

Generalized Histogram Algorithms for CUDA GPUs

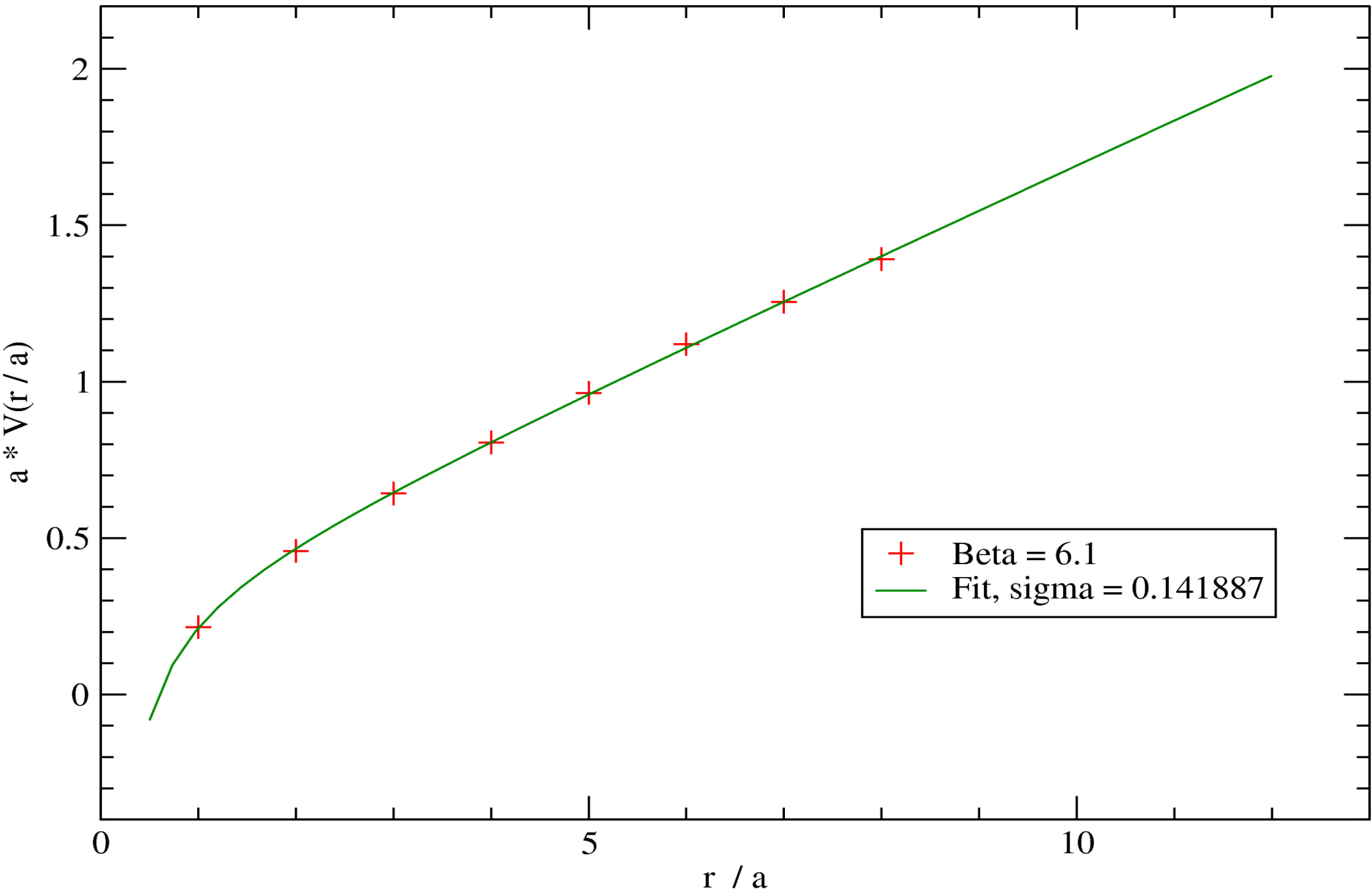
Teemu Rantalaiho,
University of Helsinki, Department of Physics
Helsinki 11.6.2012



UNIVERSITY OF HELSINKI

Static Quark-Antiquark potential

Quenched Approximation - no Tadpole Improvement



Agenda

- Introduction
 - What are histograms?
 - How do we define generalized histograms?
- Prior work and our implementation
 - Small histograms - 1-320 bins (normal)
 - Medium histograms – 320-2560 bins
 - Large histograms (Tested up to 131 072 bins)
- Performance and results
- Conclusions, future, what could we improve

Histograms

- First normal histogram:
- List of frequencies of occurrence of some sample in a set
- Pseudocode:

```
For each (bin index i) { bin[i] = 0; }
```

```
For each (input x)  
  { bin[x] = bin[x] + 1; }
```

Simple example

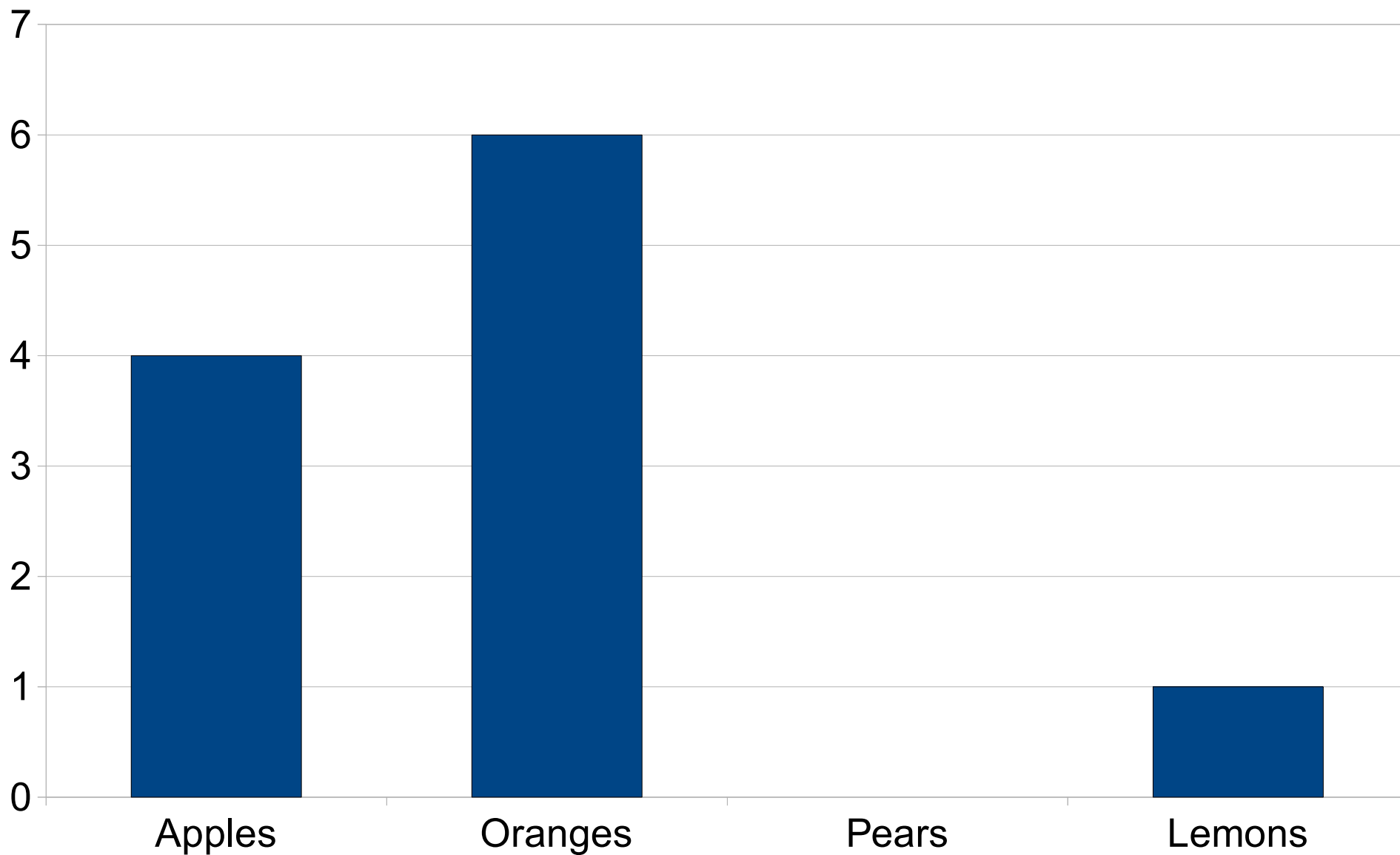
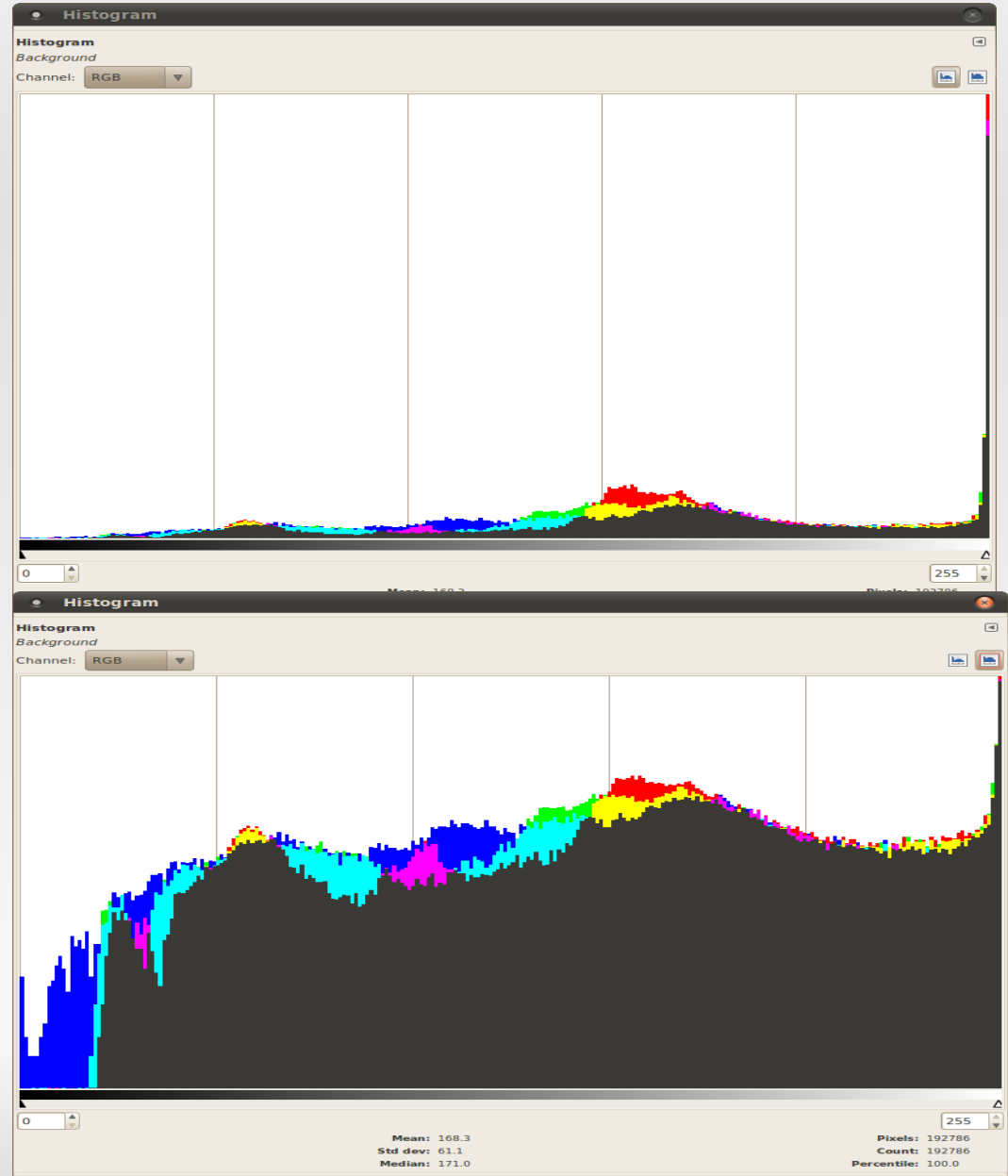


Image Processing Example



Generalized Histograms

- Sum over input key-value pairs, but restrict sum to equal keys
- "Order independent Reduce-by-key"
 - Obtainable from Thrust [<http://thrust.github.com>] for CUDA with "Sort-by-key" followed by "Reduce-by-key" (But performance is modest)
 - User defines input-transform and sum-functors
- Pseudocode:

```
For each (bin index i) { bin[i] = 0 }  
For each (input (k[j], x[j]))  
  { bin[k[j]] = bin[k[j]] + x[j] }
```
- Serial implementation completely trivial

Applications

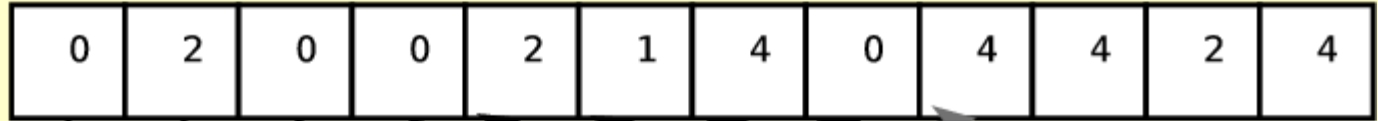
- Function evaluation
- Image processing (weighted histograms, semi-transparent pixels, feature extraction...)
- Cosmology (angle correlations of galaxies, CMB, ...)
- Molecular dynamics (radial distribution functions)
- Experimental particle physics
- Our use-case: correlation functions from lattice data

Implementation on Parallel Architectures

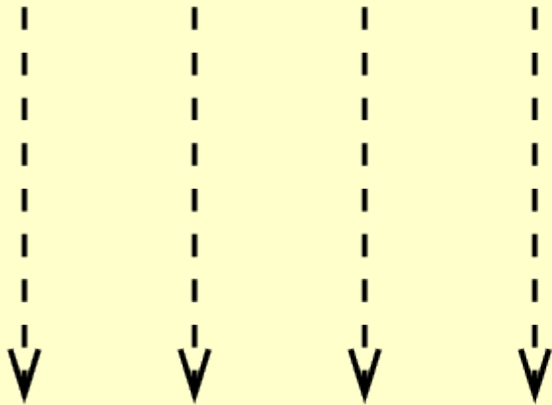
Parallel Implementation – First idea

- Every thread builds its own histogram
- Easy, fast, accumulate results at the end
- Uses a lot of local resources (shared memory)
- ~64 bins of integers for Fermi-level CUDA GPUs
- Extend to 256 bins using 8-bit bins (normal histograms) – accumulation into local memory
- Prior implementations used 16-bit bins
[Nugteren, van den Braak, Corporaal, Mesman]

