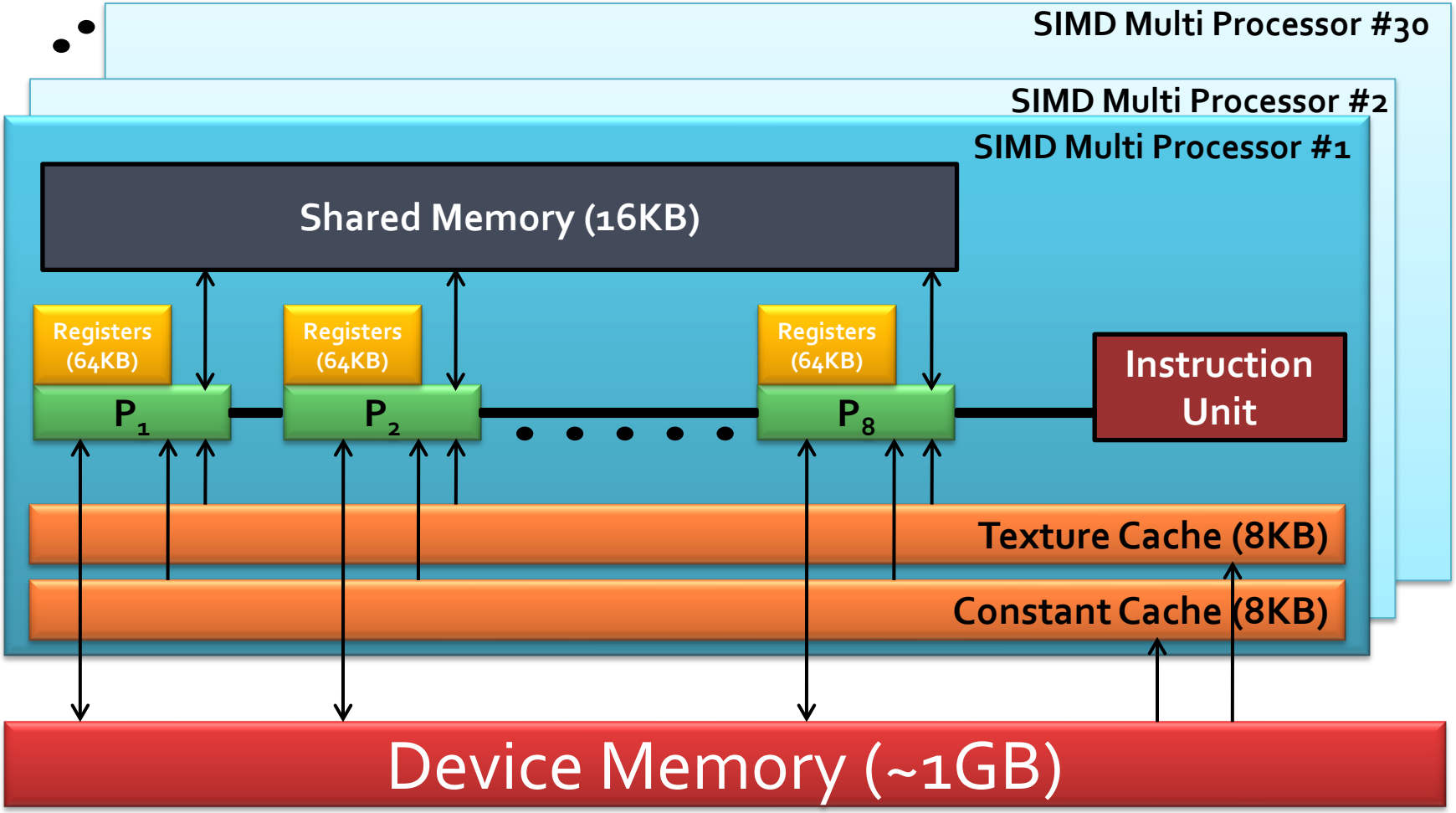
# GPU by the Numbers (GTX280)

- 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

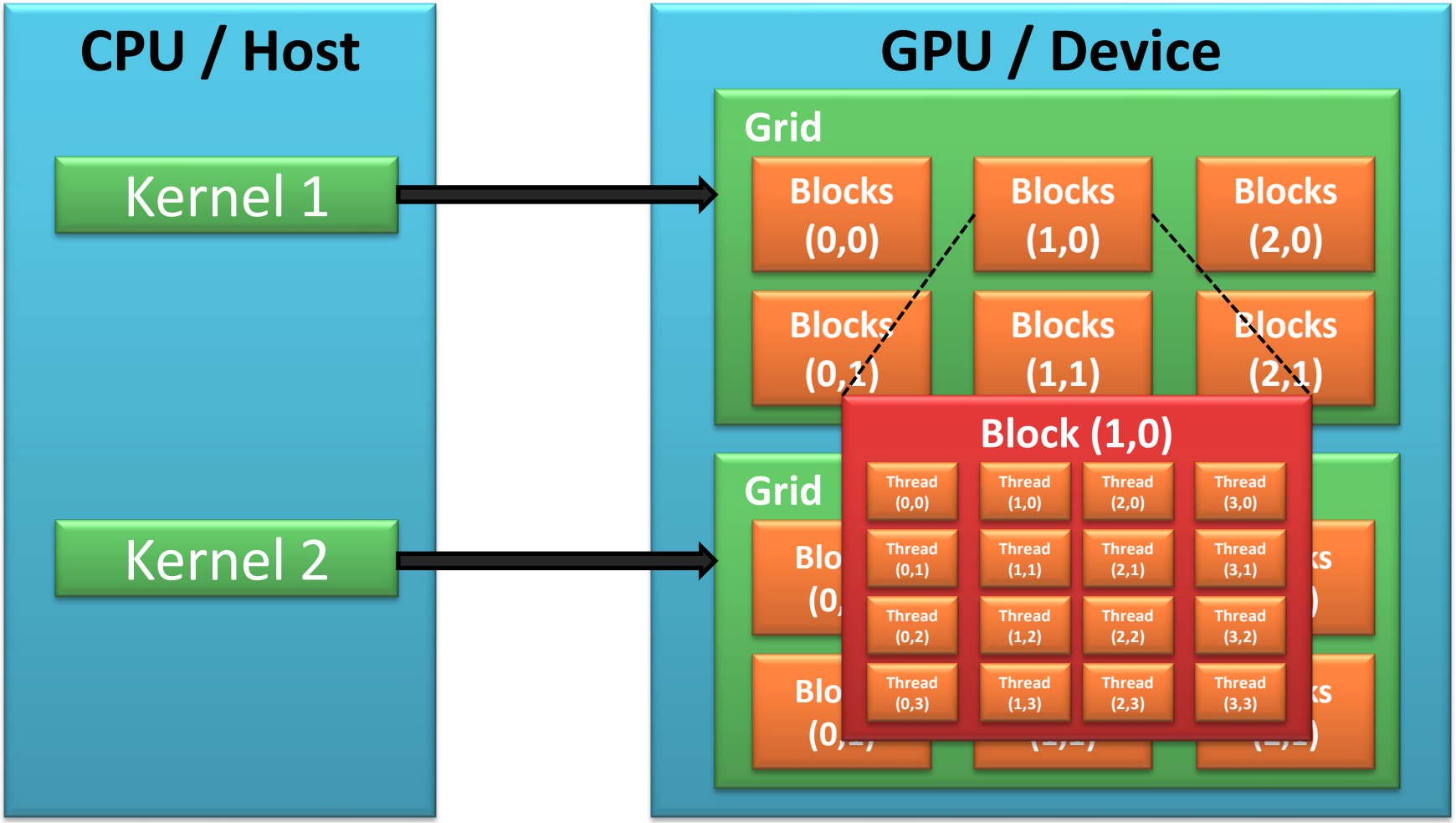
# GPU Architecture



# CUDA Hardware Architecture



# CUDA Software Architecture



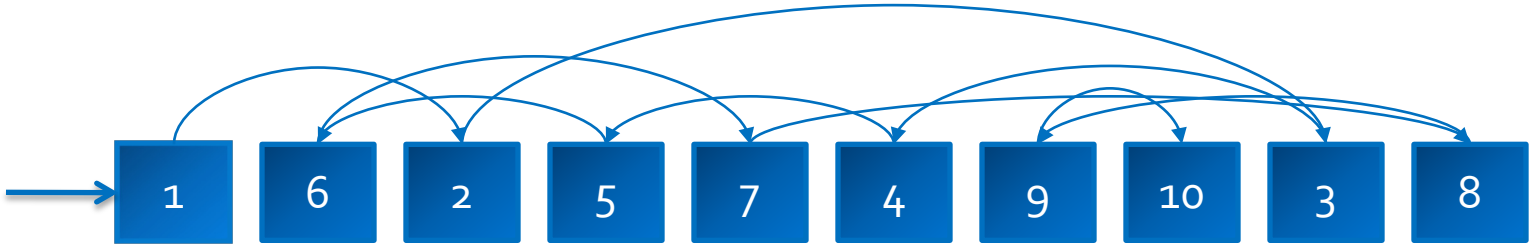
# Data-Parallel Primitives on the GPU

- 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




Unordered List

# List Ranking Algorithms

| Algorithm | Time | Work | Constants | Space |
|---|---|---|---|---|
| Serial | $O(n)$ | $O(n)$ | small | $c$ |
| Wyllie[16] | $O(\frac{n \log n}{p} + \log n)$ | $O(n \log n)$ | small | $n + c$ |
| Randomized[36, 37] | $O(\frac{n}{p} + \log n)$ | $O(n)$ | medium | $> 2n$ |
| PRAM Optimal [32, 33] | $O(\frac{n}{p} + \log n)$ | $O(n)$ | large | $> n$ |
| Miller-Reid[34] | $O(\frac{n}{p} + \log^2 n)$ | $O(n)$ | small | $5p + c$ |
| Helman-JáJá [17] | $O(\frac{n}{p} + \log n)$ | $O(n)$ | small | $n + \log n + c$ |

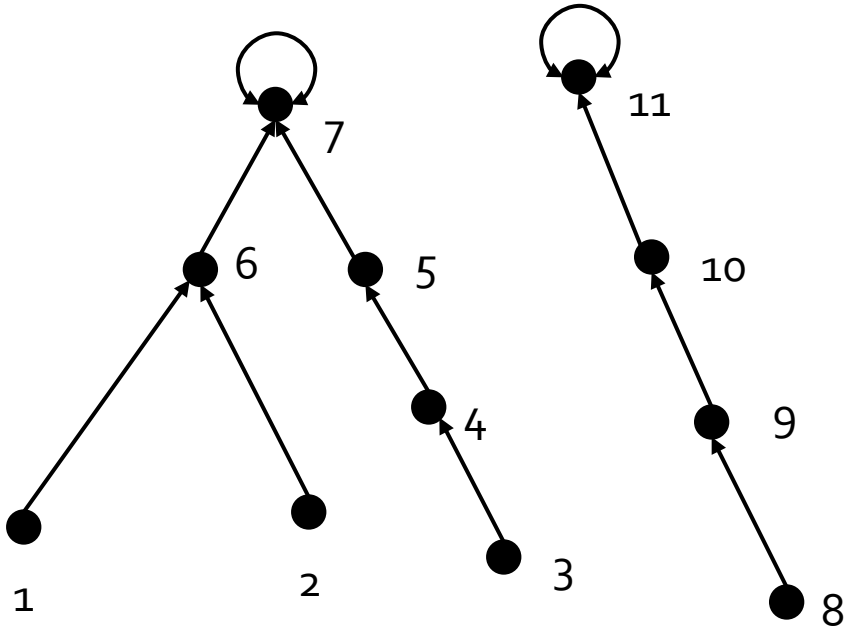
# Applications of List Ranking

- Euler Tour Technique
- Load Balancing
- Tree Contraction
- Expression Evaluation
- Planar Graph Embedding
- Etc..

# Pointer Jumping and Wyllie's Algorithm on the GPU

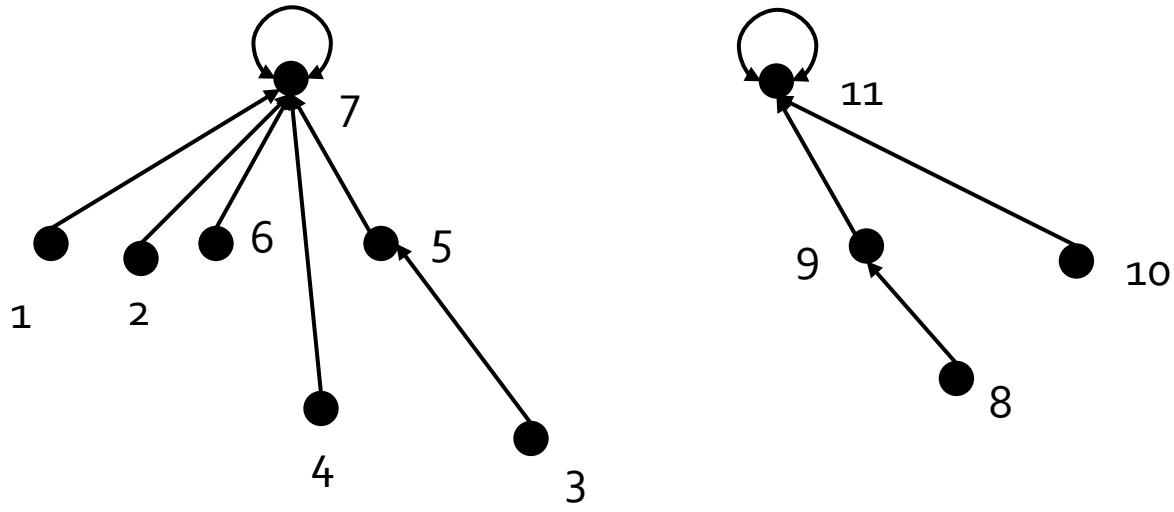
# Pointer Jumping

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



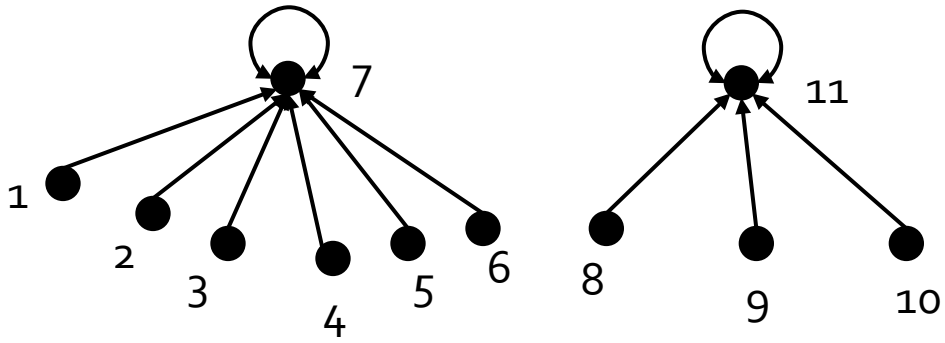
# Pointer Jumping

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



# Pointer Jumping

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



# The Algorithm

**Algorithm 2** Pointer Jumping

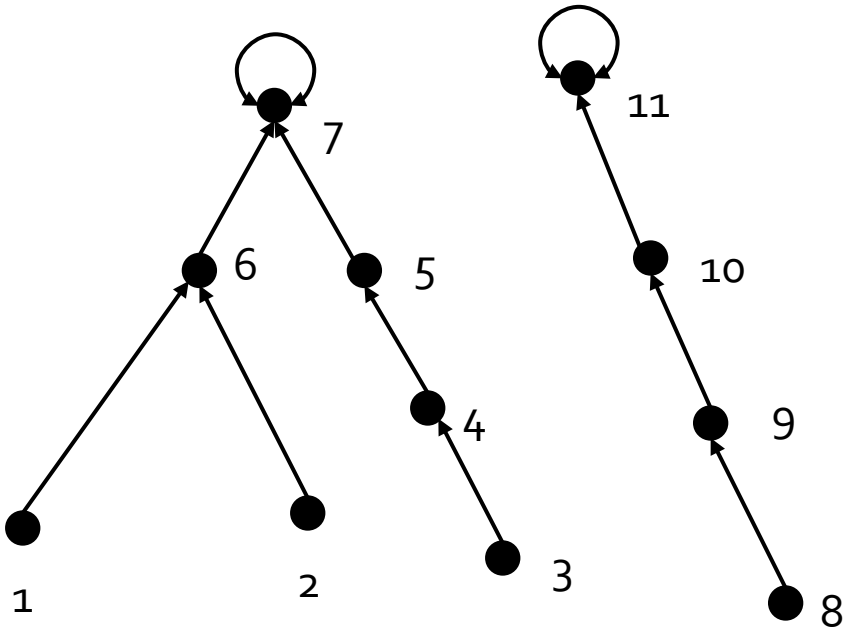
**Input:** A forest of rooted directed trees.

