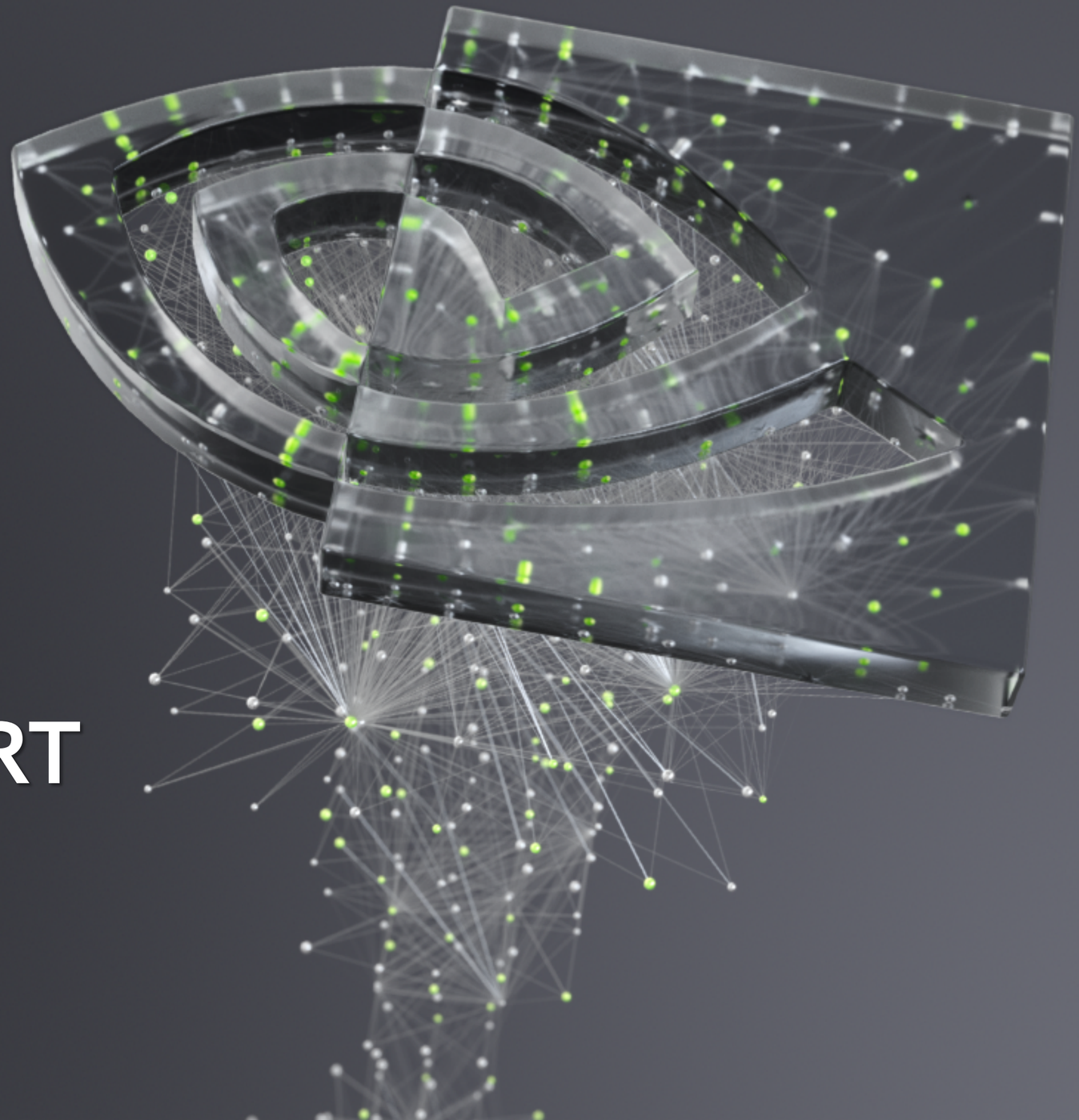


A FASTER RADIX SORT IMPLEMENTATION

Andy Adinets

AGENDA

Introduction

Radix sort

Optimizations

What makes our radix sorting implementation faster

Results

Performance comparisons, conclusion

LEAST SIGNIFICANT DIGIT RADIX SORT

Algorithm

Sorts n keys of w bits each

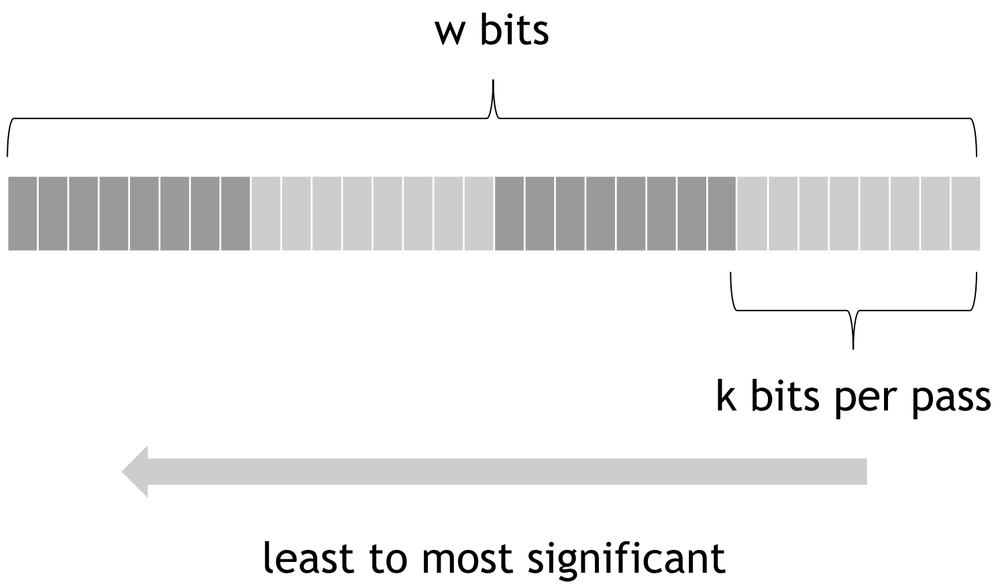
- complexity O(w * n)

