Performance Evaluation of Lock-free Data Structures on GPUs

http://www.cse.iitk.ac.in/~mainakc/lockfree.html

Prabhakar Misra and Mainak Chaudhuri

Indian Institute of Technology, Kanpur

Sketch

- Talk in one slide
- Result highlights
- Related work
- Lock-free data structures
- CUDA implementation
- Evaluation methodology
- Empirical results
- Summary

Sketch

- ➢ Talk in one slide
- ➢ Result highlights
- Related work
- Lock-free data structures
- CUDA implementation
- Evaluation methodology
- Empirical results
- Summary

Talk in One Slide

- Locks are expensive in GPUs
  - Thousands of threads cause high contention
- Lock-free data structures offer a possible way to implement irregular computations on GPUs
  - Support for dynamically changing pointer-linked data structures is important in many applications
- Large body of existing research on lock-free data structures for traditional multiprocessors
- This is the first detailed study to explore lock-free linear lists, hash tables, skip lists, and priority queues on CUDA-enabled GPUs

Result highlights

- Significant speedup on Tesla C2070 (Fermi GF100) over 24-core server execution
- Maximum speedup
  - 7.4x for linear lists
  - 11.3x for hash tables
  - 30.7x for skip list
  - 30.8x for priority queue
- Lock-free hash table shows best scalability for a wide range of operation mixes and key ranges
  - Throughput ranges from 20.8 MOPS to 98.9 MOPS

Sketch

- Talk in one slide
- Result highlights
- ➢ Related work
- Lock-free data structures
- CUDA implementation
- Evaluation methodology
- Empirical results
- Summary

Related work

- Our lock-free linear list implementation follows a variation of the Harris-Michael construction
- Our hash table implementation leverages the linear list implementation
- Our lock-free skip list construction is due to Herlihy, Lev, and Shavit
- We follow the construction due to Lotan and Shavit for our lock-free priority queue implementation
- More related works are discussed in the paper

Sketch

- Talk in one slide
- Result highlights
- Related work
- ➢ Lock-free data structures
- CUDA implementation
- Evaluation methodology
- Empirical results
- Summary

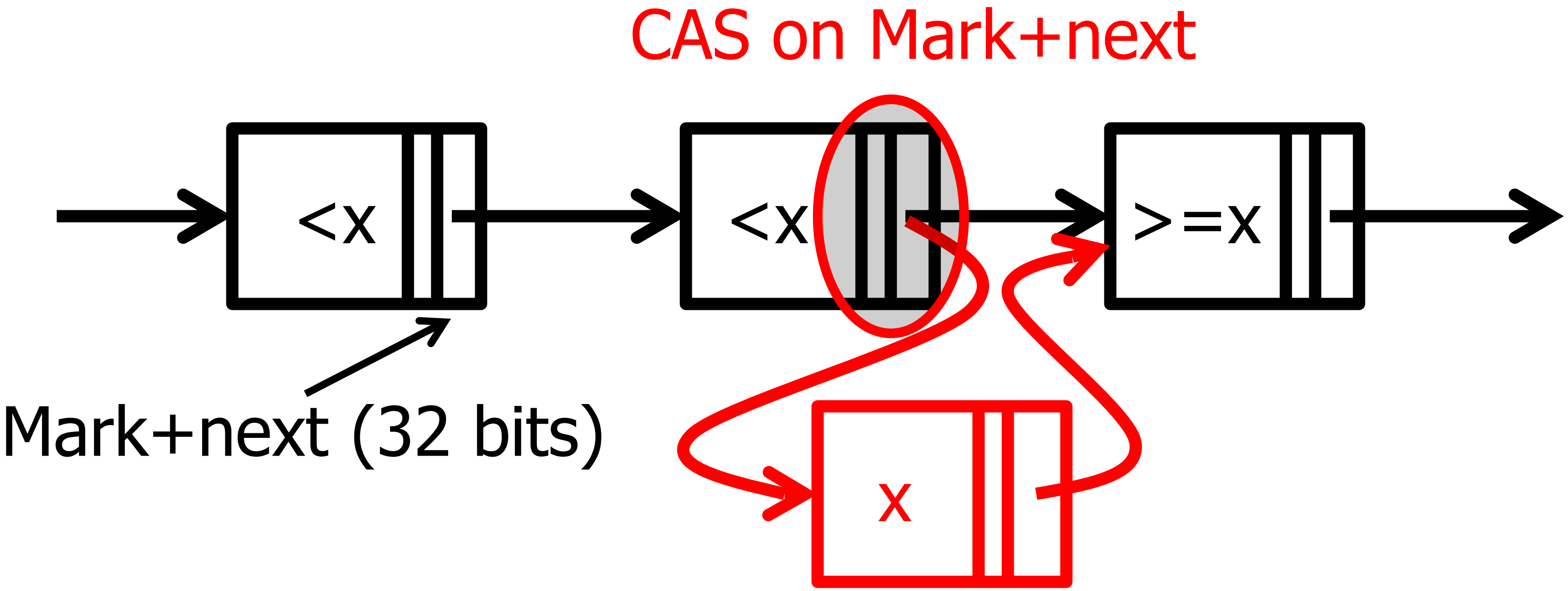
Lock-free data structures

- Linear lists, hash tables, skip lists, priority queues
  - Important computation building blocks
  - We implement a set using these data structures
  - Lock-free and wait-free operations on the set
  - Lock-free operation: infinitely often some instance of this operation finishes in finite number of steps
  - Wait-free operation: every instance of this operation finishes in finite number of steps
  - Correctness criteria: linearizable (except priority queue, which is quiescently consistent)
    - See paper for definition

Lock-free linear list

- Implemented using a sorted singly linked list
- Supported ops: add, delete, search
  - Add(x) returns 0 if x is already in the set; otherwise adds x at sorted position and returns 1
  - Delete(x) returns 0 if x is not in the set; otherwise removes x from the set and returns 1
  - Search(x) returns 0 or 1 if x is not found or found in the set
  - Add and delete are lock-free
  - Search is wait-free (just walks the list)
  - Delete only logically deletes a node by marking it
  - Subsequent add and delete operations physically remove the logically deleted nodes