**Output:** For each vertex $i$, the root $S(i)$ of the tree containing $i$

```
1: for 1 ≤ i ≤ n do in parallel
2:     while S(i) is not end of list do
3:         Set S(i) = S(S(i))
4:     end while
5: end for
```

# Pointer Jumping on the GPU

- From PRAM to the GPU
  - One thread per element
  - Successor Array Implementation




| 6 | 6 | 4 | 5 | 7 | 7 | 7 | 9 | 10 | 11 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |

# Wyllie's Algorithm

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

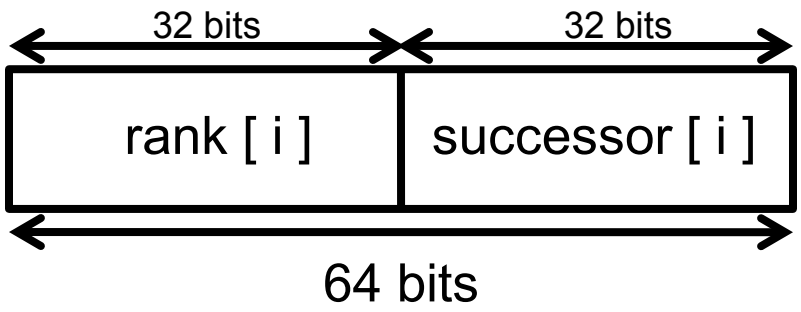
**Algorithm 3** Wyllie's Algorithm

**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

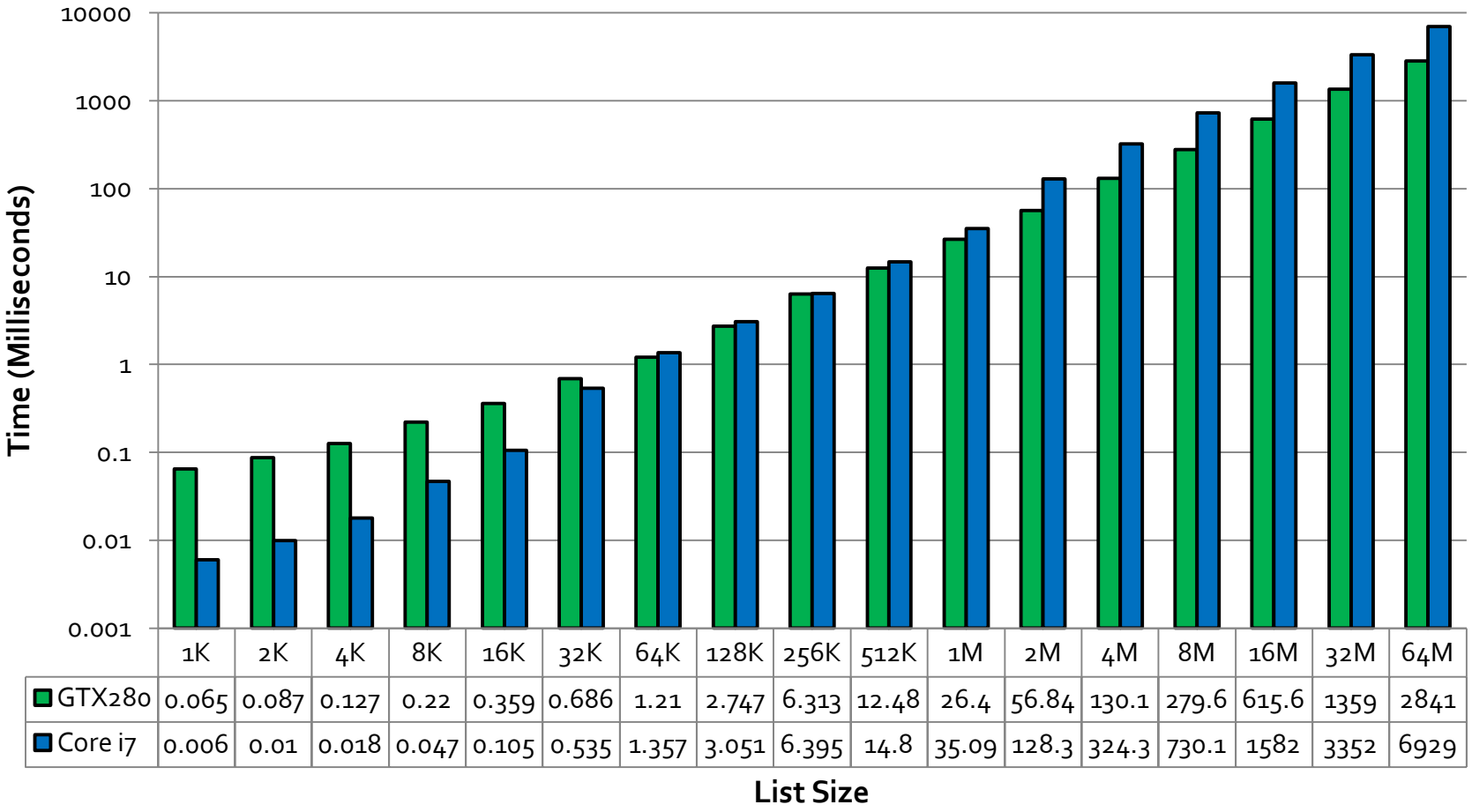
```
1: for each element in S do in parallel
2:     while S[i] and S[S[i]] are not the end of list do
3:         R[i] = R[i] + R[S[i]]
4:         S[i] = S[S[i]]
5:     end while
6: end for
```

# 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



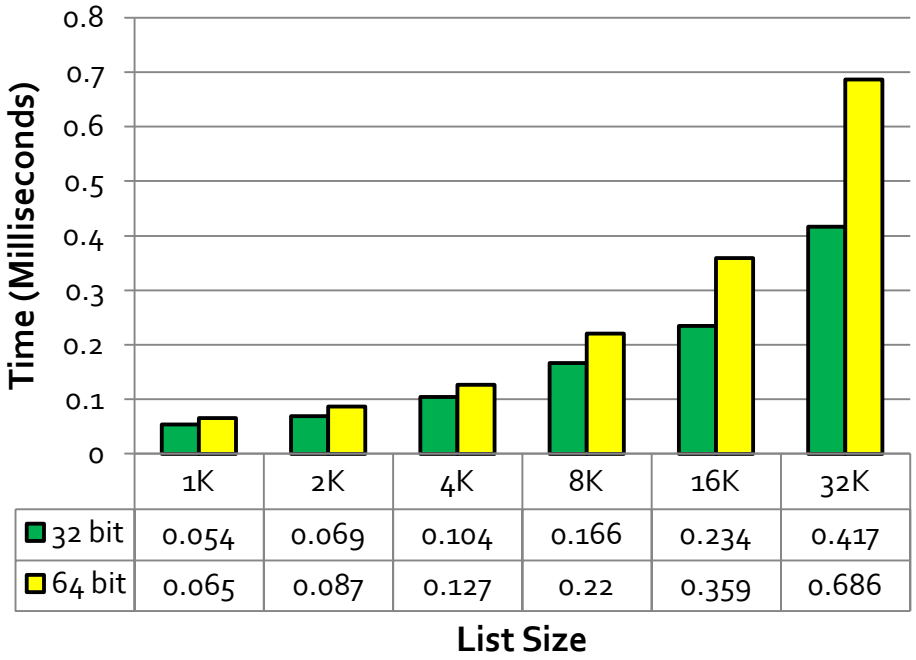
# Results of Wyllie's Algorithm




| | 1K | 2K | 4K | 8K | 16K | 32K | 64K | 128K | 256K | 512K | 1M | 2M | 4M | 8M | 16M | 32M | 64M |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| GTX280 | 0.065 | 0.087 | 0.127 | 0.22 | 0.359 | 0.686 | 1.21 | 2.747 | 6.313 | 12.48 | 26.4 | 56.84 | 130.1 | 279.6 | 615.6 | 1359 | 2841 |
| 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




| | 1K | 2K | 4K | 8K | 16K | 32K |
|---|---|---|---|---|---|---|
| 32 bit | 0.054 | 0.069 | 0.104 | 0.166 | 0.234 | 0.417 |
| 64 bit | 0.065 | 0.087 | 0.127 | 0.22 | 0.359 | 0.686 |

# Optimization II – Lock-Step

- 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.

# Optimization II – Lock-Step

**Algorithm 4** CPU-Orchestrated Wyllie's Algorithm

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

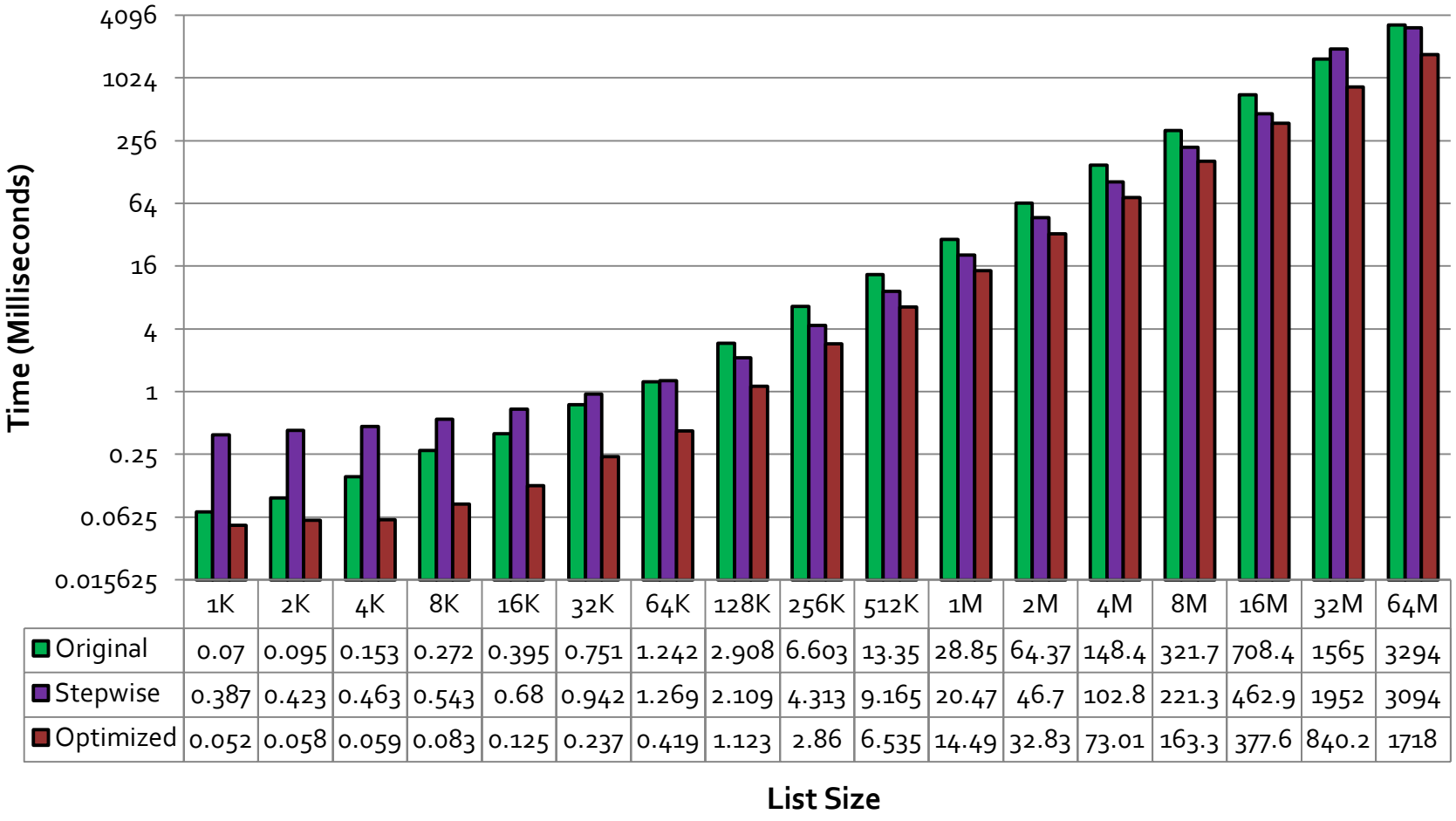
**Output:** $N$ with all the node ranks updated

```
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 r1
5:     if r1.finished = 0 then
6:         Load the successor node in local register r2
7:         if r2.finished = 0 then
8:             Set r1.next = r2.next
9:             done = 0
10:        end if
11:        Set r1.rank = r1.rank + r2.rank
12:        Synchronize and store the node in N designated to this thread with r1
13:    end if
14: end while
```

# Optimization III – Reduced Memory Operations

- Threads would load an element and it's 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



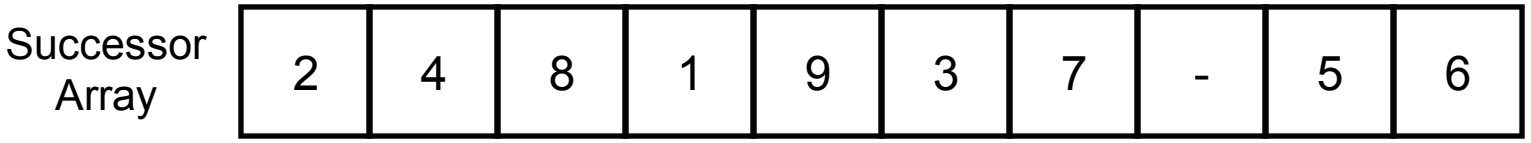

| | 1K | 2K | 4K | 8K | 16K | 32K | 64K | 128K | 256K | 512K | 1M | 2M | 4M | 8M | 16M | 32M | 64M |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Original | 0.07 | 0.095 | 0.153 | 0.272 | 0.395 | 0.751 | 1.242 | 2.908 | 6.603 | 13.35 | 28.85 | 64.37 | 148.4 | 321.7 | 708.4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