Sequence of passes

- k bits (1 digit) at a time

Each pass

- stable partition (counting sort)
- of the whole array
- main GPU primitive



LEAST SIGNIFICANT DIGIT RADIX SORT

Information

Stable sort

- preserves the order of elements with the same key

State-of-the-art in GPU sorting

- `cub::DeviceRadixSort::Sort{Keys,Pairs}[Descending]()`
- `thrust::{sort,stable_sort}[_by_key]()` (when it uses `cub::DeviceRadixSort`)

PARTITION

As performed currently

Split input into tiles

Upsweep (N reads)

per-block histogram

ScanBins

exclusive prefix sum of all digit counts

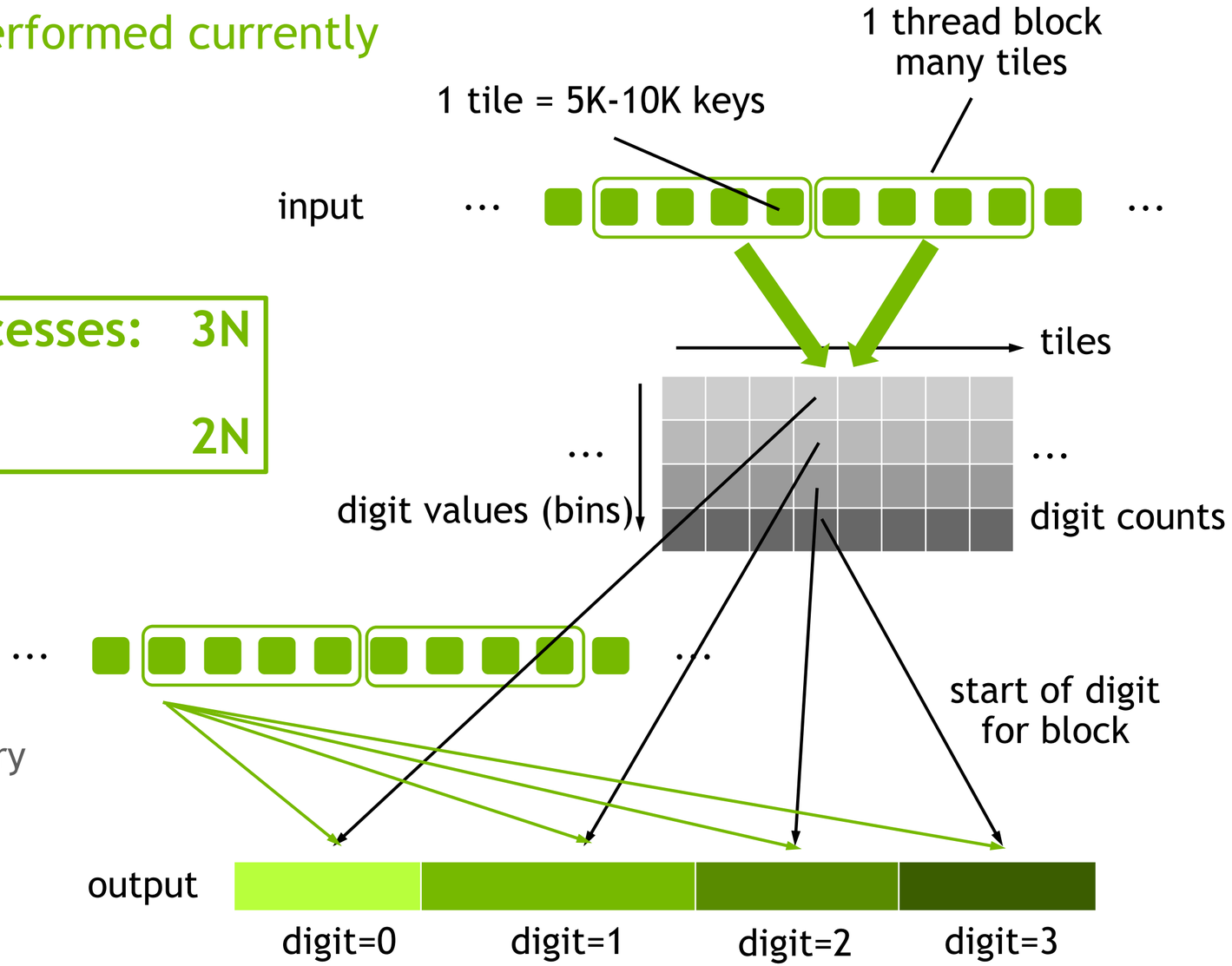
Downsweep (N reads, N writes)