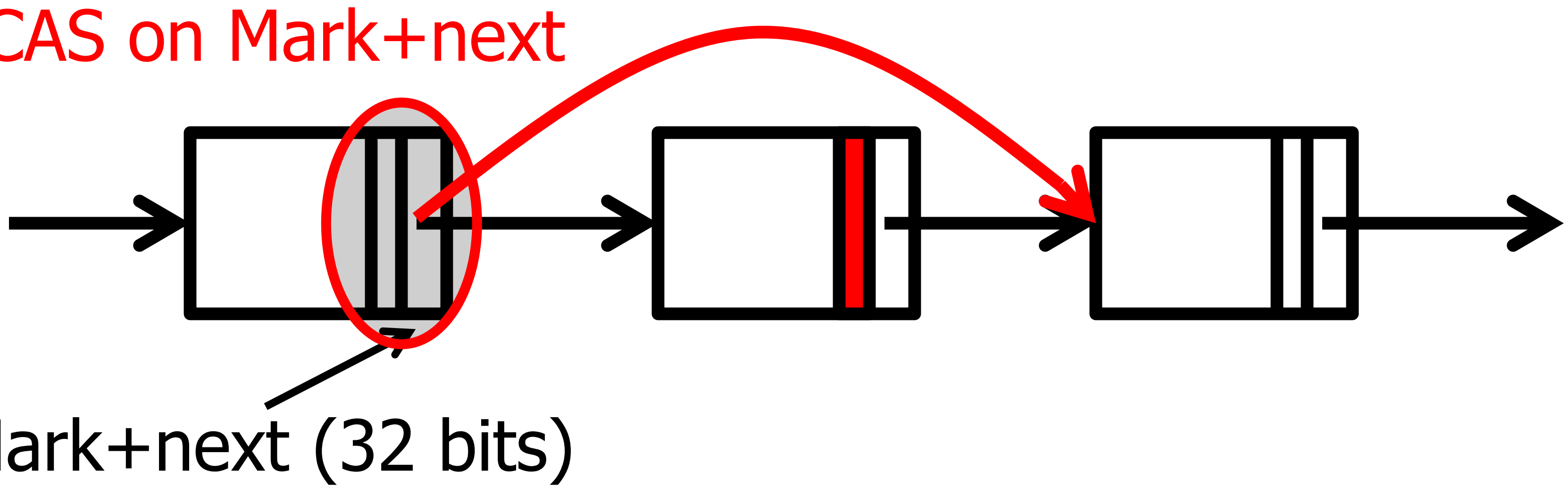
Lock-free linear list: add(x)



The Mark bit is the least significant bit of the aligned 32-bit next field

- Needed for logical deletion

Lock-free linear list: Physical delete



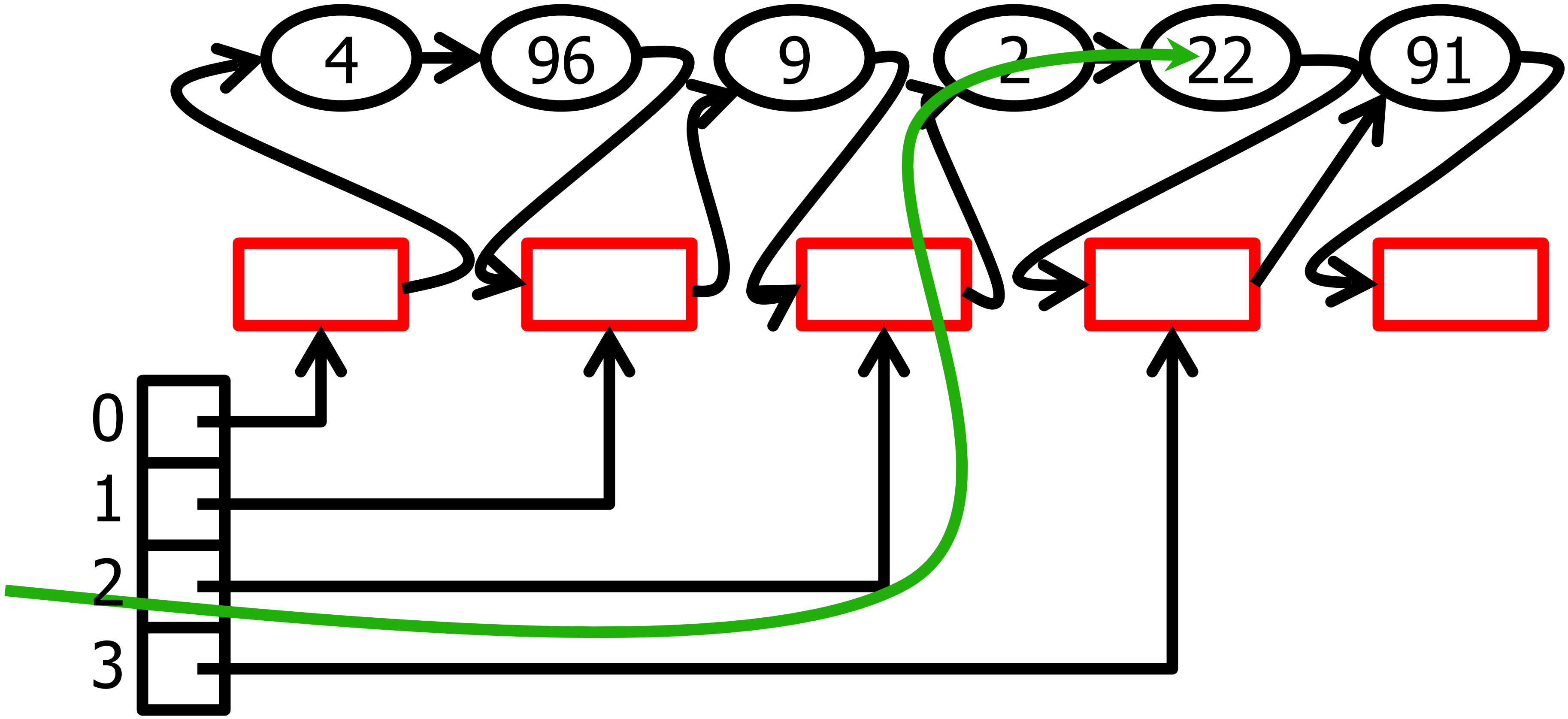
Delete(x) logically marks a node. Subsequent add or delete physically deletes it when walking the list.

