Lecture 12

Reduction

A few more CUDA issues

Sorting on GPU

Last time

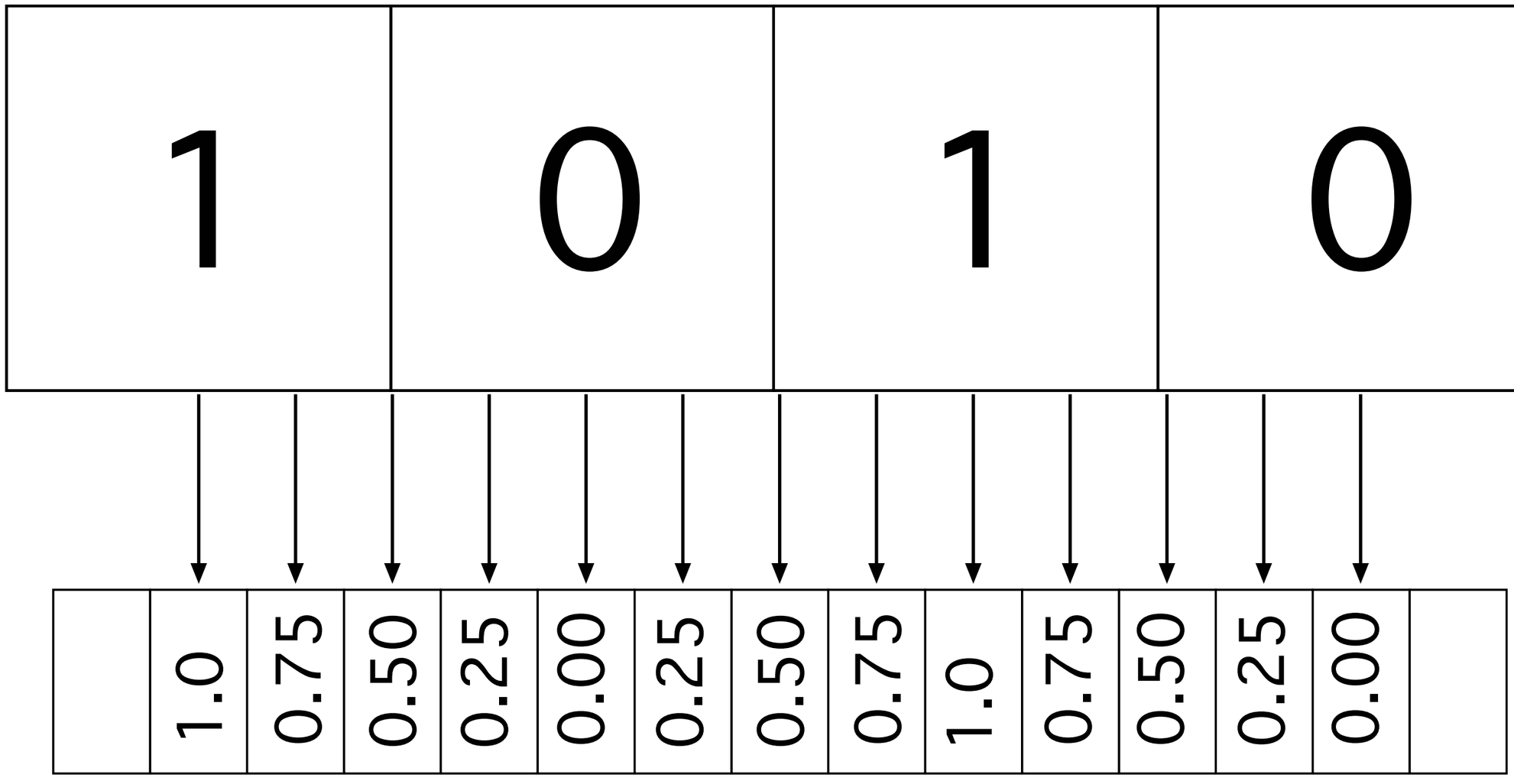
- **Coalescing**
- **Constant memory**
- **Texture memory**

The world's simplest 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

This week: Lab 4: Intro to CUDA, Mandelbrot

then

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 Kessler 1.3.1.4 and 1.5.2-1.5.4
Global sum.

Examples of reduction problems

Extracting small data from larger

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

Common problems!

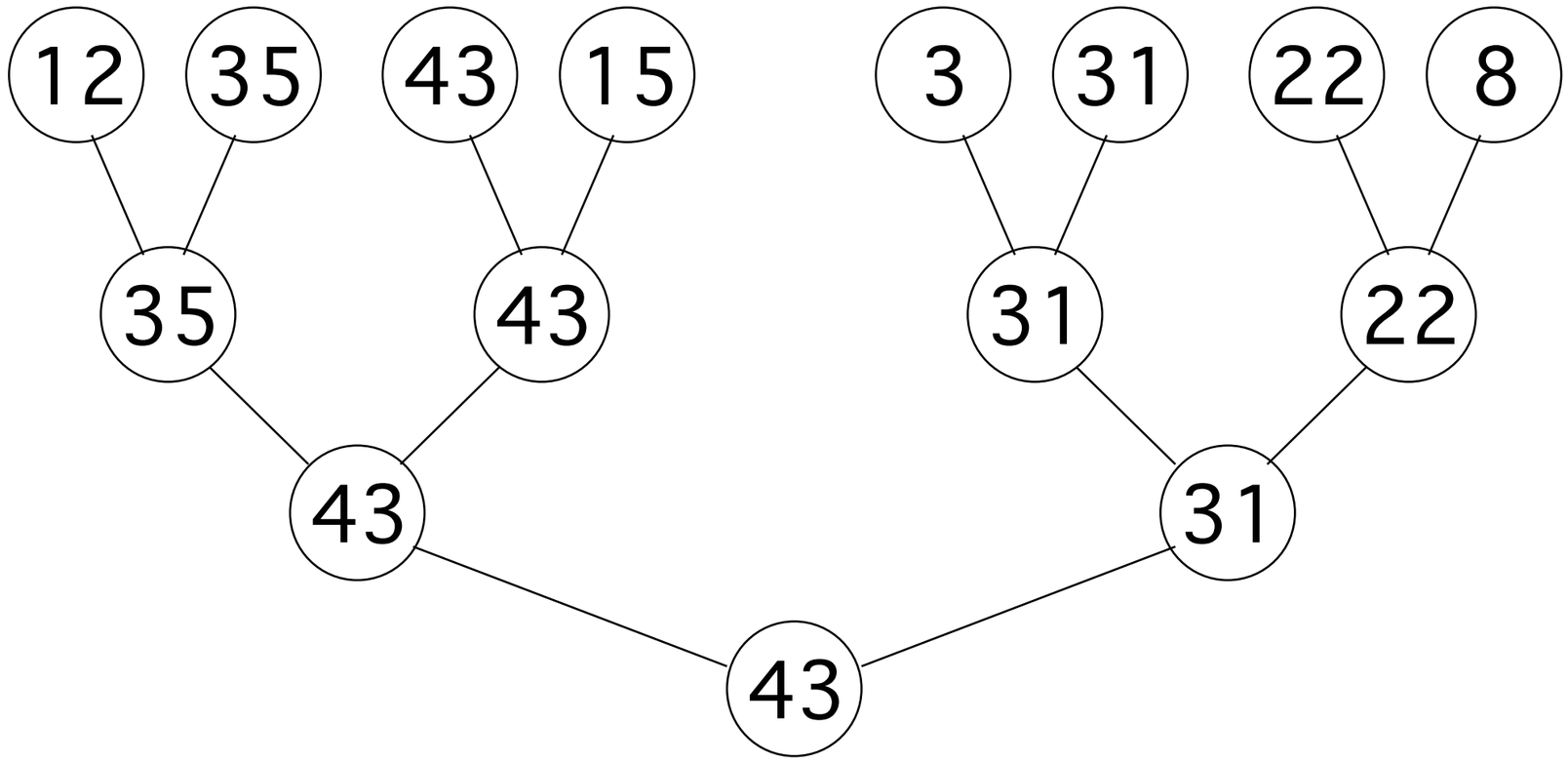
Sequentially trivial

Loop through data

Add/min/max, accumulate results

Fits badly in massive parallelism!