partition keys by digit value in shared memory

write to global memory

memory accesses: 3N

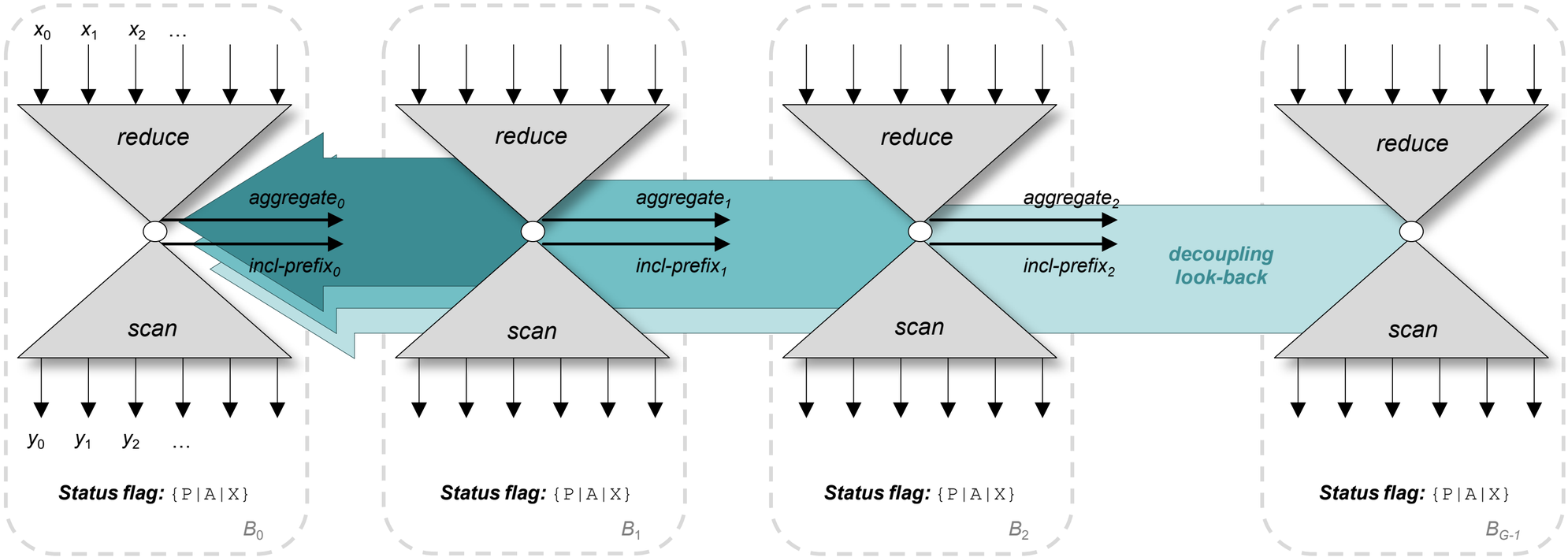
want: 2N



DECOUPLED LOOK-BACK

For prefix sums

D. Merrill, M. Garland, Single-Pass Prefix Scan with Decoupled Look-back

