Lecture 12

Reduction

A few more CUDA issues

Sorting on GPU

Last time

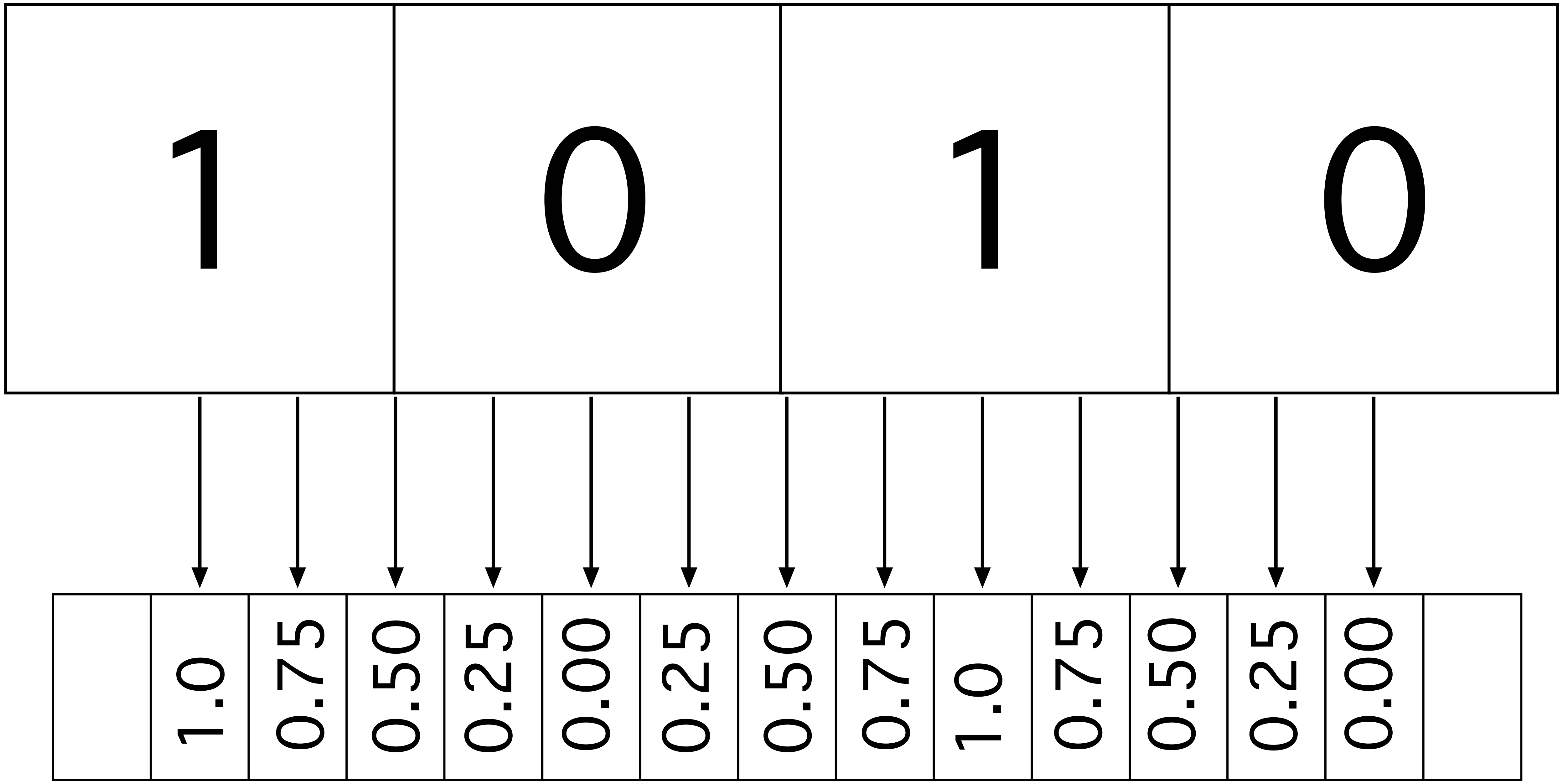
- Coalescing
- Constant memory
- Texture memory
- OpenGL interoperability

A bonus demo on texture memory

texobjdemo.cu

Simple texture memory example.

Array of numbers, accessed at non-integer coordinates.


1
0
1
0
1.0
0.75
0.50
0.25
0.00
0.25
0.50
0.75
1.0
0.75
0.50
0.25
0.00

Upcoming and ongoing labs

Lab 4: Intro to CUDA, Mandelbrot

This week.

