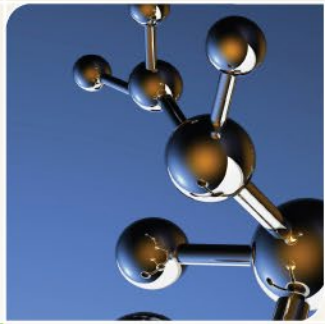
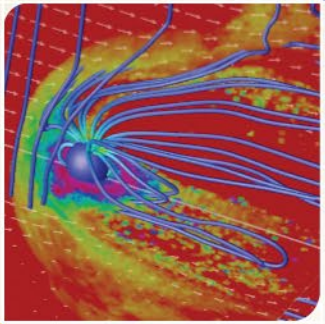
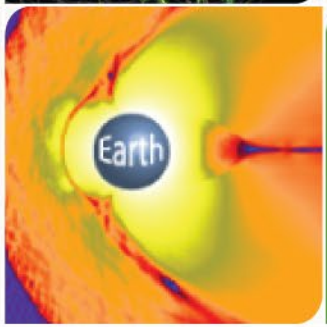
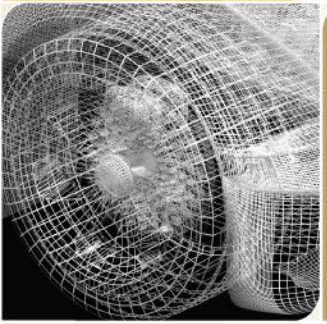







Improving Scheduling for Irregular Applications with Logarithmic Radix Binning

James Fox, Alok Tripathy, Oded Green

Georgia Tech College of Computing
Computational Science and Engineering

Additional Collaborators

James Fox

Alok Tripathy



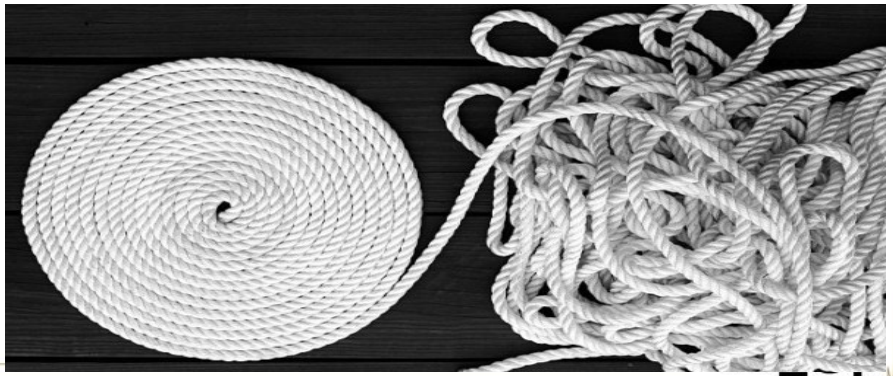
Main contributions

- Logarithmic Radix Binning
  - A new load balancing technique applicable for irregular problems
  - Same computational complexity as Prefix Summation – but better load-balancing
  - Architecture independent
  - Very simple
- One step closer to making the irregular into regular

Regular Vs. Irregular

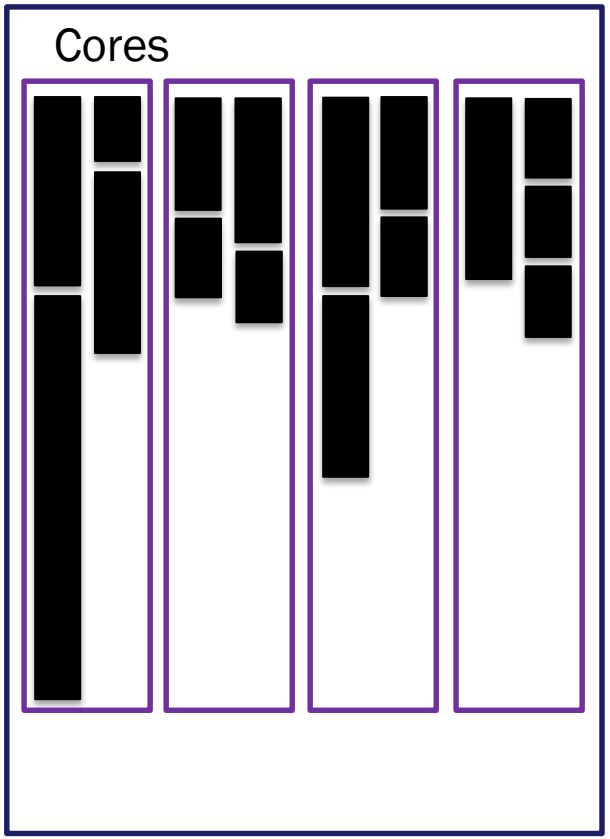
- Computational sciences
- Sequence of events is predefined
- Typically, can be analyzed offline
- Applications: linear algebra, dense matrix multiplication, image processing…