Tree-based approach

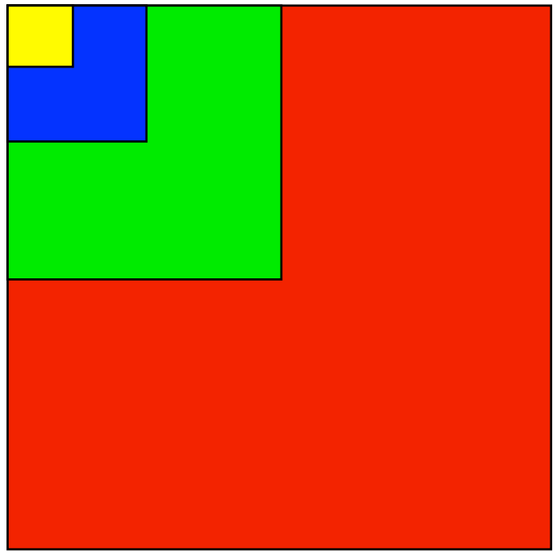


In 2D, typically 4-to-1 per level

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

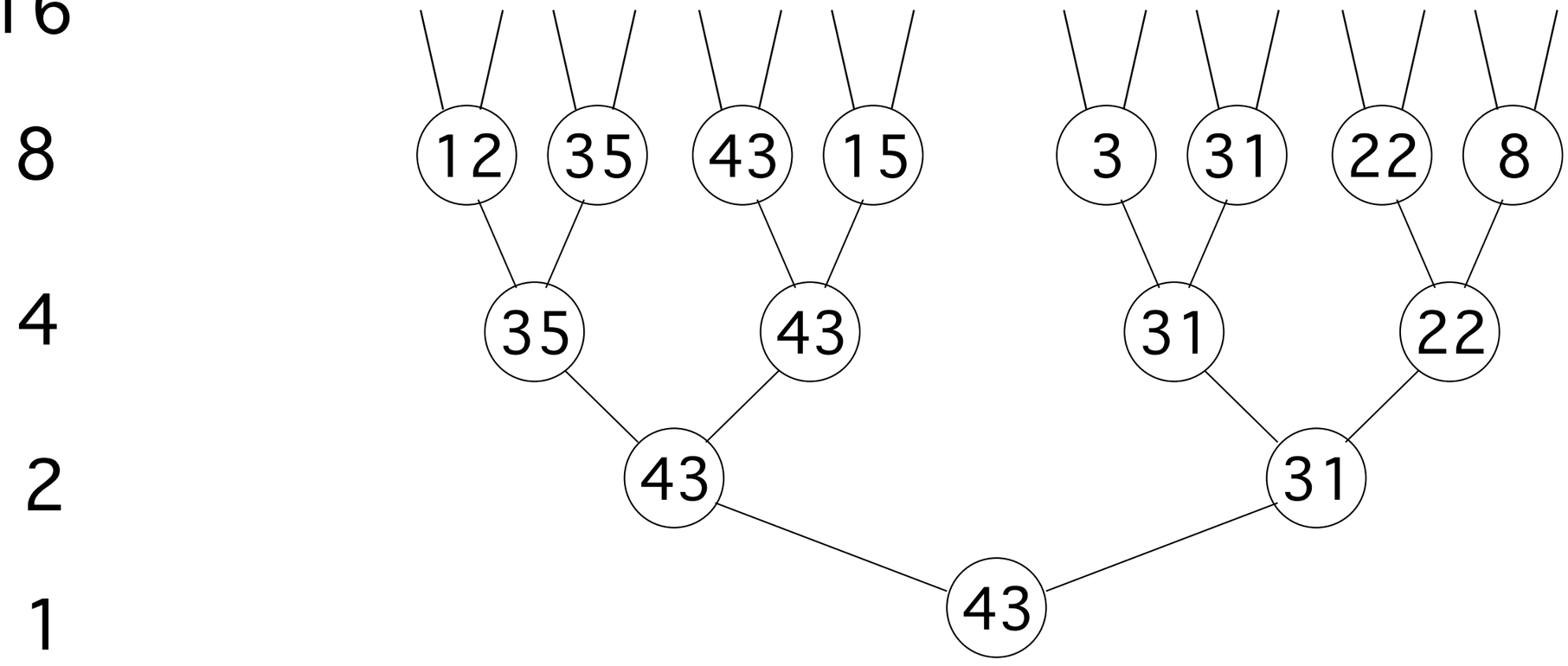


Tree-based approach

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

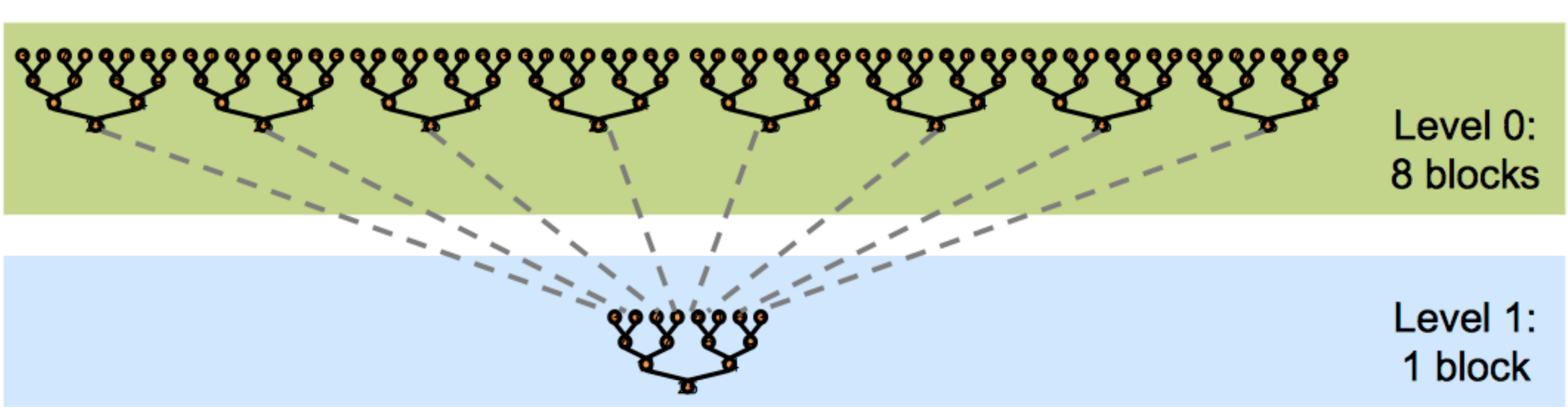
Multiple levels can be merged into one - but not all of them!

Again: You can not synchronize between blocks!