Input keys



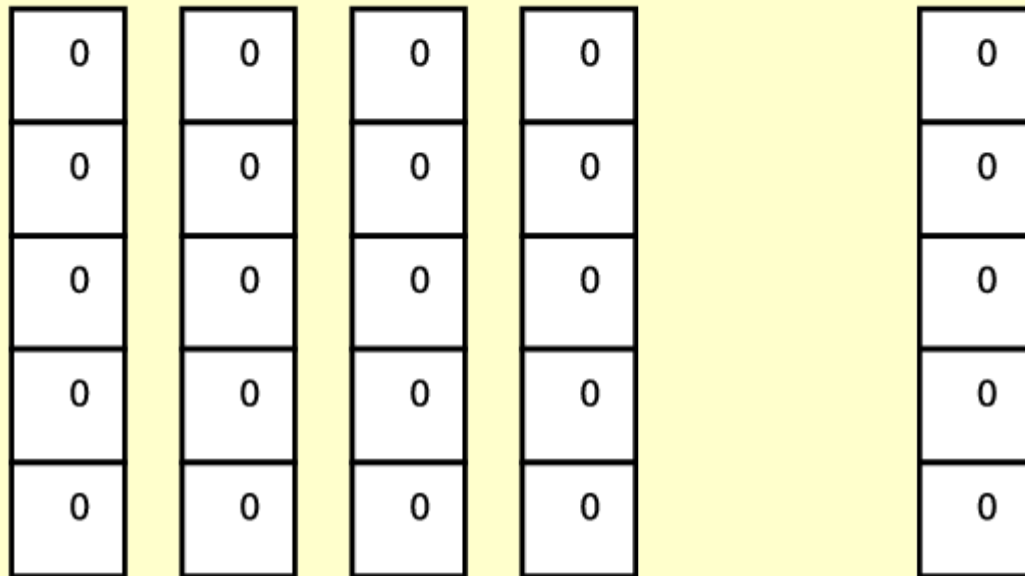
{t0} {t1} {t2} {t3}



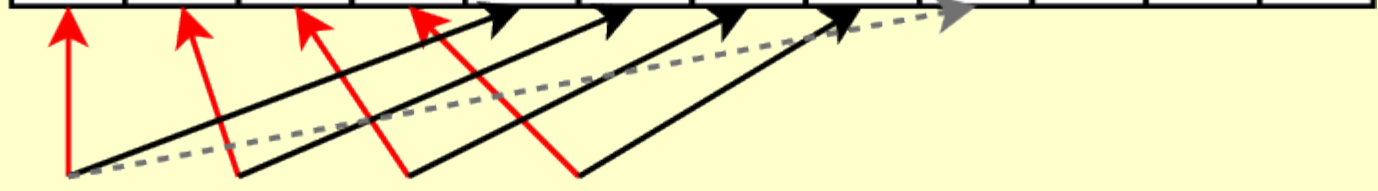
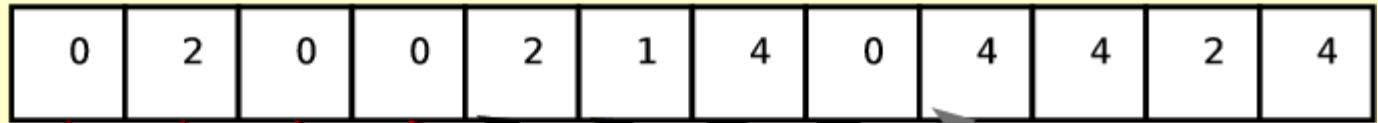
Threads