- Data analytics
- Highly data dependent
- Execution flow cannot be analyzed offline
- Applications: merging and sorting, graph algorithms, classification, sparse matrix multiplication



Known scalability issues

- Load-balancing is challenging
  - Some threads might receive heavy edges only
  - Gets tougher for large core counts
  - SIMD\SIMT programming models
    - Need to load-balance at the lane granularity
- Prefix summation can help get good partitions per core
  - Doesn't resolve SIMD/SIMT programmability



LRB resolves these problems

Logarithm Radix Binning

- Four for loops
  - Simple
  - Scalable

Algorithm 1: LRB Pseudo Code

for $i = 0 : 1 : B$ **do**
  $Bins[B] \leftarrow 0$
// Loop 2
for $i = 0 : 1 : N - 1$ **do**
  $b_i \leftarrow (\lceil log_2(w(T[i])) \rceil)$
  $atomicAdd(Bins[b_i], 1)$
// Loop 3
$prefixB[0] \leftarrow 0$
for $i = 1 : 1 : B$ **do**
  $prefixB[i] \leftarrow prefixB[i-1] + Bins[i-1]$
// Loop 4
for $i = 0 : 1 : N - 1$ **do**
  $b_i \leftarrow (\lceil log_2(w(T[i])) \rceil)$
  $pos_i = atomicAdd(prefixB[b_i], 1)$
  $T_{reordered}[pos_i] = T[i]$

Logarithm Radix Binning

- First loop – initialize bin counters
  - $O(B)$
  - $B \in \{32,64,128\}$
- Simple $O(B)$
  - Inexpensive

Algorithm 1: LRB Pseudo Code

for $i = 0 : 1 : B$ **do**
  $Bins[B] \leftarrow 0$

Logarithm Radix Binning

- Second loop – count number of instances
  - Compute expected work per task
  - Get ***log*** of the work
  - Increment counter
    - This is value between $0..B$
- Scalable $O(N)$ work

Algorithm 1: LRB Pseudo Code

```
for i = 0 : 1 : B do
    Bins[B] ← 0
// Loop 2
for i = 0 : 1 : N − 1 do
    b_i ← (⌈log_2(w(T[i]))⌉)
    atomicAdd(Bins[b_i], 1)
```

Logarithm Radix Binning

- Third loop – create bins based on counters
  - Compute expected work per task
  - Get ***log*** of the work
  - Increment counter
    - This is value between $0..B$
- Simple $O(B)$

Algorithm 1: LRB Pseudo Code

```
for i = 0 : 1 : B do
    Bins[B] ← 0
// Loop 2
for i = 0 : 1 : N − 1 do
    b_i ← (⌈log_2(w(T[i]))⌉)
    atomicAdd(Bins[b_i], 1)
// Loop 3
prefixB[0] ← 0
for i = 1 : 1 : B do
    prefixB[i] ← prefixB[i − 1] + Bins[i − 1]
```