Not all blocks are simultaneously active

(Picture by Mark Harris, NVidia)

Multiple levels per kernel run for avoiding overhead



(Picture by Mark Harris, NVidia)

Doubly interesting due to study with many optimizations:

Many possibilities:

- 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

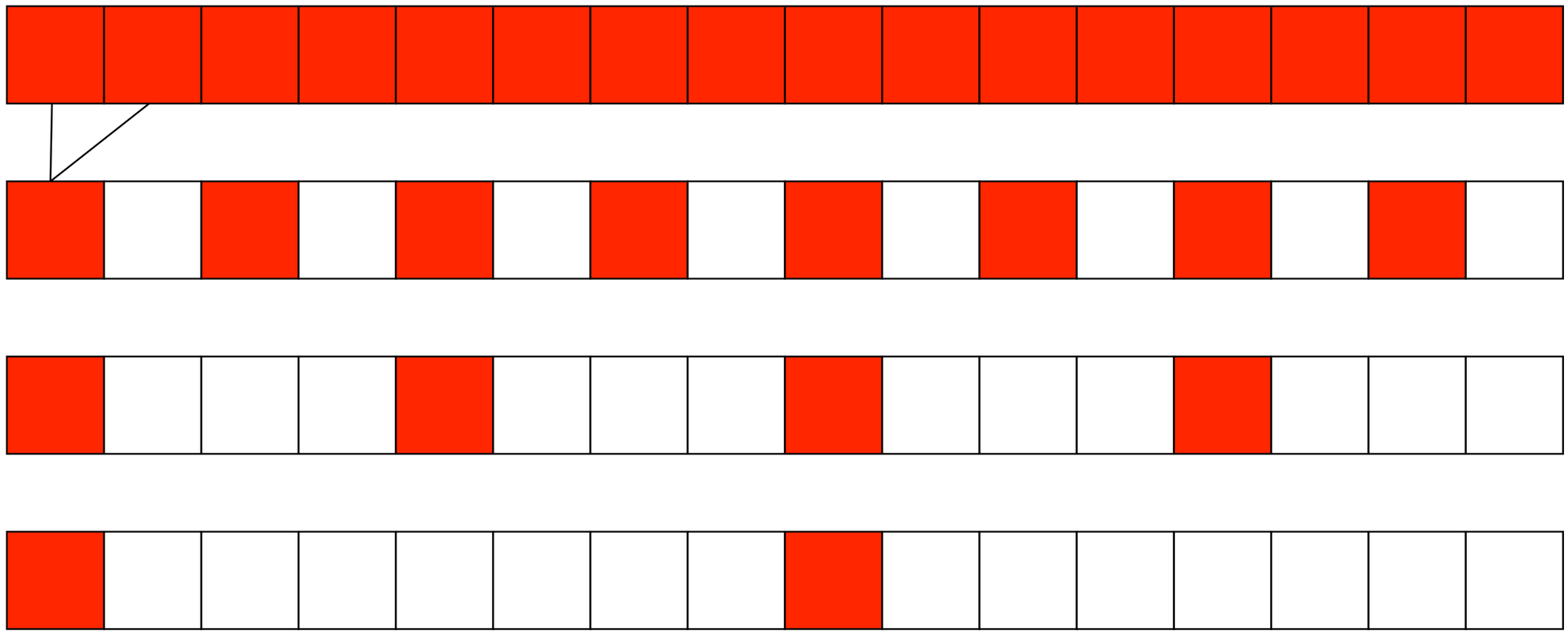
However, some of these optimizations are no longer valid.

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

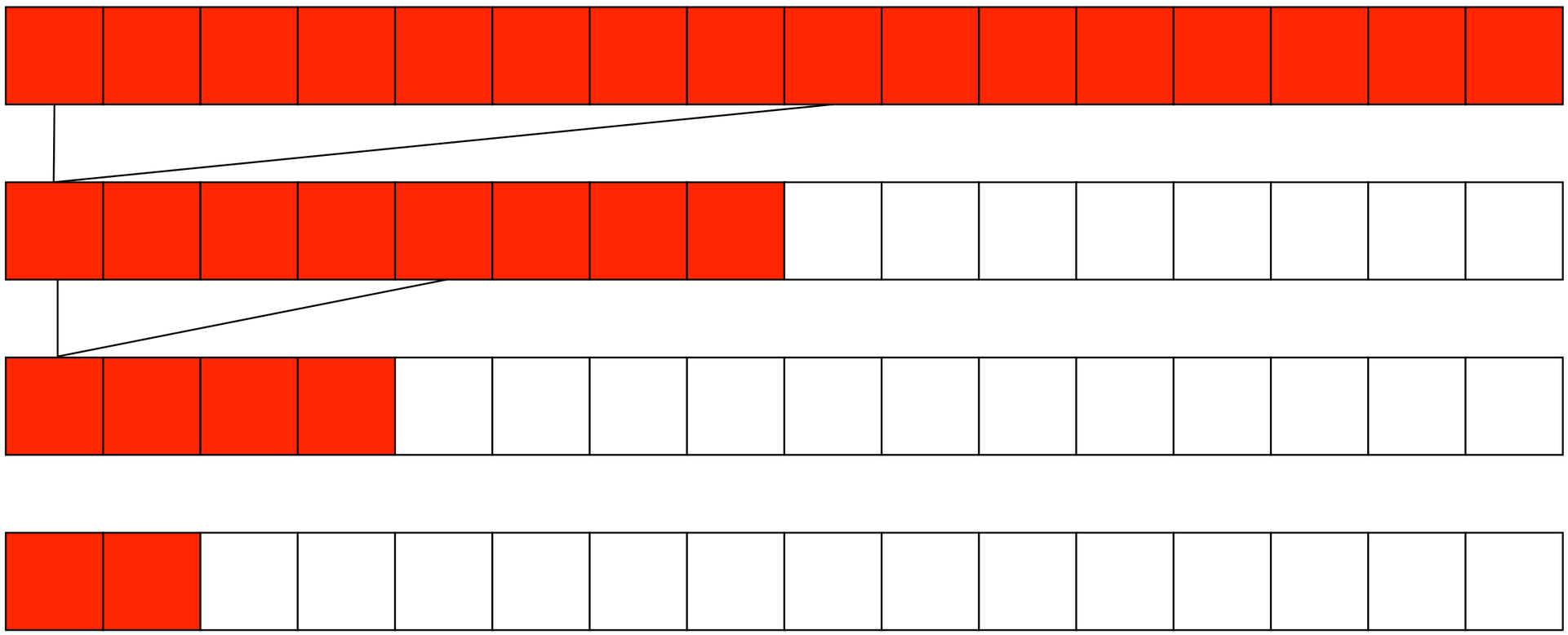
With every other thread idle/finished - half the performance.

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.

Divergent branching =

"if" statements:

Branches can be bad in GPU code!

Why?

Divergent branching in SIMD:

All branches execute all code! Data masked with result of "if".

Warp-level problem!

Can not be avoided within warps if a single thread gets a different result from others. Can be avoided if all threads in warp take same branch

Divergent warp

```
if X then  10010110
|
|        and with 10010110
|
else
|
|        and with 01101001
|
endif
```

Non-divergent warp

```
if X then  11111111
|
|
|
else
|
|
|
endif
```

Skip

Conclusions:

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

More memory

Managed memory

Atomics

Pinned memory

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. Risk for racing!

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

Not supported everywhere.

Atomic operations

A special memory access method, for avoiding conflicts and race conditions.

Available in CUDA from Compute model 1.1.