Final Result

Thread Histograms

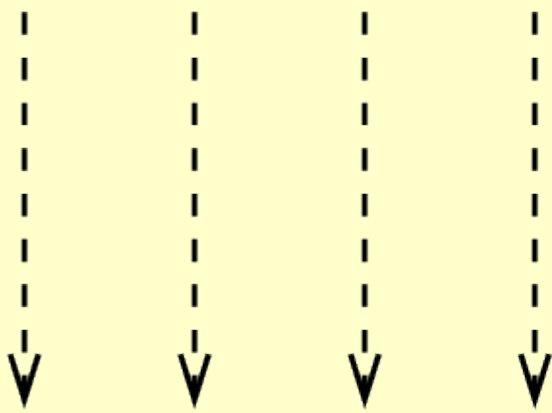


Input keys



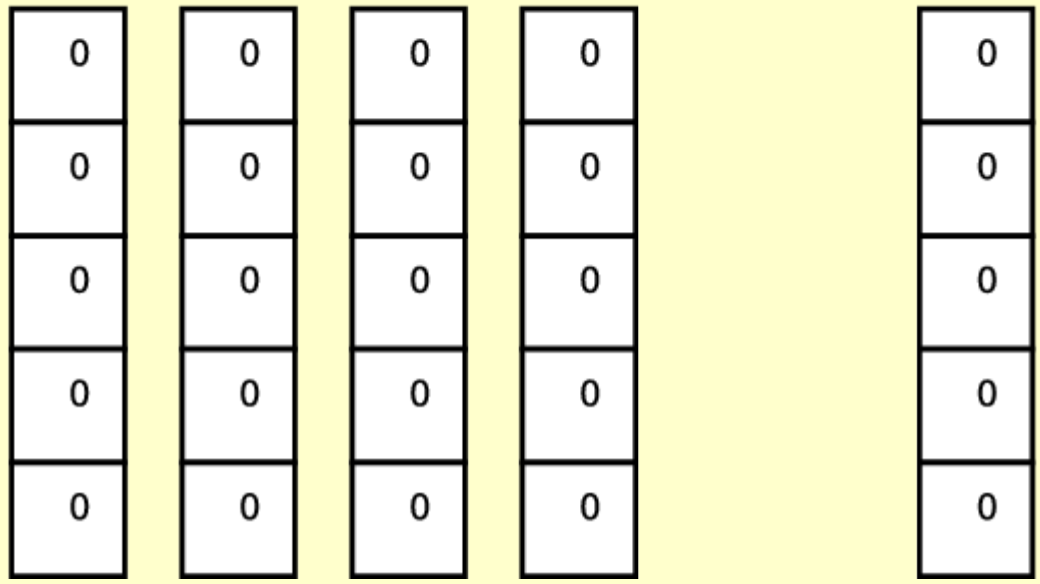
{t0} {t1} {t2} {t3}

Threads

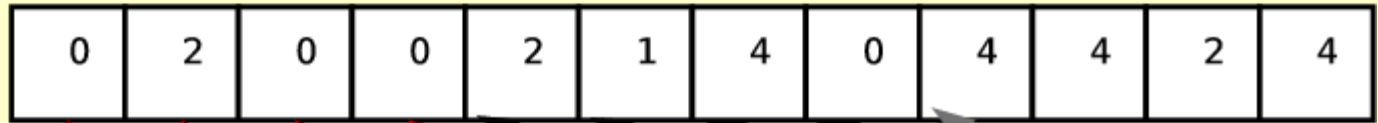


Final Result

Thread Histograms

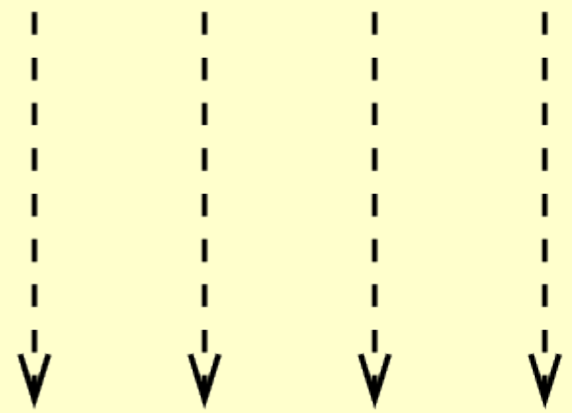


Input keys



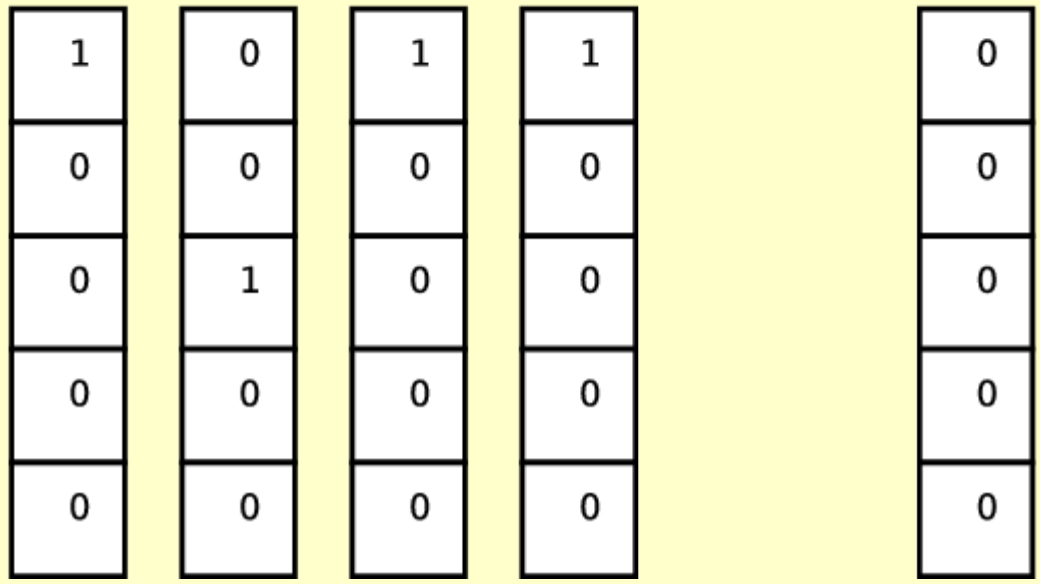
{t0} {t1} {t2} {t3}

Threads

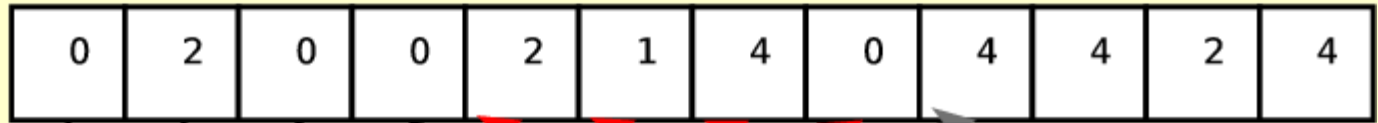


Final Result

Thread Histograms

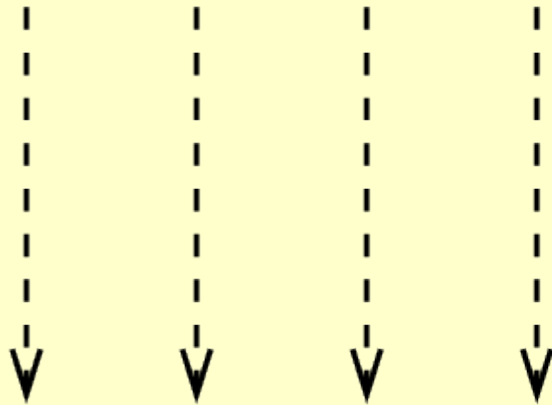


Input keys



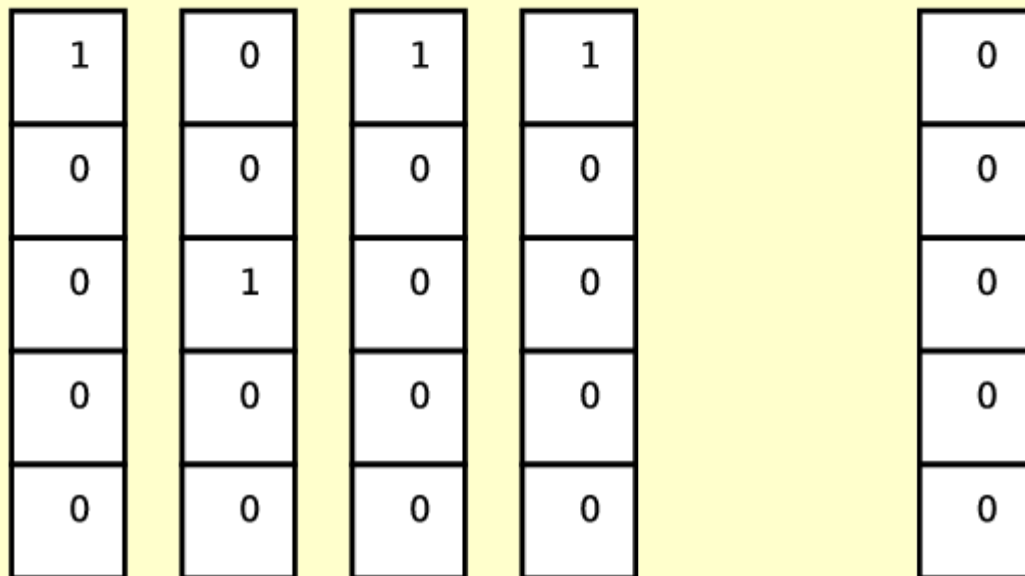
{t0} {t1} {t2} {t3}

Threads

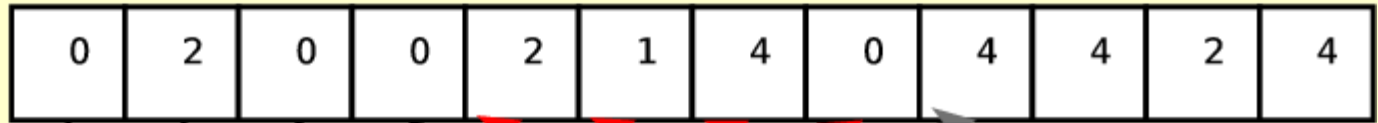


Final Result

Thread Histograms

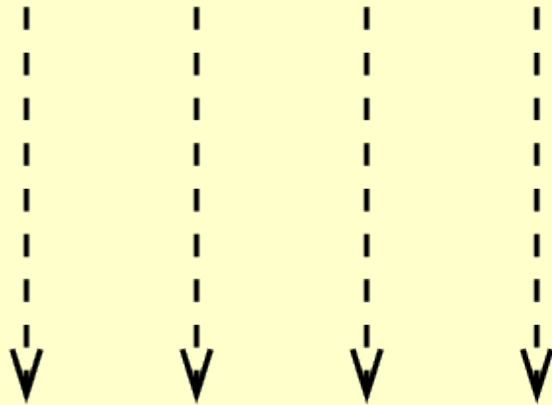


Input keys



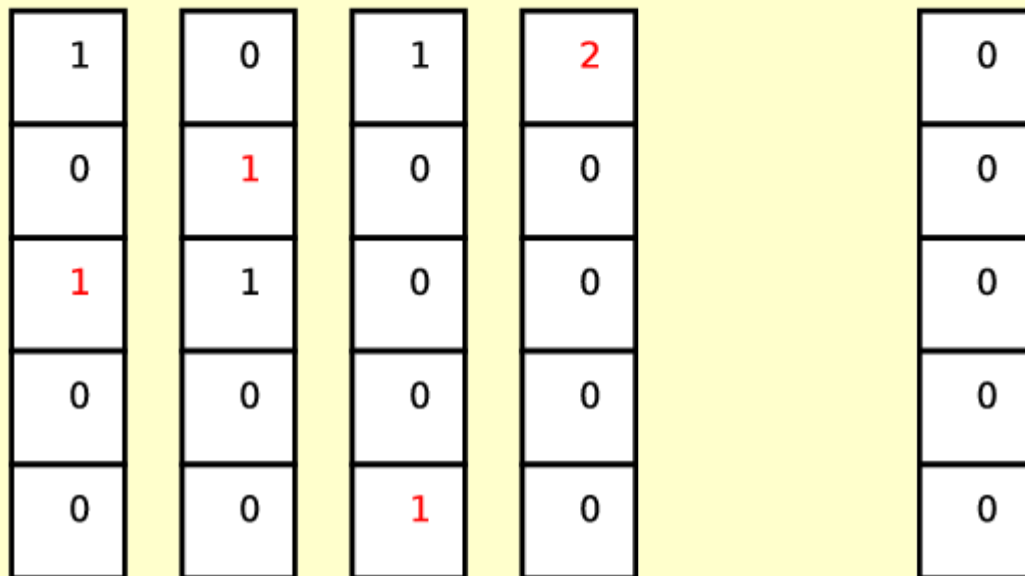
{t0} {t1} {t2} {t3}

Threads

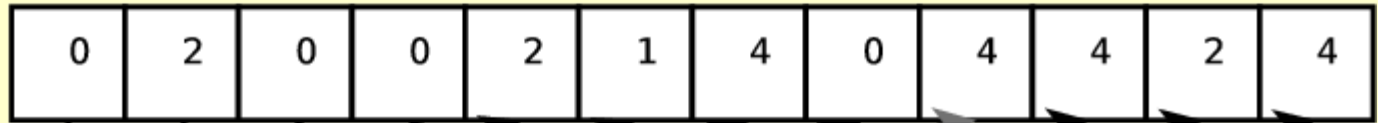


Final Result

Thread Histograms

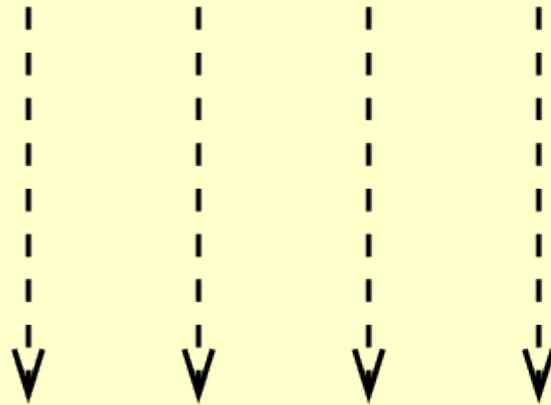


Input keys



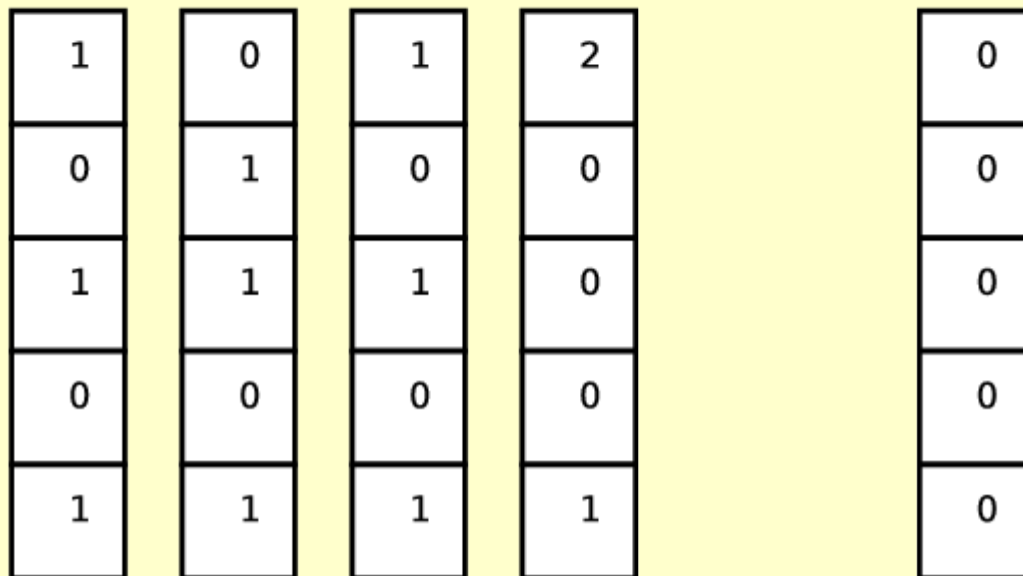
{t0} {t1} {t2} {t3}

Threads

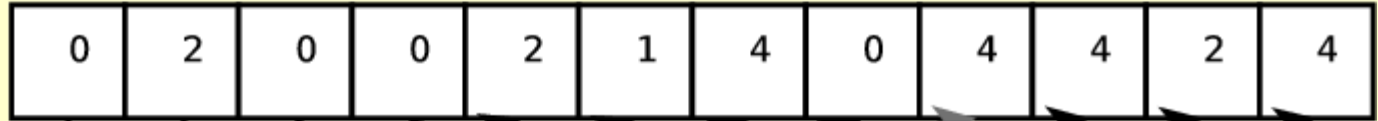


Final Result

Thread Histograms

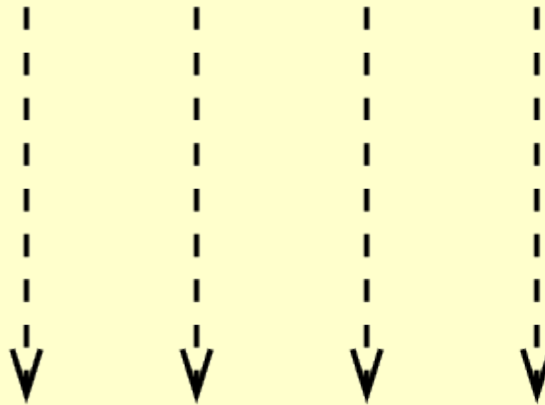


Input keys



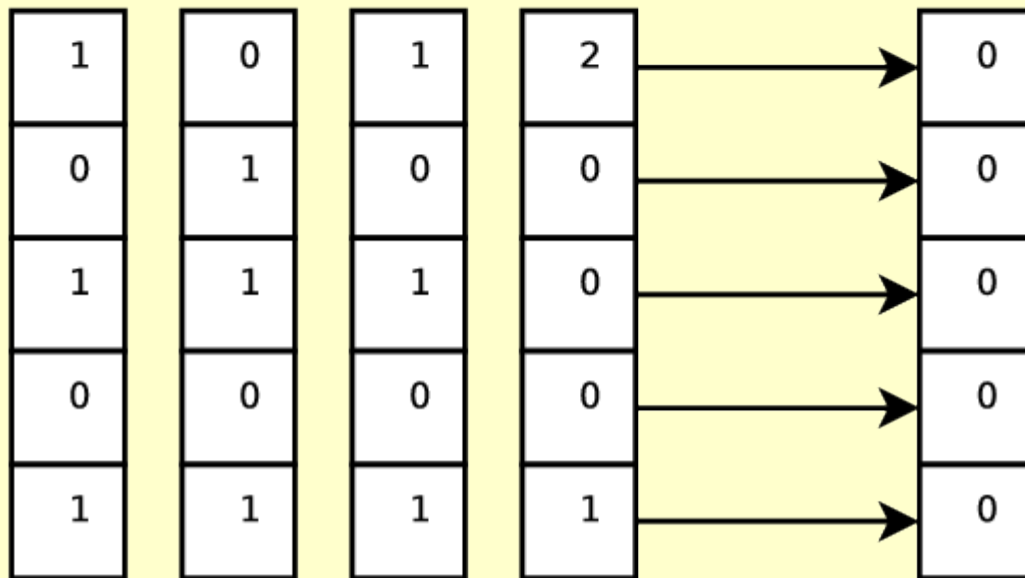
{t0} {t1} {t2} {t3}

Threads

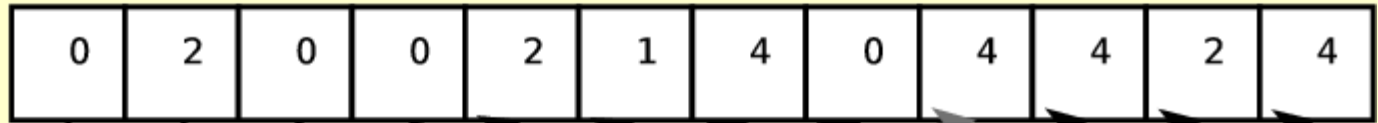


Final Result

Thread Histograms

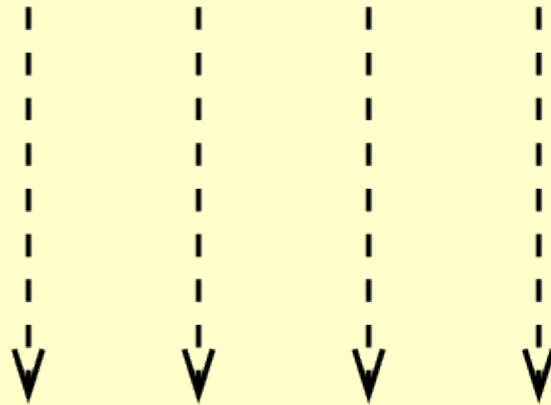


Input keys



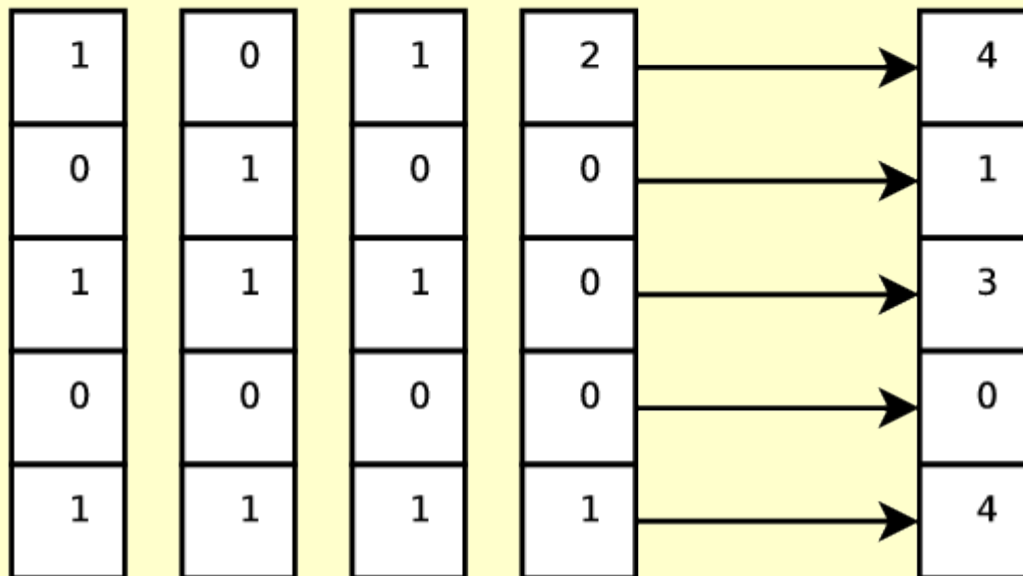
{t0} {t1} {t2} {t3}

Threads



Final Result

Thread Histograms



Medium sized Histograms

- Run out of shared memory - what can you do?
- Serialize access to shared resources
- Previous work [Podlozhnyuk - NVIDIA]:
 - One binset (histogram) per Warp (32 threads)
 - Intrawarp Collisions → Serialization
 - Ok performance for spread out keys – bad when degenerate
- We share one binset across all warps
- Degenerate key distributions open problem until now ([Shams, Kennedy] also suffer)

Partial Fix for Key Degeneracy

- Watch input keys as they come in
- Once high amount of degeneracy found (~16 degenerate keys between 32 threads)
 - Do warp-local reduction of values in shared memory
 - One thread (per unique key) collects the sum
 - Write result, free of collisions
- Expensive for well distributed keys → only apply when high degeneracy detected