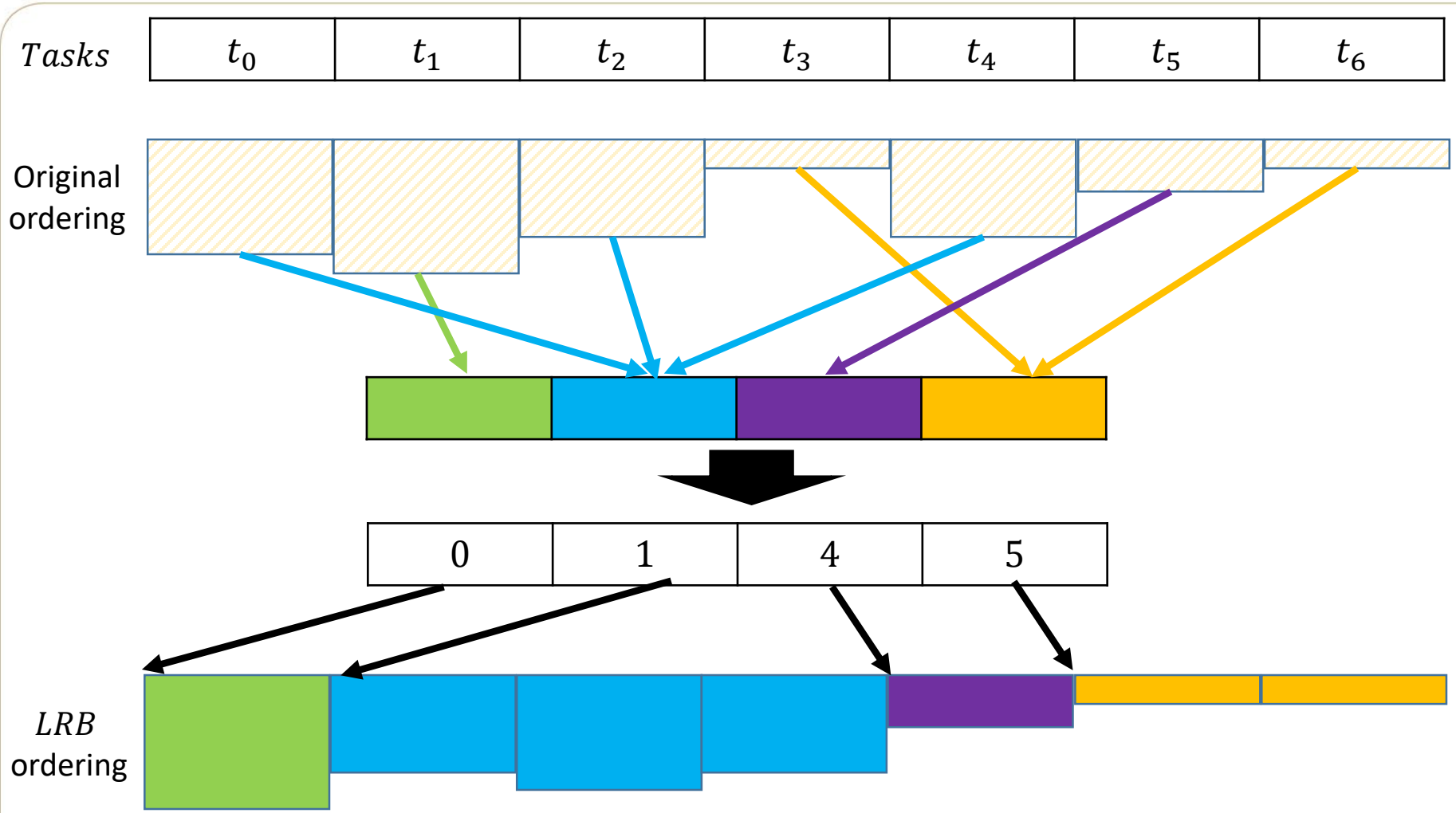
Logarithm Radix Binning

- Fourth loop – reorganize tasks in bins
  - Place the tasks into bins with a similar amount of work
- Scalable $O(N)$ work

Algorithm 1: LRB Pseudo Code

for $i = 0 : 1 : B$ **do**
  $Bins[B] \leftarrow 0$
// Loop 2
for $i = 0 : 1 : N - 1$ **do**
  $b_i \leftarrow (\lceil log_2(w(T[i])) \rceil)$
  $atomicAdd(Bins[b_i], 1)$
// Loop 3
$prefixB[0] \leftarrow 0$
for $i = 1 : 1 : B$ **do**
  $prefixB[i] \leftarrow prefixB[i-1] + Bins[i-1]$
// Loop 4
for $i = 0 : 1 : N - 1$ **do**
  $b_i \leftarrow (\lceil log_2(w(T[i])) \rceil)$
  $pos_i = atomicAdd(prefixB[b_i], 1)$
  $T_{reordered}[pos_i] = T[i]$

Logarithm Radix Binning – High Level



- New LRB ordering is not perfect, but it is good enough.