Lock-free hash table

- Leverages lock-free linear list construction
  - Implemented as a single linear list
  - An array of pointers stores the starting point (head node) of each bucket
  - The head node of each bucket stores a special key
  - Add, delete, and search operations on a bucket start at the head node of that bucket
  - Number of buckets is constant and fixed at the time of CUDA kernel launch
- Supports the same three operations as lock-free linear list

Lock-free hash table

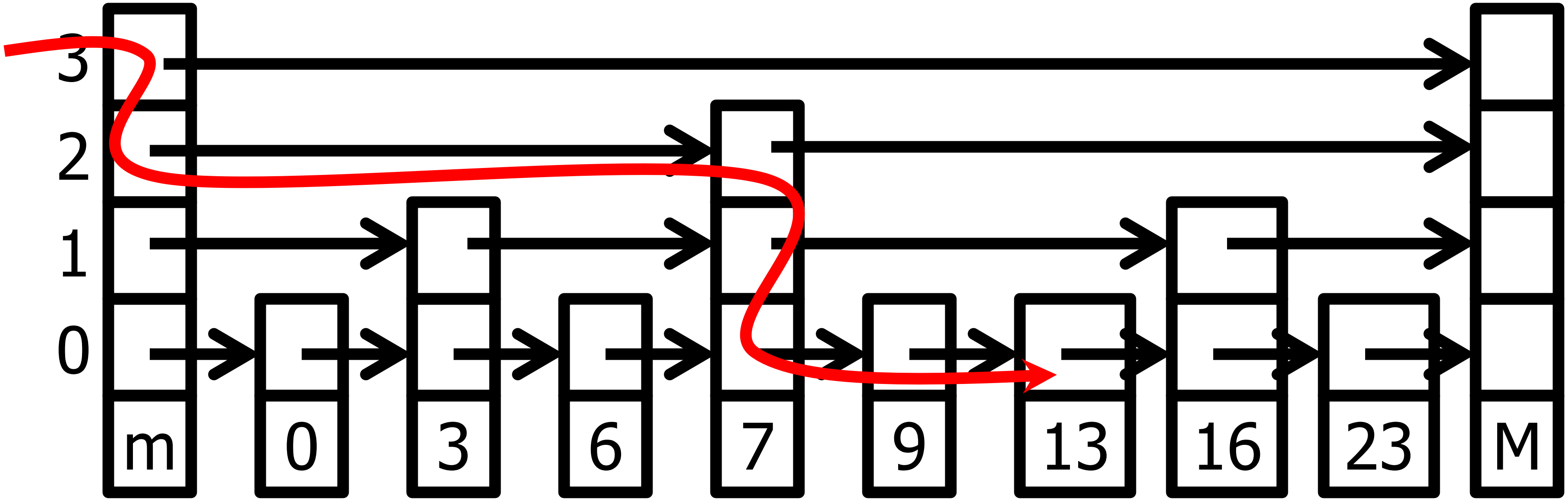
Delete(22): 22 mod 4 = 2



Skip list

- A skip list is a hierarchy of linear lists
  - Keys present in level n+1 form a subset of the keys present in level n
  - Given that a key is present in level n, there is a probability p of finding the key in level n+1
  - When a new key is inserted, the maximum level up to which this key can be present is decided by a random number r with expected value 1/(1-p)
- A skip list offers expected logarithmic search complexity

Skip list



- Keys are kept sorted at the lowest-level list
  - Head and tail nodes maintain the smallest and largest keys
- Upper-level lists provide probabilistic short-cuts into the lower-level lists leading to an expected logarithmic search time

Lock-free skip list

- Leverages lock-free linear list implementation
- Additional complications in linking up or removing multiple nodes in different lists
  - Not possible to make multiple Mark+next field modifications atomic using single-word CAS
  - Depending on the traversal path of Add and Delete some middle level node of a marked key may get physically removed while leaving the other levels unchanged: violates the subset property

Lock-free skip list

- Add is made linearizable by adding a key bottom-up
- Delete is made linearizable by logically marking the levels of the key to be deleted top-down
- A key is defined to be present in the set if it is found unmarked in the lowest-level list
- Two major performance bottlenecks
  - Large number of CAS operations
  - Complex code structure leading to significant volume of control flow divergences

Lock-free priority queue

- Supports two operations on the underlying set: Add and DeleteMin
- Leverages lock-free skip list due to its logarithmic search complexity guarantee
  - Makes the Add operation to have expected logarithmic time
  - DeleteMin walks the lowest-level list until an unmarked key is found, which it marks logically using CAS and calls Delete of skip list on that key
- New performance bottleneck
  - Heavy contention near the head due to concurrent DeleteMin operations

Sketch

- Talk in one slide
- Result highlights
- Related work
- Lock-free data structures
- ➢ CUDA implementation
- Evaluation methodology
- Empirical results
- Summary

CUDA implementation

- Extensive use of atomicCAS
- All data structures use a generic node class
  - All of them build on the basic linear list
- Large number of nodes are pre-allocated
  - Pointers to these are stored in an array
  - A global index points to the next free node
  - An Add operation executes an atomicInc on this index and uses the node pointed to by the pointer at the returned index
- Deleted nodes are not reused
  - Requires an implementation of an elaborate solution to the ABA problem
  - Left to future research

Sketch

- Talk in one slide
- Result highlights
- Related work
- Lock-free data structures
- CUDA implementation
- ➢ Evaluation methodology
- Empirical results
- Summary