To use it, specify model with

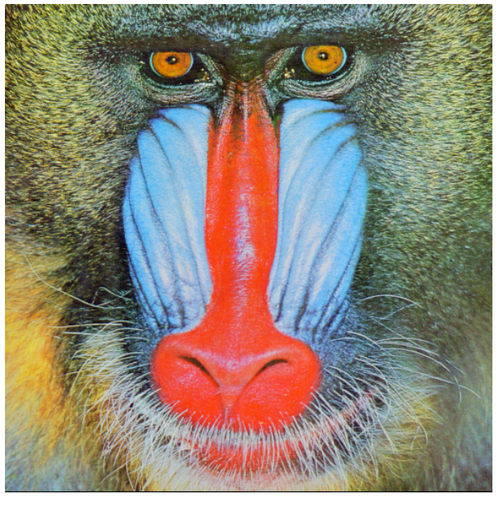
```
-arch compute_11
```

(or higher)

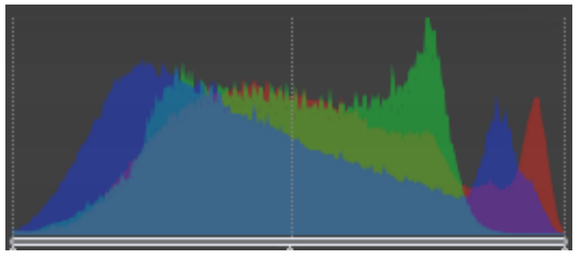
Example: Histogram

Simple method for gathering statistics about a set of data. Much data in, little out.

Common in image processing.



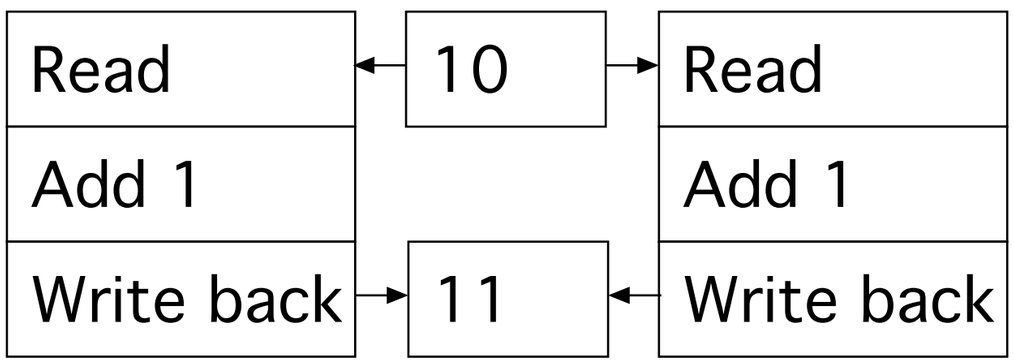
for all elements i in a[]
h[a[i]] += 1



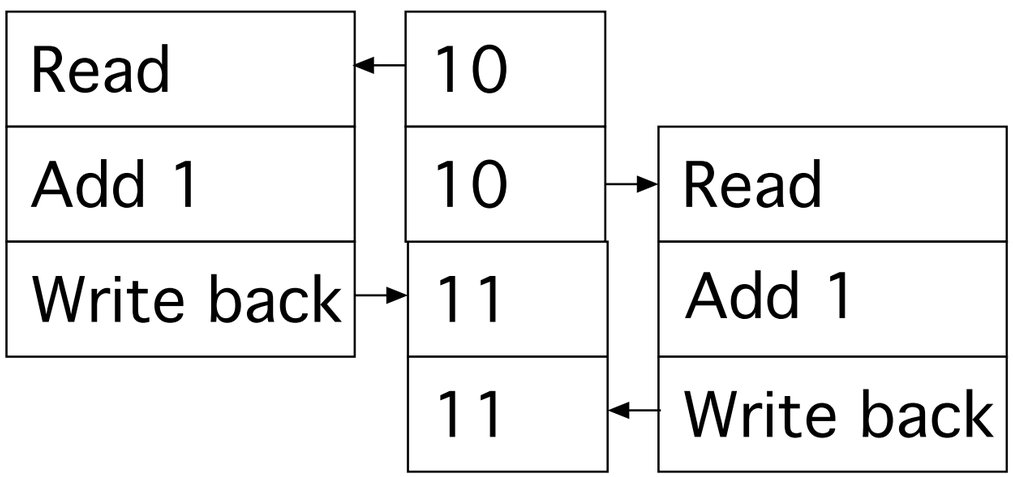
Histogram memory conflicts

If you try to parallelize these operations, multiple threads will write simultaneously at the same item

Non-atomic operations will read h[a[i]], add 1, and write back.



Unknown write order



Write unsynchronized values in sequence

Solution: Atomics

Read - modify - write in *one* operation

Guaranteed not to be subject to racing

atomicAdd, atomicSub, atomicExch, atomicMin, atomicMax, atomicInc, atomicDec, atomicCAS, atomicAND, atomicOR, atomicXor

More types in Fermi and up

But it comes for a cost!

Slower than other operations

Global memory only as of Compute Capability 1.1

Shared memory atomics in modern GPUs.

Simpler but slower than reduction solutions!

Example: Find maximum

for all elements i in a[]
maxValue = max(maxValue, a[i])

Easy? Yes! Parallel? No!

All threads will write to the same memory element!

Use atomics? Very slow! All write at the same time, must wait -> sequential performance!

Solution: Use reduction instead!

Atomic conclusions

Simplifies some operations

Serializes conflicting operations

Can hurt performance! Don't overuse!

More exotic optimizations and tools

Pinned memory

Multiple streams

Not where you start but let's not ignore the options.

Pinned memory

Can boost performance for memory transfer

Page-locked memory

So far: malloc() and cudaMalloc()

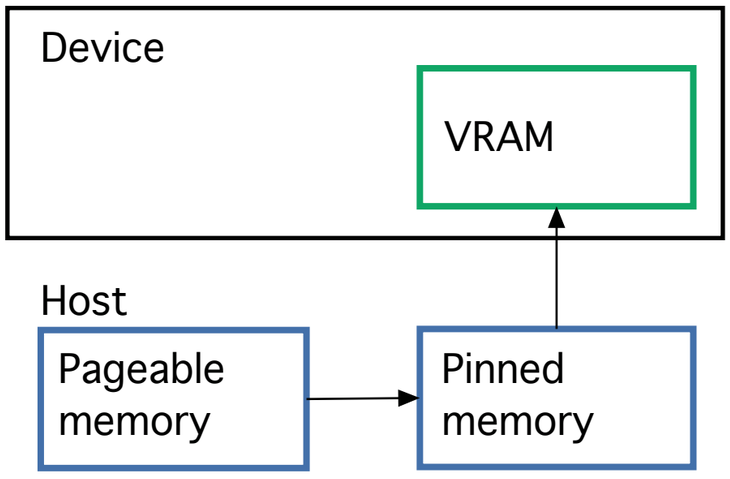
New call: cudaHostAlloc()

Allocated page-locked memory! Fixed physical location!

Pinned memory

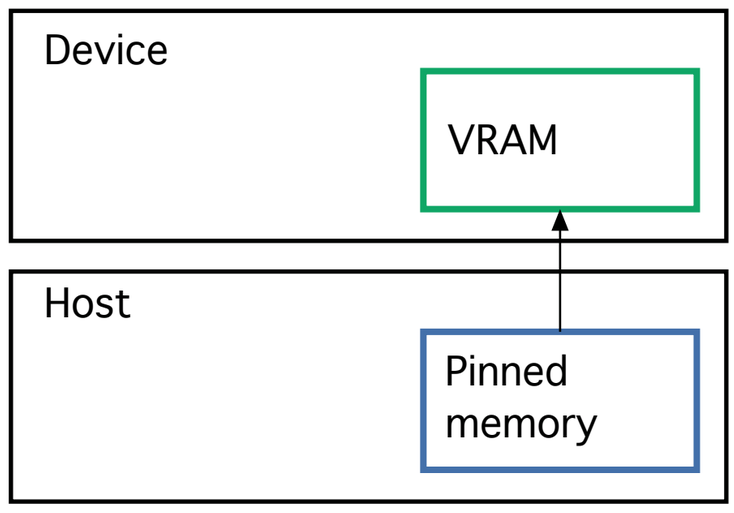
Page-locked memory is a limited resource!