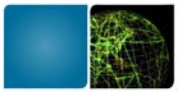
Logarithm Radix Binning - Features

- Given two tasks, u and v, in bin i we know the following:
  - $2^i \leq w(u), w(v) < 2^{i+1}$
  - This means that $w(v)$ is never more than twice as big or small as $w(u)$
    - Or vice-versa

Complexity Analysis

- Work: $O(N+B) = O(N)$
- Time: $O\left(\frac{N}{P} + B\log(P)\right) = O\left(\frac{N}{P} + \log P\right)$
- Storage: $O(N+B) = O(N)$

- Parallel Prefix Summation:
- Work: $O(N)$
- Time: $O\left(\frac{N}{P} + \log(P)\right)$
  - Requires P synchronizations!
- Storage: $O(N+B) = O(N)$


Systems

1) NVIDIA GV-100 GPU

- 80 SMs, 5120 SPs (CUDA cores)
- 6MB LLC
- 16GB MCDRAM
- PCI-E

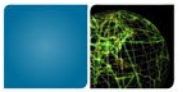
2) Intel KNL processor

- 64 threads, 256 threads, 4-way SMT
- 45MB LLC
- 16GB of MCDRAM – roughly 400 GB/s BW
- 96GB of DDR4 - roughly 100 GB/s BW



Experiments

1. Time comparison with parallel prefix summation
   - Even though PPS doesn't solve load-balancing…
2. Accelerating Segmented Sort on the GPU
3. Accelerating PageRank
   - Will not cover because of time limitations
   - Cool results as we ran one thread per vector-lane… over 4k threads…

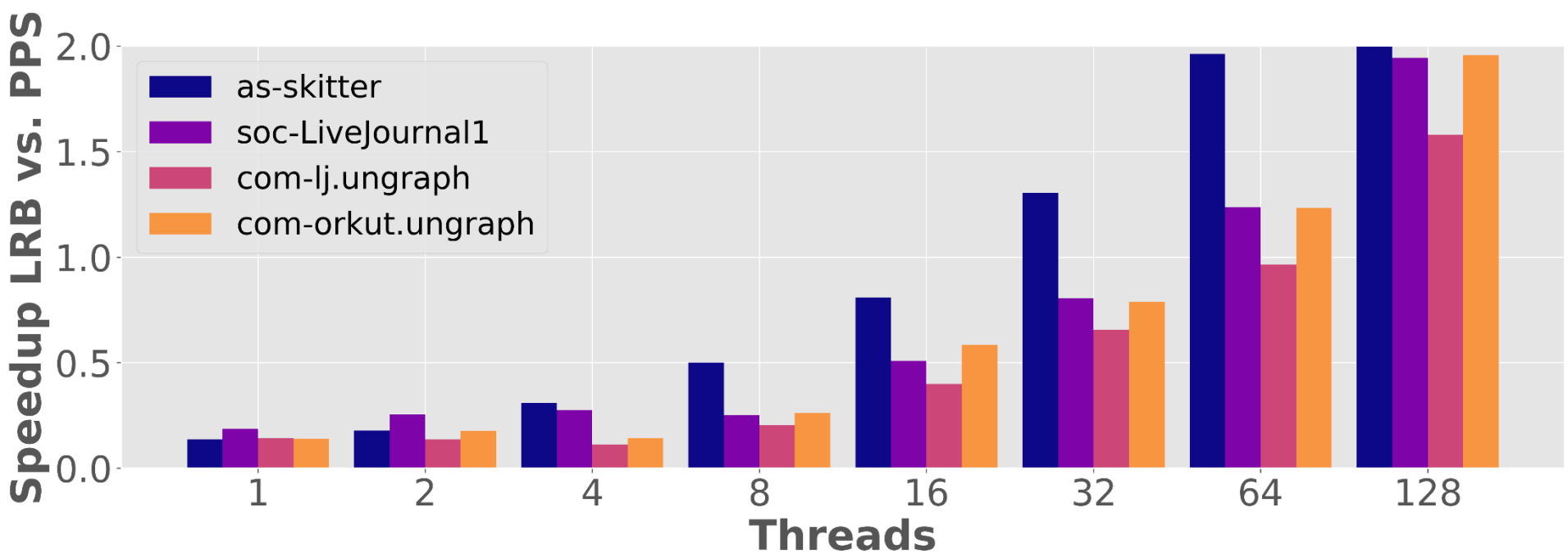

Experiment 1

- Load balance an array of length N
- Prefix summation use a binary search to find N/P partition points – we do not time this
  - Partitions are near equal in size
  - Does not ensure good SIMD\SIMD placement
- We only focus on the execution times