Evaluation methodology

- Experiments are done on two platforms
  - Tesla C2070 card featuring one GF100 Fermi GPU
    - 14 streaming multiprocessors (SM), each having 32 CUDA cores; thread blocks map to SMs
    - 1.15 GHz core frequency and 1.49 GHz memory frequency
    - 48 KB shared memory and 16 KB L1 cache per thread block; 768 KB globally shared L2 cache
  - Quad processor SMP, each processor having six cores (Intel X7460 CPU) running at 2.66 GHz
    - 16 MB L3 cache shared by six cores in each processor
    - Lock-free implementations use POSIX threads and rely on x86 cmpxchg instruction for realizing the atomicCAS primitive

Evaluation methodology

- Each data structure is evaluated on
  - A range of integer keys [0, 100), [0, 1000), [0, 10000), and [0, 100000)
    - Keys are generated uniformly at random from the range; these are input arguments to the operations
  - Two different mixes of supported operations
  - Different number of operations ranging from 10000 to 100000 in steps of 10000
- Number of thread blocks and threads per block for the CUDA kernel are optimized
  - In most cases, the number of thread blocks is such that each thread carries out one operation
  - 64 threads per block for linear list and 512 threads per block for the rest

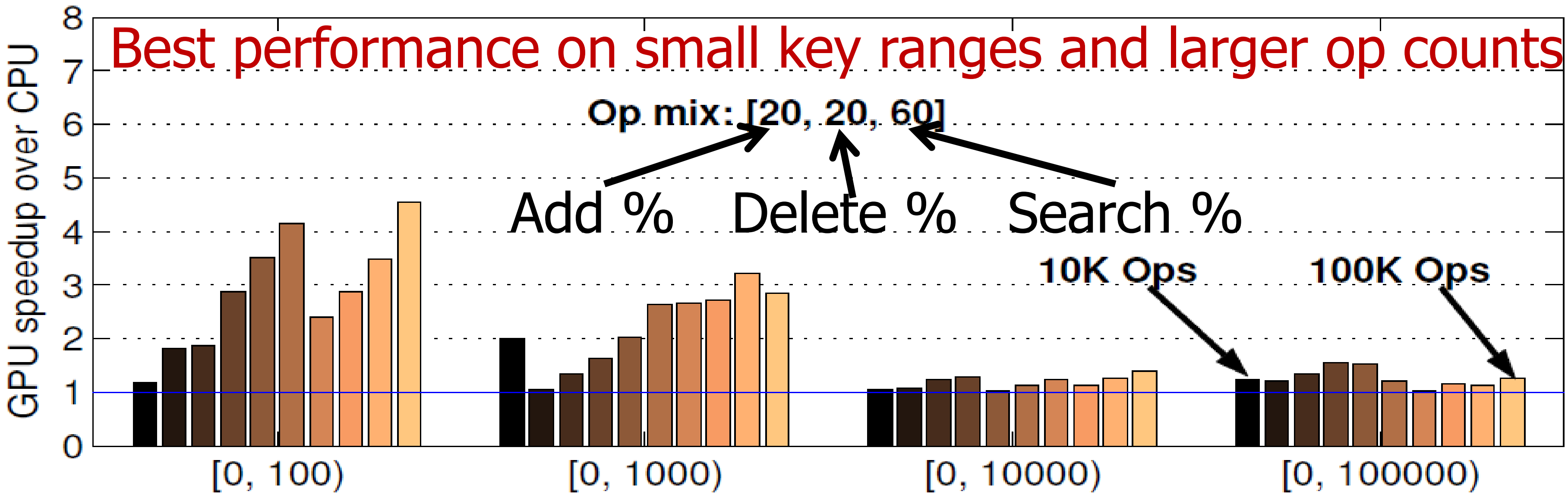
Evaluation methodology

- For evaluation on CPU, thread count that offers the best performance is picked
  - 24 threads do not always offer the best
- In summary, each experiment shows results using the best performance on the GPU as well as on the CPU
- Lock-free hash table uses ten thousand buckets
- Lock-free skip list uses p=0.5 and 32 levels
  - Lock-free priority queue leverages the lock-free skip list that uses the same parameters

Sketch

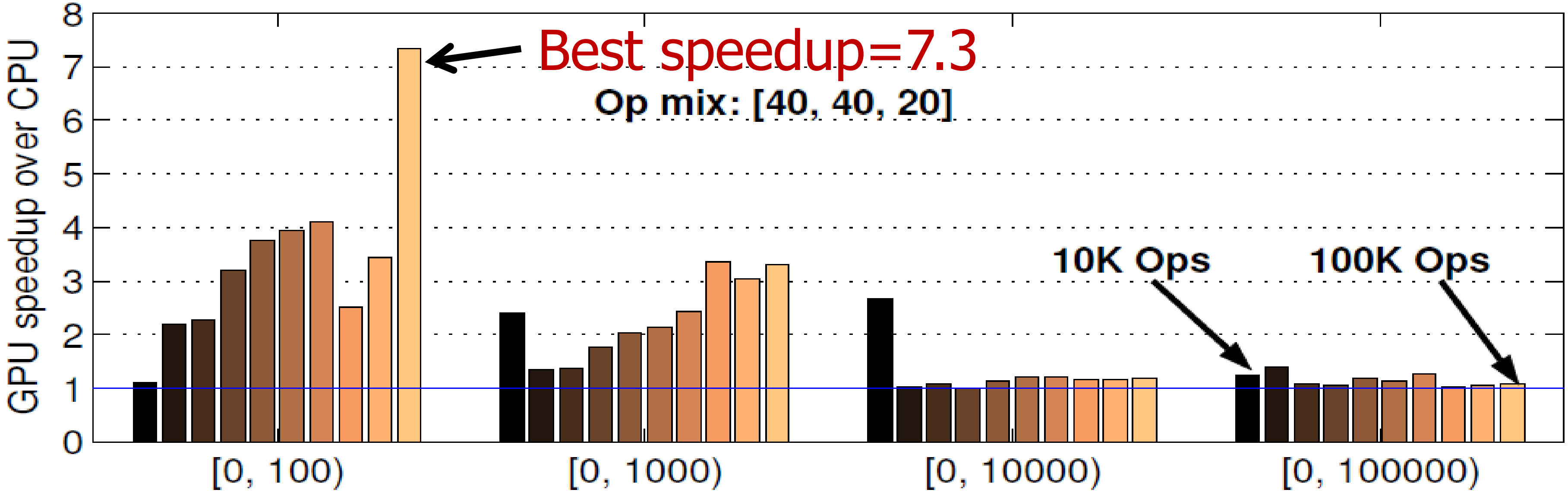
- Talk in one slide
- Result highlights
- Related work
- Lock-free data structures
- CUDA implementation
- Evaluation methodology
- ➢ Empirical results
- Summary