# Where is the performance bottleneck?

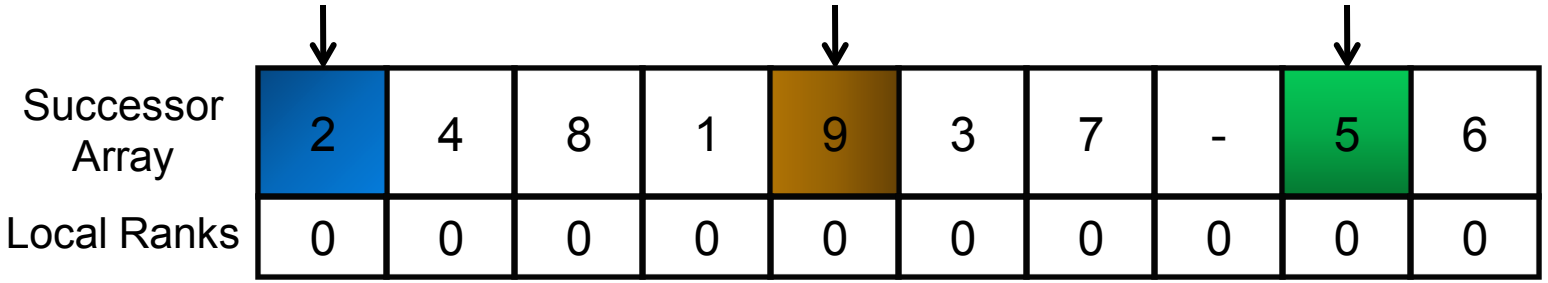
- Wyllie's Algorithm is not optimal in PRAM model
  - $O(n \log n)$ work for something that can be done in $O(n)$
- Expensive synchronization on GPU (64 bit packing and write operations)
- Random Reads from Global Memory
  - ~400-600 GPU cycles of read latency for each element
- There has to be a better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

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Successor Array | 2 | 4 | 8 | 1 | 9 | 3 | 7 | - | 5 | 6 |
| Local Ranks | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 |

Step 2. **Traverse** the List until the next splitter is met and **increment** local ranks as we progress

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Successor Array | 2 | 4 | 8 | 1 | 9 | 3 | 7 | - | 5 | 6 |
| Local Ranks | 0 | 0 | 1 | 2 | 0 | 1 | 2 | 0 | 0 | 1 |

Step 2. **Traverse** the List until the next splitter is met and **increment** local ranks as we progress

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Successor Array | 2 | 4 | 8 | 1 | 9 | 3 | 7 | - | 5 | 6 |
| Local Ranks | 0 | 3 | 1 | 2 | 0 | 1 | 2 | 3 | 0 | 1 |

Step 3. **Stop** When all elements have been assigned a local rank

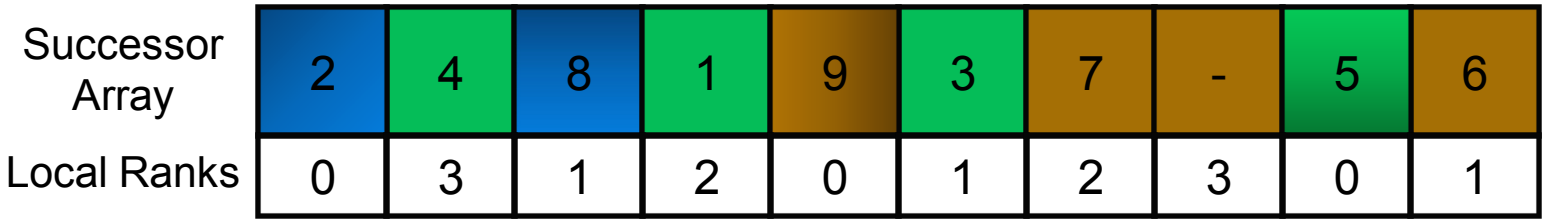
| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Successor Array | 2 | 4 | 8 | 1 | 9 | 3 | 7 | - | 5 | 6 |
| Local Ranks | 0 | 3 | 1 | 2 | 0 | 1 | 2 | 3 | 0 | 1 |

Step 4. **Create** a new list of splitters which contains a **prefix value** that is equal to the local rank of it's predecessor

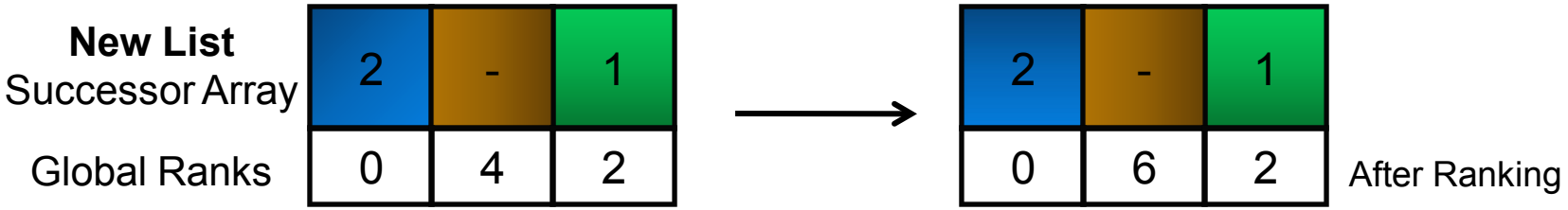
| Successor Array | 2 | 4 | 8 | 1 | 9 | 3 | 7 | - | 5 | 6 |
|---|---|---|---|---|---|---|---|---|---|---|
| Local Ranks | 0 | 3 | 1 | 2 | 0 | 1 | 2 | 3 | 0 | 1 |

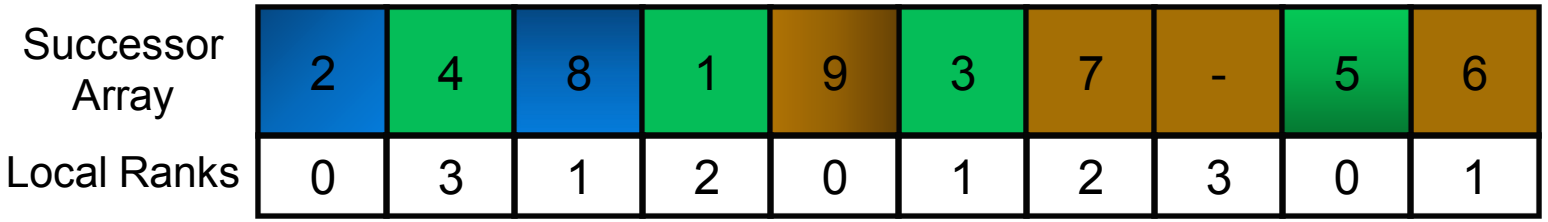
Step 4. **Create** a new list of splitters which contains a **prefix value** that is equal to the local rank of it's predecessor

| **New List** Successor Array | 2 | - | 1 |
|---|---|---|---|
| Global Ranks | 0 | 4 | 2 |

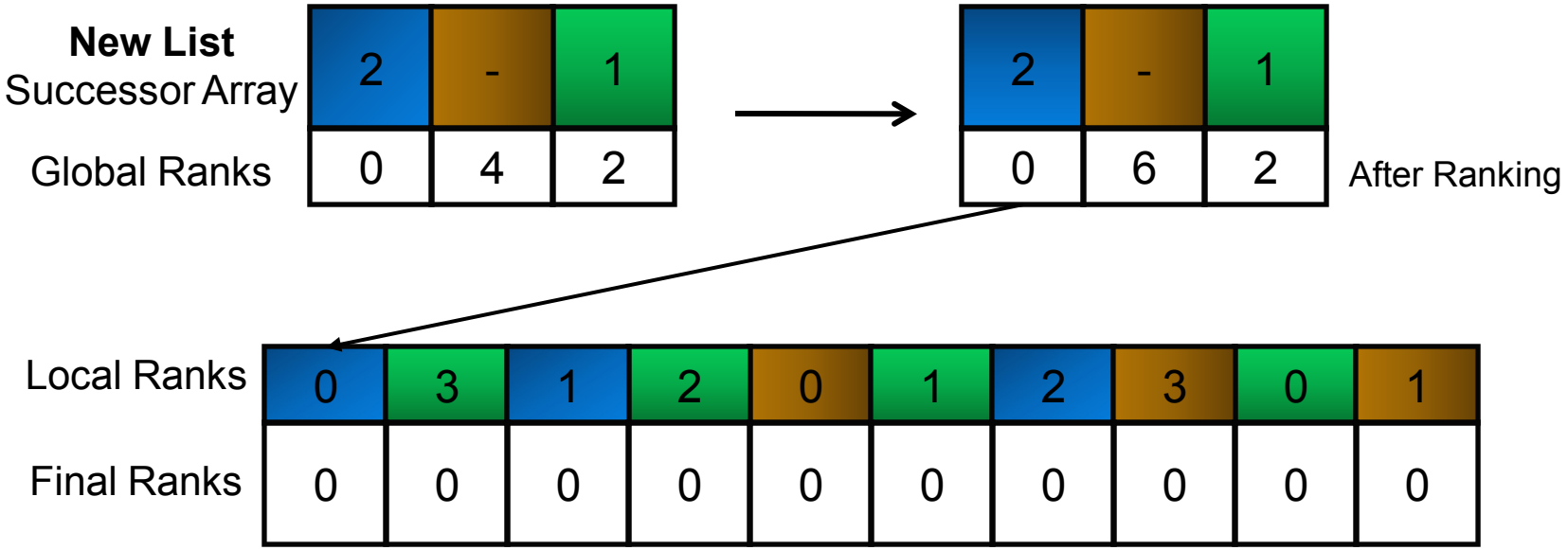


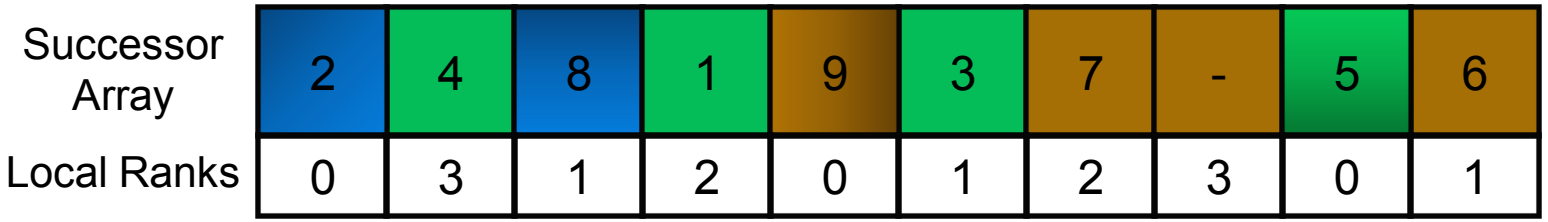

Step 5. Scan the global ranks array **sequentially**



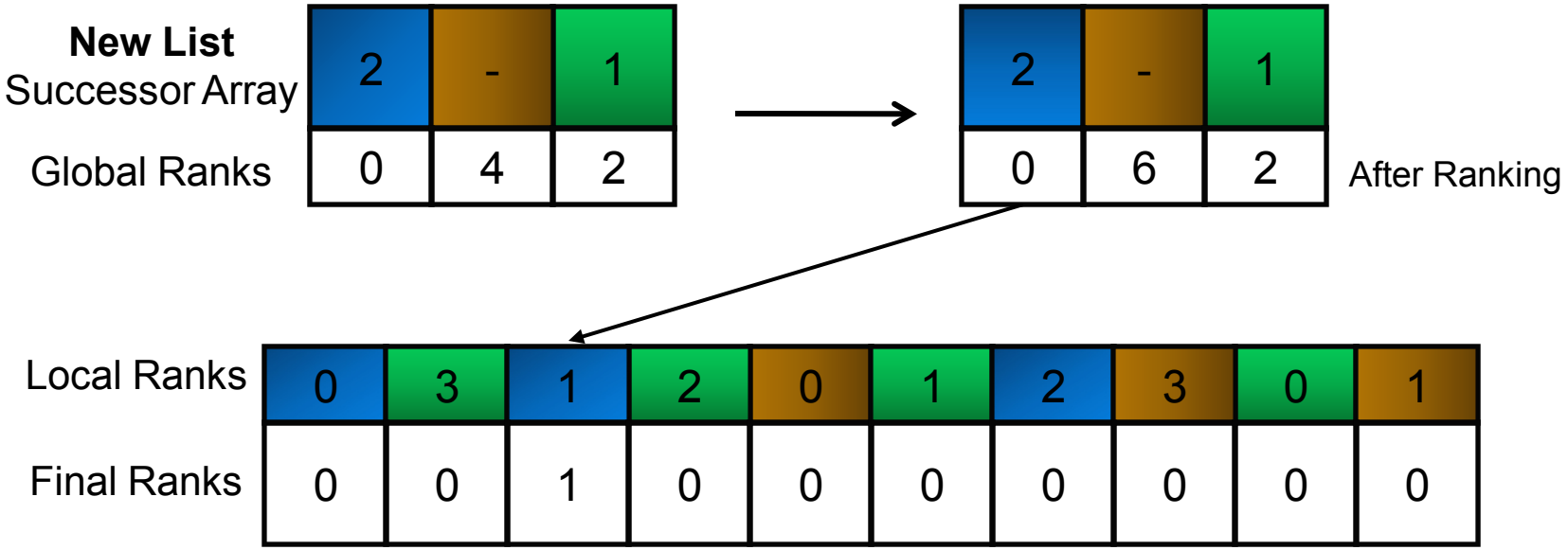


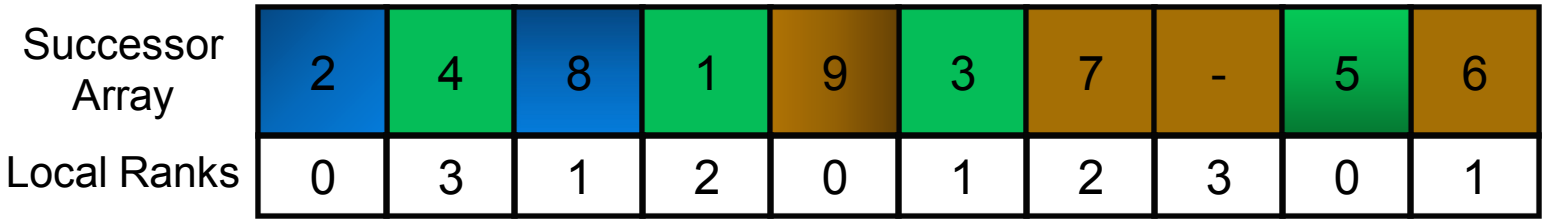

Step 6. Add the global ranks to the corresponding local ranks to get the final rank of the list.



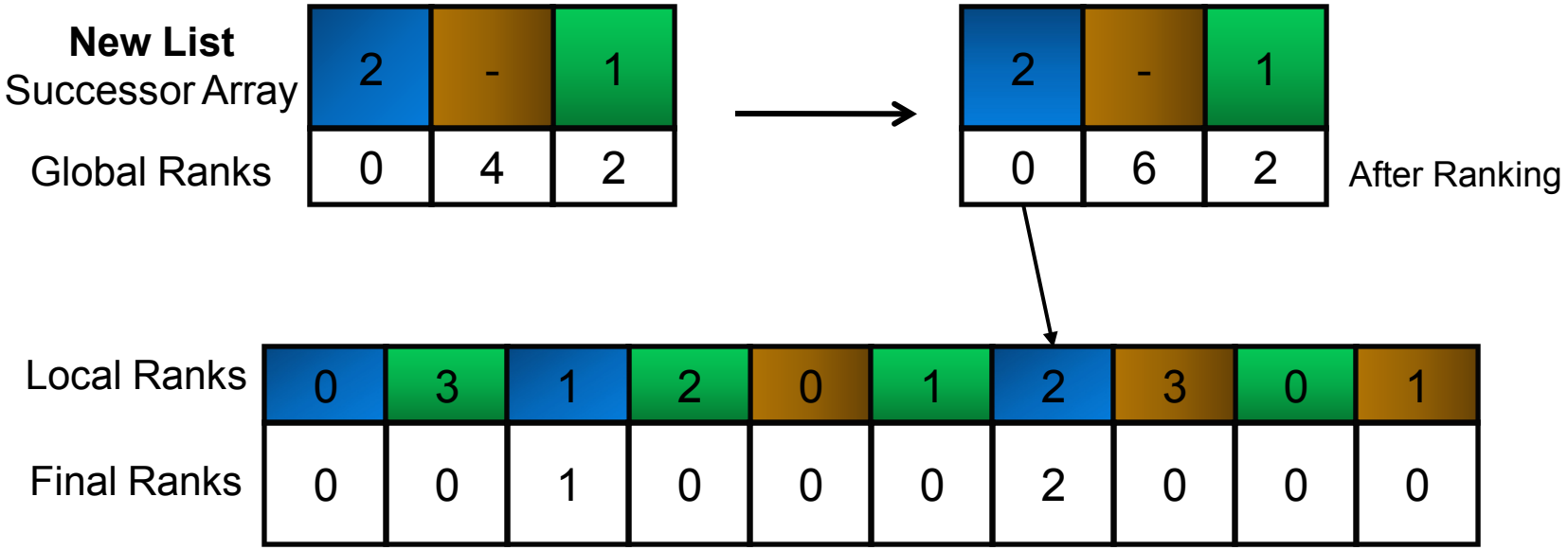



Step 6. Add the global ranks to the corresponding local ranks to get the final rank of the list.