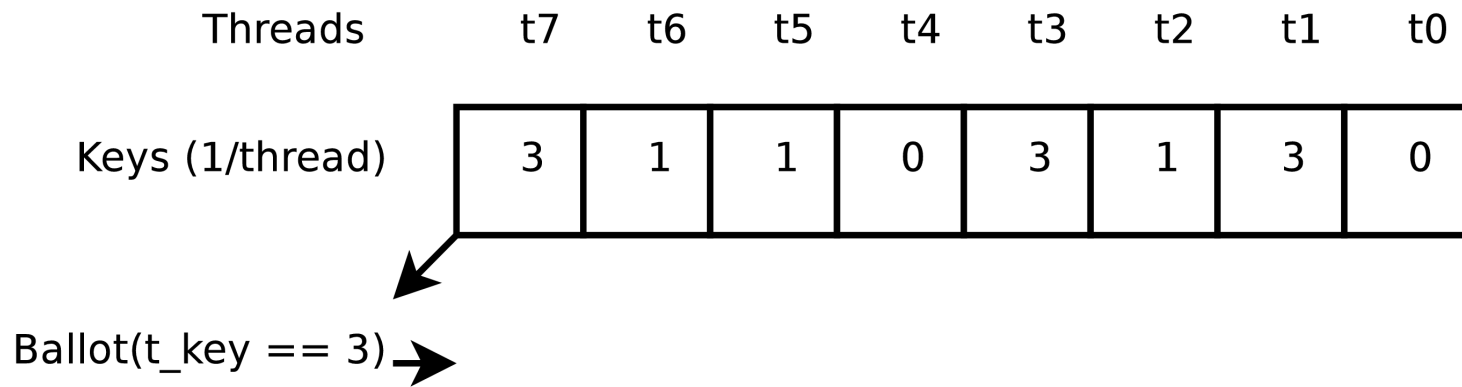
Warp Reduction

- Resolve all threads (in the same warp) that have the same key
- Do a parallel reduction between those threads
 - Run reductions of different keys in parallel
 - Take advantage of warp-vote functions
- Finding group-masks is expensive
 - Use ballot for every unique key and collect bitmask
- Normal histogram: Result is population count of bitmask

Group-ID Example


Threads	t7	t6	t5	t4	t3	t2	t1	t0
Keys (1/thread)	3	1	1	0	3	1	3	0

Group-ID Example

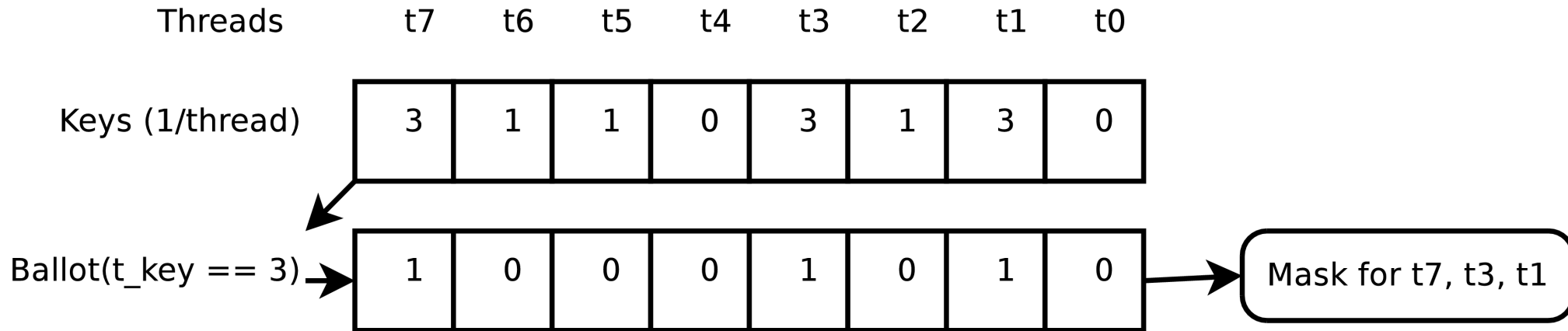


Group-ID Example

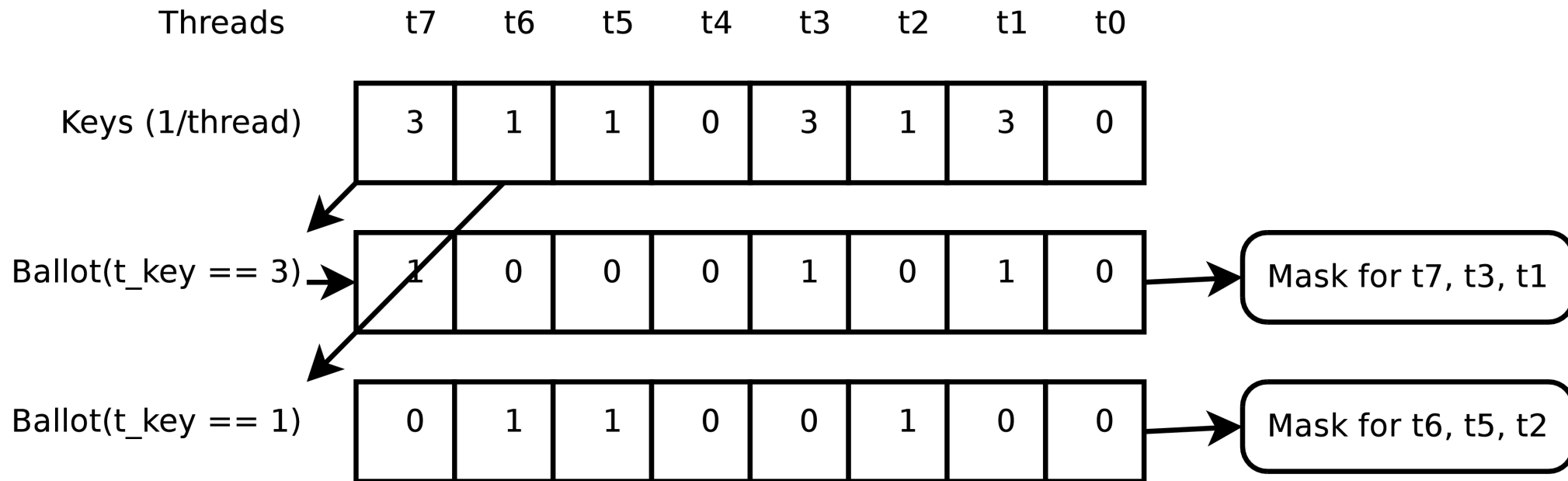
Threads	t7	t6	t5	t4	t3	t2	t1	t0
Keys (1/thread)	3	1	1	0	3	1	3	0
Ballot($t_key == 3$)	1	0	0	0	1	0	1	0