Lock-free linear list



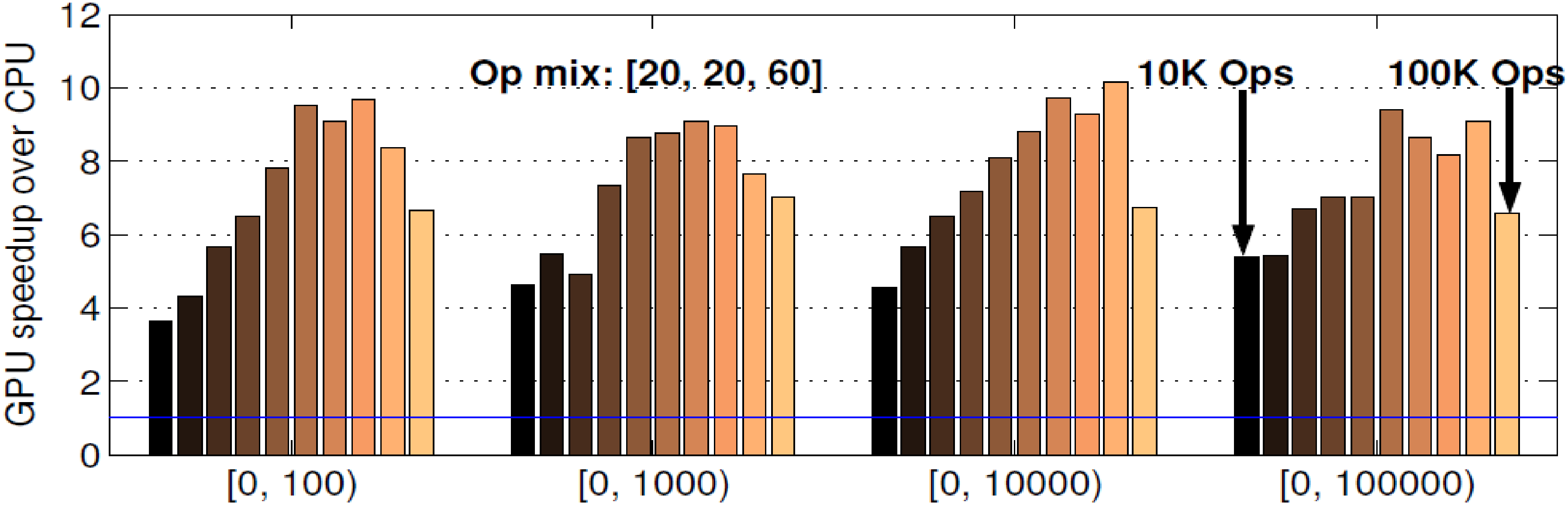
Best performance on small key ranges and larger op counts