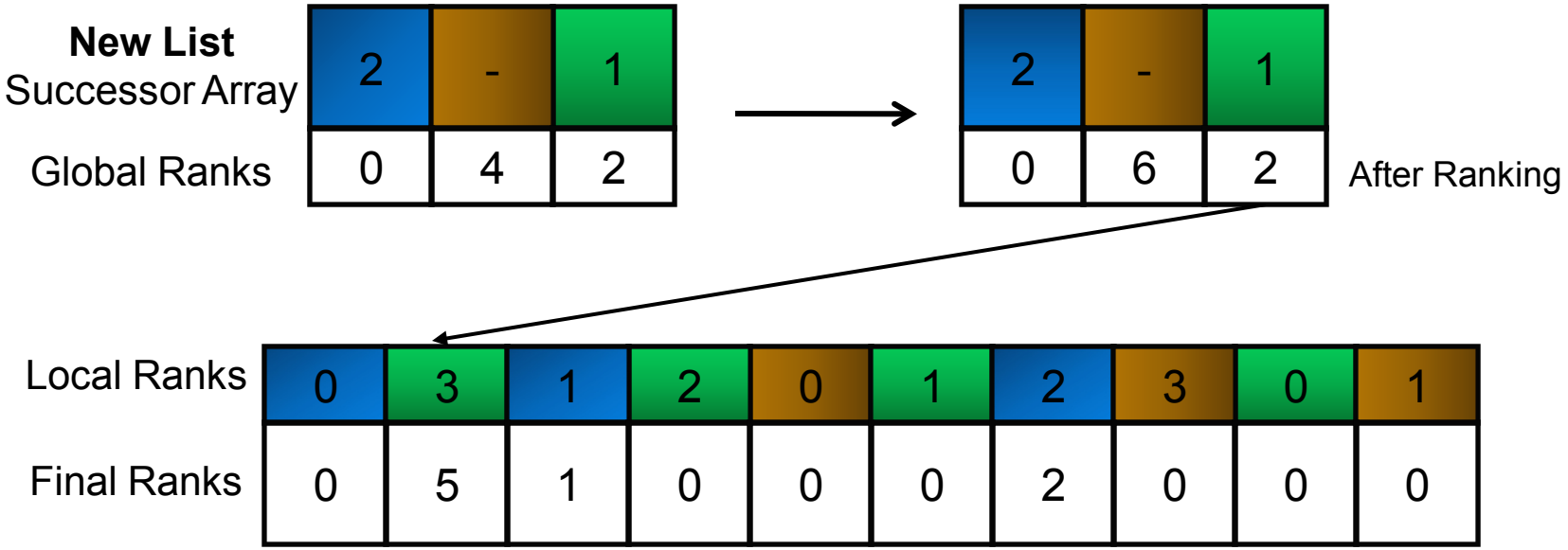


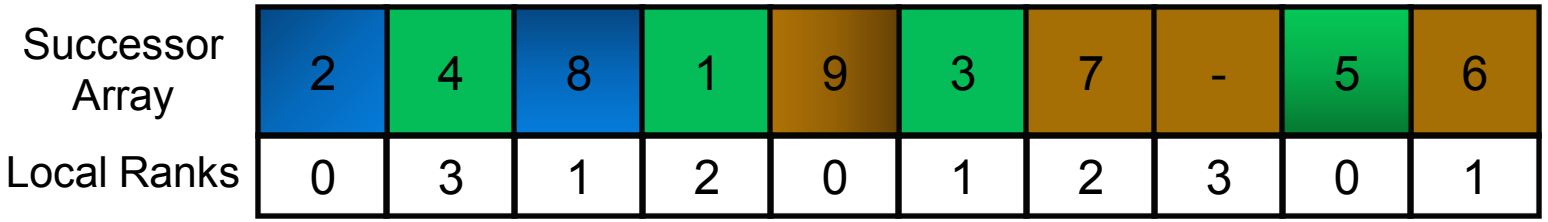

Step 6. Add the global ranks to the corresponding local ranks to get the final rank of the list.



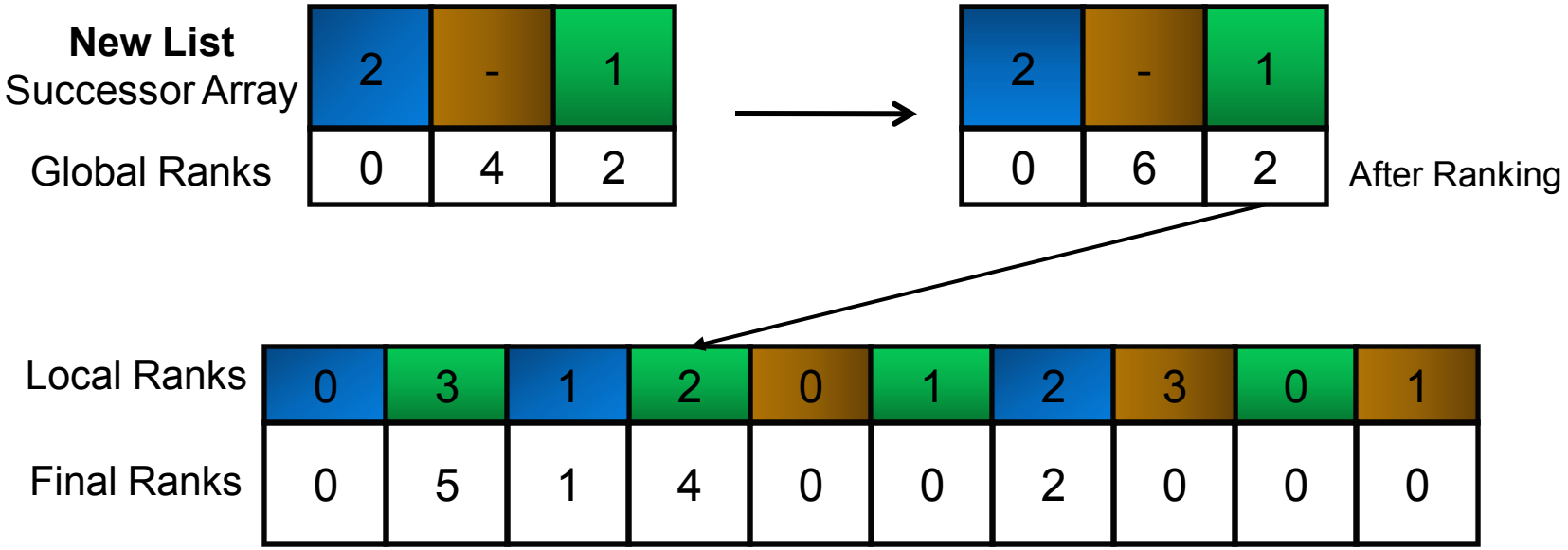
| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Successor Array | 2 | 4 | 8 | 1 | 9 | 3 | 7 | - | 5 | 6 |
| Local Ranks | 0 | 3 | 1 | 2 | 0 | 1 | 2 | 3 | 0 | 1 |

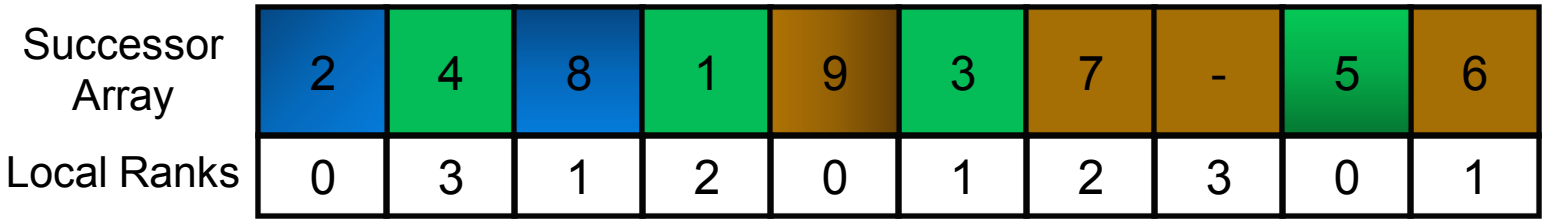
Step 6. Add the global ranks to the corresponding local ranks to get the final rank of the list.



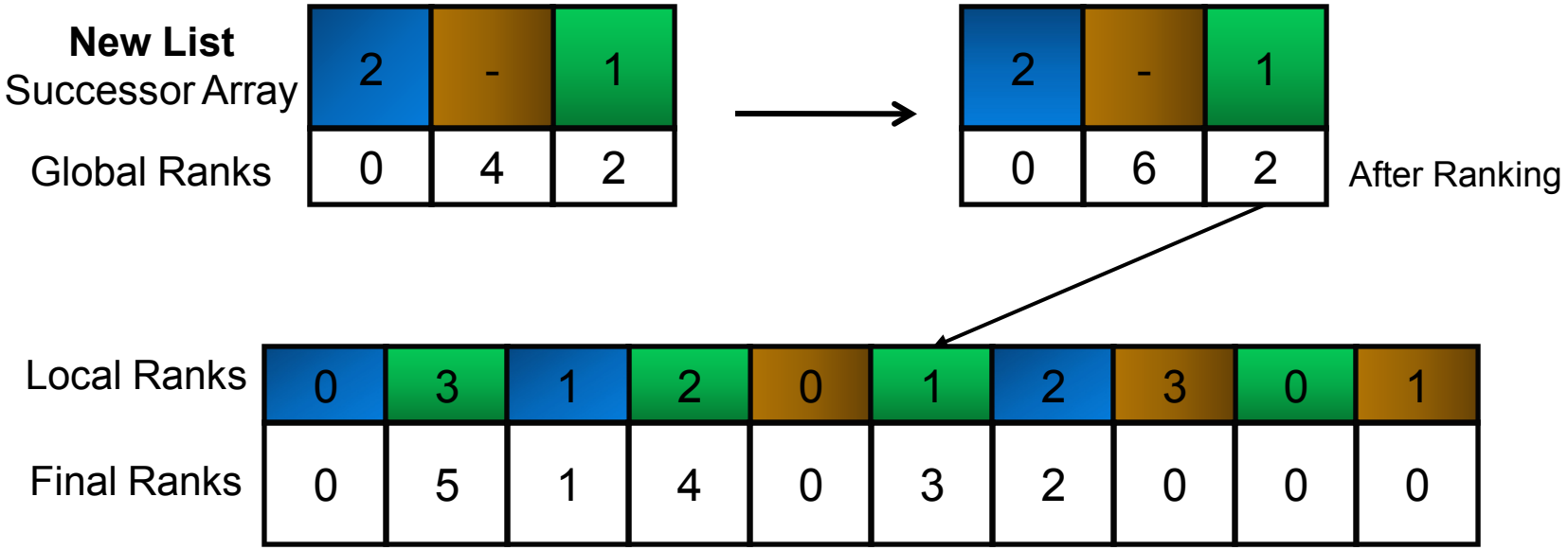


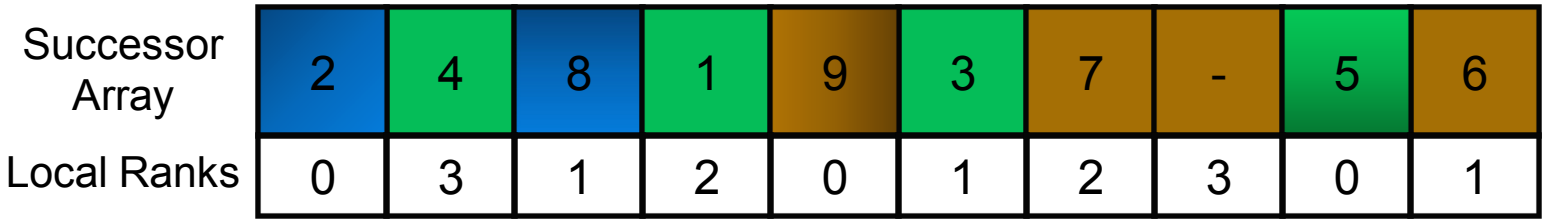

Step 6. Add the global ranks to the corresponding local ranks to get the final rank of the list.



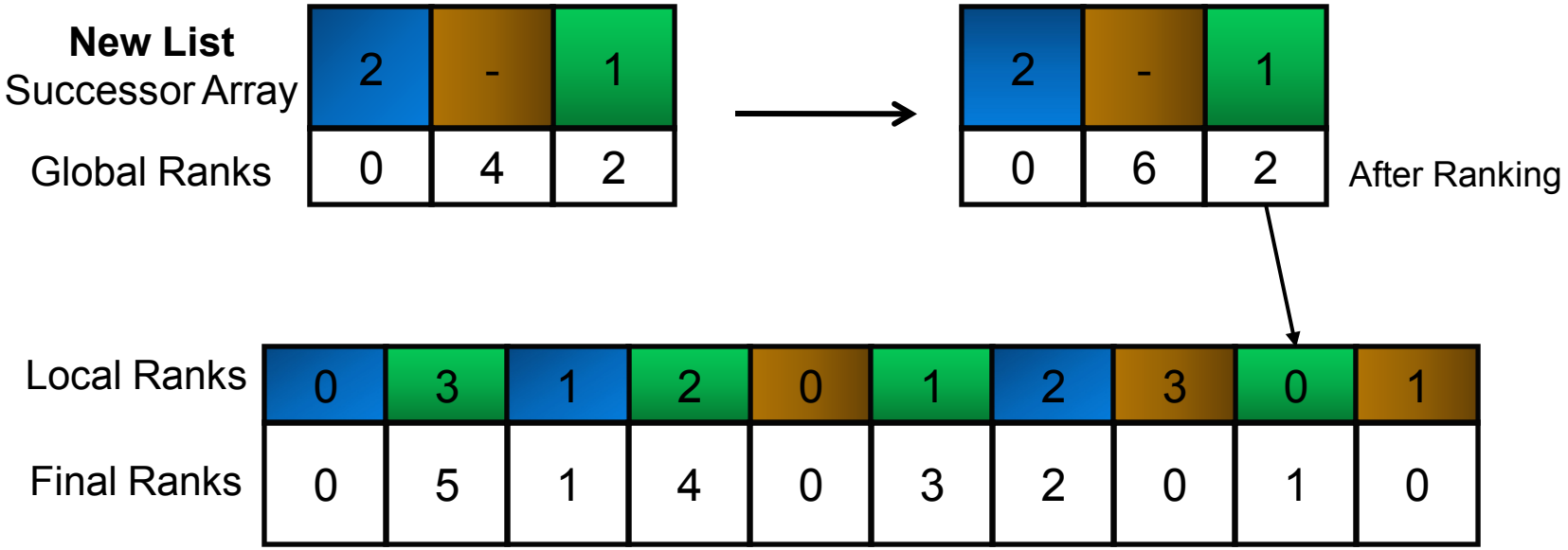



Step 6. Add the global ranks to the corresponding local ranks to get the final rank of the list.