Lab 5: Image filtering.

Shared memory in focus!

Lab 6: Reduction and sorting with OpenCL.

Lecture questions

1) How can you efficiently compute the average of a dataset with CUDA?

2) In what way does bitonic sort fit the GPU better than many other sorting algorithms?

3) What is the reason to use pinned memory?

4) What problem does atomics solve?

Reduction

Parallelizing problems of limited parallel nature

Problem seen in Kesser 1.3.1.4 and 1.5.2-1.5.4
Global sum.

Examples of reduction algorithms

Extracting small data from larger

- Finding max or min
- Calculating median or average
- Histograms

Common problems!

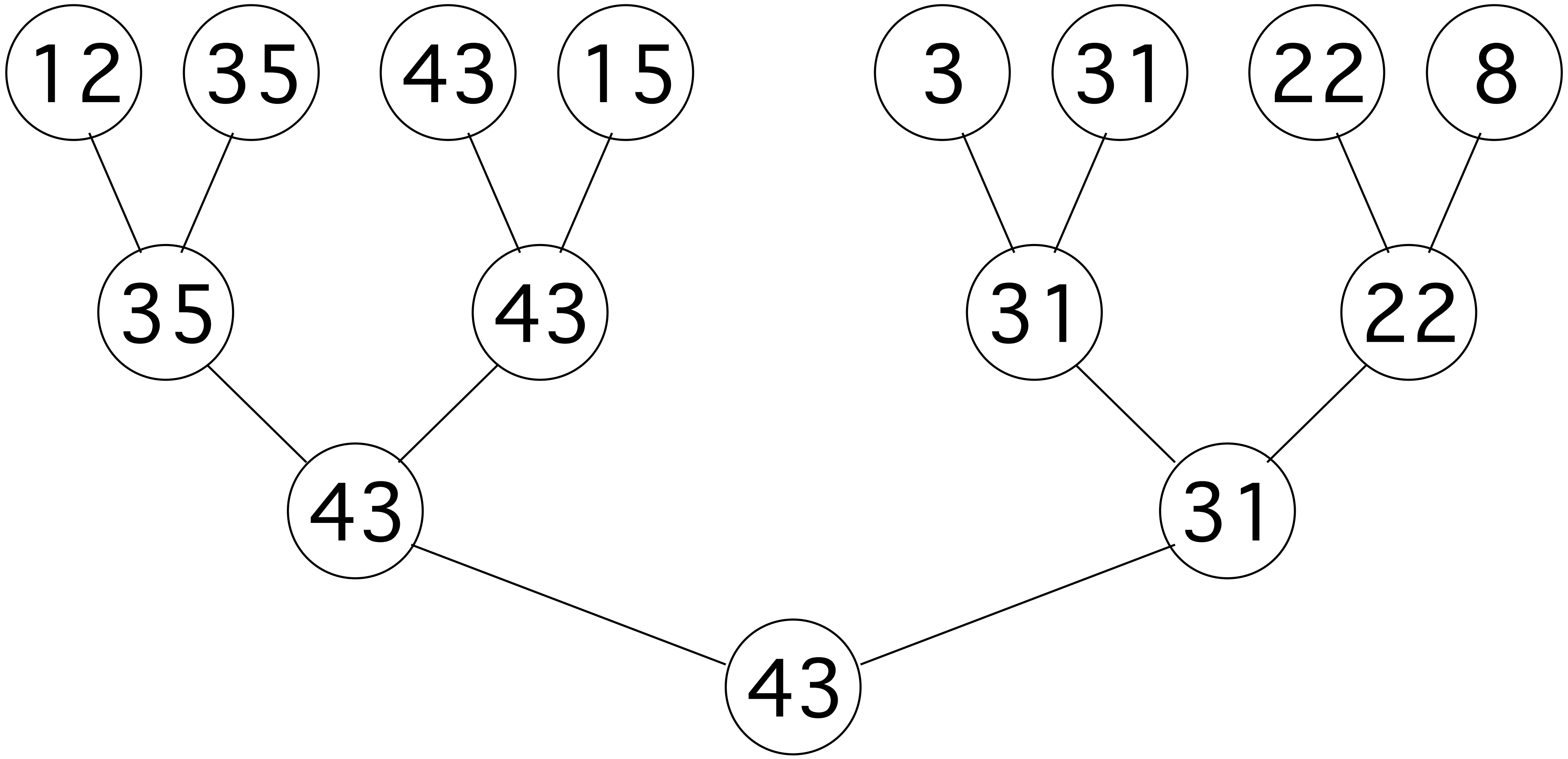
Sequentially trivial

Loop through data

Add/min/max, accumulate results

Fits badly in massive parallelism!