No major difference between search-heavy and add/delete-heavy op strings



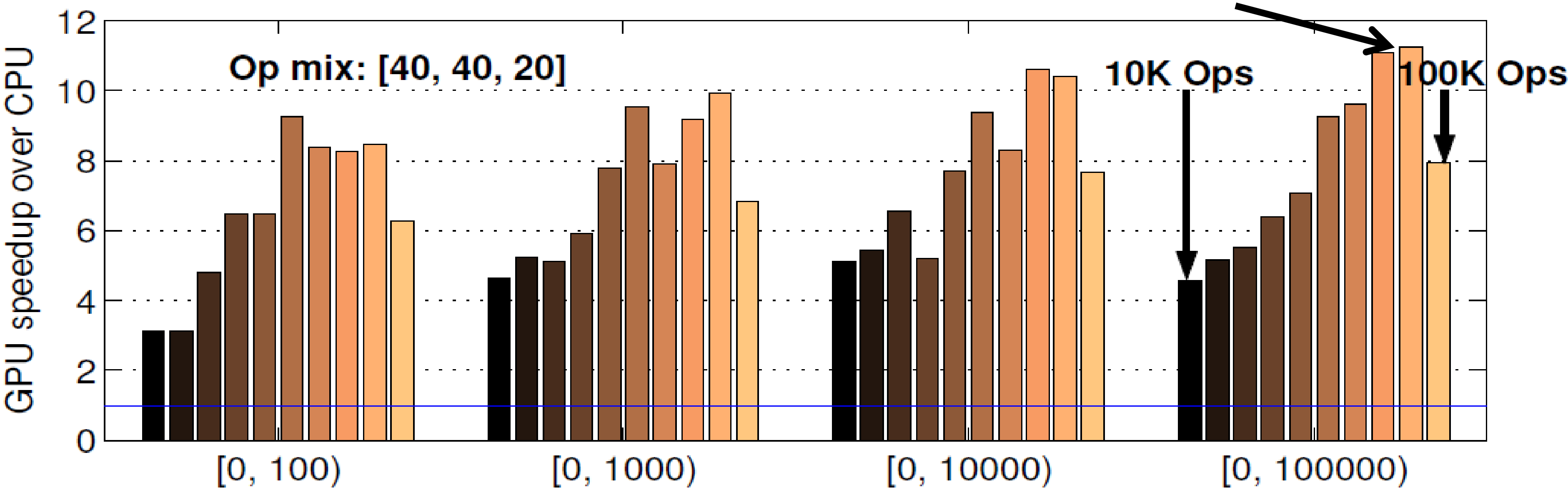
Best speedup=7.3

Lock-free hash table

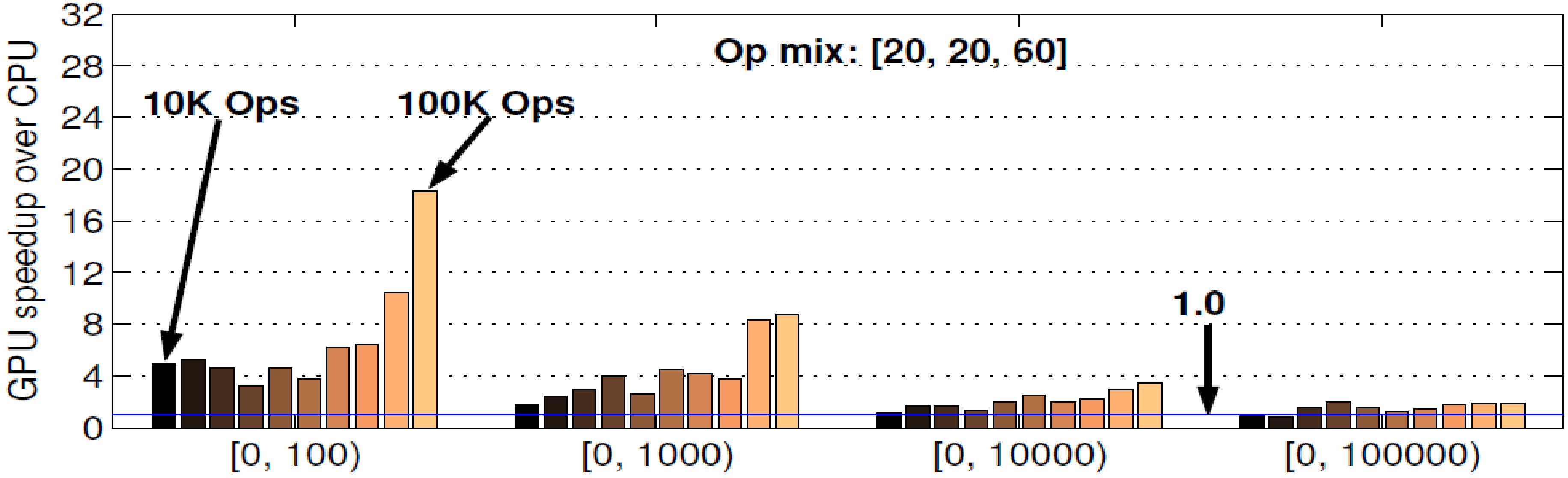


Consistent speedup across all key ranges and op-mixes

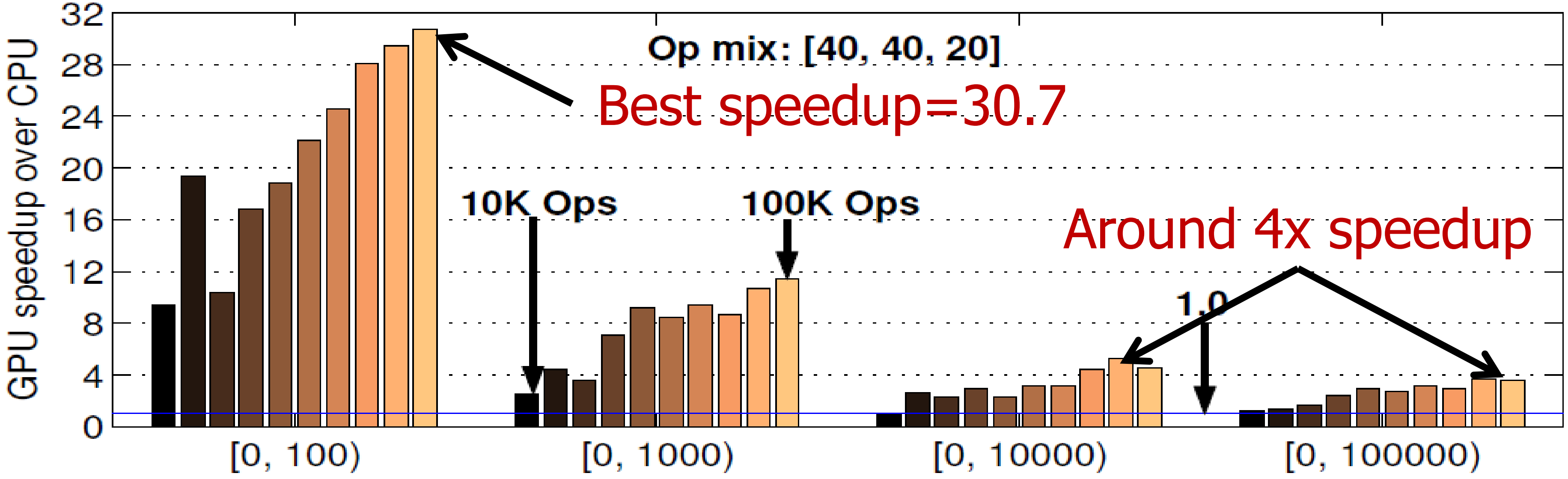
Best speedup = 11.3



Lock-free skip list



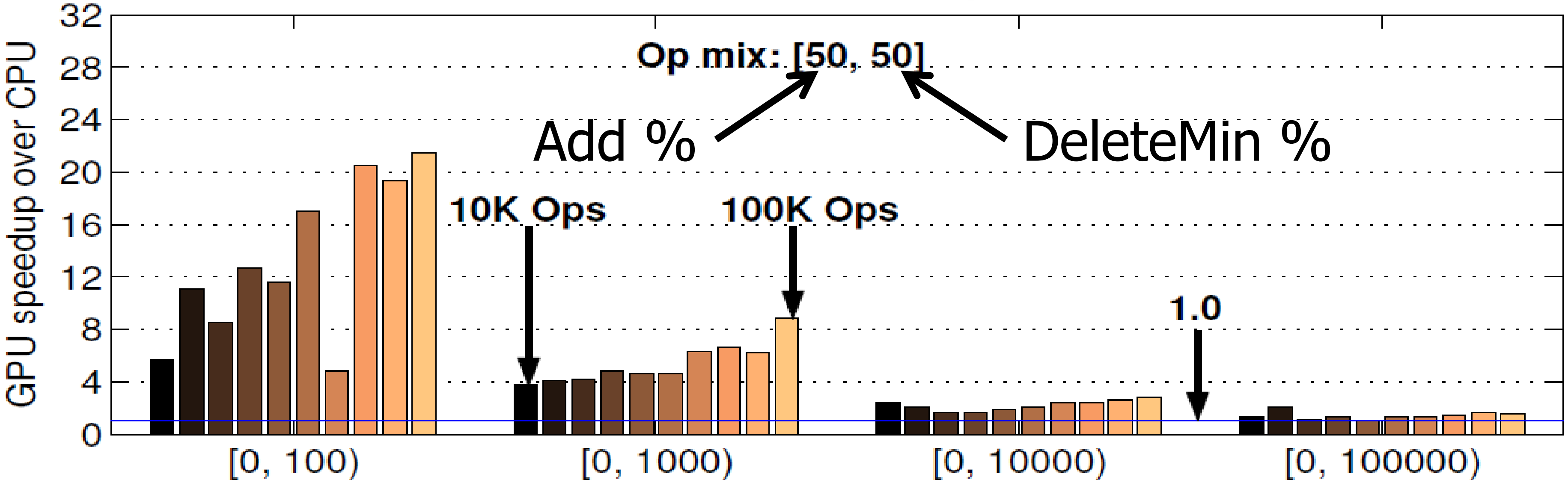
[illegible]