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



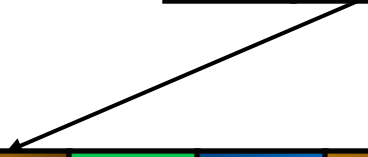
| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Successor Array | 2 | 4 | 8 | 1 | 9 | 3 | 7 | - | 5 | 6 |
| Local Ranks | 0 | 3 | 1 | 2 | 0 | 1 | 2 | 3 | 0 | 1 |

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

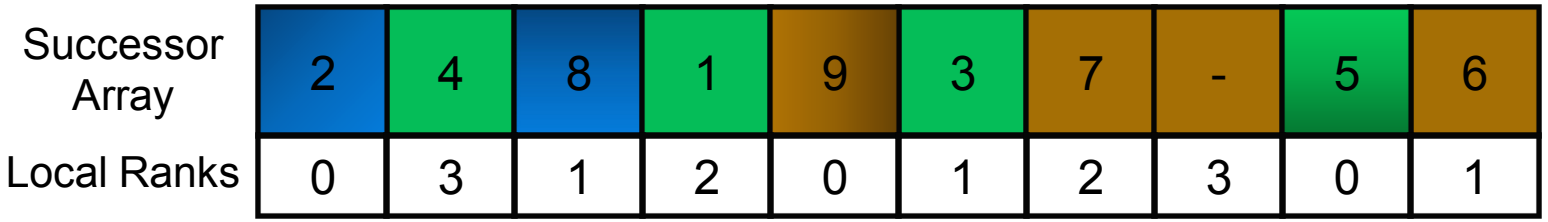
| **New List** Successor Array | 2 | - | 1 |
|---|---|---|---|
| Global Ranks | 0 | 4 | 2 |

| | | | |
|---|---|---|---|
| | 2 | - | 1 |
| | 0 | 6 | 2 |

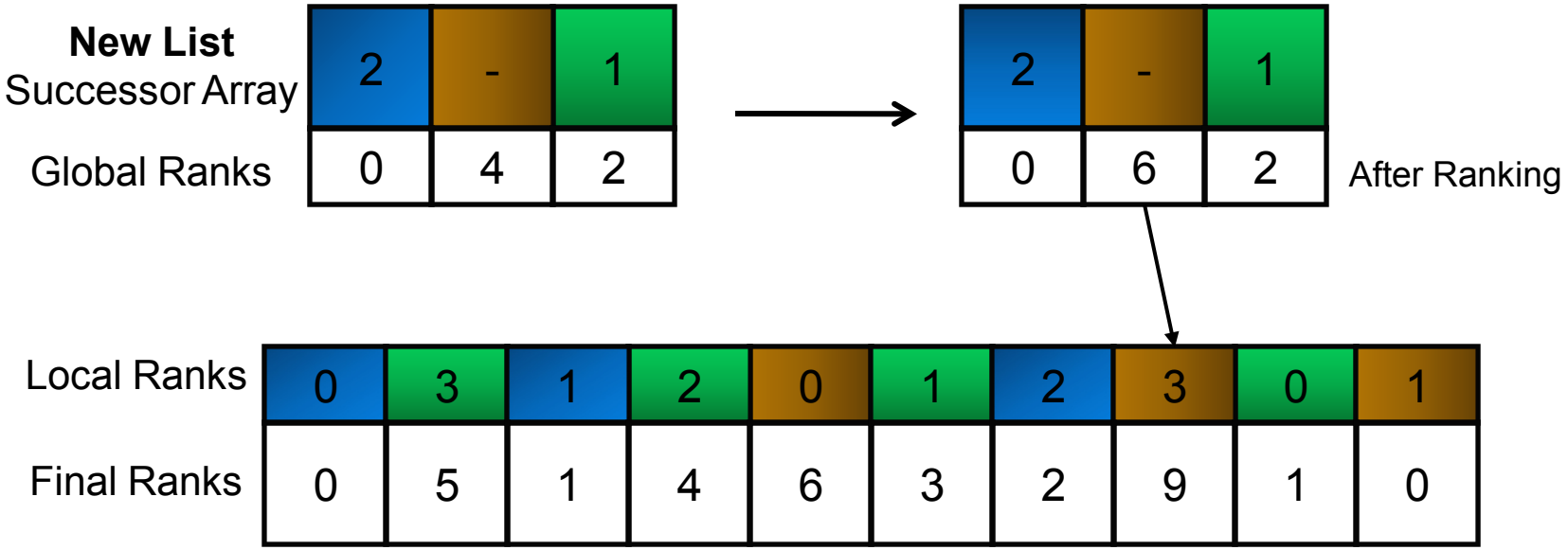
After Ranking

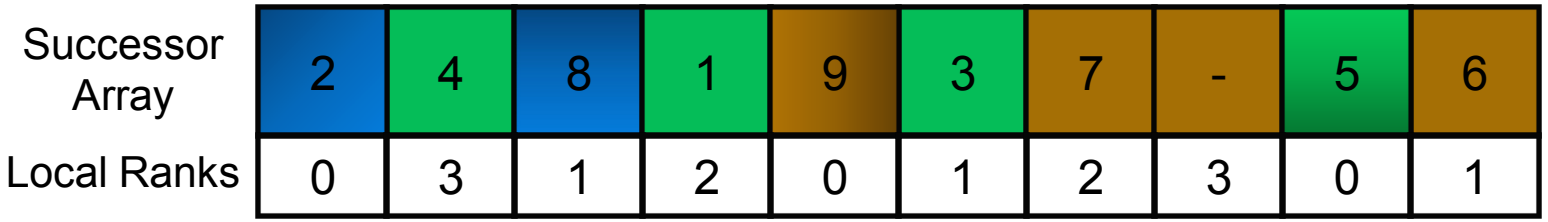


| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Local Ranks | 0 | 3 | 1 | 2 | 0 | 1 | 2 | 3 | 0 | 1 |
| Final Ranks | 0 | 5 | 1 | 4 | 6 | 3 | 2 | 0 | 1 | 0 |

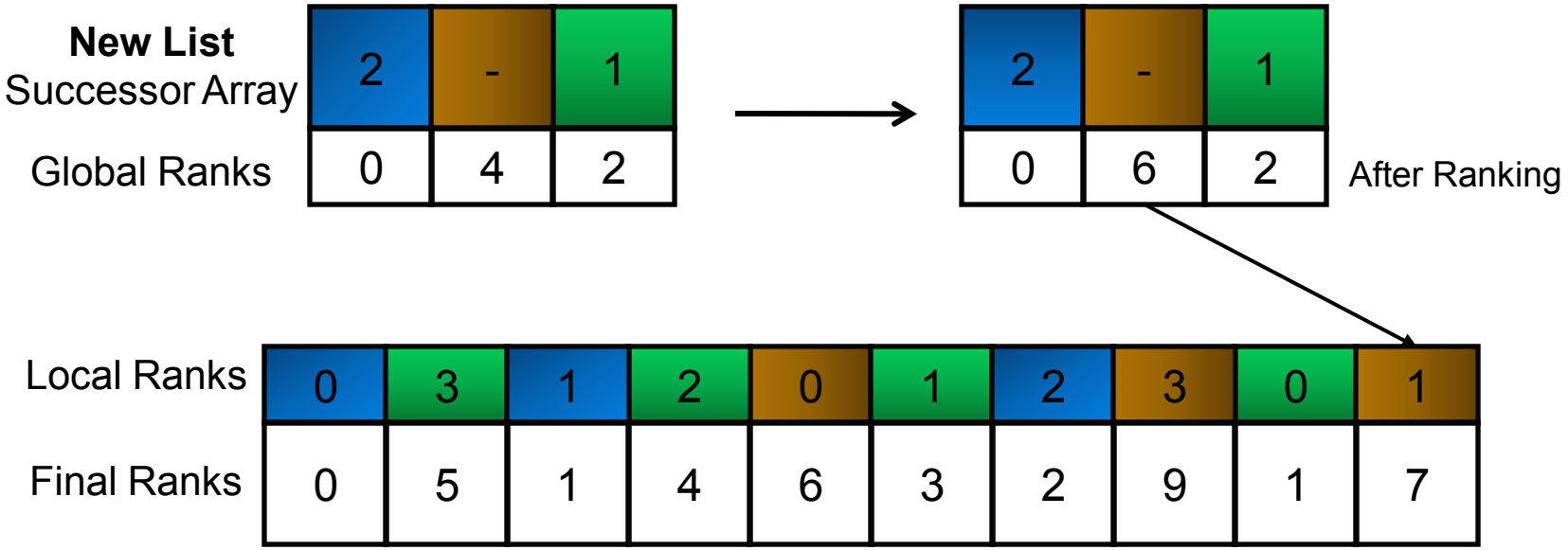


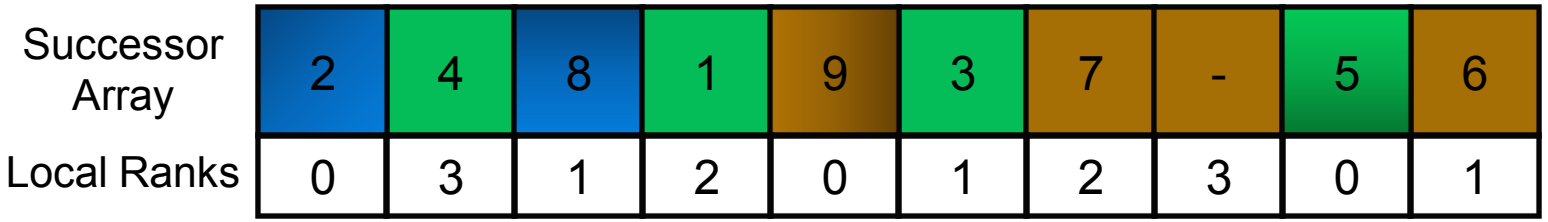

Step 6. Add the global ranks to the corresponding local ranks to get the final rank of the list.



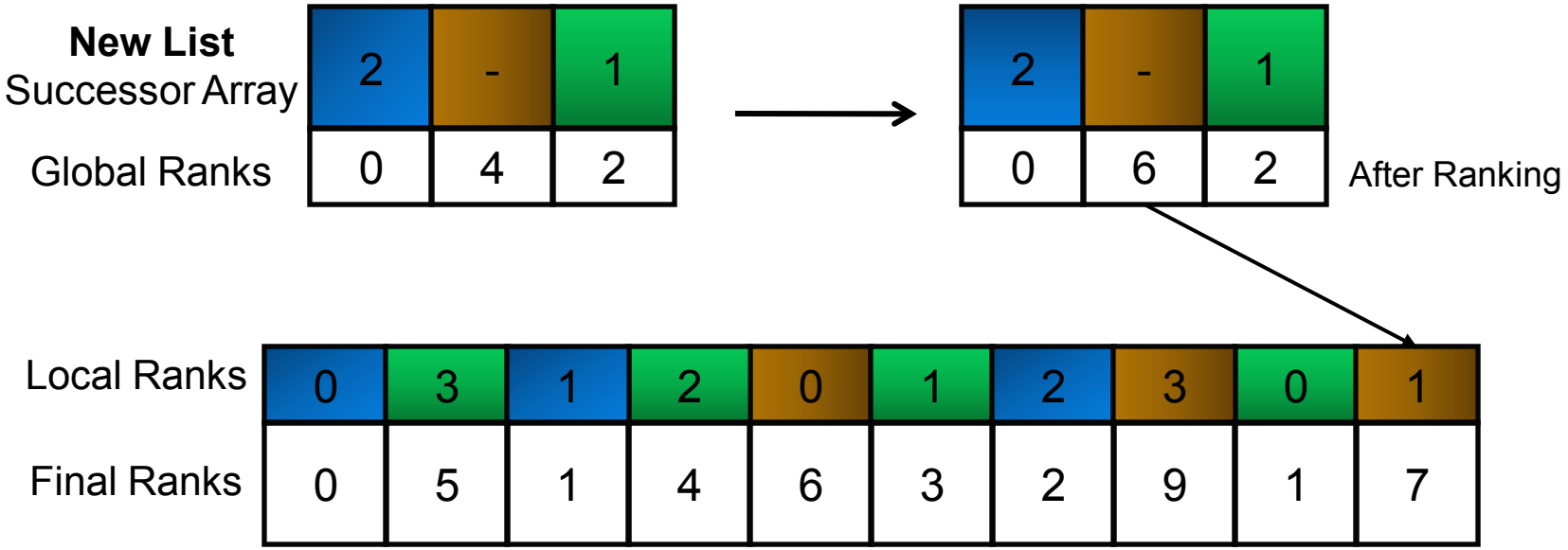



Step 6. Add the global ranks to the corresponding local ranks to get the final rank of the list.






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

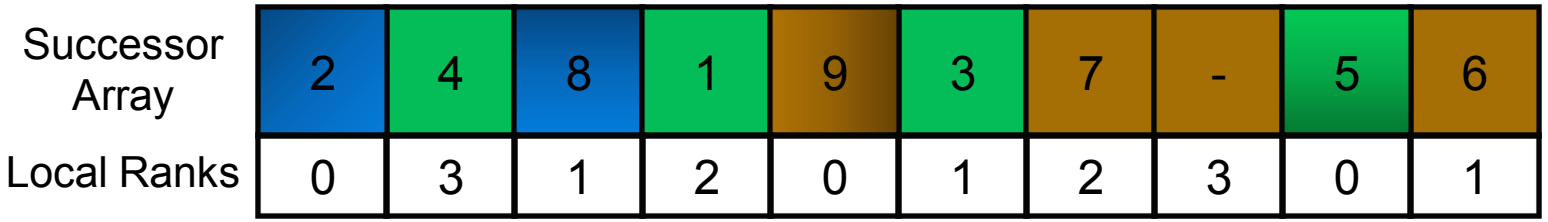


# Is this suitable for the GPU?

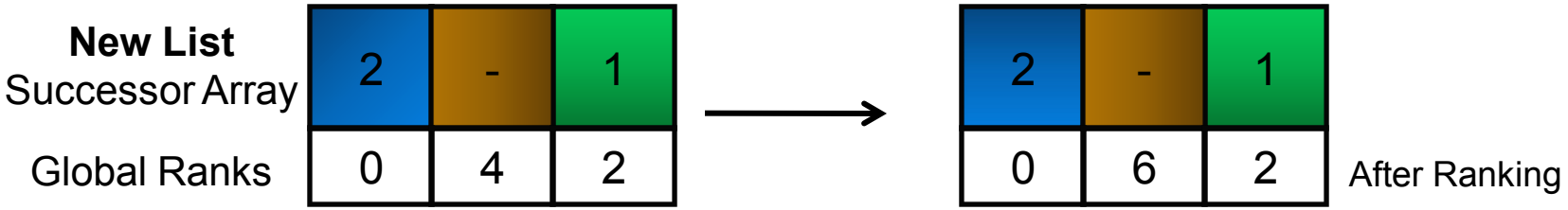
- Assume you select n/log n splitters (for large number of threads)
- The resulting list itself will be very large
- Will slow down the computation as that has to be computed sequentially
- Amdahl's Law comes into effect
  - Large portion of the time is spent in serial section of the Code