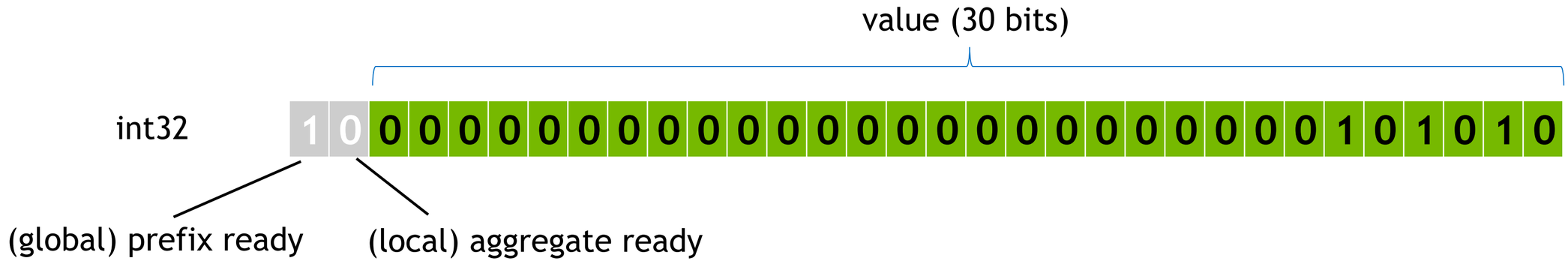


DECOUPLED LOOK-BACK

For partition

2^k-way partition: 2^k decoupled look-backs

distribute digit values between threads



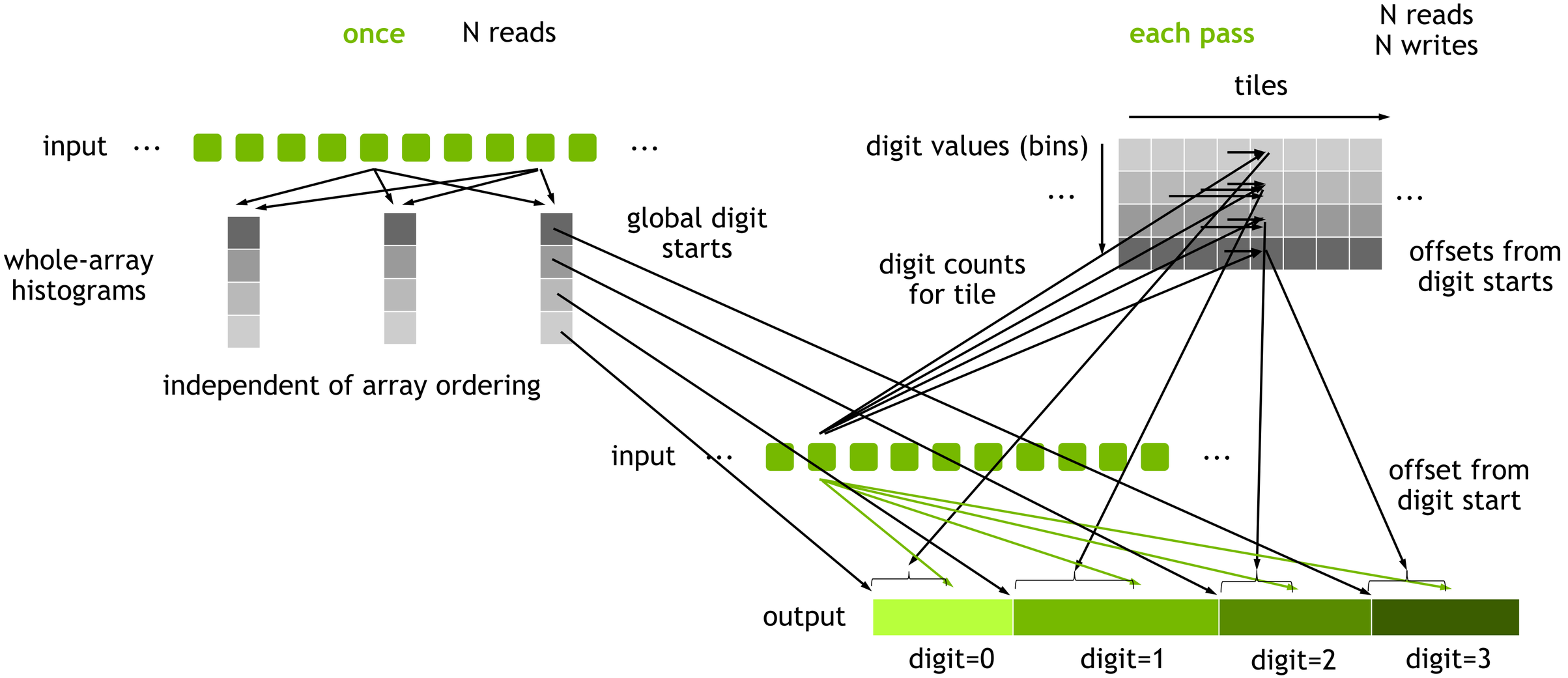
For >= 2^{30} keys, invoke multiple kernels