- Inputs are real world graph and task lengths are the size of the edge lists

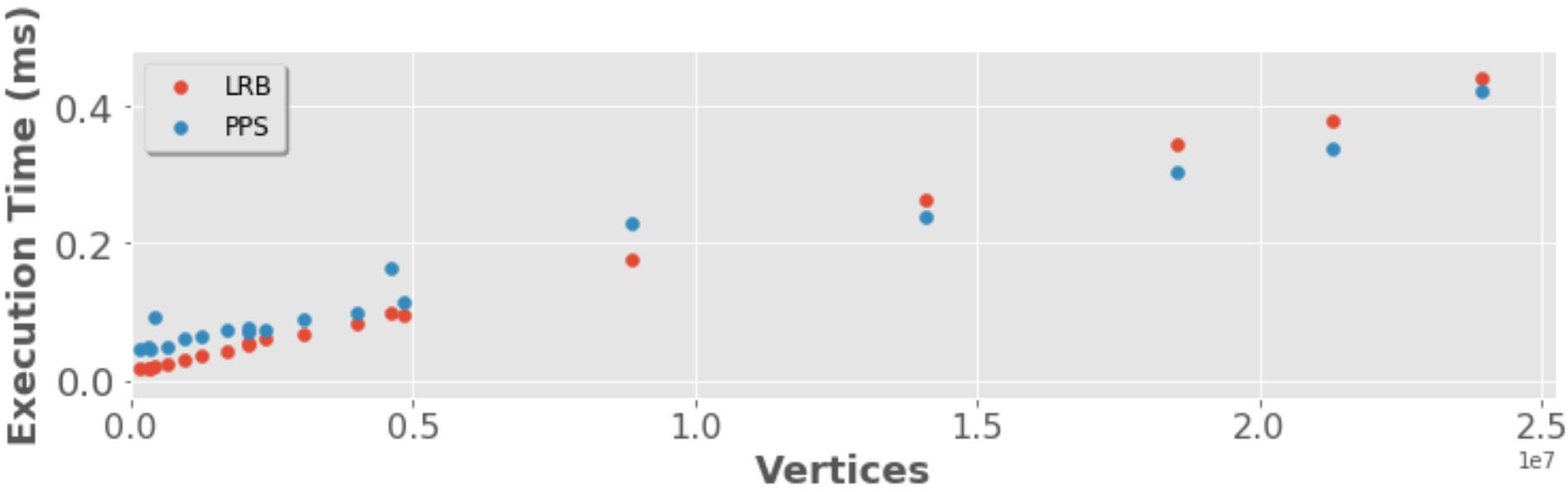
CPU Comparison - Speedup

- Small thread counts – prefix sum is faster
- Large thread counts – LRB is faster
  - This is the more interesting problem