For non-pinned memory, CUDA copies it internally to page-locked memory, then DMA to GPU. Transfer time goes up!



Picture based on an NVidia article

Normal, pageable data transfer



Pinned data transfer

Pinned memory, streams, overlapping computation

Pinned memory is part of an optimization approach with overlapping computations

No longer just a slight speedup of data transfer!

cudaMemCpyAsynch() can copy locked memory asynchronously!

Multiple streams

CUDA commands are placed in a queue, a *stream*!

These are the same queues as you can post CUDA events to.

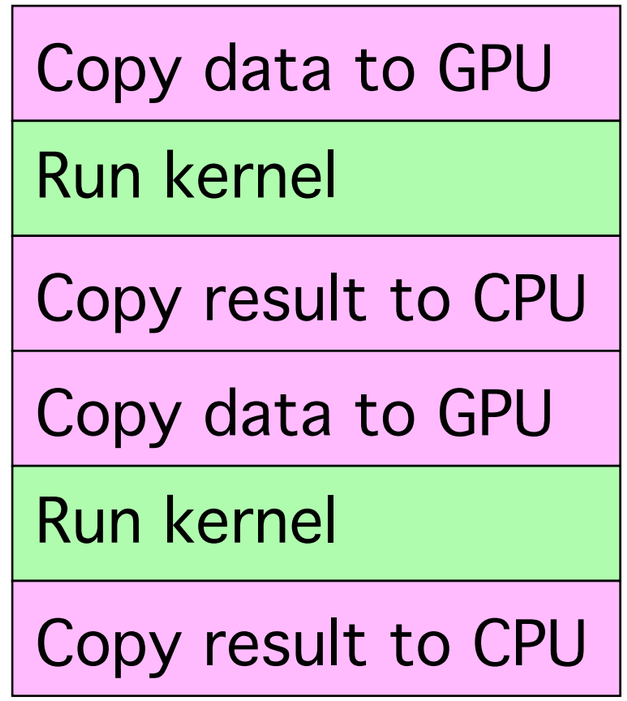
We usually only use the default CUDA stream.

Multiple CUDA streams can be used to overlap work - especially computing and data transfers!

Single stream computation

The kernel can not run until the data is transferred.

For this example, 2/3 data transfer, 1/3 computation



Dual stream computation

While one stream runs a kernel, the other stream performs data copying,

More time for computing, in this example kernels are running 1/2 of the time instead of 1/3.

Copy data to GPU	
Run kernel	Copy data to GPU
Copy result to CPU	Run kernel
Copy data to GPU	-
Run kernel	Copy result to CPU
-	Copy data to GPU
Copy result to CPU	Run kernel
	-
	Copy result to CPU

Not all devices...

Asynchronous data copying as well as concurrent execution is not guaranteed...

so make a device query!

CU_DEVICE_ATTRIBUTE_ASYNCH_ENGINE_COUNT:
Can we copy memory asynch?

CU_DEVICE_ATTRIBUTE_CONCURRENT_KERNELS:
Can we run multiple kernels?

Debugging CUDA

Let's get a bit more efficient when your code doesn't work

- Catch error codes
- printf() from kernels
- cudagdb

Catch those error codes

```
// Check for errors everywhere
err = cudaMalloc( (void**)&ad, csize );
// If the GPU won't even take our data we are toasted
if (err) printf("cudaMalloc %d %s\n", err, cudaGetErrorString(err));
...
dim3 dimBlock( blocksize, 1 );
dim3 dimGrid( 1, 1 );
hello<<<dimGrid, dimBlock>>>(ad, bd);
// Most important thing to check? Did the kernel run at all?
err = cudaPeekAtLastError();
if (err) printf("cudaPeekAtLastError %d %s\n", err, cudaGetErrorString(err));
```

and pass them to cudaGetErrorString() for an explanation

printf() from kernels

Yes - printf() if legal in a kernel since Compute Capability 2.0

But don’t try to print 100000 messages per second...

More advanced debugger tools

There are more tools to help you out there!

cudagdb

Variant of the GDB debugger

Allows breakpoints and single-stepping CUDA kernels!

Sorting on GPUs

Revisiting some algorithms from lecture 6:

Some not-so-good sorting approaches

Bitonic sort

QuickSort

Concurrent kernels and recursion

Adapt to parallel algorithms

Many sorting algorithms are highly sequential

Suitable for parallel implementation?

- **Data driven execution**
- **Data independent execution**

Data driven execution

Computing pattern depends on data

Usually harder to parallellize!

Example: QuickSort.

Data independent execution

Known computing pattern

Easier to parallellize - always the same plan

Example: Bitonic sort

Bubble sort

Loop through data, compare neighbors

Extremely sequential

Inefficient

Parallel version: Bubble sort with odd-even transposition method

Compare all items pairwise

Two phases, ”odd phase” and ”even phase” (shifted one step)

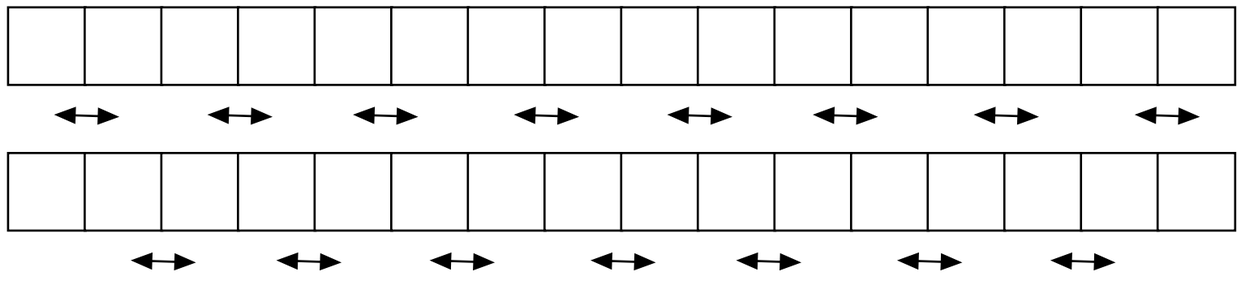
Bubble sort, parallel version

Bubble sort with odd-even transposition method

Compare all items pairwise

Two phases, ”odd phase” and ”even phase” (shifted one step)

Fully sorted after n phases



$O(n^2)$

Suitable for GPU?

Not as bad as it seems at first look:

- **Data independent**
- **Excellent locality**
- **Appears to have possibilities to use shared memory but with some costly transfers at edges between blocks.**
- **But certainly not optimal at very large sizes**

”Better” algorithms don’t necessary beat this all that easily!

Rank sort

Count number of items that are smaller

Easy to parallelize:

- **One thread per item**
- **Loop through entire data**
- **Store in index decided from count of number of smaller items.**

Suitable for GPU?