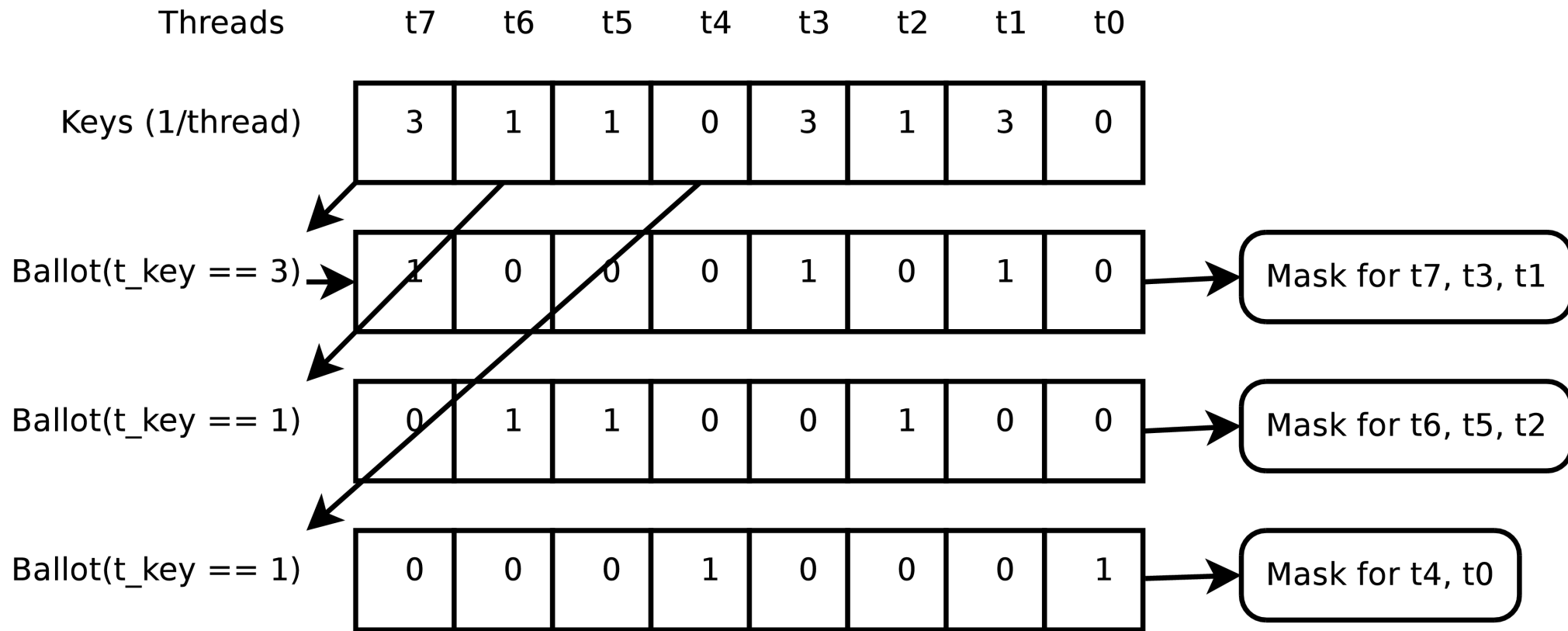
Group-ID Example



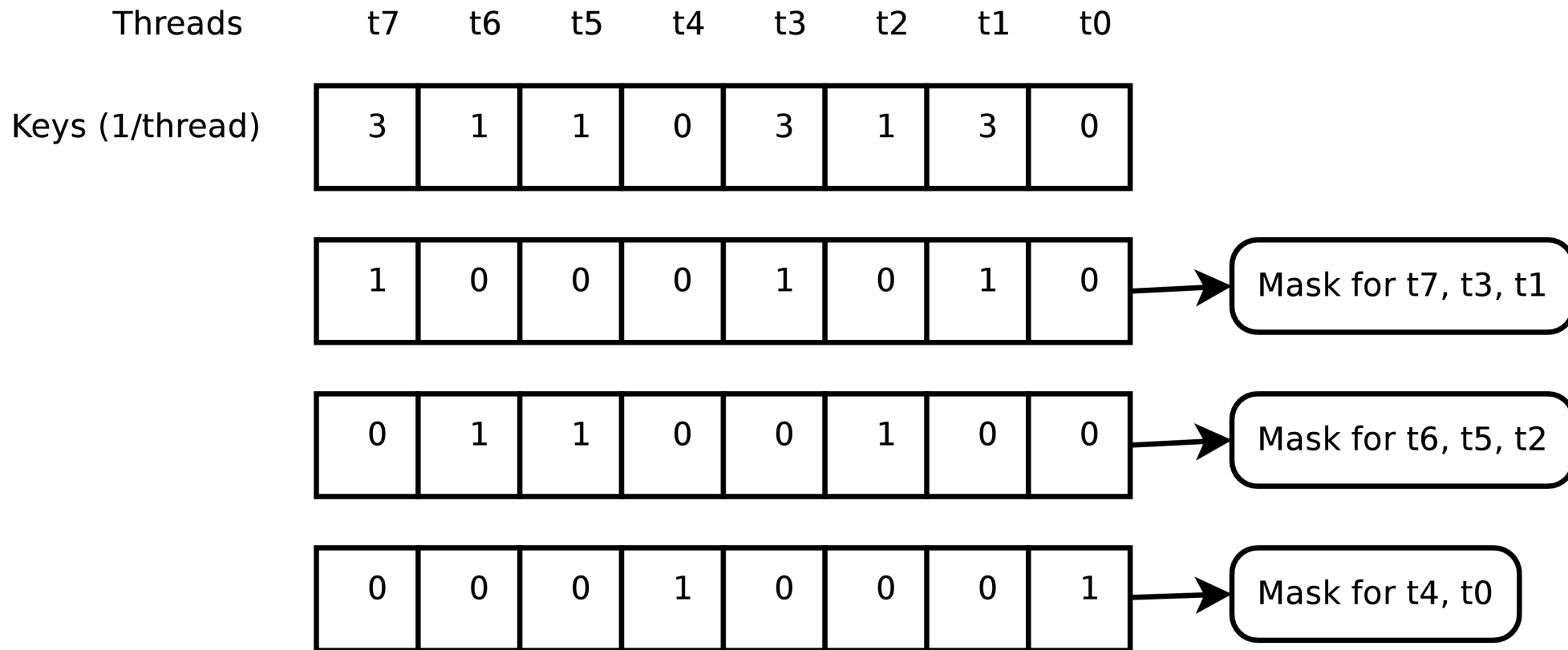
Group-ID Example



Group-ID Example



Group-ID Example



Large Histograms

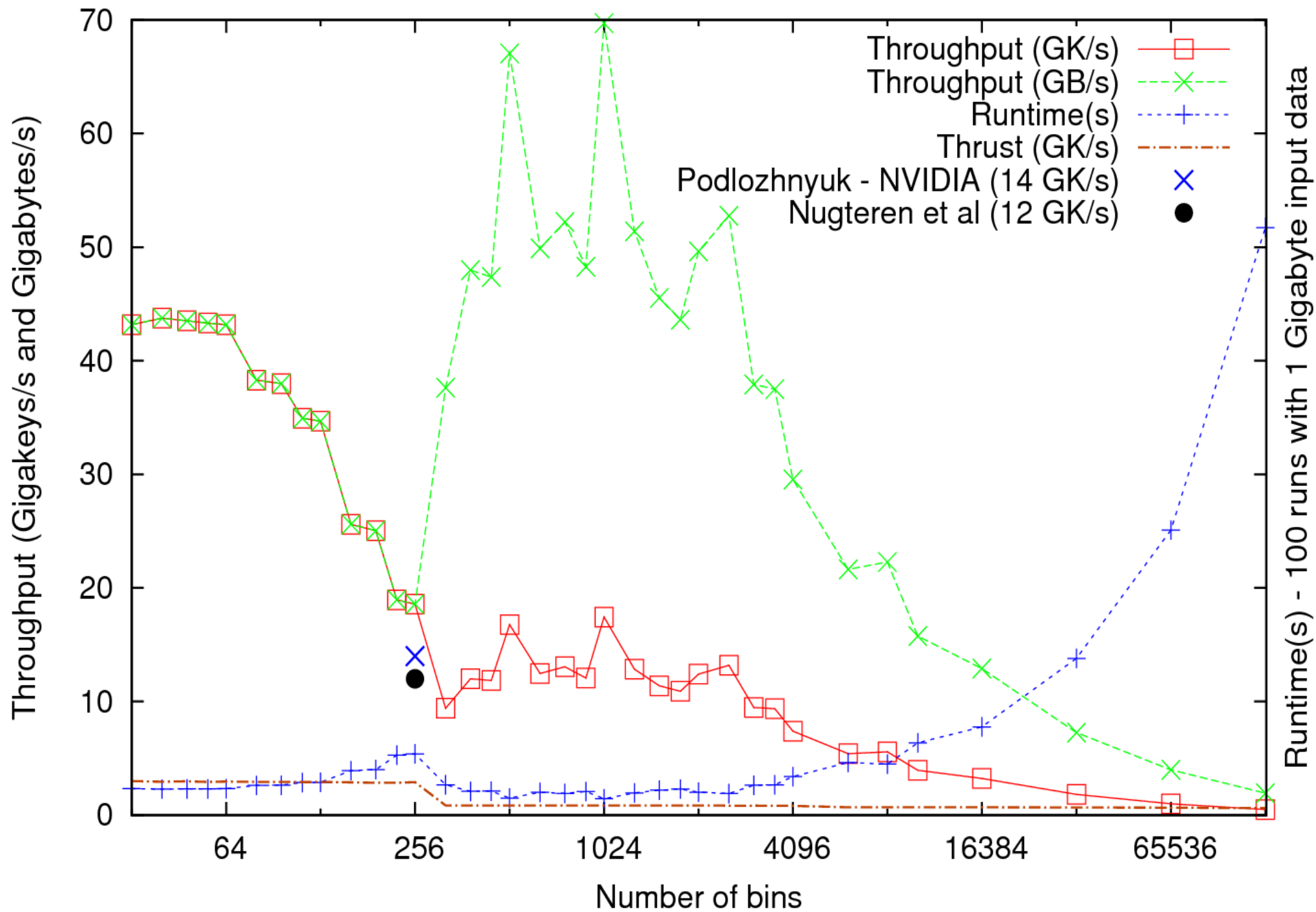
- At around 2500 bins we run low on shared memory
- First solution: Multiple passes of the medium histogram algorithm
 - Improved occupancy and cache help
- With very large histograms (~100 000 bins) too many passes
 - Resort to global memory atomics
 - For generalized histograms use a per-warp hashtable in shared memory
- Adaptive warp-reduction for degenerate keys

Large Histograms

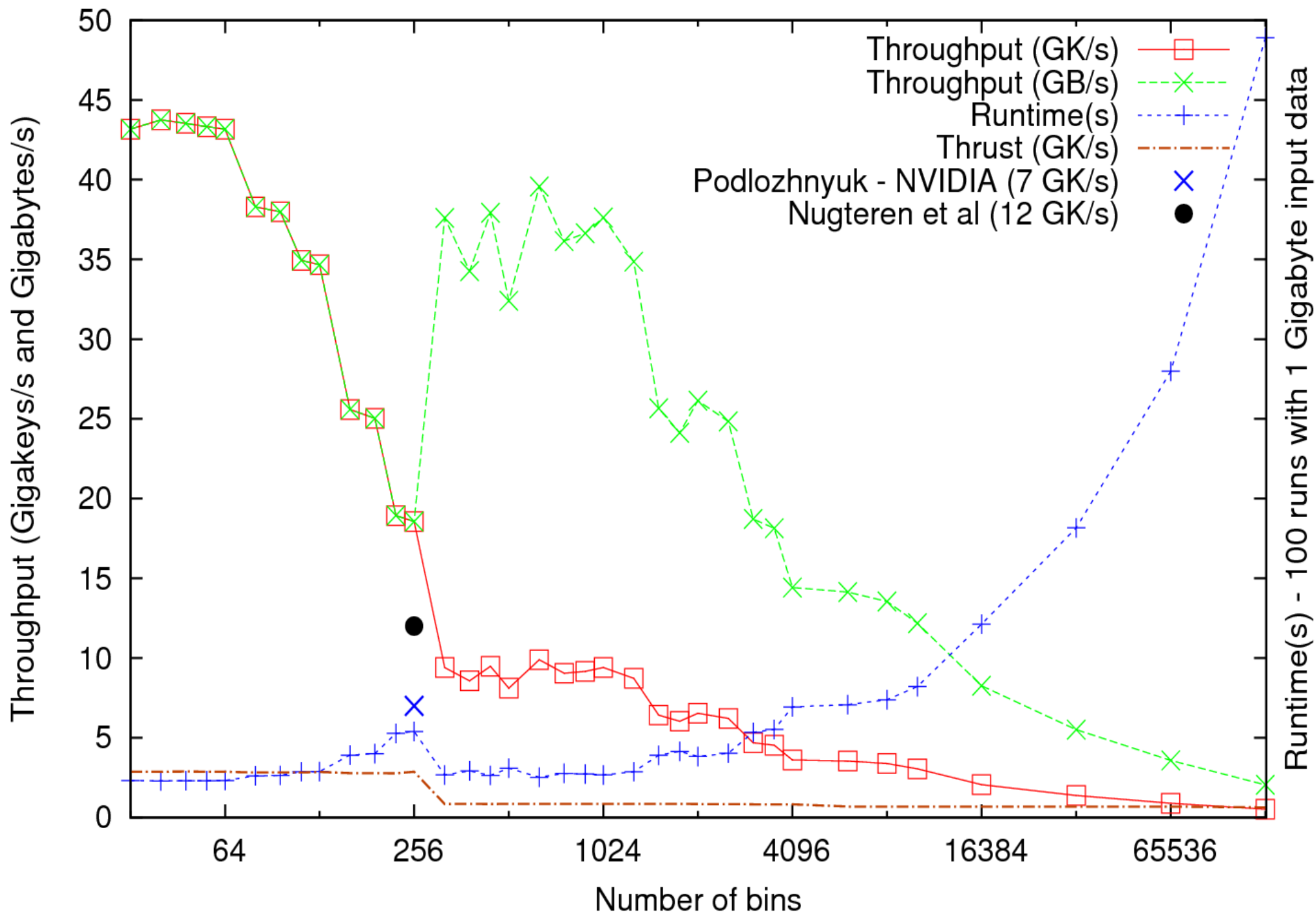
- Performance drops to about same level as thrust at around 100 000 bins
 - CPU should be ~2x faster here
 - Even as is, could be useful to use GPU
 - No data transfers across pci Express BUS
 - Complex key-value resolving code can amortize slow histogram code
- Global atomics will be faster in Kepler
 - Coupled with warp-reduction, could be very competitive
- Multiwarp hashtable-based algorithm TBD

Performance
With Tesla M2070 (ECC on)
(Except some references)

Histogram Performance for uniform random keys
Tested on Tesla M2070 (ECC enabled) - CUDA 4.0.17



Histogram Performance for texture-data keys
Tested on Tesla M2070 (ECC enabled) - CUDA 4.0.17



Histogram Performance for nearly degenerate keys
Tested on Tesla M2070 (ECC enabled) - CUDA 4.0.17

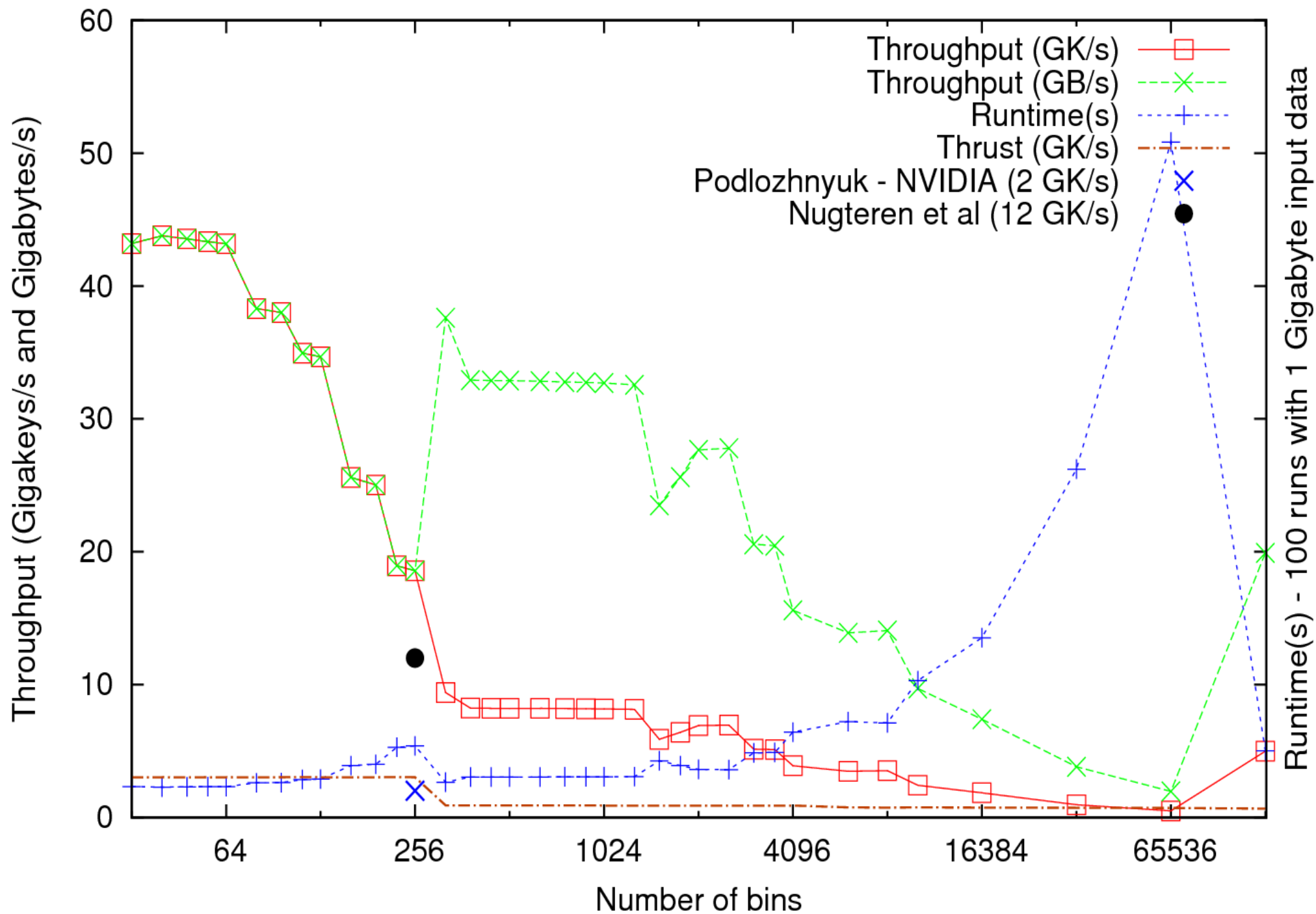


Image Histograms

- Comparisons against NVIDIA's NPP library [<http://developer.nvidia.com/npp>]
- Our one channel (monochrome) 8-bit histogram with 256 bins reaches 18GK/s, NPP gets 8GK/s → 125% improvement
- Our three-channel histogram with 256 bins per channel reaches 12GK/s, NPP reaches 6GK/s → 2x improvement

Image sizes – 8bit one channel



Image size	NPP	Our method	Improvement
1024x1024	3.56 GK/s	5.7 GK/s	60%
2048x2048	6.64 GK/s	11.76 GK/s	77%
4096x4096	7.6 GK/s	16.3 GK/s	114%
8192x8192	7.99 GK/s	17.97 GK/s	125%

Generalized histogram performance

- Test-case: Sum over every row in a fp32 matrix
 - Thrust: Reduce-by-key solution ~ 1.8 GK/s
 - Thrust: Normal reduce ~ 20 GK/s (80 GB/s)
 - Our algorithm 2-7x faster than "thrust::reduce_by_key()" in this case

Matrix-size	Histogram Time	Gigakeys/s
50 x 1 000 000	4.73 ms	10.58
500 x 100 000	4.06 ms	12.33
5000 x 10 000	12.13 ms	4.12

API Example

```
#include "cuda_histogram.h"

// minimal test - 1 key per input index
struct test_xform {
    __host__ __device__
    void operator()(int* in, int i, int* keys, int* res, int nres) const {
        *keys++ = in[i];
        *res++ = 1;
    } };

struct test_sumfun {
    __device__ __host__ int operator() (int res1, int res2) const {
        return res1 + res2;
    } };

// Host function - d_data contains keys, out will have result
int main (int* d_data, int nData, int* out) {
    test_xform xform; test_sumfun sum;
    callHistogramKernel<histogram_atomic_inc, 1>
        (d_data, xform, sum, 0, nData, 0, out, 256);
}
```

Features

- CudaStreams (Optional)
- Output to GPU or CPU memory
- Multiple keys / input
- Multidimensional input ranges
- Arbitrary transform / binary operators through function objects
- Accumulates on top of previous result
- User-supplied temporary arrays (Optional)

Conclusions, future work,...

- Generalized histogram implementation for CUDA GPUs
 - Very generic
 - Outperforms existing algorithms in all cases
 - Available under Apache V.2 License
[<https://github.com/trantalaiho/Cuda-Histogram>]
- Optimize for Kepler (need HW first)
- Needs marketing
- Further testing, "skip-key"-support, control over loop-unrolling...