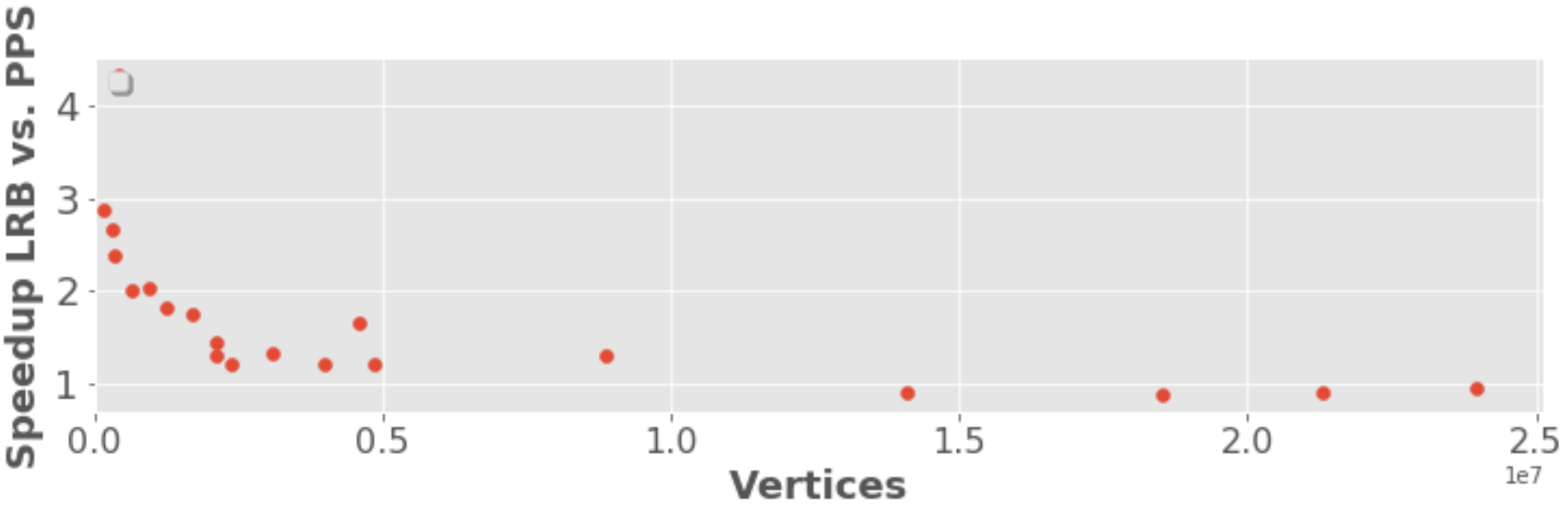
GPU Comparison – Execution Times

- In comparison with CUB's prefix implementation
  - Very optimized



GPU Comparison - Speedup

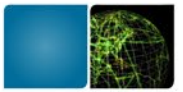
- For smaller inputs, can be up-to 3X faster
- For larger inputs, roughly 5% slower



Experiment 2: Accelerating Segmented Sort



- Rather than sorting a single array of length M, we need to sort N arrays of length M
  - One example is sorting N rows in a CSR
  - Its expected that Segmented Sort for CSR will be faster the sorting COO
    - Locals sorts vs. a global sort

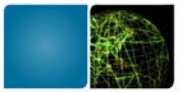

Sorting Comparisons

- CUB – optimized framework for basic primitive
  - Radix based
  - Two different test cases: COO and segmented sorts
- ModernGPU - optimized framework for sorting and DB operations
  - Merge-Sort based
- BB-Sort
  - Hou *et al., "Fast Segmented Sort on the GPU", ICS'17*
  - *Highly optimized kernels*
  - Several hundred to thousands of lines of code
  - Has some load-balancing but not as fine grain as LRB

Segmented Sort – soc-LiveJournal1

- Rows – 4,847,571
- NNZ – 68,993,773
- Average adjacency size 14

	Time(ms)
ModernGPU-SegSort	13.128
CUB-SortPair	31.47
CUB-SegSort	1923
BB-Sort	11

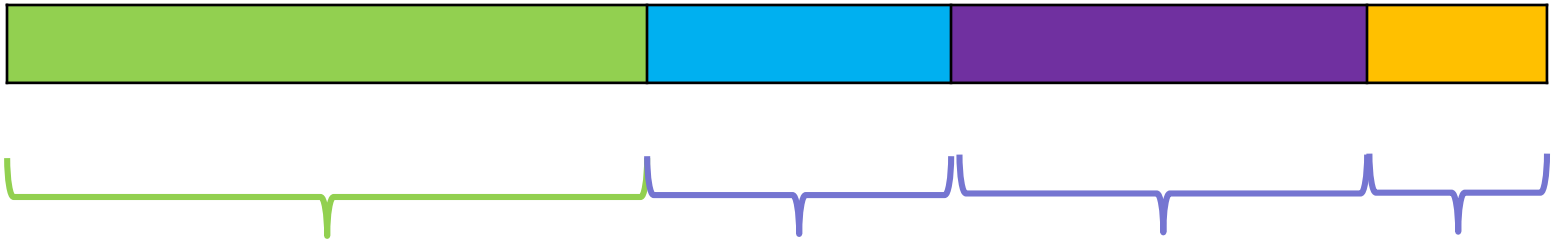

Segmented Sort – soc-LiveJournal1

- CUB's segmented sort works really well if the segments are fairly large

	Time(ms)	
ModernGPU-SegSort	13.128	1.04X
CUB-SortPair	31.47	2.40X
CUB-SegSort	1923	177.3X
BB-Sort	11	0.84X
LRB-SegSort	13.06	

Our Segmented Sort

- We did not implement any merge or sort kernels!
- Instead we used the existing sorting kernels in CUB:
  - Large adjacency arrays sorted using a device wide SegmentSort
  - Small adjacency arrays sorted in the small L1 caches use thread-block sort.
    - One for each bin size



Device segmented sort – for rows with more than 4k entries

Thread block granularity. One kernel for each bing



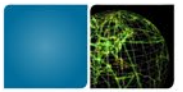
Our Segmented Sort

- This would not be possible without LRB
- Thread-block sorts are templated functions that require shared-memory size, number of threads, number of elements.
- LRB accounts for less than 3% of execution time
- We would probably benefit from the highly optimized kernels in BB.