Tree-based approach



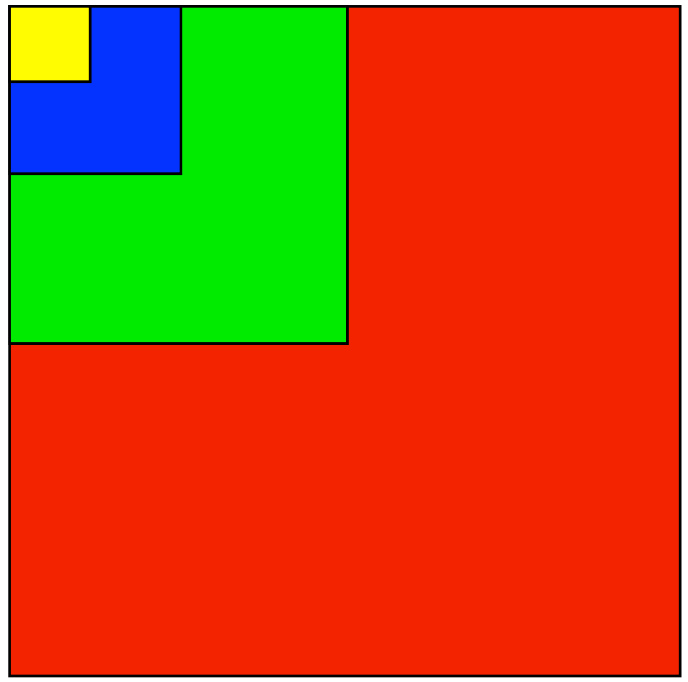
In 2D, typically 4-to-1 per level

Pyramid hierarchy

47	2	3	57	5	12	7	8
10	20	6	13	14	15	16	17
19	11	21	22	23	68	25	26
38	29	64	31	32	33	35	34
37	28	39	49	53	42	41	52
46	1	48	40	61	51	44	43
55	71	4	58	69	62	50	60
30	65	66	67	24	59	70	56