Thank you

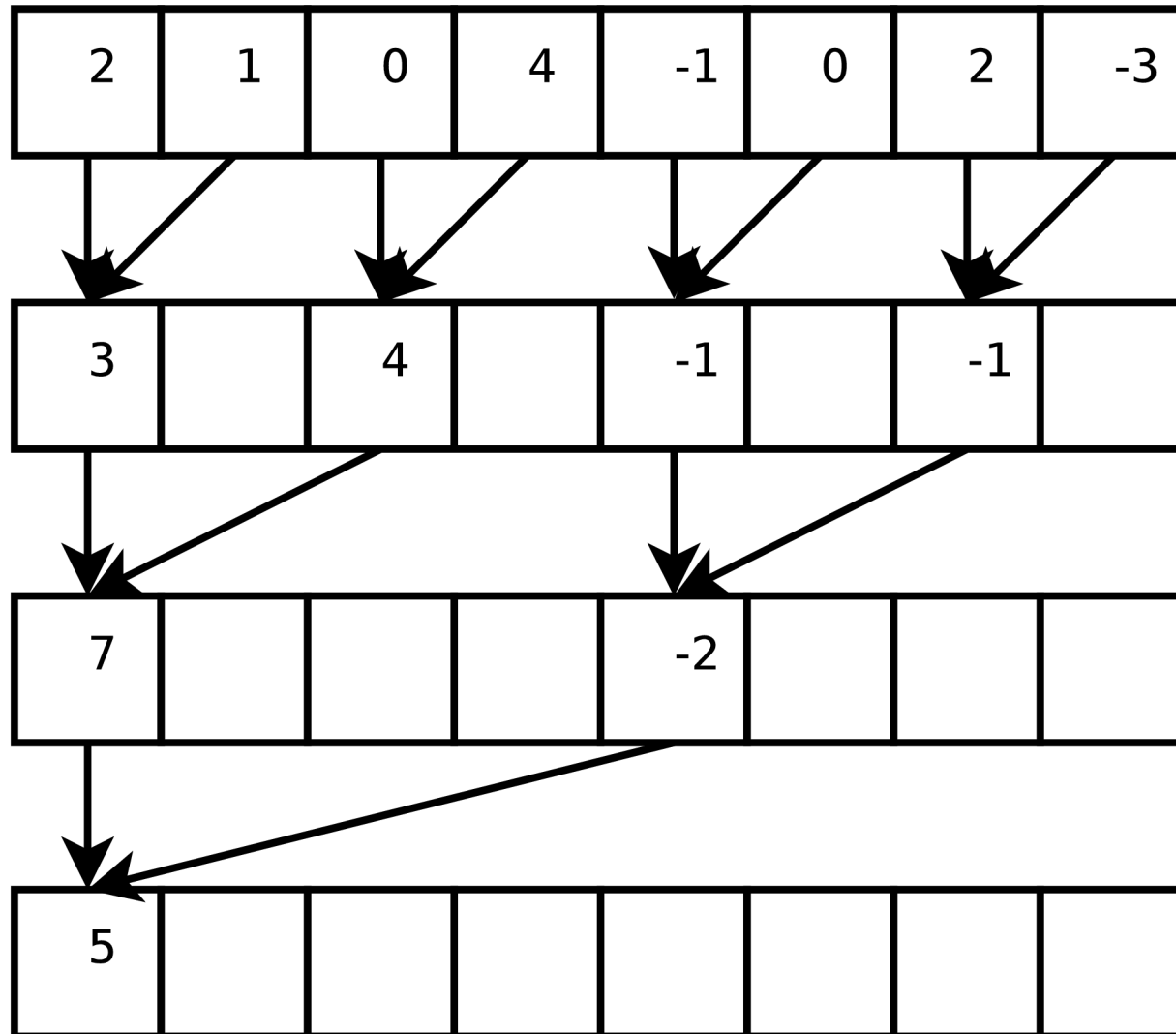
- Ask me questions
 - Right now
 - From teemu.rantalaiho@helsinki.fi
 - Grab me by the elbow later

Extra slides – Warp Reduction details

Warp reduction by key

- Inputs for each thread:
 - A key-value pair
 - Bitmask which tells which threads belong to our group
- Idea: As a normal parallel reduction
 - Using bitmask, give each thread order ID in group
 - Even threads within group sum values from odd
 - Shrink group by dividing by two
 - Odd threads write their value to shared memory
 - Repeat until whole group consumed

Normal parallel reduce



Warp Key-Reduce Example

Inputs

t3	t2	t1	t0
x3	x2	x1	x0

Thread 1

1	0	1	1
---	---	----------	---

Thread 2

0	1	0	0
---	---	---	---

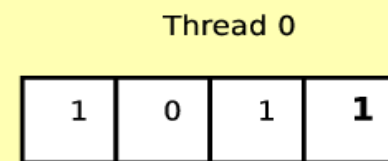
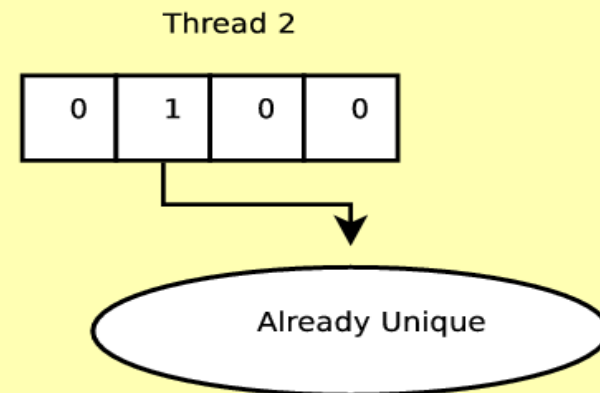
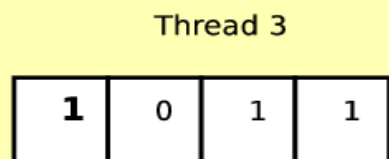
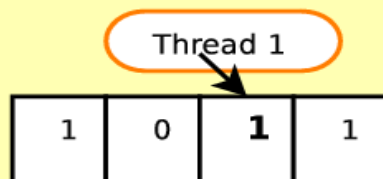
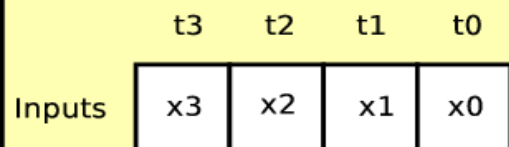
Thread 3