Only handles per-digit prefix sum

still need the starting position for each digit

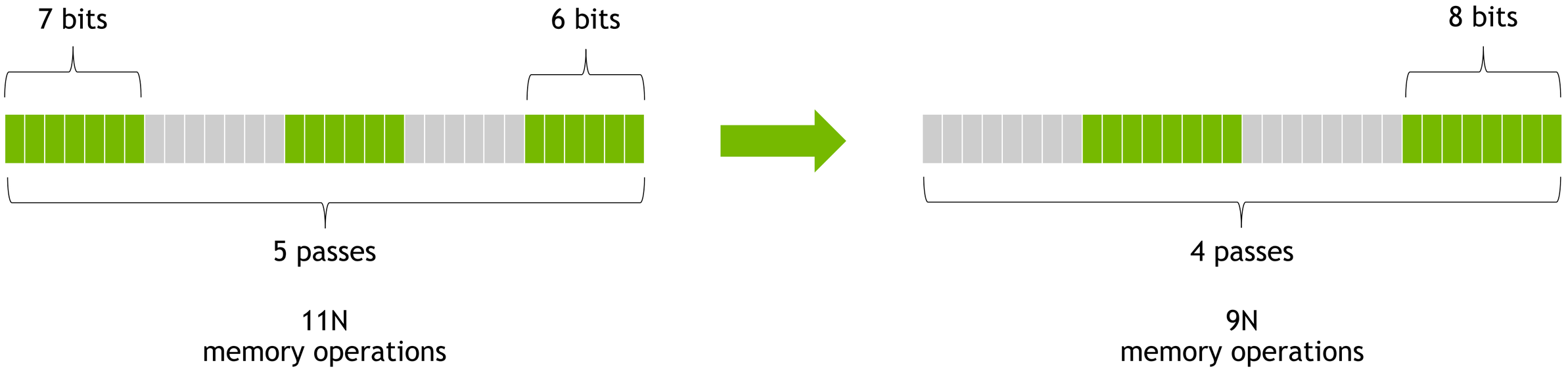
ONESWEEP

Histogram Pass + Decoupled Look-back



LARGER PASSES, FEWER PASSES

uint32: 4 passes x 8 bits



larger passes => larger histograms

more shared memory

more global memory operations

STABLE RANKING

As performed in cub

Partition keys by current digit

- in shared memory
- preserves the order of the keys with the same digit

Implementation in cub

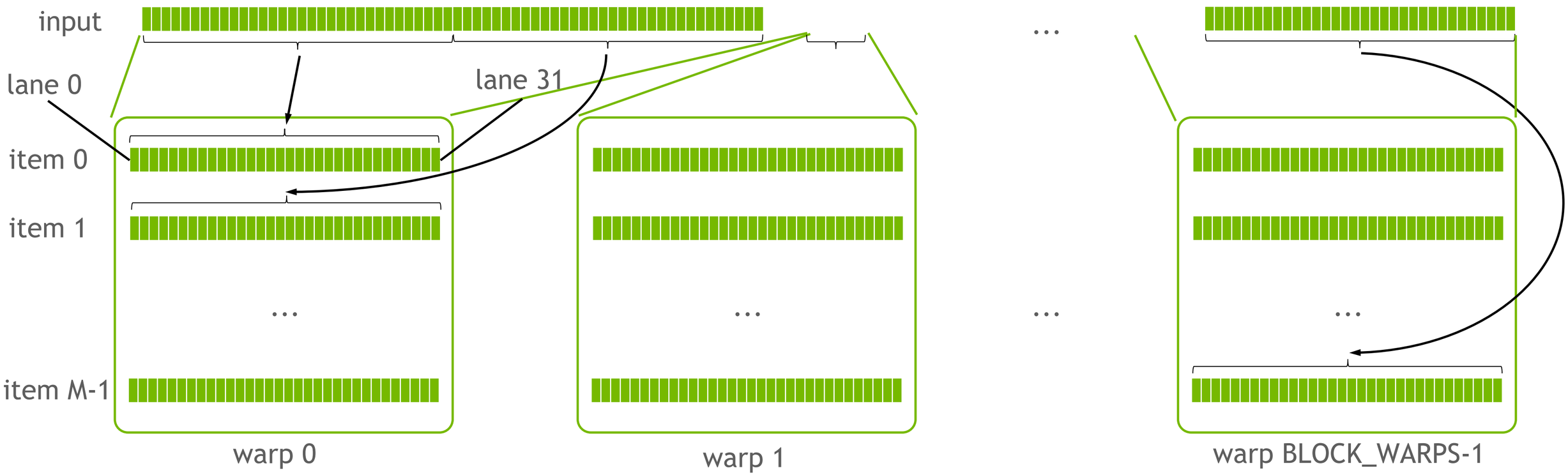
- https://github.com/thrust/cub/blob/master/cub/block/block_radix_rank.cuh

ORDER OF KEYS FOR RANKING

Match-based ranking in cub

idx -> (warp, item, lane)