Again, not as bad as it seems at first look:

- **Data independent**
- **Excellent locality - especially good for broadcasting (e.g. constant memory). Also suitable for shared memory.**
- **Again, $O(n^2)$: Will grow at very large sizes**

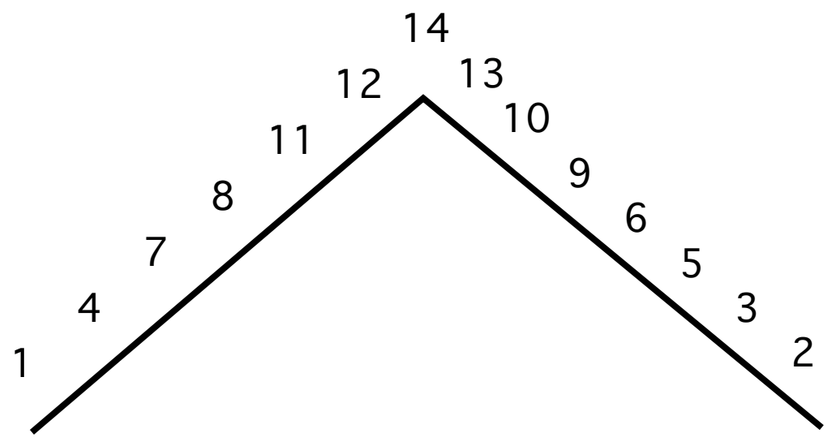
Two bad ones that are not quite as bad as they seem.

N parallel iterations may beat NlogN sequential ones!

Bitonic sort

(As described in Kessler 2.3)

Bitonic set: Two monotonic parts in different direction.



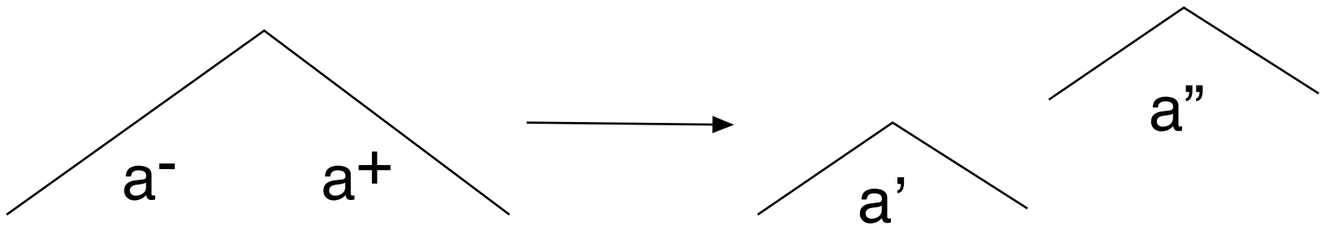
Bitonic sort

(According to Batcher:) Let a be a bitonic set with a maximum at k, consisting of two monotonic parts, one increasing, a^- (from item 1 to k) and one decreasing, a^+ (k+1 to n)

Then two new sets can be constructed as

$$a' = \min(a_1, a_{k+1}), \min(a_2, a_{k+2})\ldots$$
$$a'' = \max(a_1, a_{k+1}), \max(a_2, a_{k+2})\ldots$$

These two sets are also bitonic and $\max(a') \leq \min(a'')$!

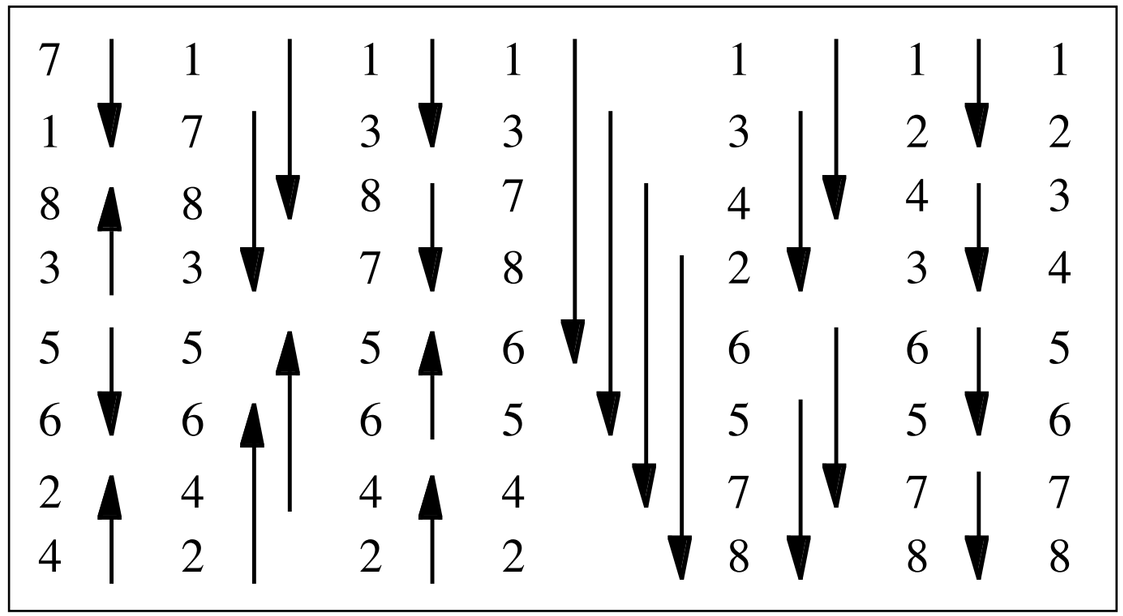


Bitonic sort by divide-and-conquer

Bitonic sort works on a bitonic sequence: partially sorted

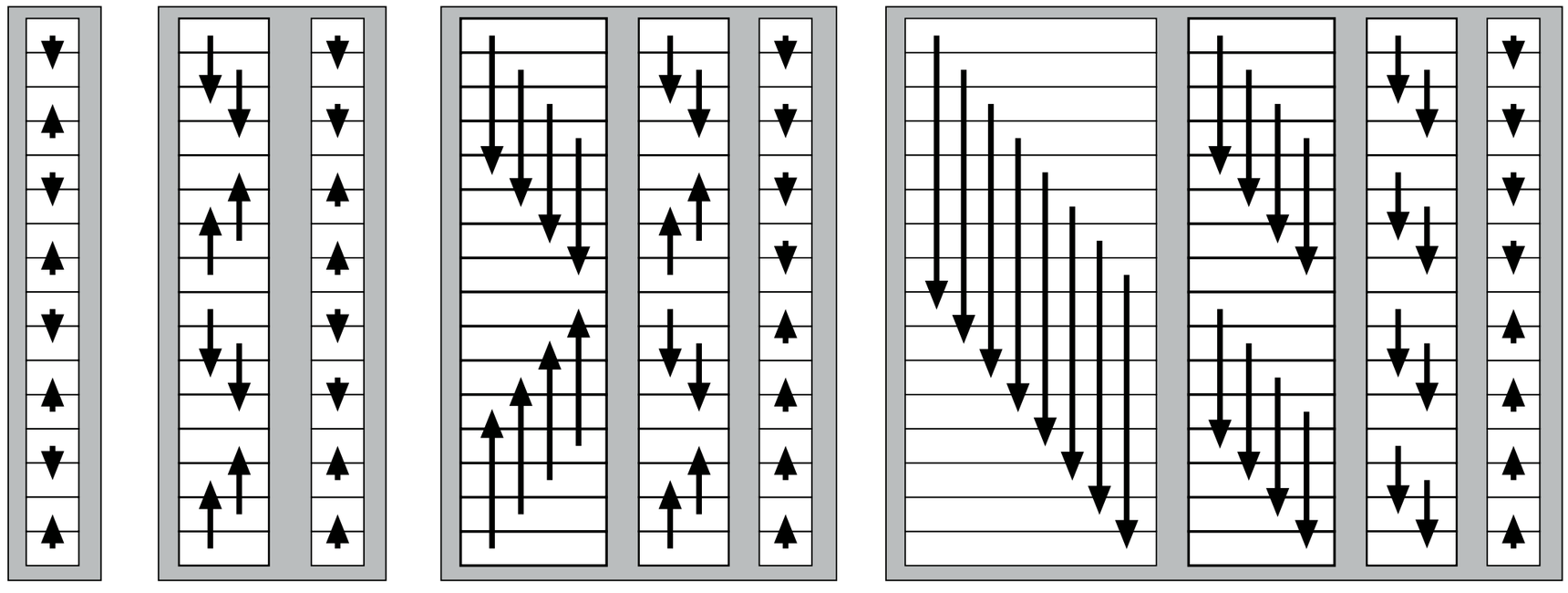
The parts must be sorted. Sort them by bitonic sort!