1	0	1	1
----------	---	---	---

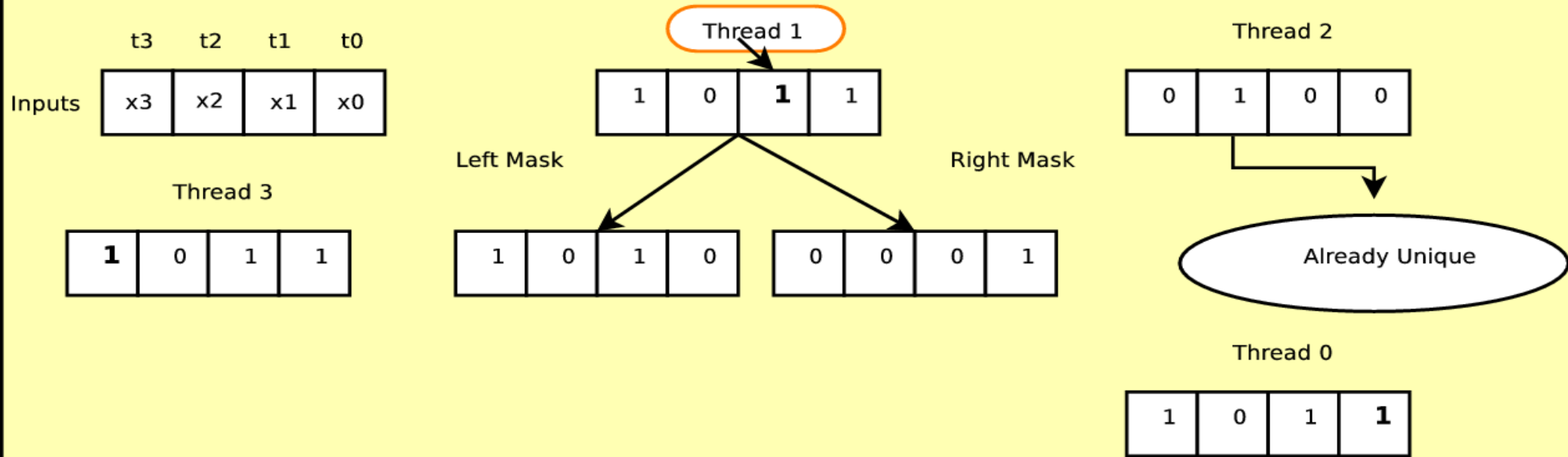
Thread 0

1	0	1	1
---	---	---	----------

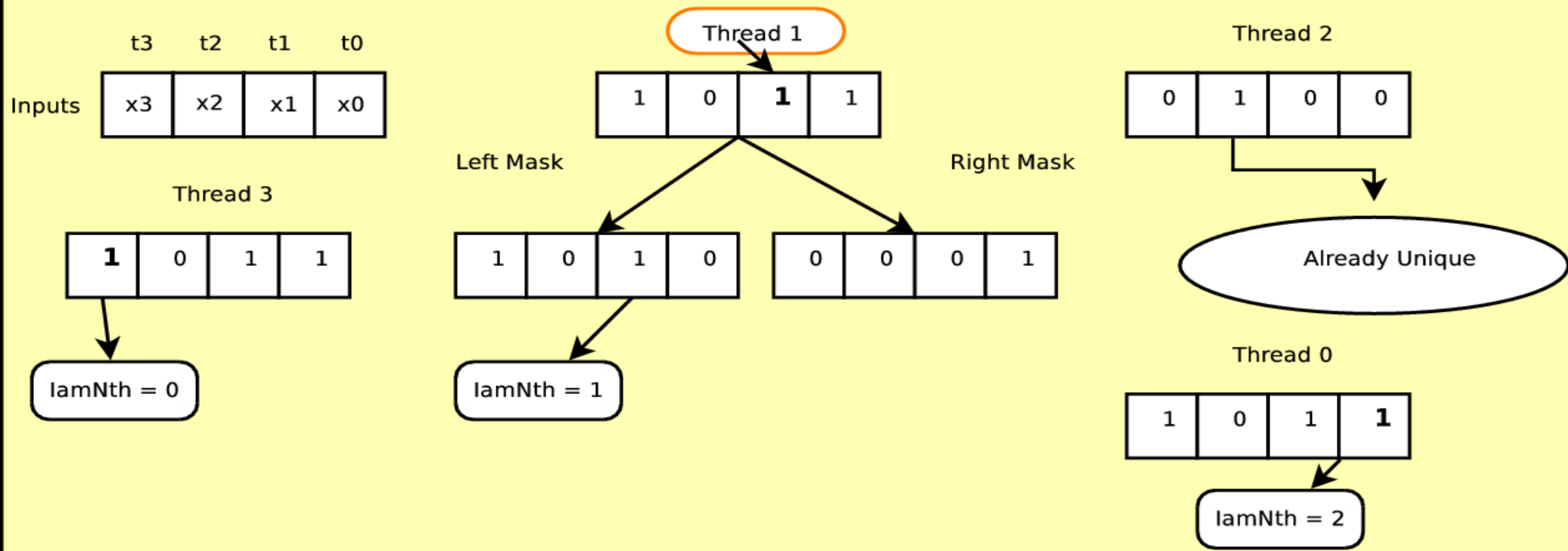
Warp Key-Reduce Example



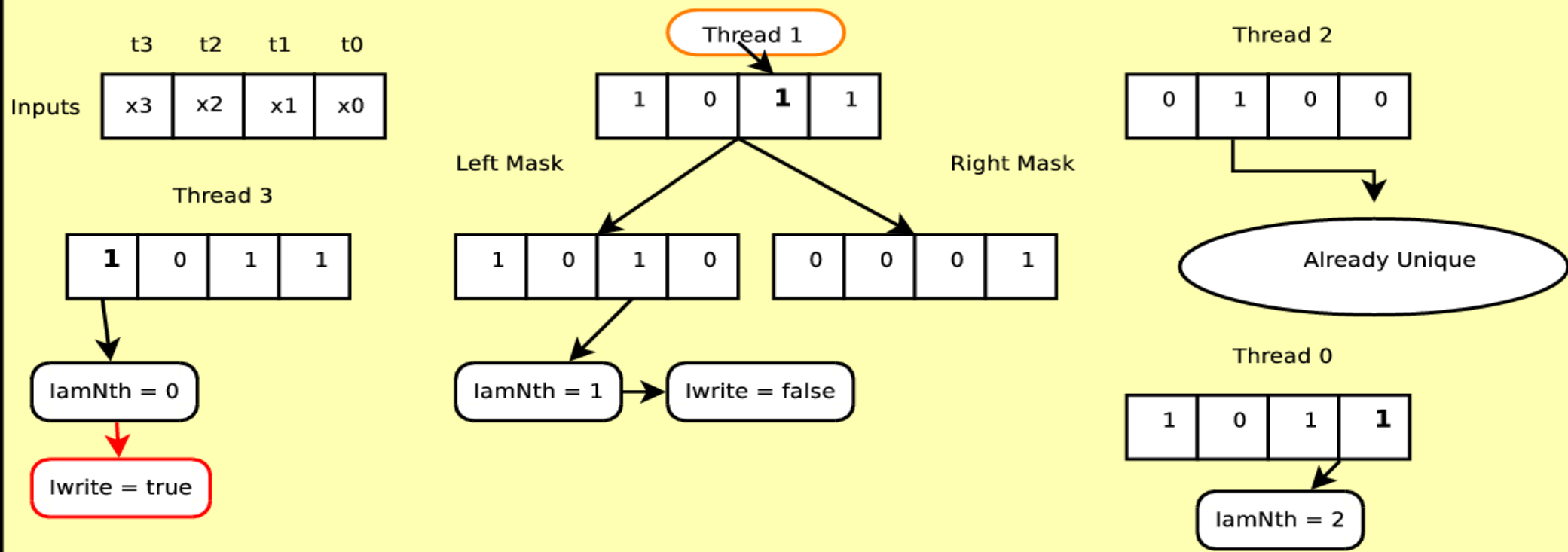
Warp Key-Reduce Example



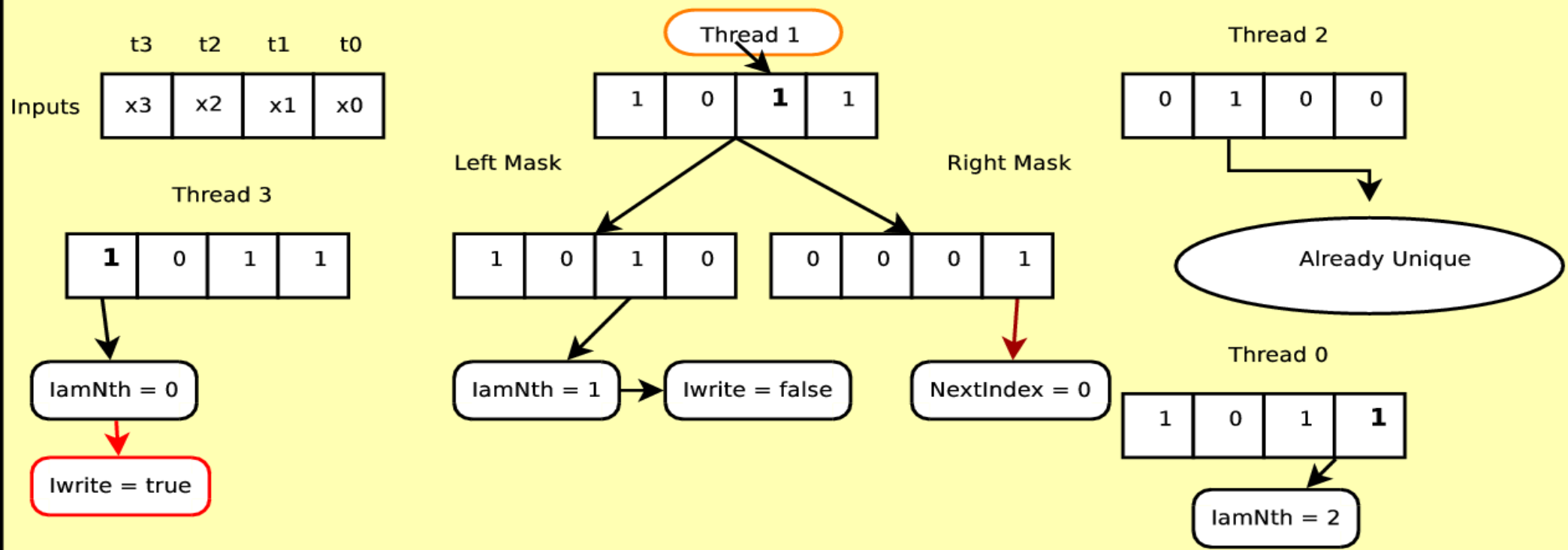
Warp Key-Reduce Example



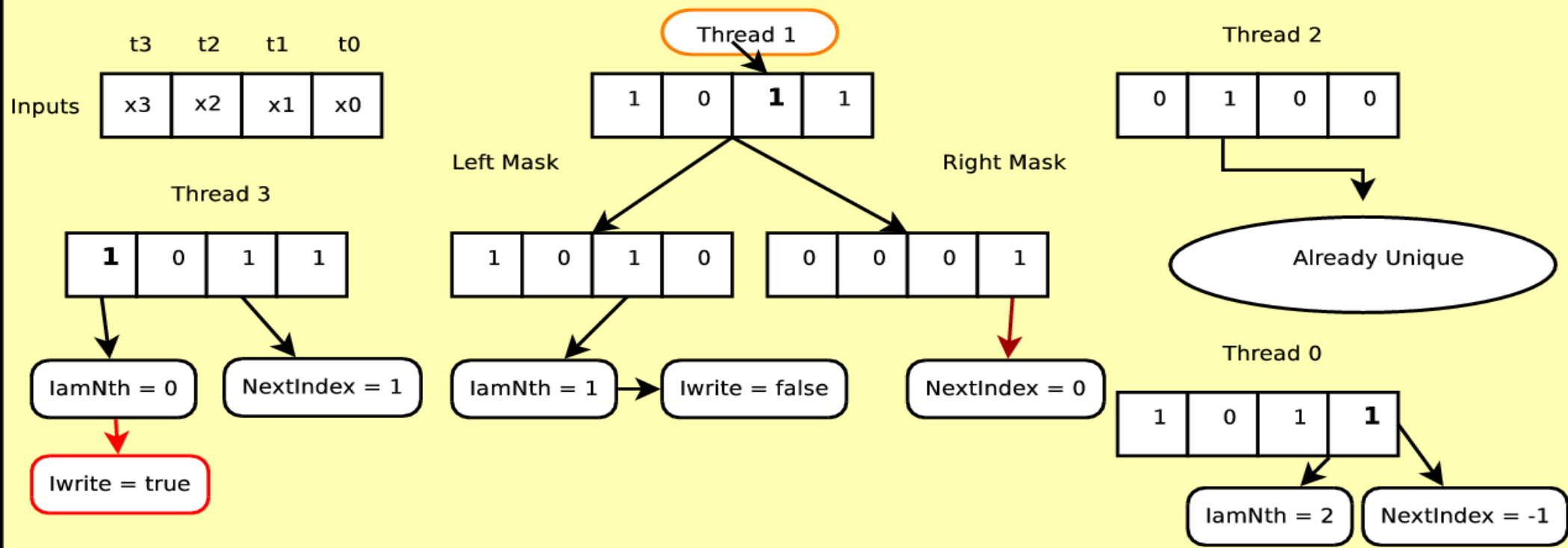
Warp Key-Reduce Example



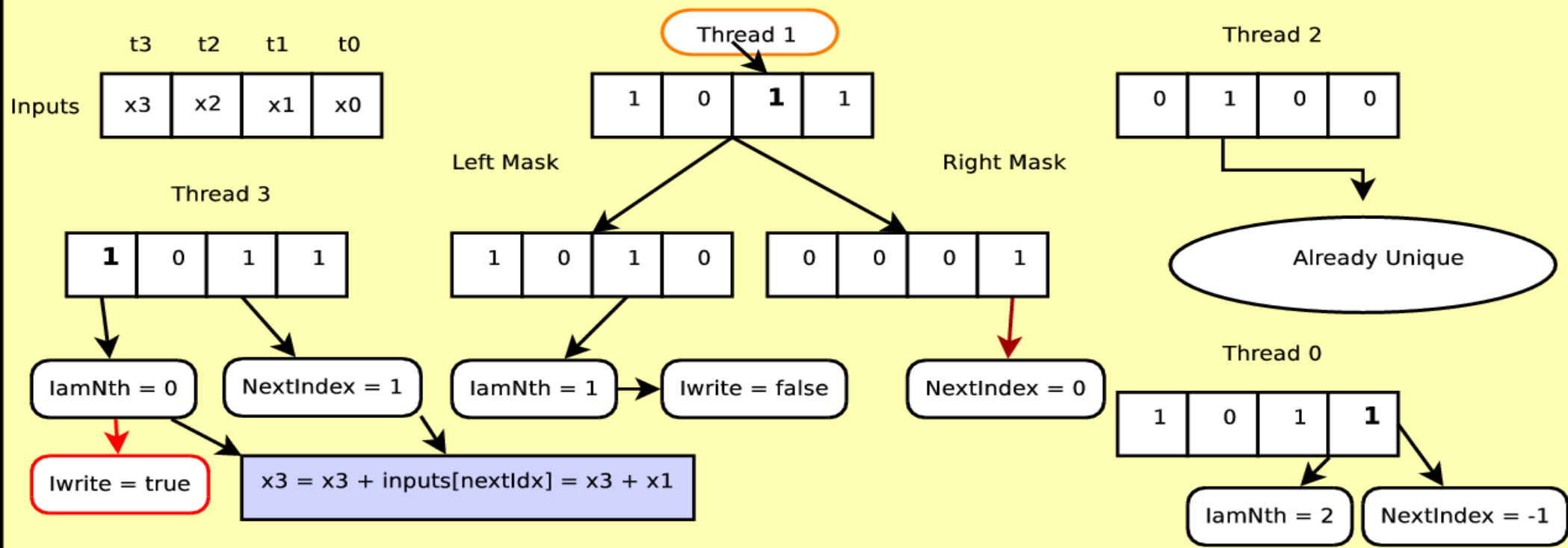
Warp Key-Reduce Example