→

47	57	15	17
38	64	68	35
46	49	61	52
71	67	69	70

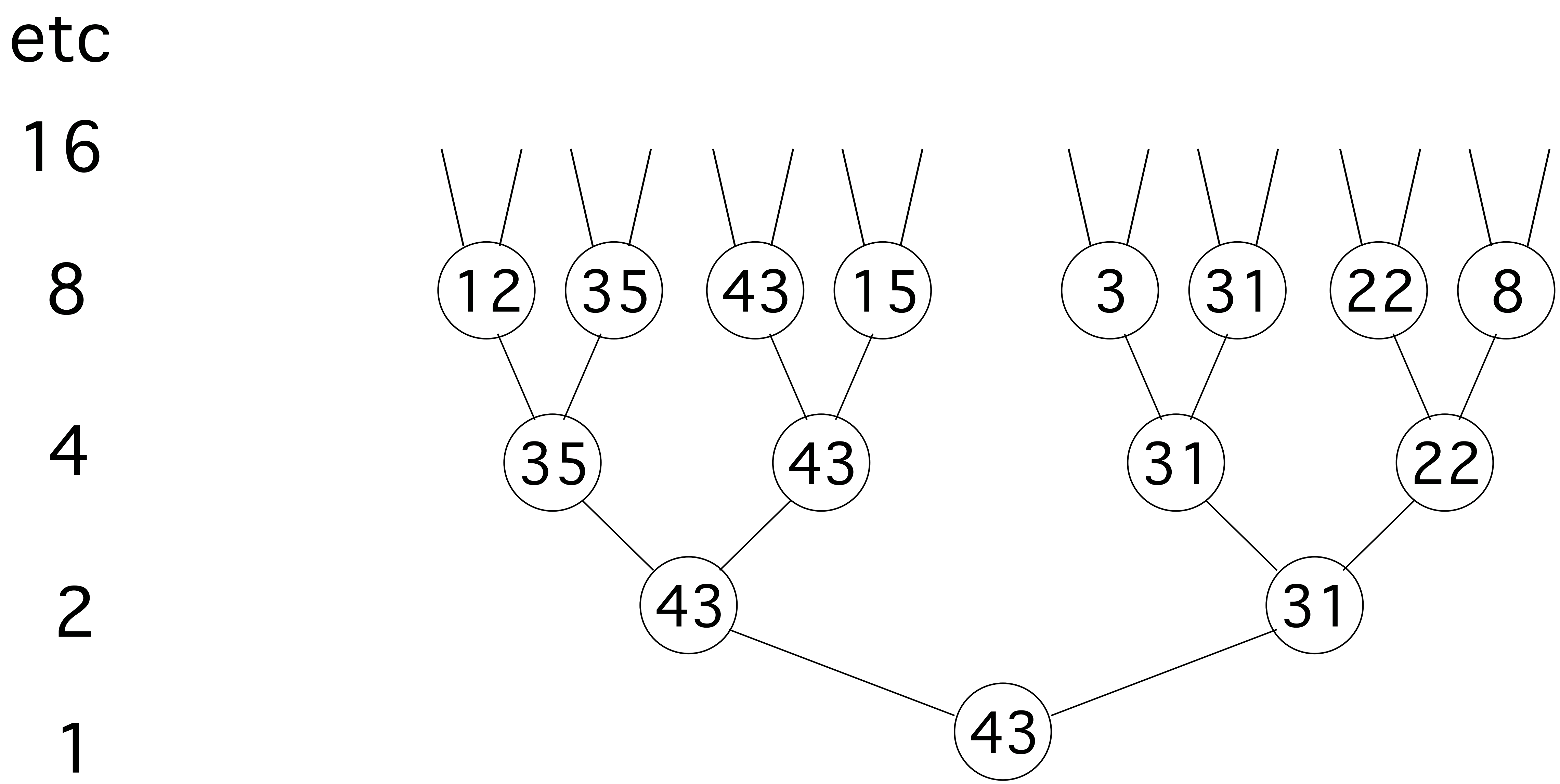


Tree-based approach

Each level parallel! Can be split onto large numbers of threads

but

the parallelism is reduced for each level, and the results need to be reorganized to a smaller number of threads!


etc
16
8
4
2
1
12
35
43
15
3
31
22
8
35
43
31
22
43
31
43

Multiple kernel runs for varying size!

For n = k downto 0 do
Launch 2^n kernels

Multiple levels can be merged into one - but not all of them!

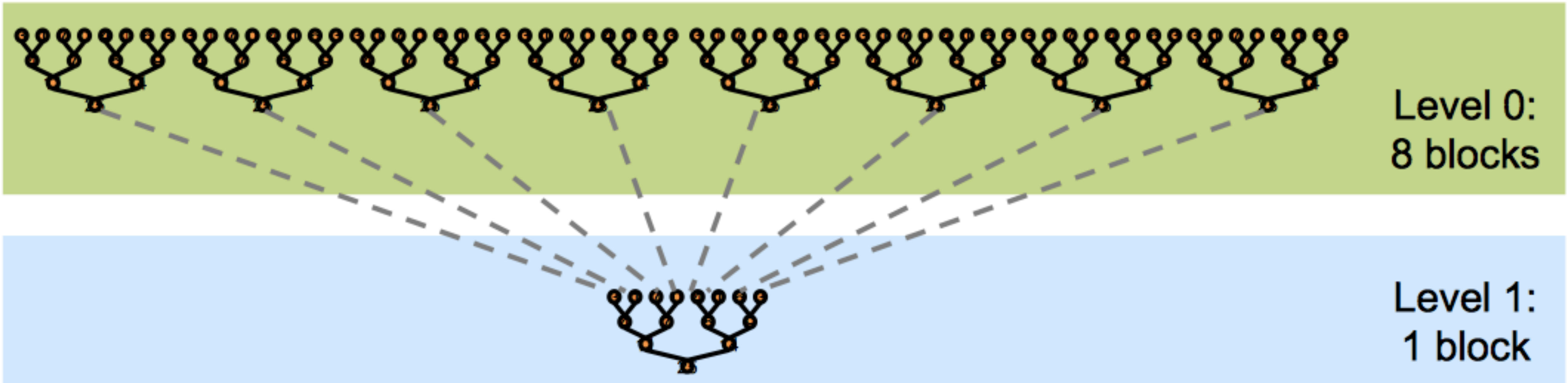
Important note: You can not synchronize between blocks!