row-major

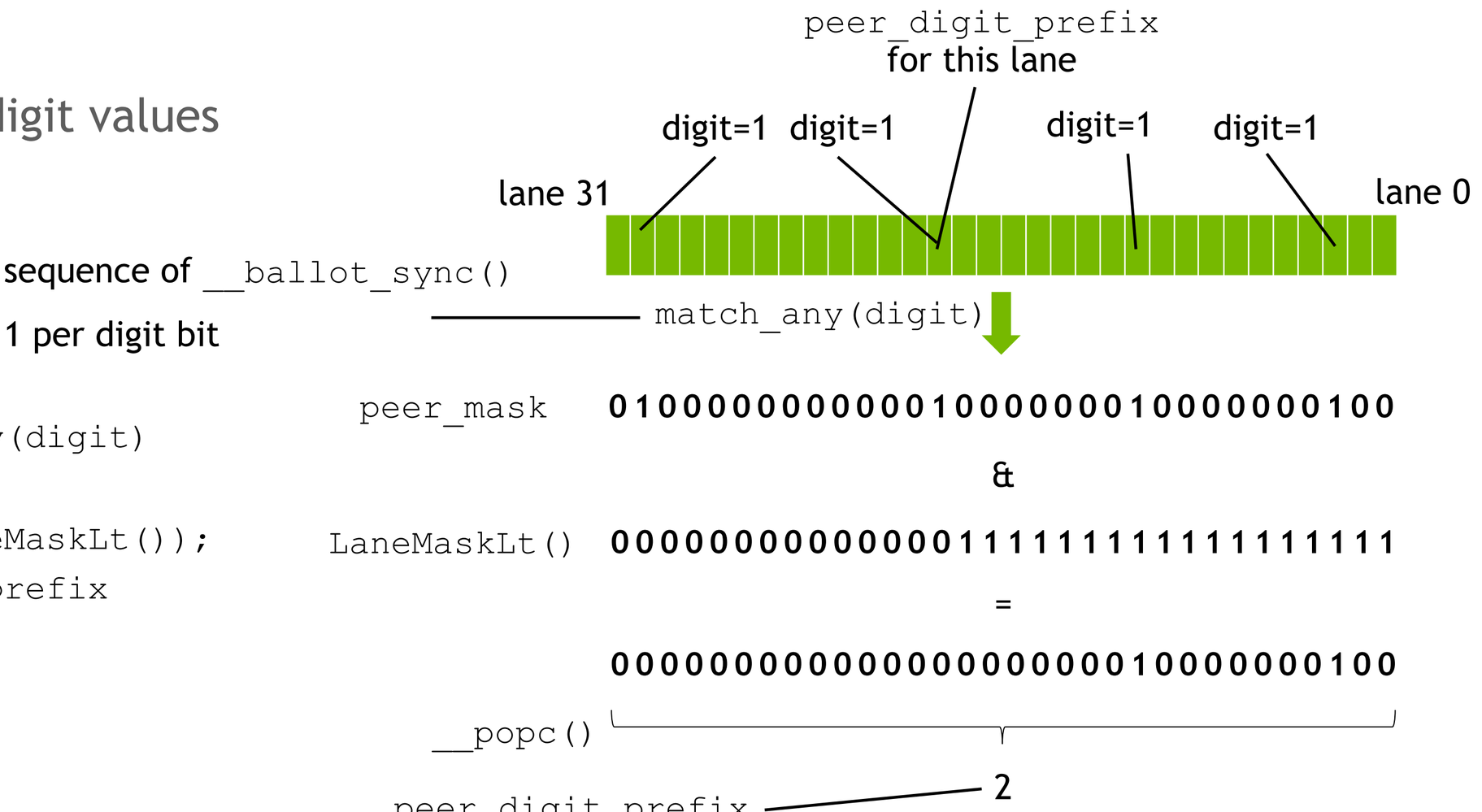


PEER DIGIT PREFIX

As computed in cub

in parallel for different digit values

```
int peer_mask = match_any(digit)
int peer_digit_prefix =
  __popc(peer_mask & LaneMaskLt());
rank[item] = peer_digit_prefix
```



STABLE RANKING (2)

As performed in cub

Warp digit prefix

- warp-private digit histogram
- incremented by leader
- add to rank

Within thread block

- exclusive prefix sum of warp digit counts
- add to rank

STABLE RANKING

Optimized for decoupled look-back

Want digit counts earlier

to unblock other thread blocks

Compute them earlier, move match() to later

warp-private histograms with `atomicAdd()`

sum them up => **digit counts**

modify the rest of the algorithm accordingly

FASTER MATCH

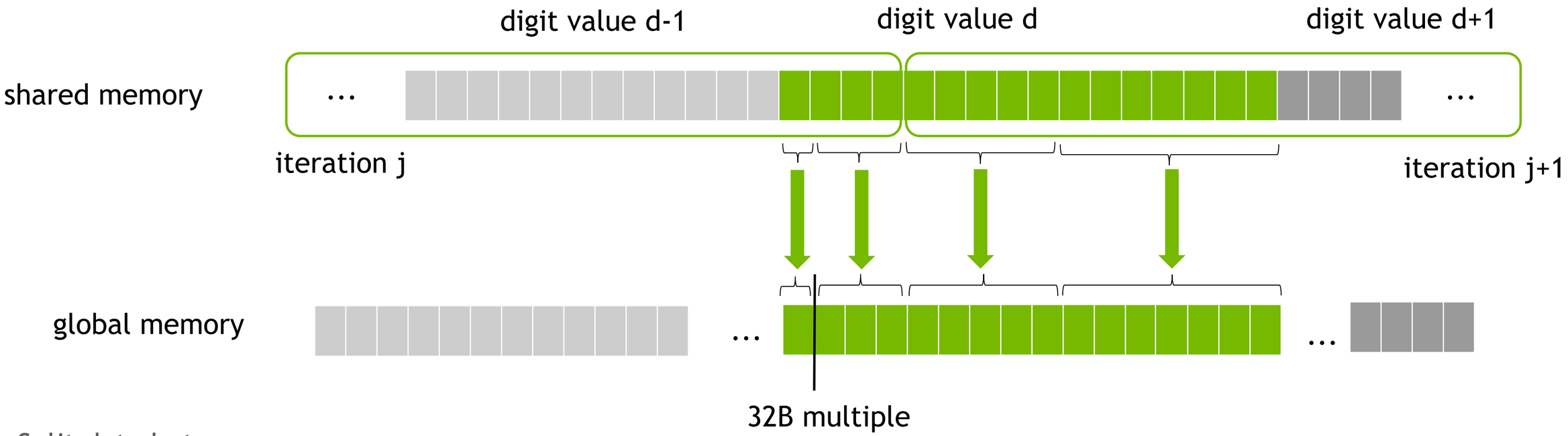
Using atomicOr()

Mask of threads in the warp with the same digit value

```
__shared__ int shared_match_masks[WARPS][DIGITS];  // init with 0
int* match_masks = shared_match_masks[warp];

...

atomicOr(match_masks[digit], 1 << lane);
__syncwarp(~0);
int peer_mask = match_masks[digit];
int leader = (WARP_THREADS - 1) - __clz(peer_mask);  // highest-order bit set

...
// update counters, __shfl_sync() at the end

if (lane_id == leader) match_masks[bin] = 0;
__syncwarp(~0);
```