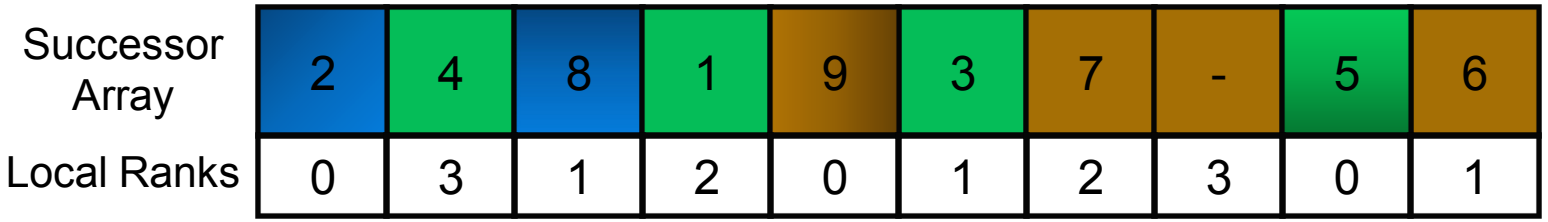
# Solution: Recrusive H-J

- 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

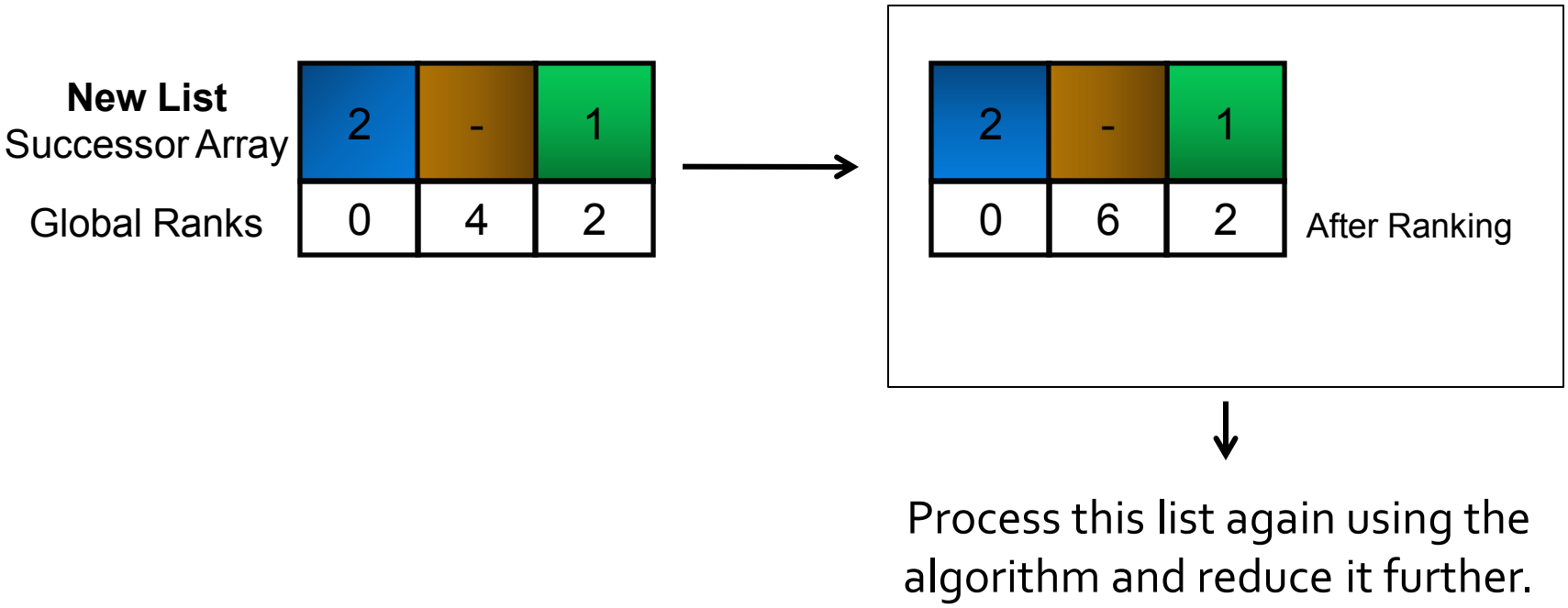



Make step 5 **recursive** to allow the GPU to continue processing the list in parallel






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.

# GPU Implementation

- 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

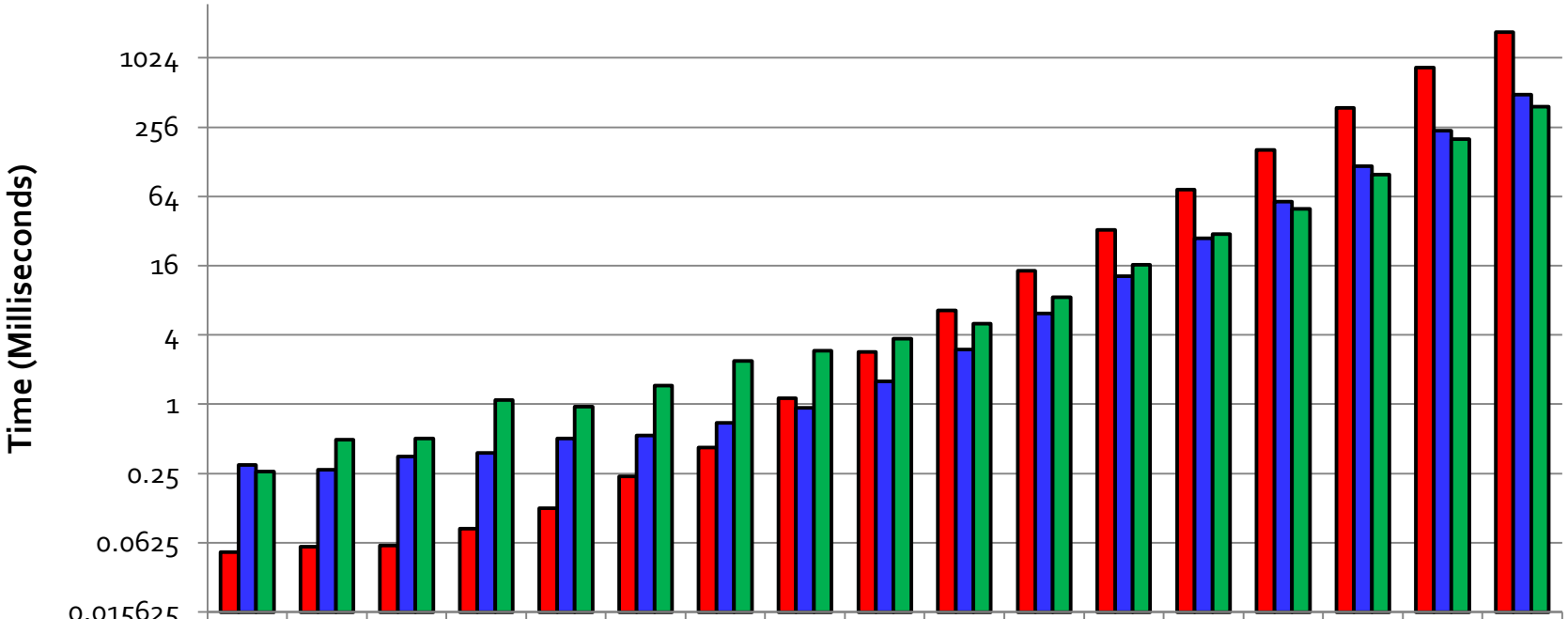
# When do we stop?

- 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 M.

# Results – Runtime w.r.t Wyllie's



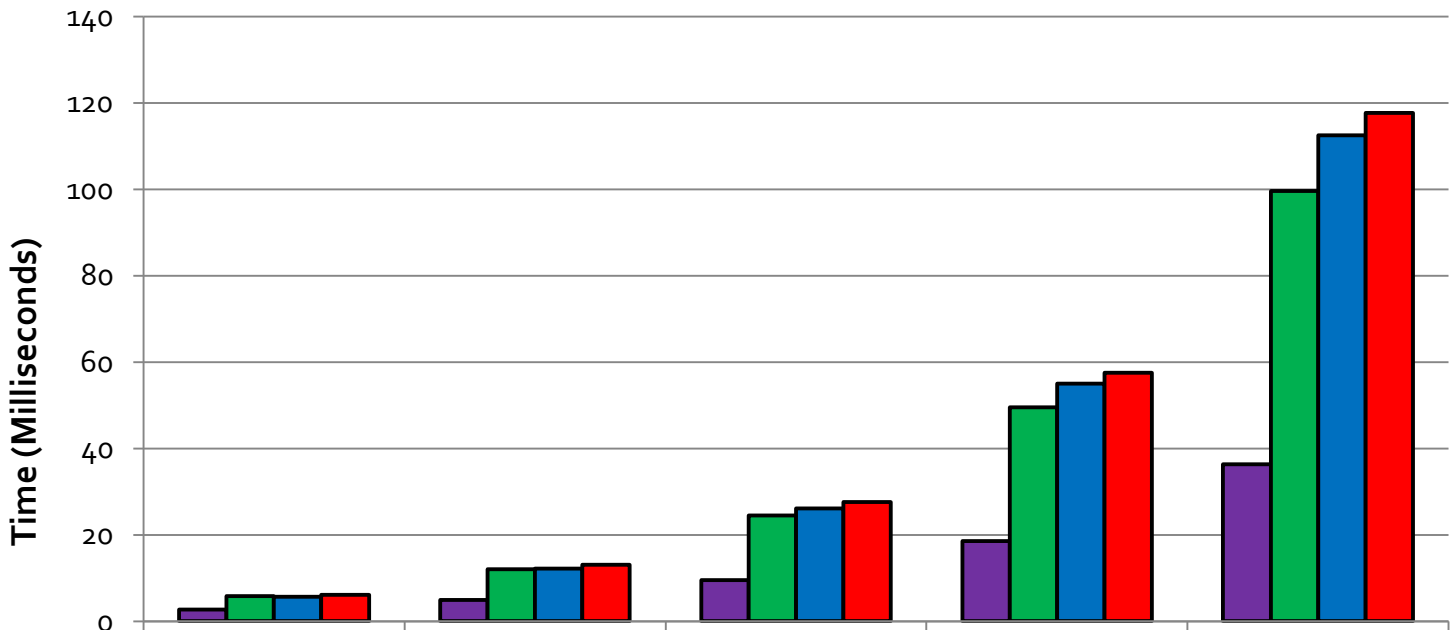

| | 1K | 2K | 4K | 8K | 16K | 32K | 64K | 128K | 256K | 512K | 1M | 2M | 4M | 8M | 16M | 32M | 64M |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Wyllie (RO) | 0.052 | 0.058 | 0.059 | 0.083 | 0.125 | 0.237 | 0.419 | 1.123 | 2.86 | 6.535 | 14.49 | 32.83 | 73.01 | 163.3 | 377.6 | 840.2 | 1718 |
| RHJ (log n) | 0.297 | 0.27 | 0.351 | 0.378 | 0.501 | 0.538 | 0.692 | 0.926 | 1.576 | 3.001 | 6.188 | 13.04 | 27.64 | 57.52 | 117.8 | 239.7 | 488.9 |
| RHJ (2log2n) | 0.259 | 0.493 | 0.501 | 1.089 | 0.954 | 1.444 | 2.369 | 2.914 | 3.731 | 4.992 | 8.557 | 16.32 | 30.09 | 49.75 | 99.08 | 200.7 | 388.2 |

List Size

# Runtime Performance Analysis

- 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

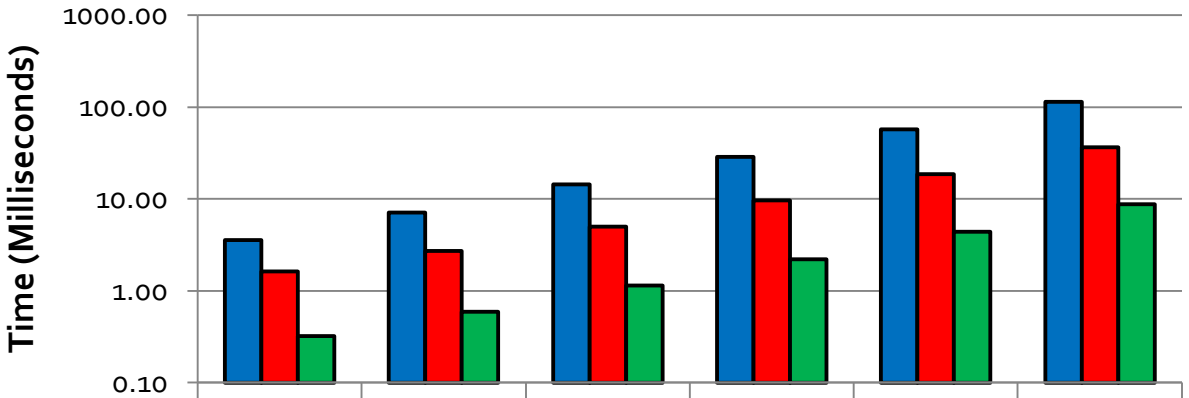



| | 1M | 2M | 4M | 8M | 16M |
|---|---|---|---|---|---|
| Ordered List | 2.719 | 4.956 | 9.56 | 18.615 | 36.432 |
| Ordered/Unbalanced | 5.897 | 12.111 | 24.523 | 49.585 | 99.716 |
| Random/Balanced | 5.649 | 12.211 | 26.164 | 54.992 | 112.59 |
| Random List | 6.188 | 13.038 | 27.638 | 57.516 | 117.771 |

**List Size**

# Results – Ordered Lists

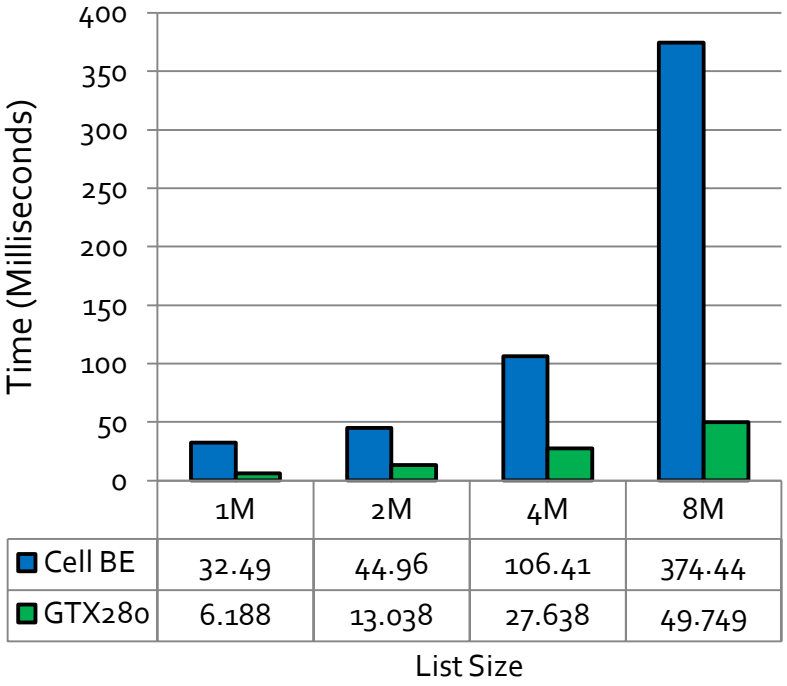
- Ordered Lists are Lists with successor of element $i = i+1$
- List Ranking on Ordered List is equivalent to scan



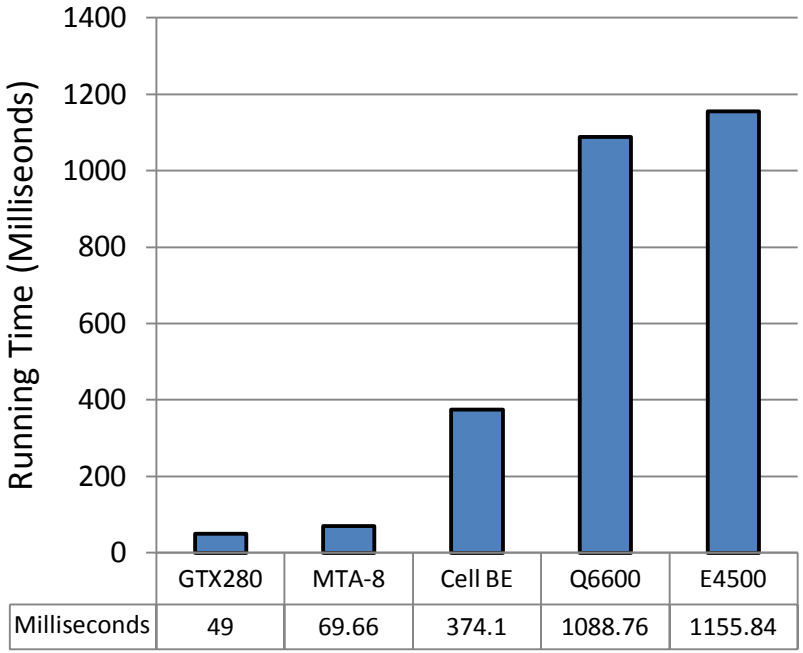

| | 512K | 1M | 2M | 4M | 8M | 16M |
|---|---|---|---|---|---|---|
| CPU | 3.55 | 7.12 | 14.27 | 28.55 | 56.62 | 113.96 |
| GPU RHJ | 1.61 | 2.72 | 4.96 | 9.56 | 18.62 | 36.43 |
| CUDPP | 0.32 | 0.59 | 1.13 | 2.21 | 4.37 | 8.68 |