Best speedup=30.7

Around 4x speedup

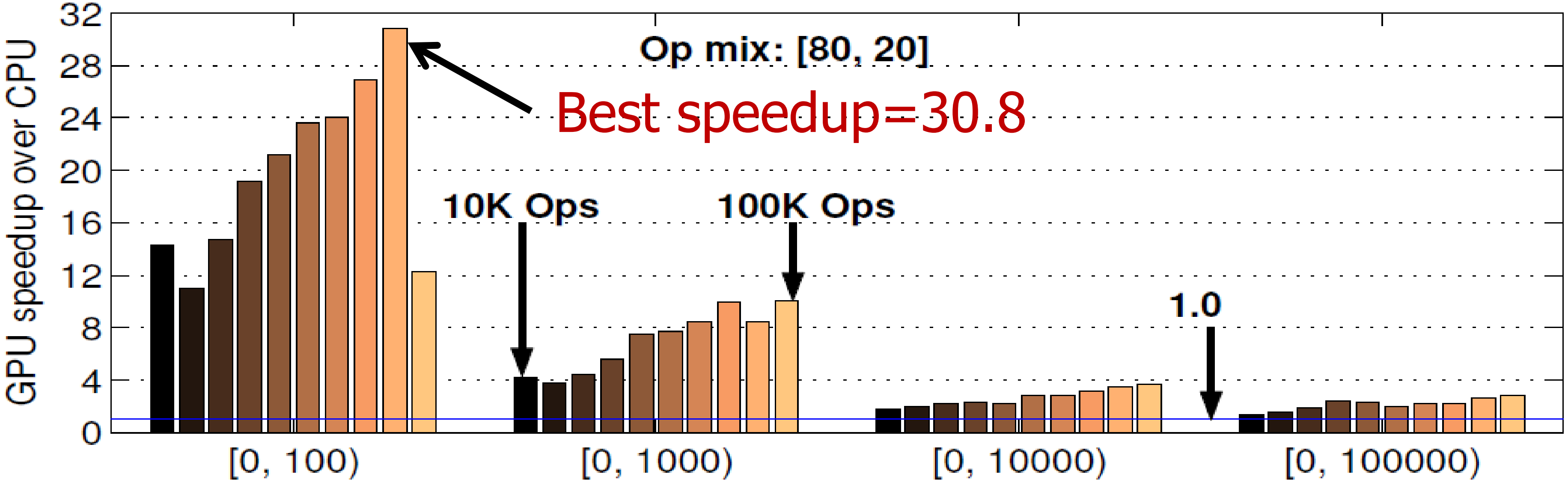
Lock-free priority queue



Add %

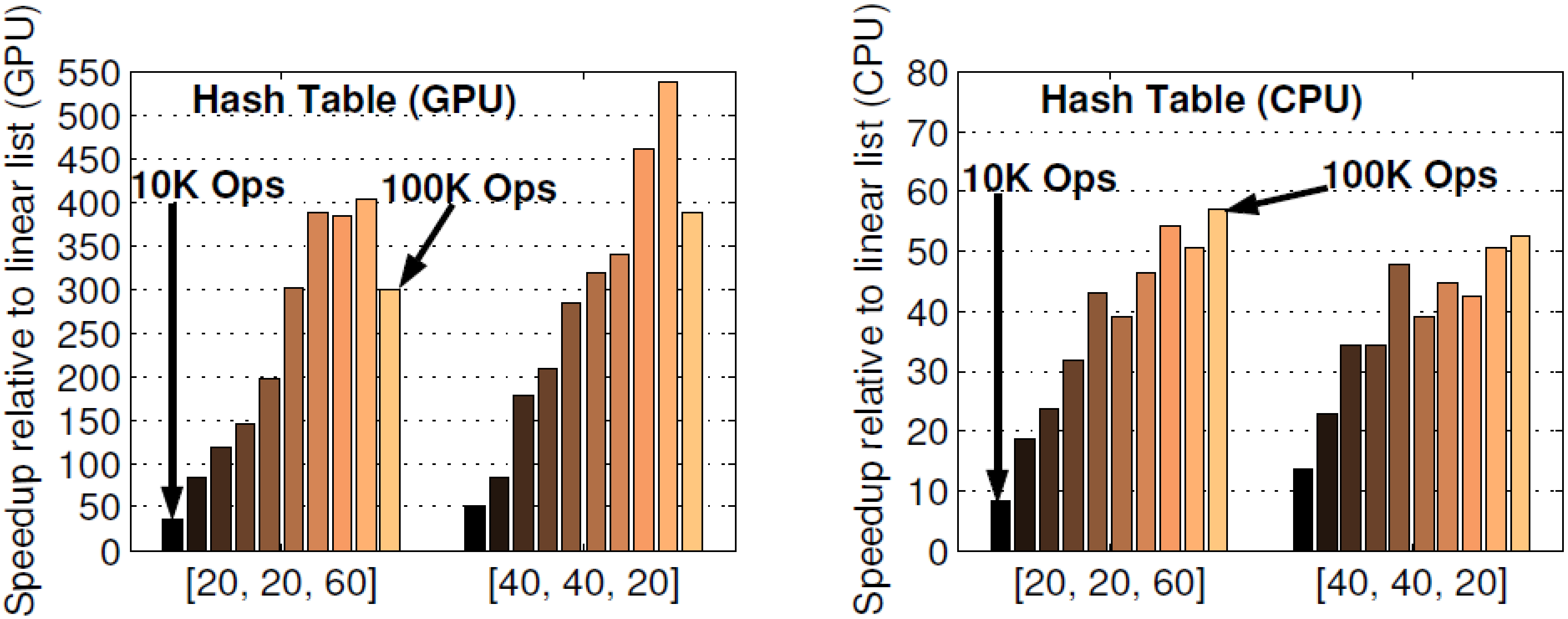
DeleteMin %

Trends are similar to skip list: speedup increases with Add %



Best speedup=30.8

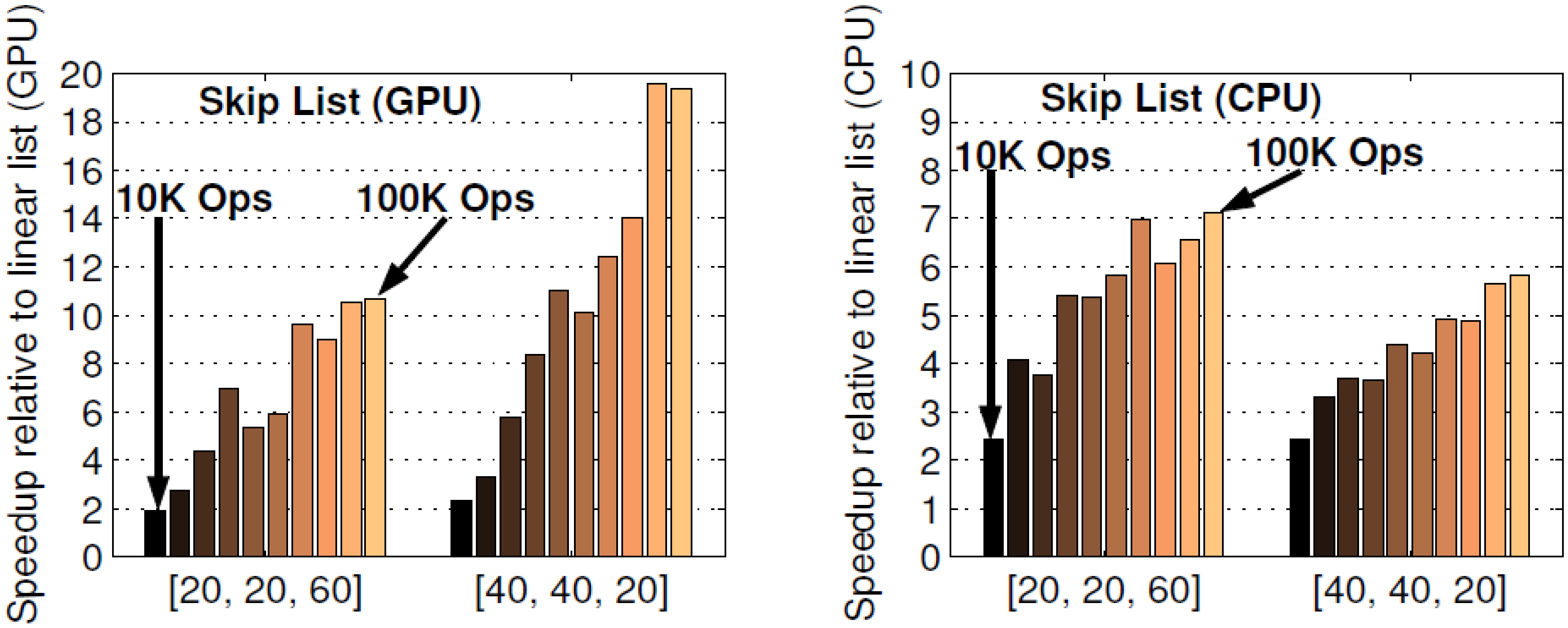
Hash table vs. linear list



The data are shown for the largest key range

- On GPU, the hash table is 36x to 538x faster than linear list
- On CPU, the hash table is only 8x to 54x faster than linear list
- GPU exposes more concurrency in the lock-free hash table

Skip list vs. linear list



- ❖ On GPU, the skip list is 2x to 20x faster than linear list
- ❖ GPU still exposes more concurrency than CPU for skip list
- ❖ Hash table shows far better scalability than skip list

Throughput of hash table

- Hash table is the best performing data structure among the four we have evaluated
  - For the largest key range, on a search-heavy op mix [20, 20, 60], the throughput ranges from 28.6 MOPS to 98.9 MOPS on the GPU
  - For an add/delete-heavy op mix [40, 40, 20], the throughput range is 20.8 MOPS to 72.0 MOPS
- Nearly 100 MOPS on a search-heavy op mix

Sketch

- Talk in one slide
- Result highlights
- Related work
- Lock-free data structures
- CUDA implementation
- Evaluation methodology
- Empirical results

➢ Summary

Summary

- First detailed evaluation of four lock-free data structures on CUDA-enabled GPU
- All four data structures offer moderate to high speedup on small to medium key ranges compared to CPU implementations
- Benefits are low for large key ranges in linear lists, skip lists, and priority queues
  - Primarily due to CAS overhead and complex control flow in skip lists and priority queues
- Hash tables offer consistently good speedup on arbitrary key ranges and op mixes
  - Nearly 100 MOPS throughput for search-heavy op mixes and more than 11x speedup over CPU

Summary

- Further improvement requires two key architectural innovations in GPUs
  - Fast atomics and high synchronization throughput
    - Helpful for all kinds of scalable implementations
  - Reduction in control flow divergence overhead
    - Helpful for complex lock-free constructions such as skip lists and priority queues

http://www.cse.iitk.ac.in/~mainakc/lockfree.html

Thank you