Additional Experiments

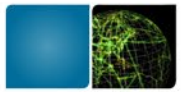
- For ModernGPU:
  - Reordered the segments using LRB
  - Used MGPU's SegmentSort
  - Twice as fast
    - Unfortunately, the segments are not in the original order. Artifact of the API.

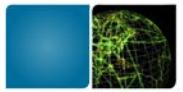

LRB Summary

- Showed a new load-balancing mechanism for irregular problems.
- LRB has a time complexity and execution similar to PPS
  - LRB has better task partitioning
- Works well for a wide range of applications: segmented sorting, page-rank, triangle counting, BFS and more.

- LRB makes irregular execution one step closer to regular execution!

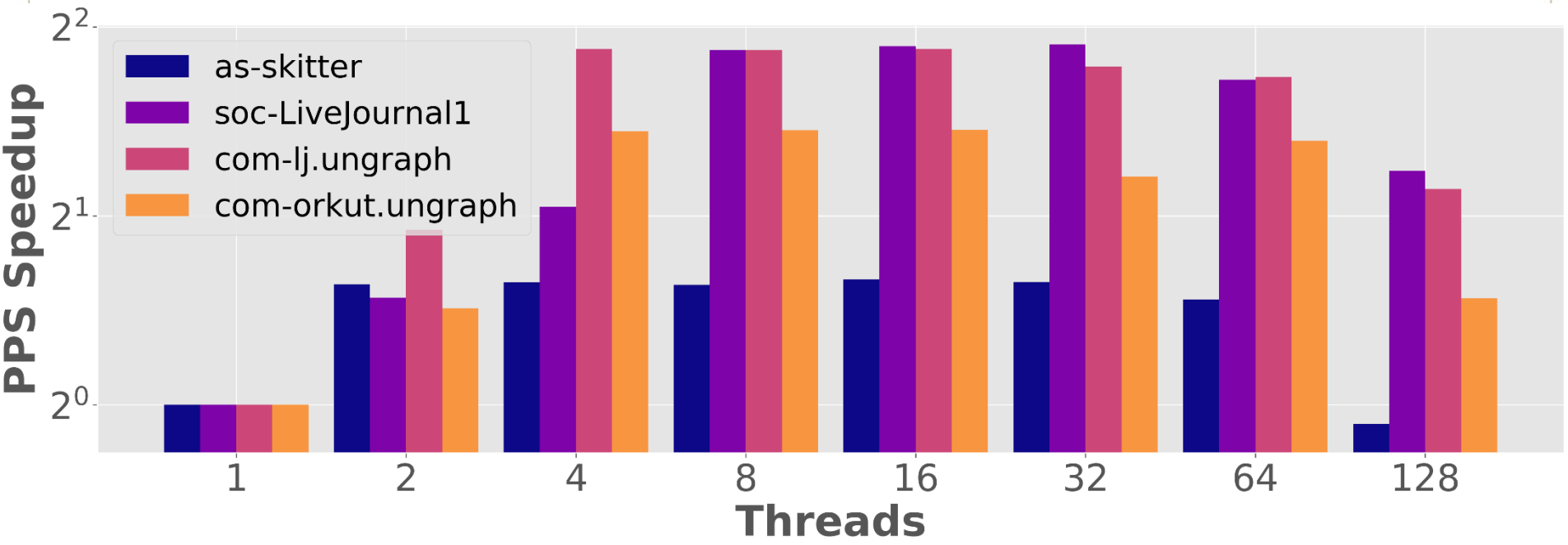

Thank you


Backup Slides

CPU Scalability – Parallel Prefix Sum

- Does not scale very well
- OpenMP sync is very costly



CPU Scalability – LRB

- Scales with the number of threads
- Could still do better