Bitonic sort example



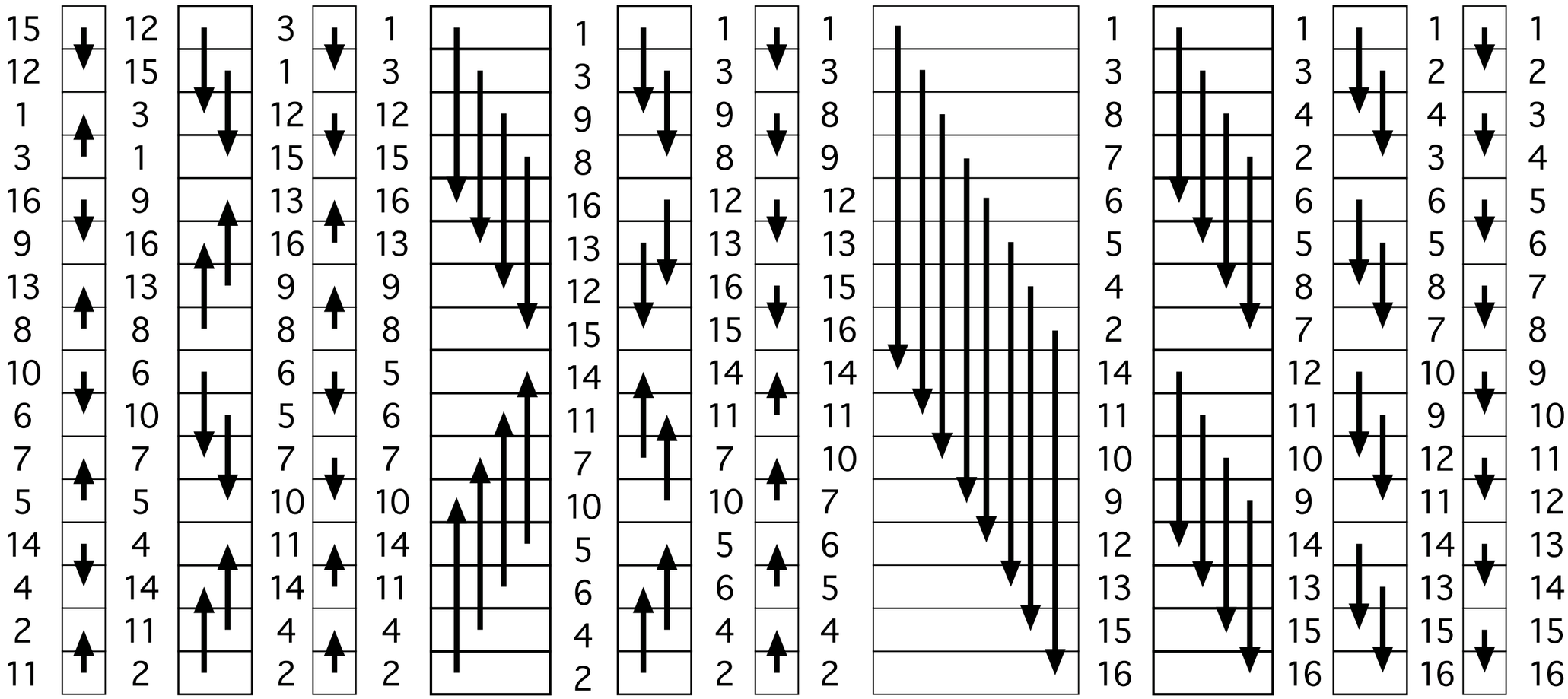
Bigger example

The problem scales nicely, uniformly



More stages gives longer stages

(Image inspired by one from Wikipedia)


15 12 3 1 1 1 1 1 1 1 1
12 15 1 3 3 3 3 3 2 2 2
1 3 12 12 9 8 9 8 4 4 3
3 1 15 15 8 9 8 12 2 3 4
16 9 13 16 16 12 12 13 6 6 5
9 16 16 13 13 13 13 15 5 5 6
13 13 9 9 12 16 15 4 8 8 7
8 8 8 8 15 15 16 2 7 7 8
10 6 6 5 14 14 14 14 12 10 9
6 10 5 6 11 11 11 11 11 9 10
7 7 7 7 7 7 10 10 10 12 11
5 5 10 10 10 10 7 9 9 11 12
14 4 11 14 5 5 6 12 14 14 13
4 14 14 11 6 6 5 13 13 13 14
2 11 4 4 4 4 4 15 15 15 15
11 2 2 2 2 2 2 16 16 16 16

Get those steps right

Step length

Step direction

Comparison direction

Calculated from stage number and stage length

Code examples

Sequential:

Recursive example

Iterative example

Parallel:

CUDA example (not optimized)

Bitonic sort

- **Data independent, no worst case**

- **Fast: $O(n \cdot \log^2 n)$ (Why?)**

- **Good locality in some parts**

but

- **Big leaps in addressing for some parts**

What about those big leaps?

**Small leaps: Can be computed within one block.
Shared memory friendly.**

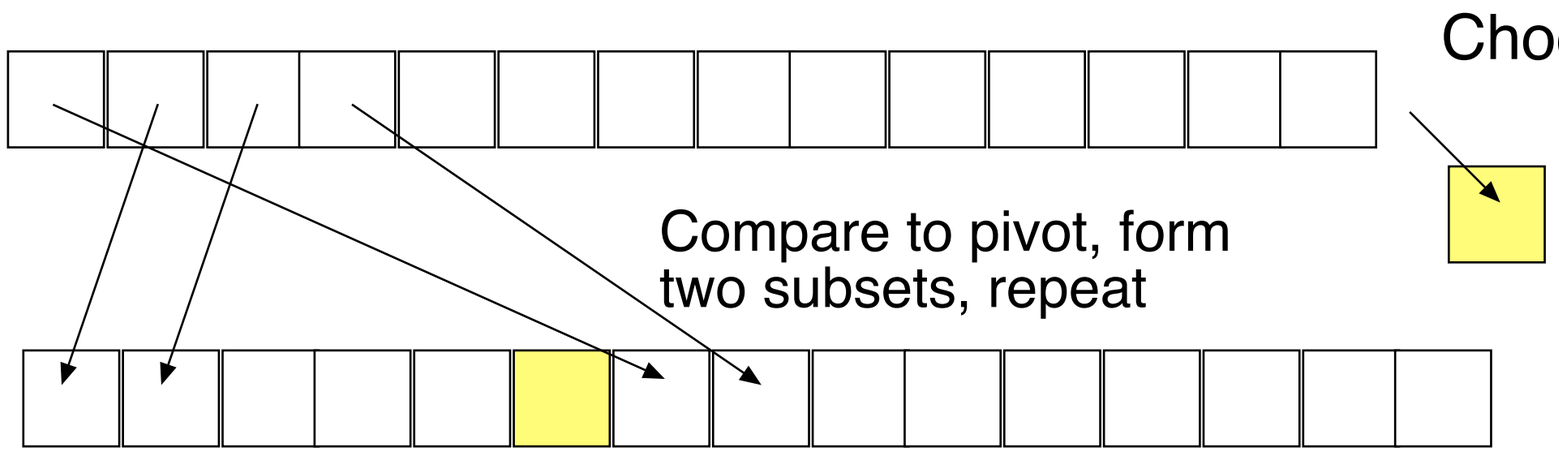
Big leaps (>number of threads/block): No synchronization possible between blocks!