List Size

# Comparison with recent Results




| | 1M | 2M | 4M | 8M |
|---|---|---|---|---|
| Cell BE | 32.49 | 44.96 | 106.41 | 374.44 |
| GTX280 | 6.188 | 13.038 | 27.638 | 49.749 |




| | GTX280 | MTA-8 | Cell BE | Q6600 | E4500 |
|---|---|---|---|---|---|
| Milliseconds | 49 | 69.66 | 374.1 | 1088.76 | 1155.84 |

**Fastest single-chip List Ranking implementation (at time of publishing)**

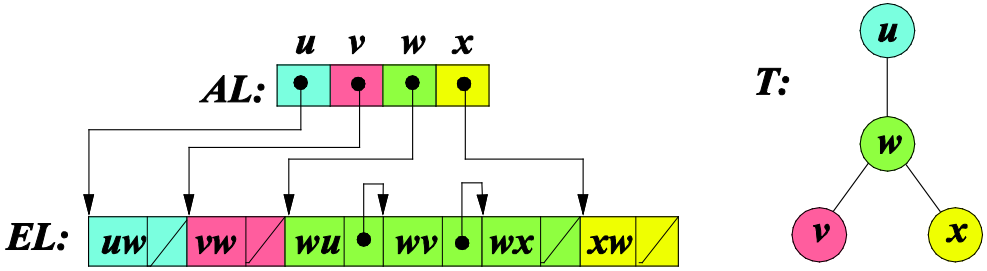
# Applications of List Ranking

# Euler Tour Technique

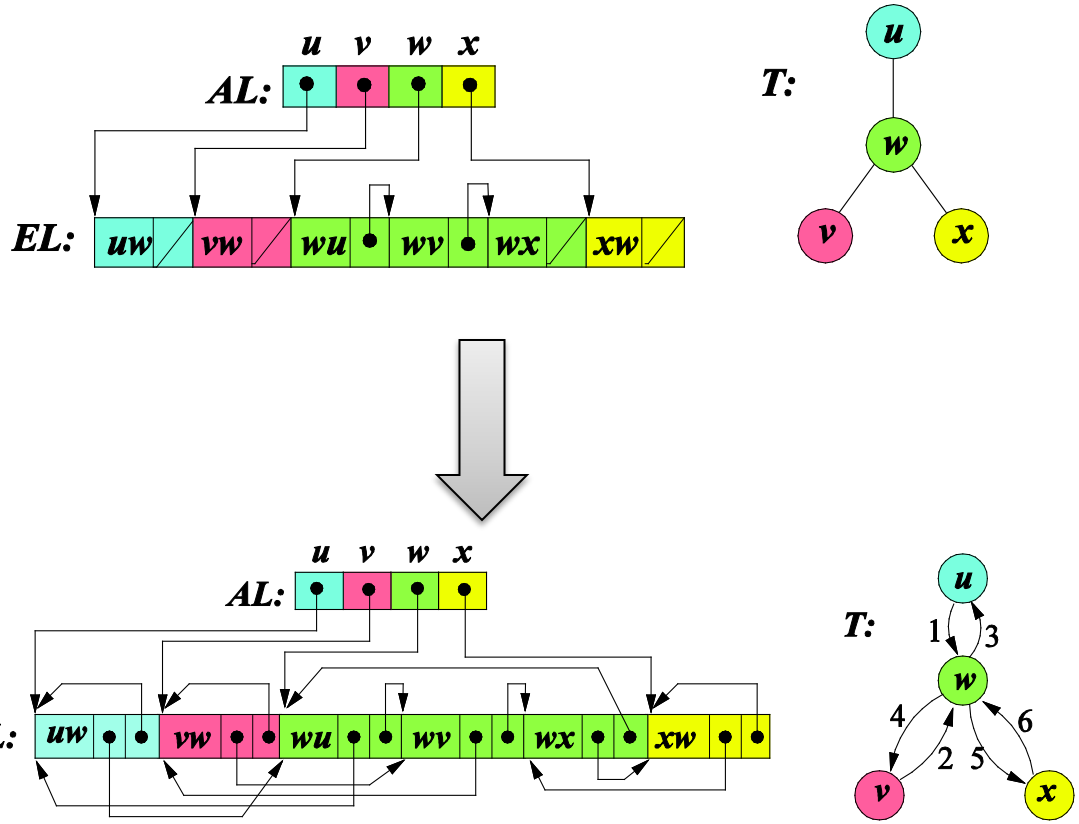
- Parallel Processing of Graphs
  - Utilizes List Ranking
- Can be used for a variety of Tree Computations
  - Parent Finding
  - Traversal
  - Vertex Levels
  - Subtree Size

# 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)



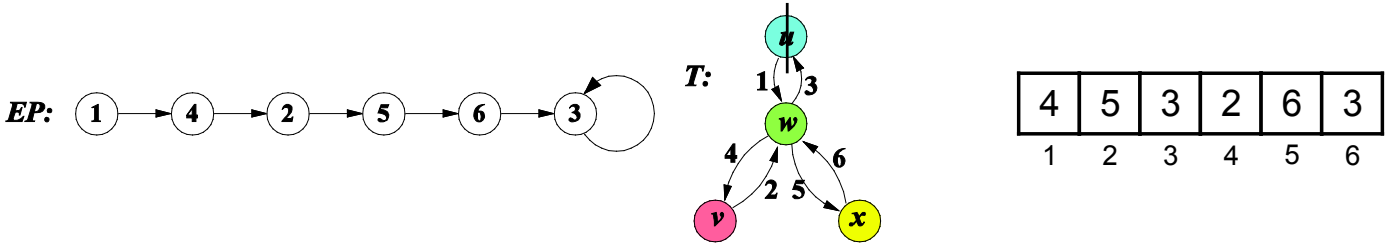
# Creating an Eulerian Graph from T




Each edge is replaced with a set of anti-parallel arcs, and an additional pointer for each edge pointing to their anti-parallel sibling

# Euler Path

- 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:

$$\mathrm{EP}[e]=(\mathrm{EL}[e].sib).next$$

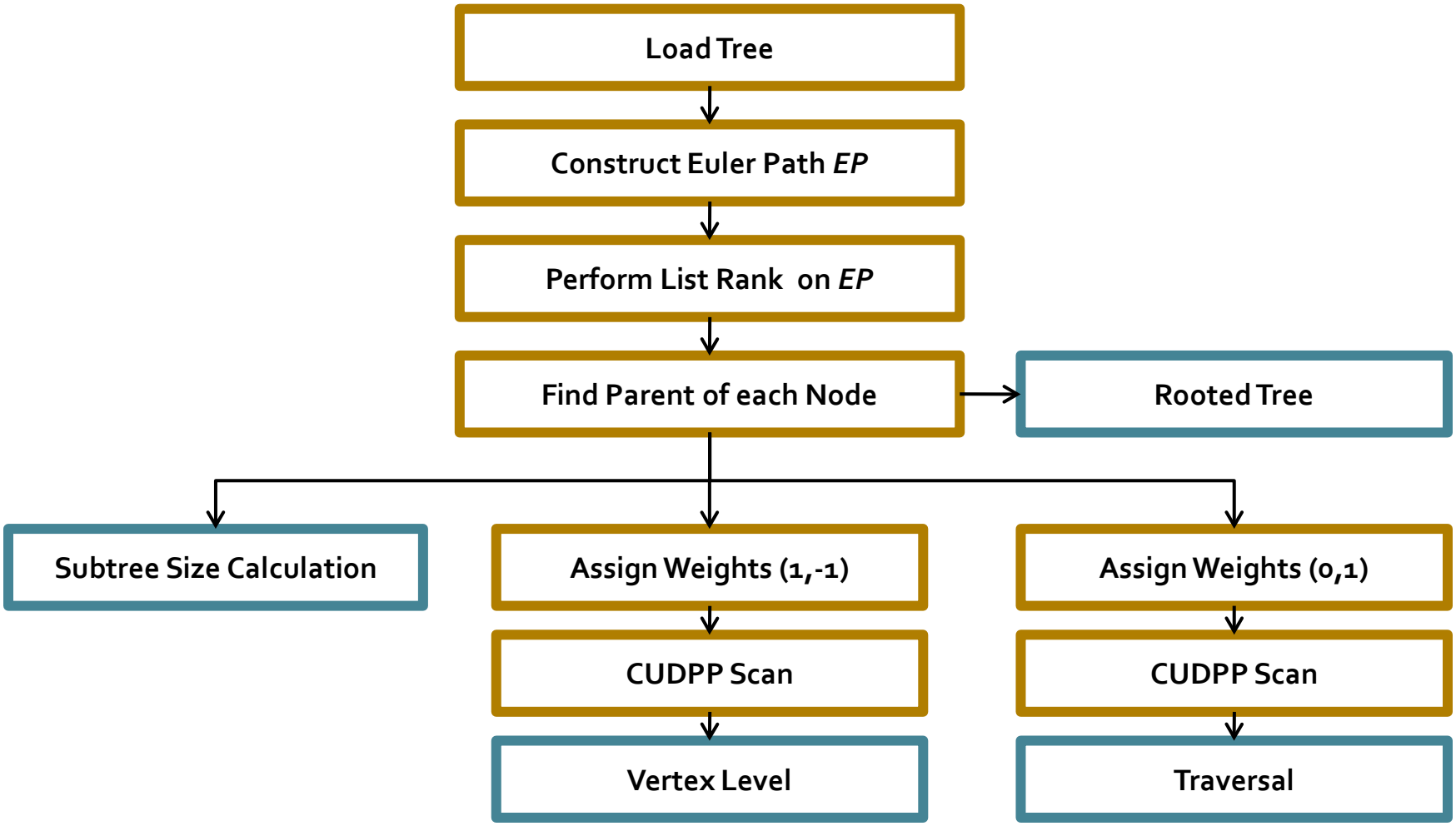
# Tree Computations

- 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 Par[y]=x
  - Subtree Size
    - Subtree[i]=(Rank[ij]-Rank[ji]+1)/2 where j= Par[i]

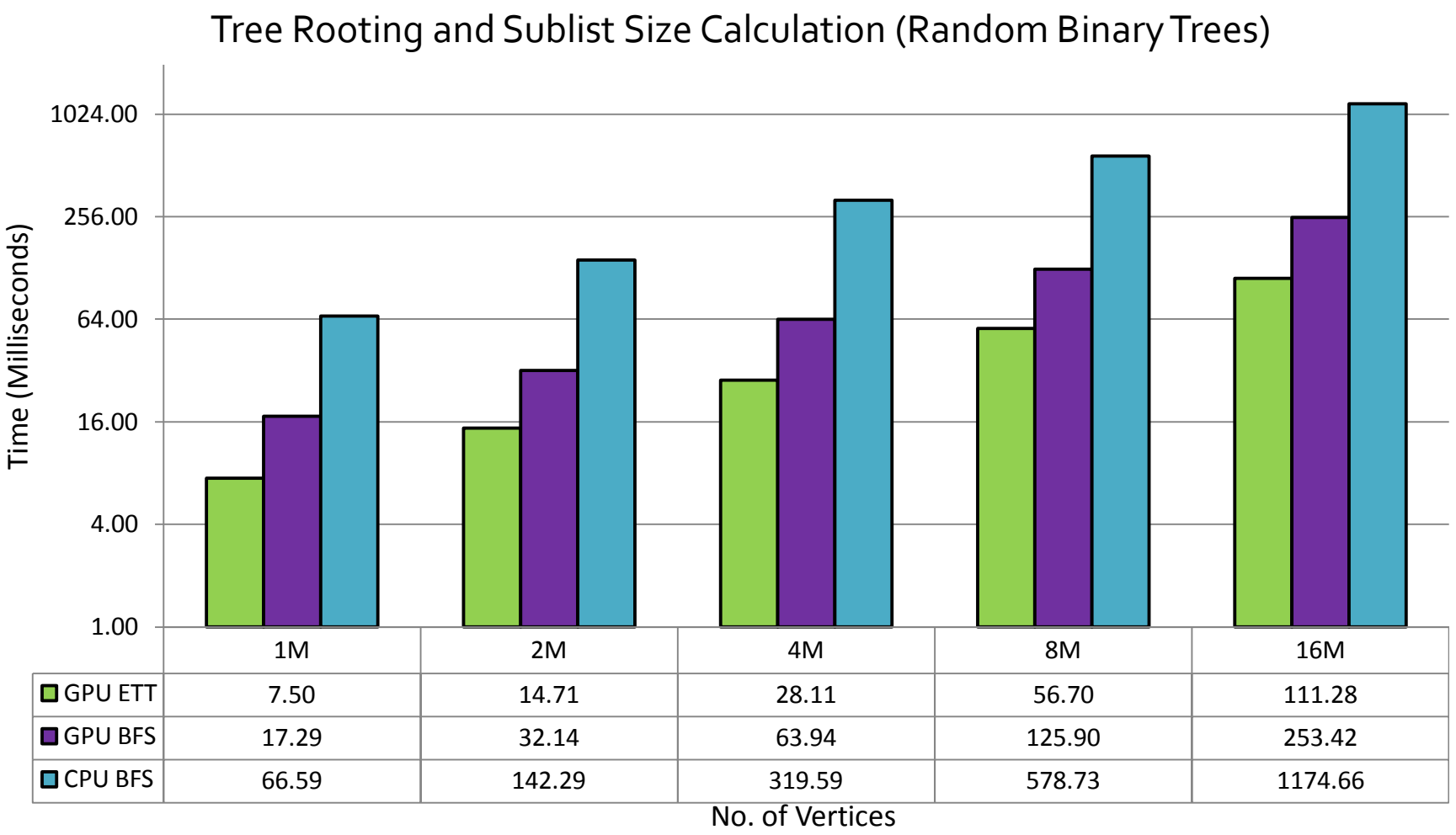
# Tree Computations

- 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

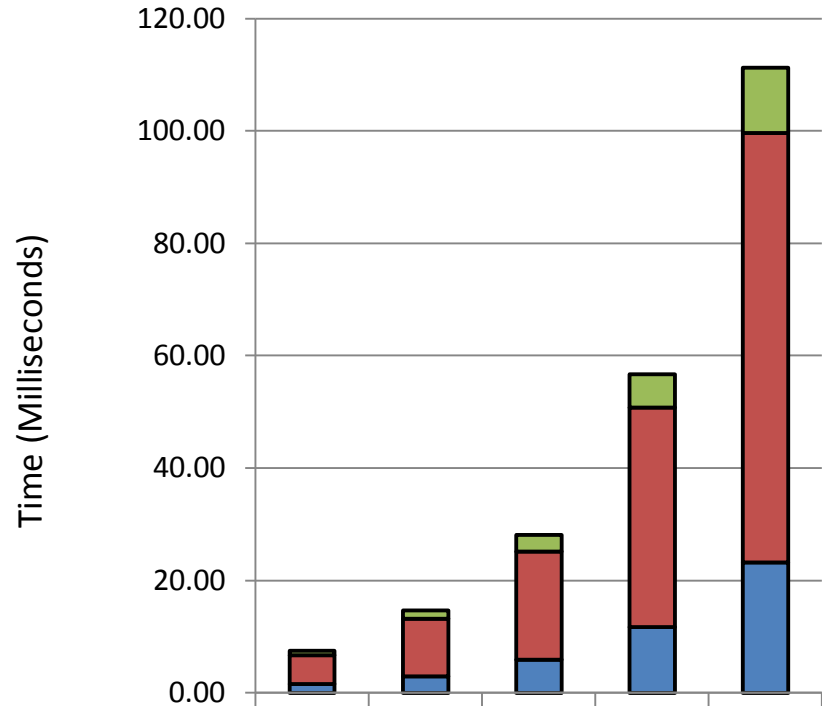

Load Tree
Construct Euler Path *EP*
Perform List Rank on *EP*
Find Parent of each Node
Rooted Tree
Subtree Size Calculation
Assign Weights (1,-1)
CUDPP Scan
Vertex Level
Assign Weights (0,1)
CUDPP Scan
Traversal