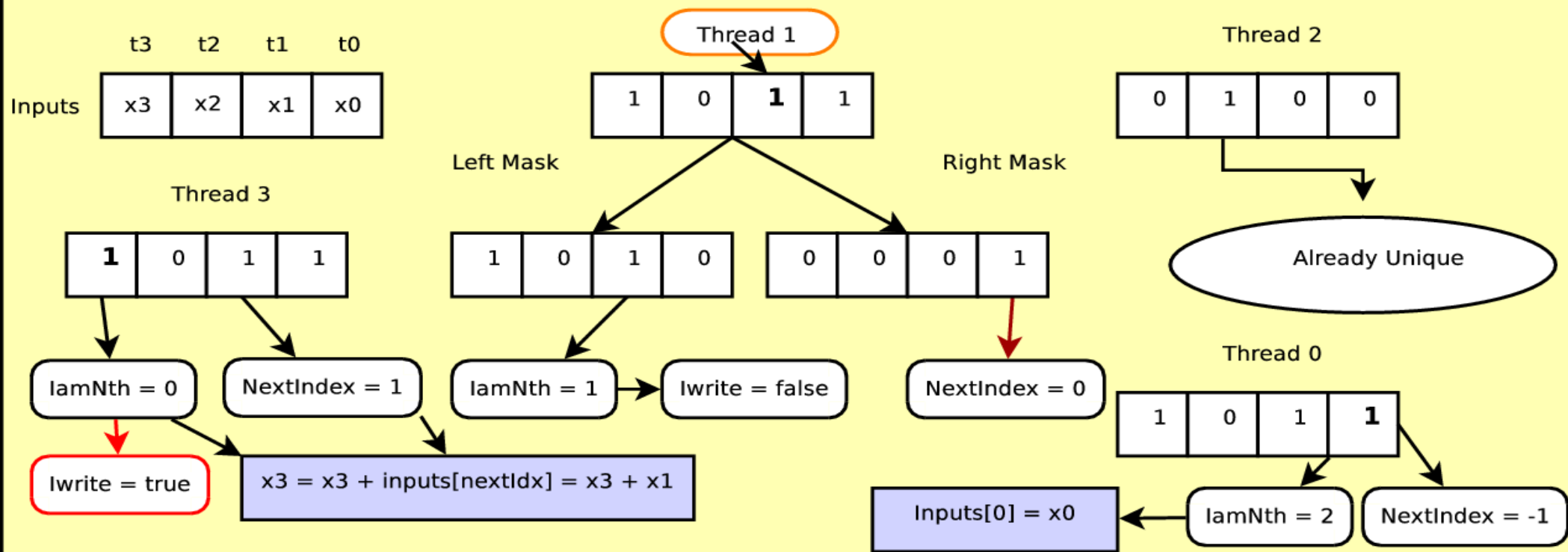
Warp Key-Reduce Example



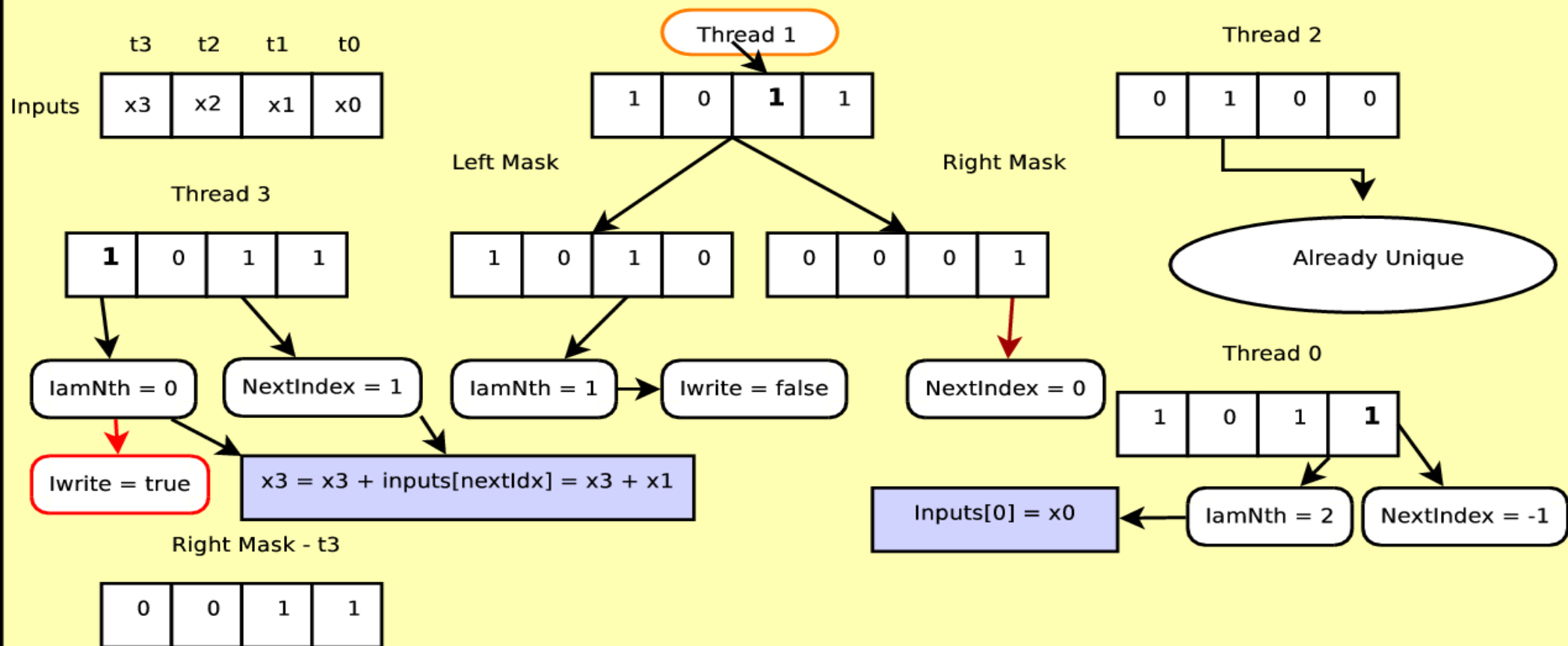
Warp Key-Reduce Example



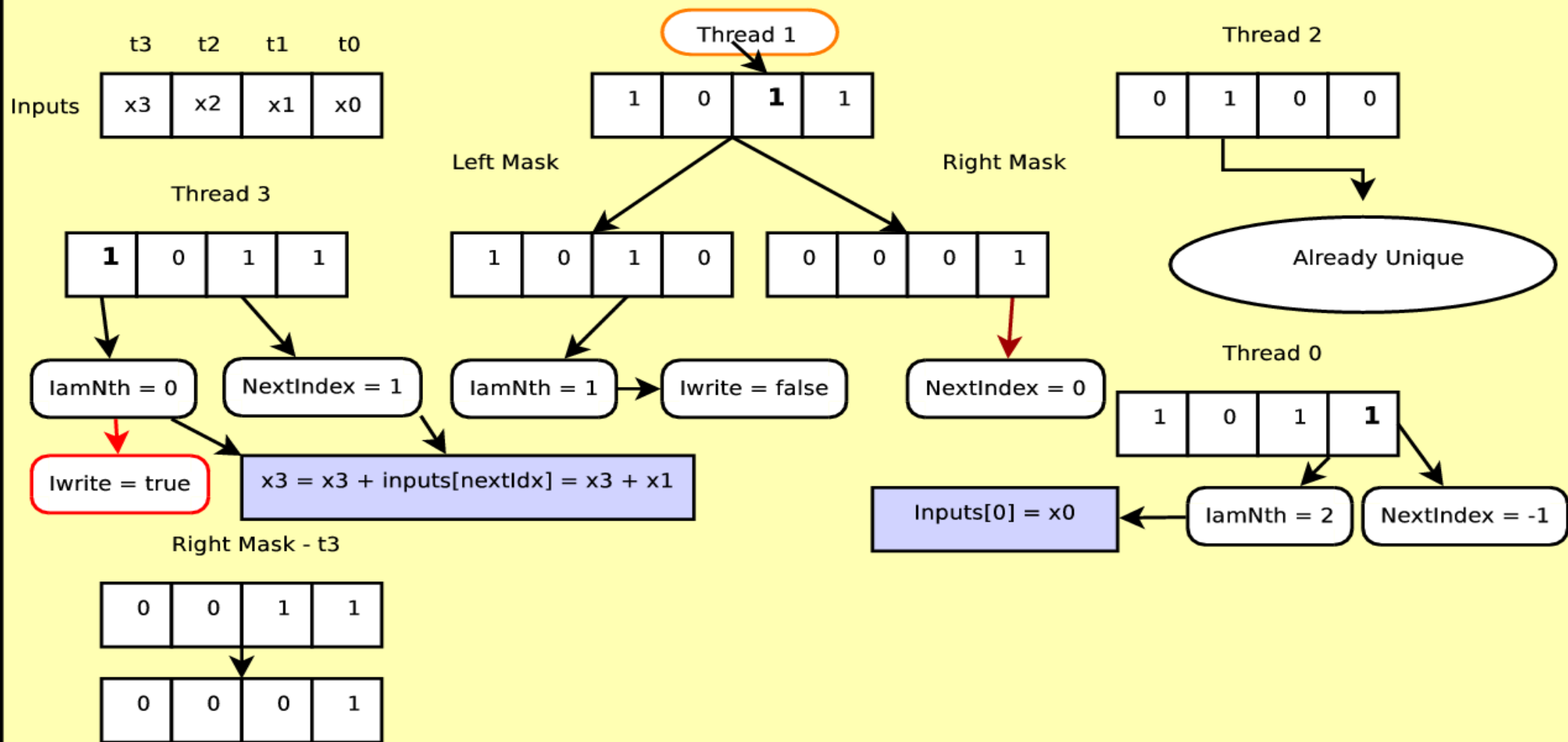
Warp Key-Reduce Example



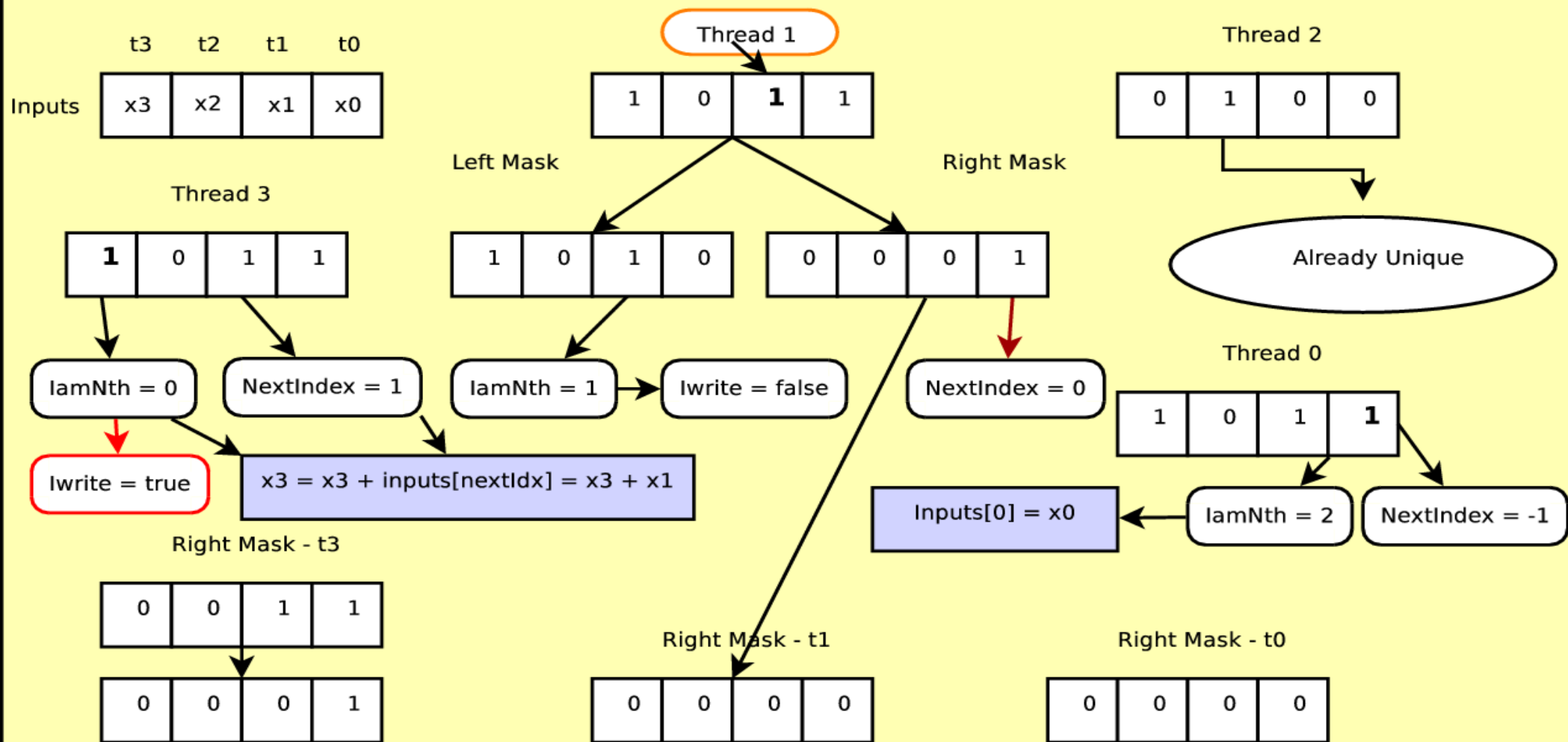
Warp Key-Reduce Example



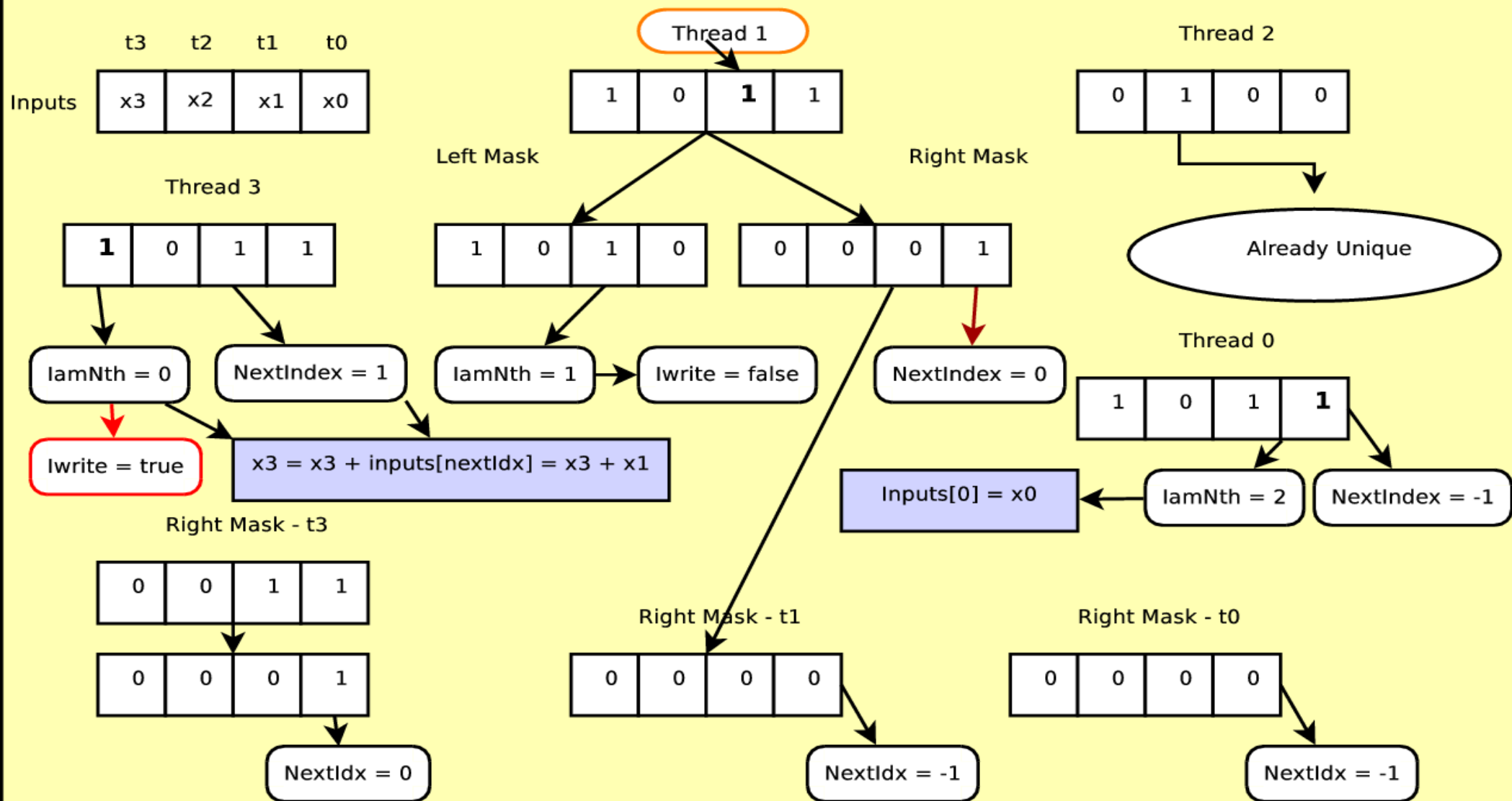
Warp Key-Reduce Example



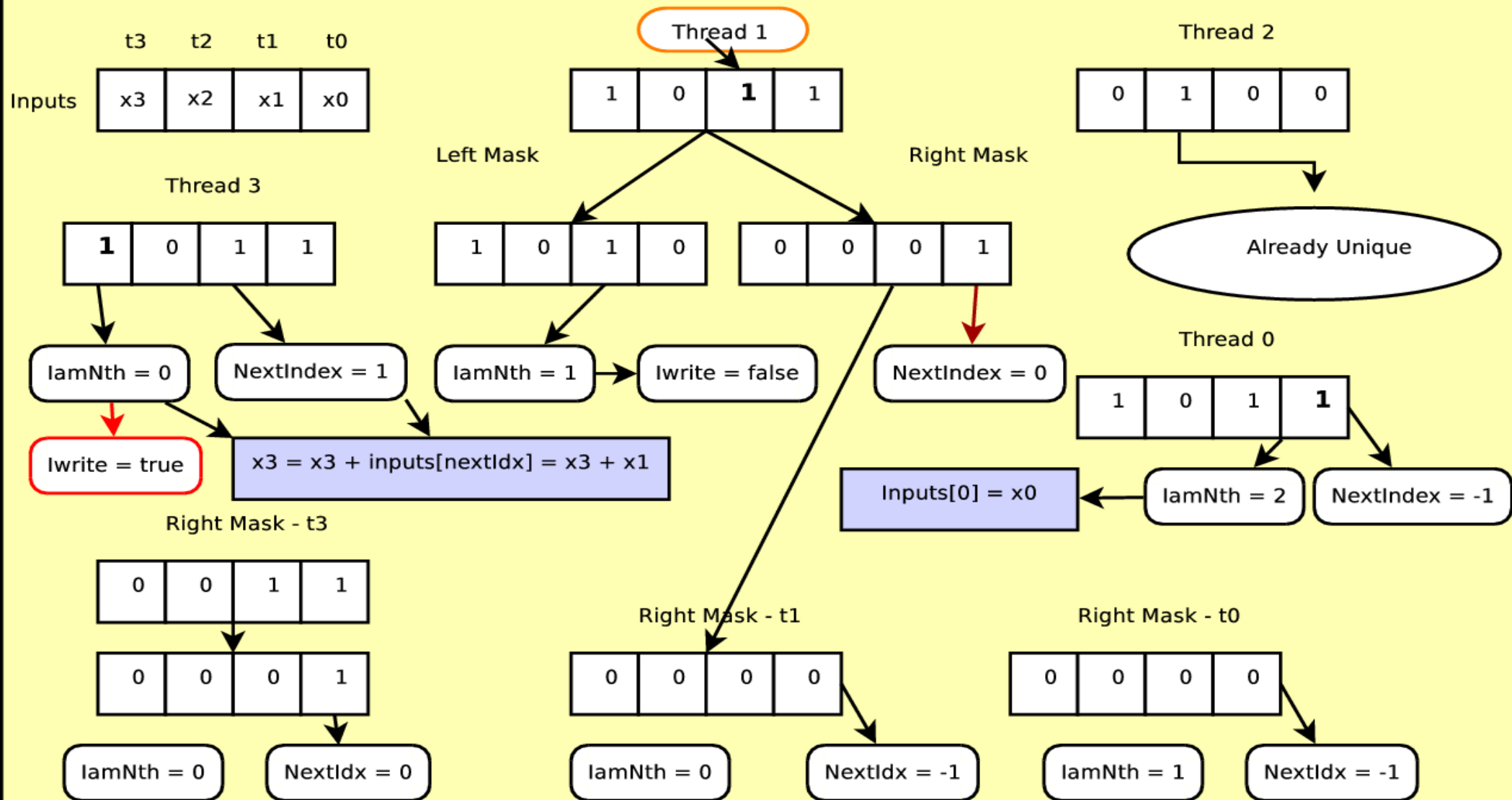
Warp Key-Reduce Example



Warp Key-Reduce Example



Warp Key-Reduce Example



Warp Key-Reduce Example