Why?

- **Complex hardware**
- **Risk for deadlock between blocks that are not simultaneously active**

(Picture by Mark Harris, NVidia)

Multiple levels per kernel run for avoiding overhead



(Picture by Mark Harris, NVidia)

Many important optimizations:

- Avoid "if" statements, divergent branches
- Avoid bank conflicts in shared memory
- Loop unrolling to avoid loop overhead (classic old-style optimization!)

Huge speed difference reported by Harris

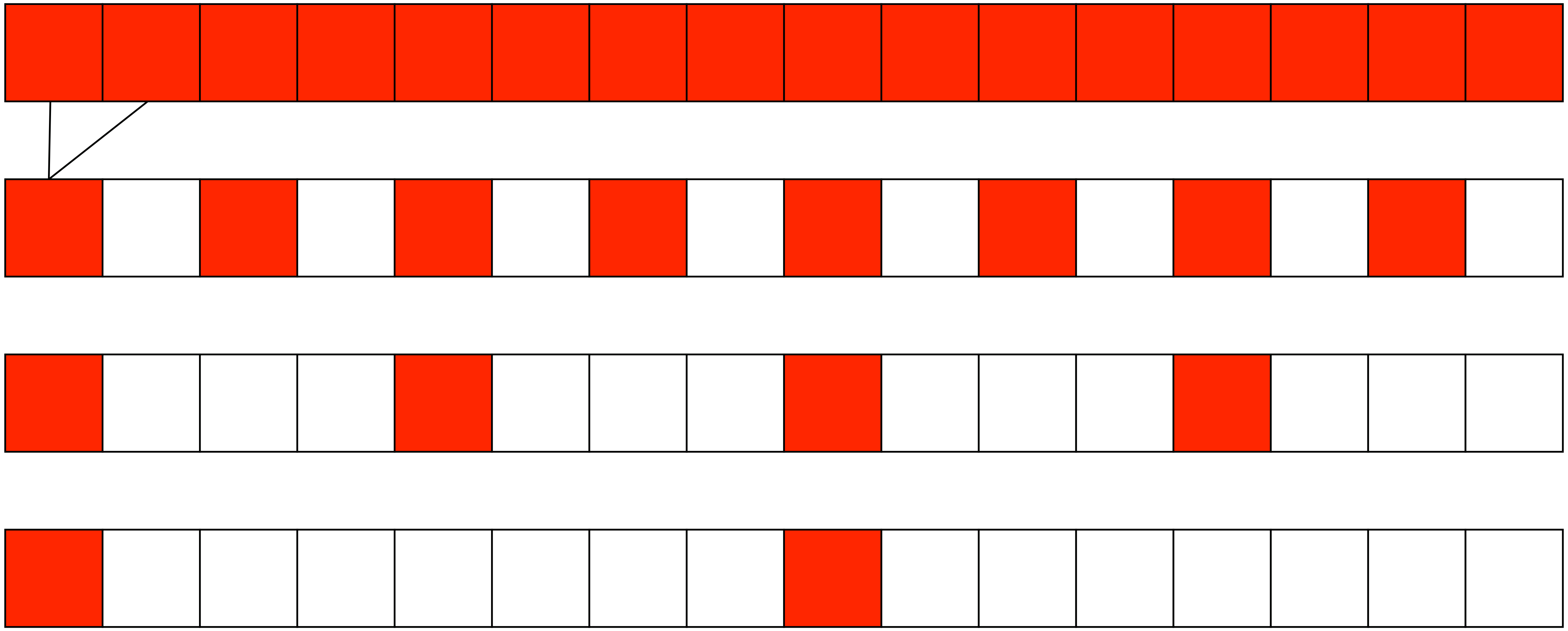
	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x	8.34x
Kernel 5: unroll last warp	0.536 ms	31.289 GB/s	1.8x	15.01x
Kernel 6: completely unrolled	0.381 ms	43.996 GB/s	1.41x	21.16x
Kernel 7: multiple elements per thread	0.268 ms	62.671 GB/s	1.42x	30.04x

Alternative: Reduction in many levels, but making sure idle threads are *dense*!