DIRECT WRITE-OUT

Writing keys to global memory



Split data between warps

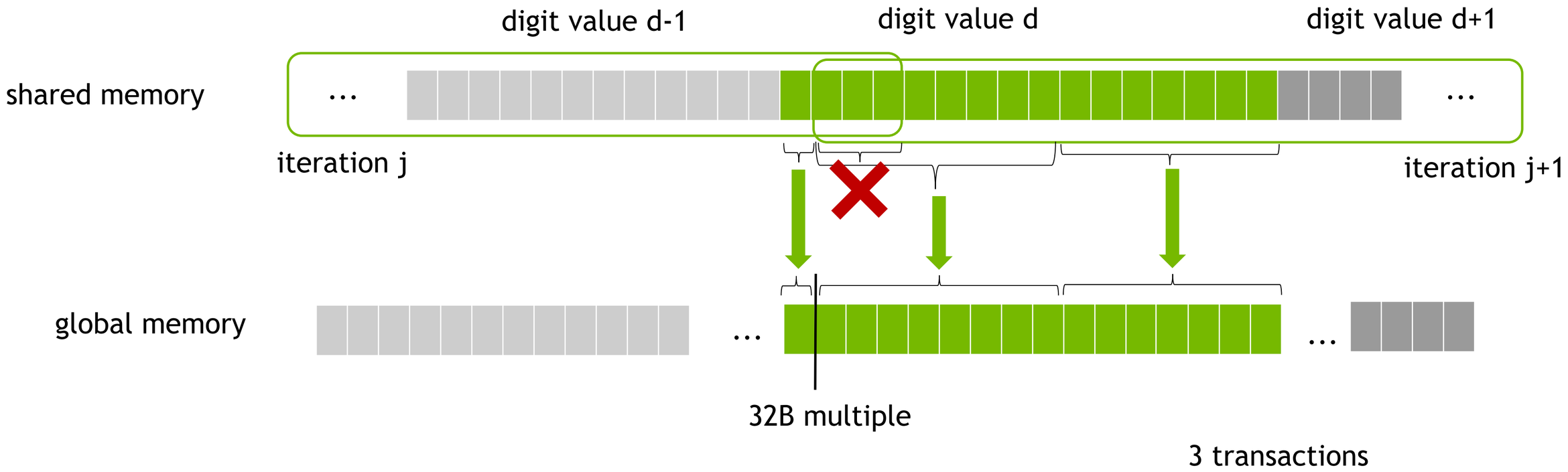
Each thread computes the key's digit value and writes the key to global memory

Problem

misalignment => unnecessary transactions

ALIGNED WRITE-OUT

Writing keys to global memory




```
if (lane_id == 31 && global_idx % 8 != 7)
```

last `(global_idx % 8 + 1)` threads don't write

pick up in the next iteration

OPTIMIZATIONS

For onesweep

Decoupled look-back (3N -> 2N memory operations per pass)

Larger passes, fewer passes (6-7 bits x 5 passes -> 8 bits x 4 passes)

Compute digit counts earlier

- helps with decoupled look-back

Minor optimizations

- match based on `atomicOr()`
- optimized writeout (32-bit keys)

SETUP

For performance comparison

CUDA 10.1.243

V100-SXM2

cub

https://github.com/thrust/cub

commit 6552e4d429c194e11962feb638abf87bcf220af0 (Feb 20, 2020)

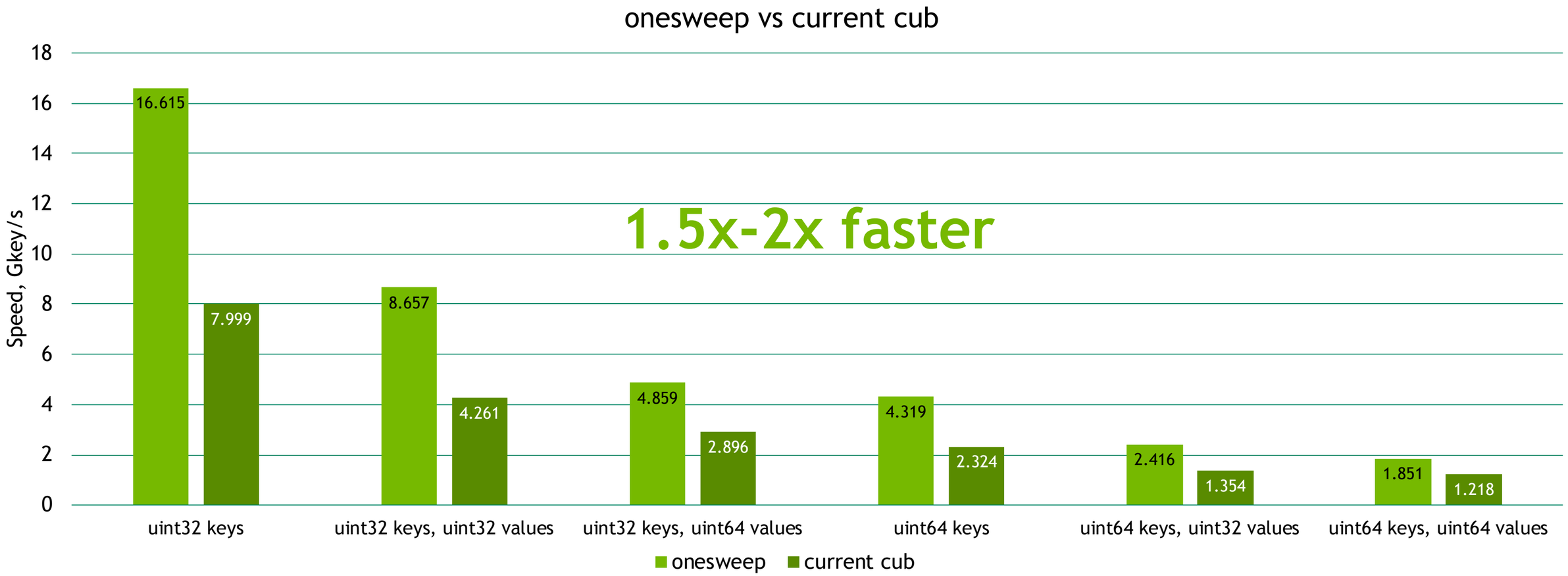
onesweep

implemented inside cub

Sort 64M random elements

PERFORMANCE COMPARISON

V100, 64M elements



NVIDIA

OPTIMIZATIONS

For current cub sorting

Match characteristics of Partition and Downsweep

1 tile / thread block

- better performance

ScanBins -> DeviceScan::ExclusiveSum()