# Results for GPU




| | 1M | 2M | 4M | 8M | 16M |
|---|---|---|---|---|---|
| GPU ETT | 7.50 | 14.71 | 28.11 | 56.70 | 111.28 |
| GPU BFS | 17.29 | 32.14 | 63.94 | 125.90 | 253.42 |
| CPU BFS | 66.59 | 142.29 | 319.59 | 578.73 | 1174.66 |

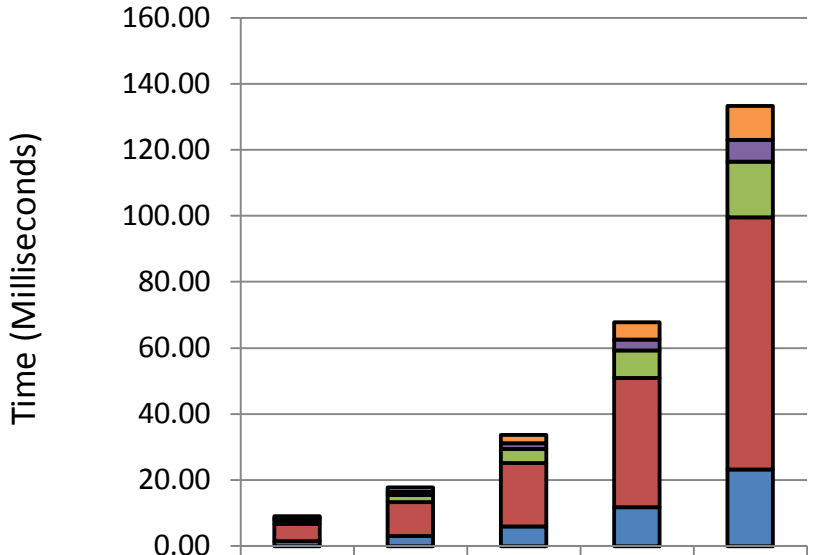
# Timing Breakup




| | 1M | 2M | 4M | 8M | 16M |
|---|---|---|---|---|---|
| FindParent | 0.77 | 1.49 | 2.93 | 5.88 | 11.70 |
| List Ranking | 5.21 | 10.26 | 19.28 | 39.17 | 76.35 |
| EP Construction | 1.52 | 2.96 | 5.90 | 11.65 | 23.23 |

No. of Vertices

Parent Finding




| | 1M | 2M | 4M | 8M | 16M |
|---|---|---|---|---|---|
| Last Kernel | 0.66 | 1.30 | 2.60 | 5.17 | 10.36 |
| CUDPP Scan | 0.55 | 0.98 | 1.72 | 3.31 | 6.50 |
| FindParent | 1.12 | 2.16 | 4.24 | 8.40 | 16.90 |
| List Ranking | 5.21 | 10.26 | 19.28 | 39.17 | 76.35 |
| EP Construction | 1.52 | 2.96 | 5.90 | 11.65 | 23.23 |

No. of Vertices

Preorder Traversal

# Performance Prediction for Irregular Applications on the GPU

# GPU Performance Model

- 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

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

$$T(K) = \frac{C(K)}{R} \text{ seconds}$$

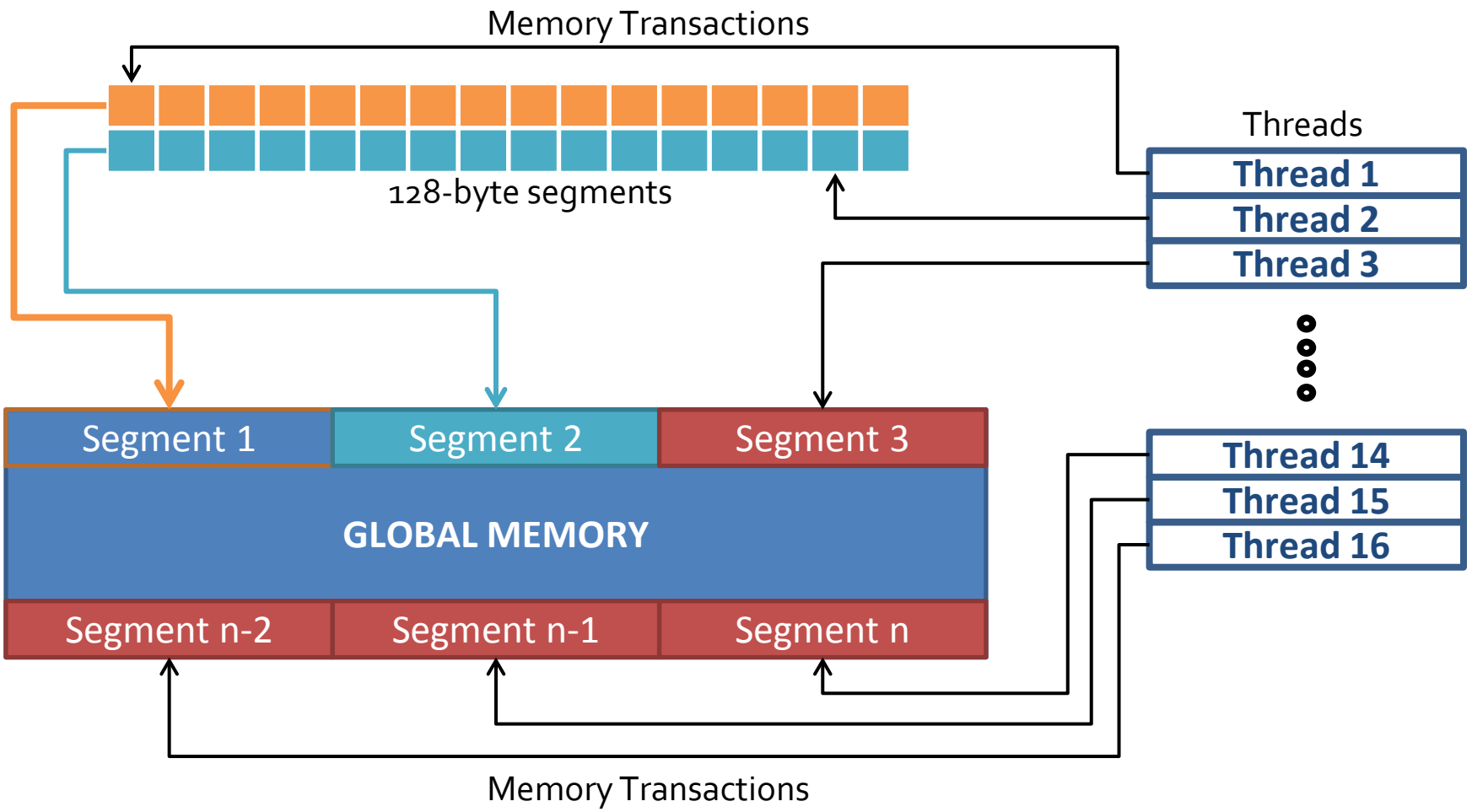
# Estimating Cycles

- 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

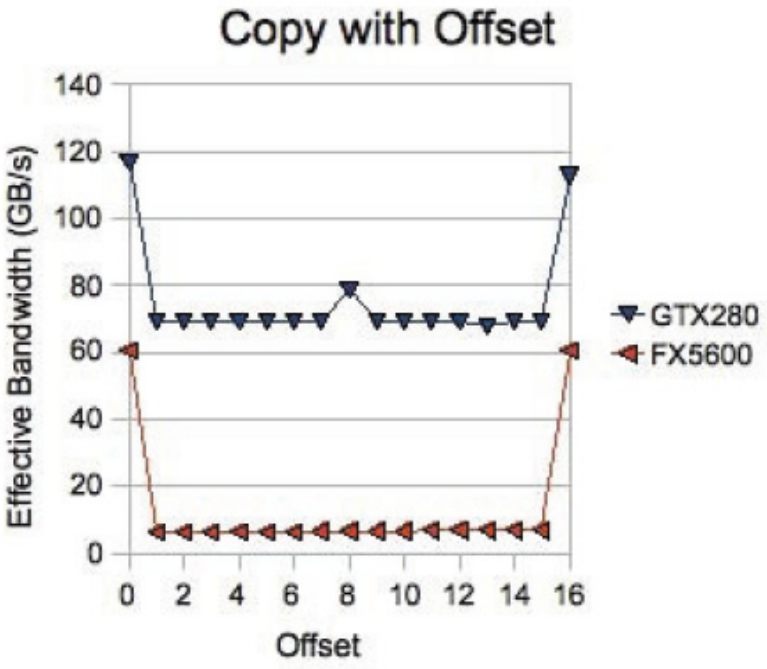
# Global Memory Coalescing

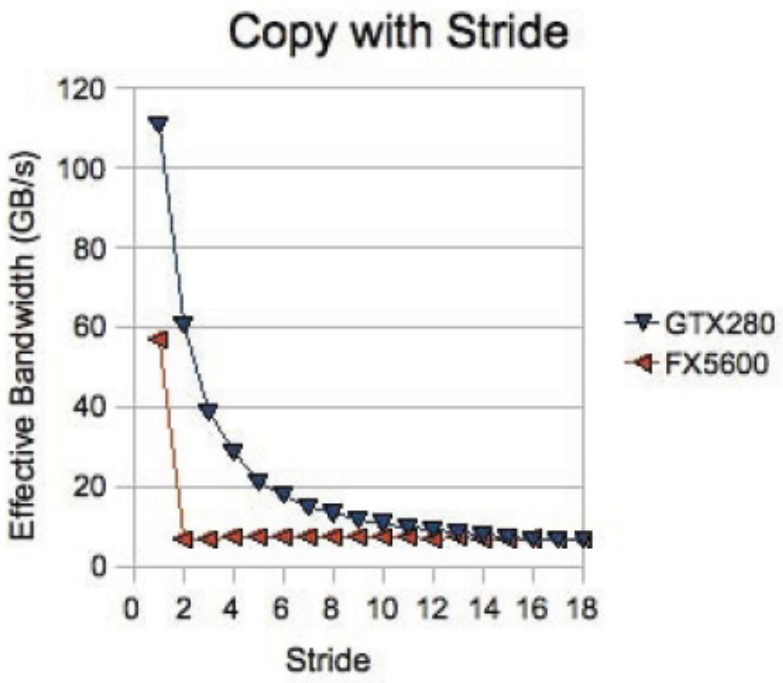
- Global Memory can fetch up to 128 bytes in a single transaction
  - If all threads of a half-warp access elements in 128 byte segment at the same time – then bandwidth improves
- Unlikely to happen with irregular memory applications
- What's the effect of un-coalesced memory accesses?

# Global Memory Coalescing



# Effect on Bandwidth



# 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

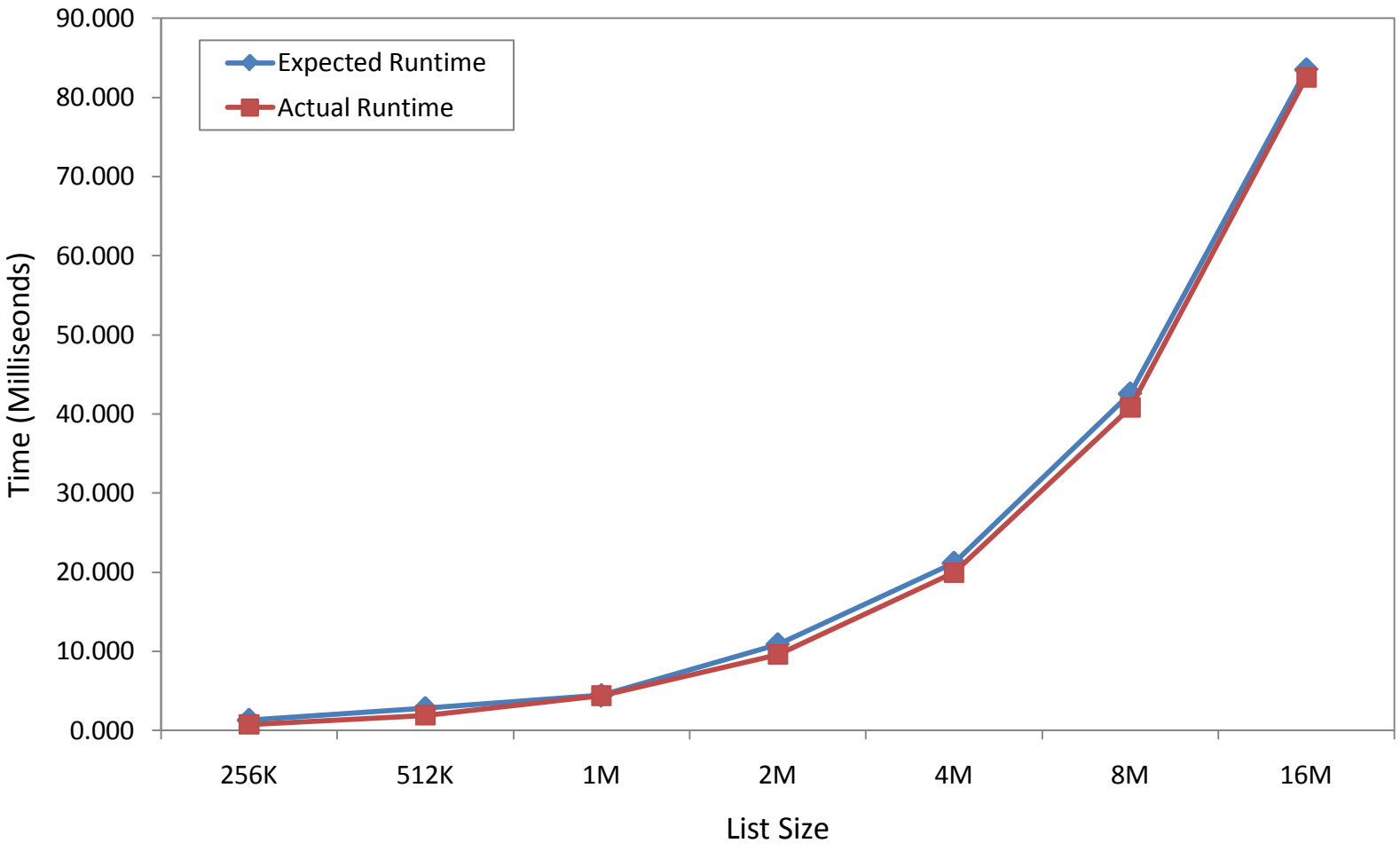
# Modeling List Ranking

- Calculations

$$= \frac{N}{512 \times 30} \times 16 \times \frac{32}{8 \times 4} \times 4 \log N \times 3 \times 500 \times \frac{1}{1.3 \times 10^9} \; sec$$

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

# Accuracy of the Model



# 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

# Chronology of Work

- 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!