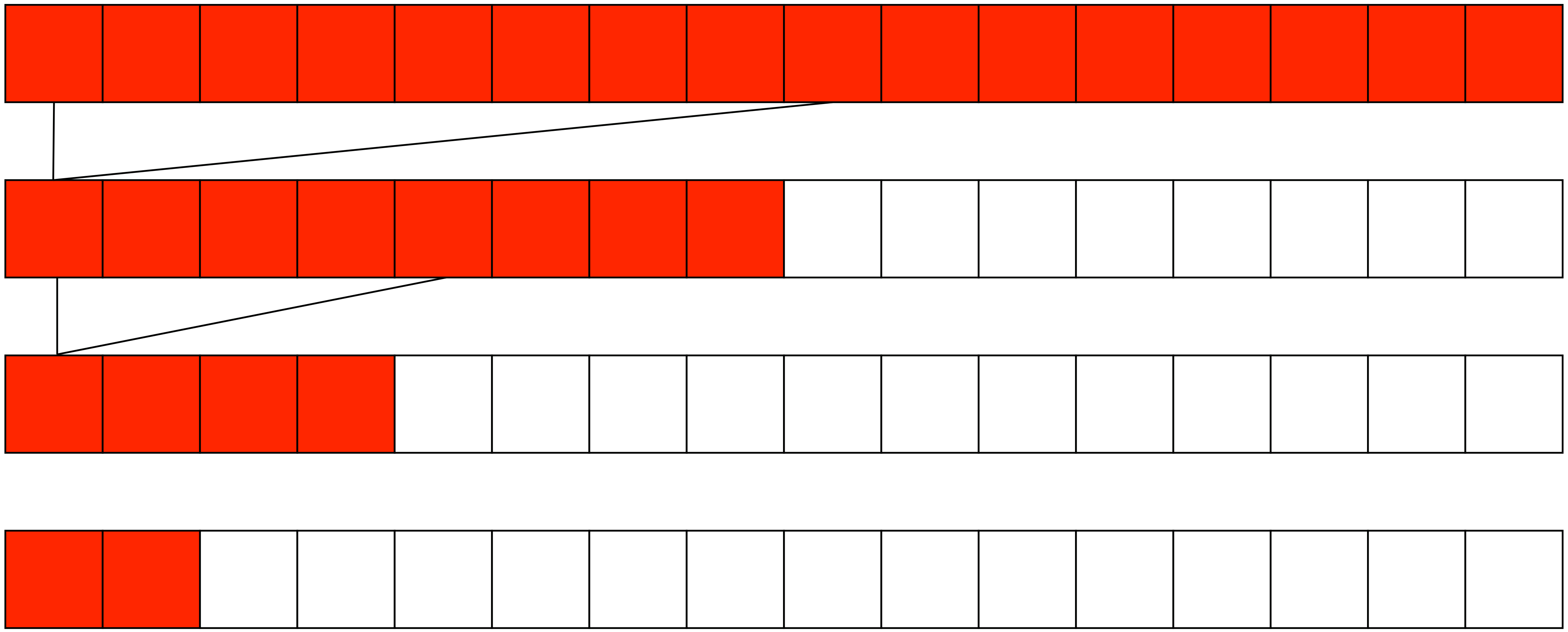
With every other thread idle/finished - half the performance.

With every other *warp* idle finished - good performance!

Skip every other thread over and over in same kernel - waste!



Keep active threads together - better!



Threads and memory both behave like this for coalescing.

Conclusions:

- Multiple kernel runs for varying problem size
- Multiple kernel runs for synchronizing blocks
- Optimizing matters! Not only shared memory and coalescing!

Managed memory

Makes read/write memory as easy as constant!

New, simpler Hello World!

```
#include <stdio.h>

const int N = 16;
const int blocksize = 16;

__global__
void hello(char *a, int *b)
{
 a[threadIdx.x] += b[threadIdx.x];
}

__managed__ char a[N] = "Hello \0\0\0\0\0\0";
__managed__ int b[N] = {15, 10, 6, 0, -11, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
```

```
int main()
{
 printf("%s", a);
 dim3 dimBlock( blocksize, 1 );
 dim3 dimGrid( 1, 1 );
 hello<<<dimGrid, dimBlock>>>(a, b);
 cudaDeviceSynchronize(); // Synchronize

 printf("%s\n", a);
 return EXIT_SUCCESS;
}
```

Managed memory

Managed memory must be declared __managed__

Memory accessible both from CPU and GPU.

Do not expect performance penalty (but always be ready for surprises).