- faster for large arrays of digit counts

Updated policy settings

- 23 items / thread
- 256 threads / block
- match-based ranking also for 6 bits

EFFECTS OF INDIVIDUAL OPTIMIZATIONS

Sort 64M uint32 keys, V100

Optimization	Improvement	Performance, Gkey/s
current cub version		7.999
+ changes from the previous slide	+ 30.8%	10.460
+ decoupled look-back (cub's match-based ranking, 23 items/thread)	+ 32.5%	13.862
+ 8 bits / pass	+ 11.0%	15.393
+ compute digit counts earlier	+ 3.4%	15.917
+ `atomicOr()`-based match	+ 1.9%	16.213
+ aligned writeout (46 items/thread) (onesweep)	+ 2.5%	16.615

CONCLUSION

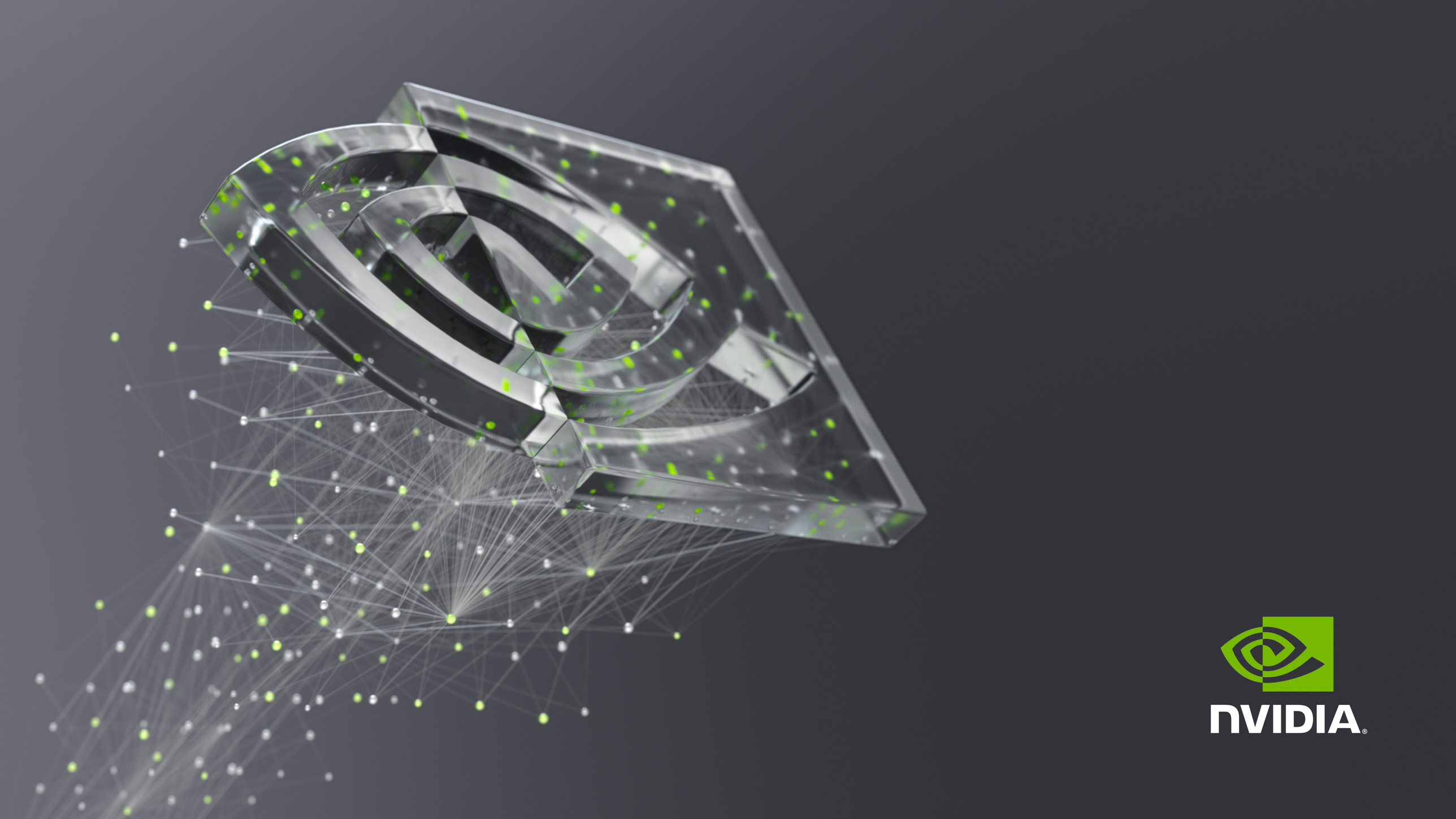
Onesweep is being integrated into cub

Some beta-testers

GPU-Accelerated Genome Assembly: A Deep Dive into Clara Genomics Analysis SDK [S21968]

Feel free to contact me with questions or comments

aadinets@nvidia.com


NVIDIA