But we *must* synchronize!

-> multiple kernel runs!

QuickSort

Very popular algorithm for sequential implementation



Data driven, data dependent reorganization, non-uniform

Fancy name - nobody expect QuickSort to be nothing but optimal

QuickSort is

Fast: O(n·logn) in typical cases

$O(n^2)$ in the worst case

Data driven, data dependent reorganization, non-uniform

QuickSort on GPU

Initially ignored as impractical

CUDA implementations exist

Data driven approaches increasingly suitable as GPUs become more flexible

Parallel QuickSort

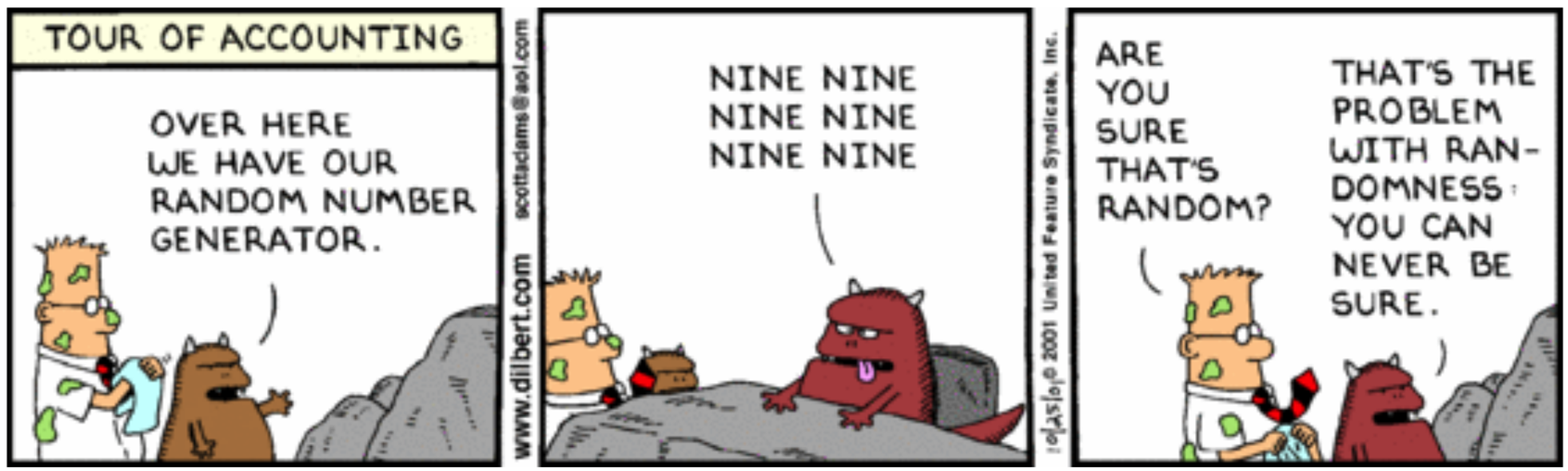
Several stages to consider:

- **Pivot selection. Usually just grab one.**
- **Comparisons**
- **Partitioning**
- **Concatenate result**

Pivot selection

If we could always pick a pivot that splits the data in half…

but you can't do that without sorting! (Or a histogram.) But how about a random one?



There is a worst case caused by bad pivots. Live with it!

Comparisons

Easy to parallelize

One thread per comparison not unreasonable!
(GPUs don’t have a problem with many threads!)

No problem!

Partitioning

The big problem!

Sequential partitioning: Bad!

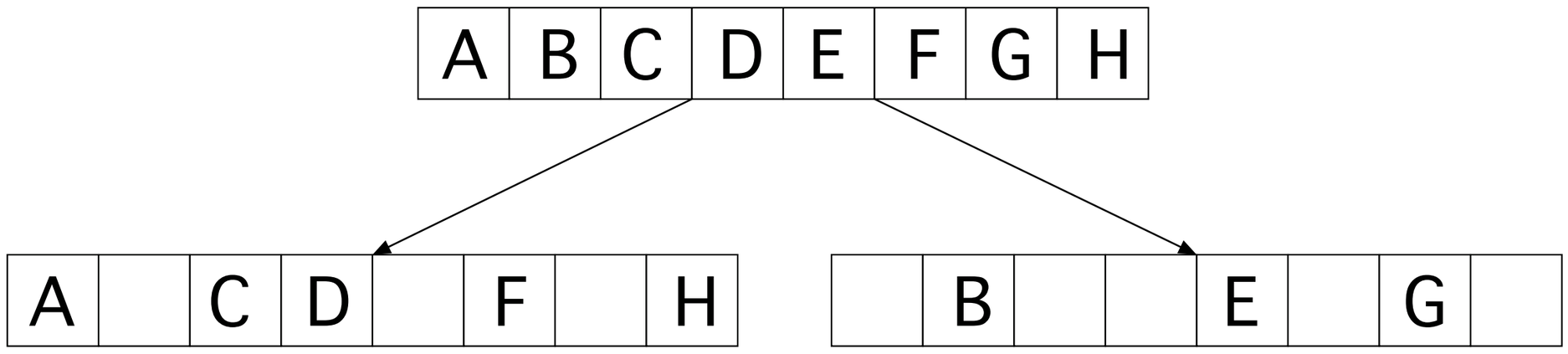
Parallel partitioning 1: Atomic fetch & increment. (GPUs have atomics!)

Parallel partitioning 2: Divide and conquer

In-place sorting not feasible

Split to two list of same size as original. Massive number of threads!

Then we must pack to smaller size.



Packing to smaller size not trivial

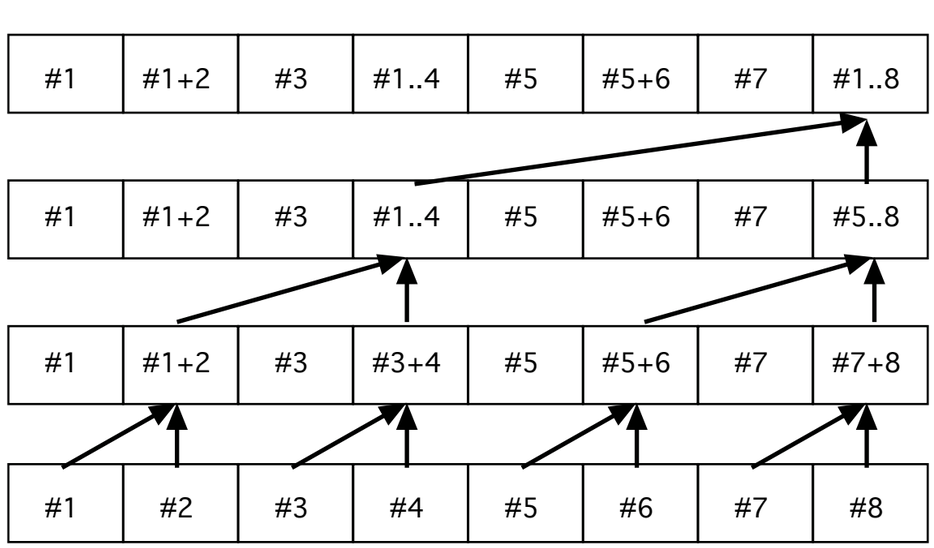
Data dependent

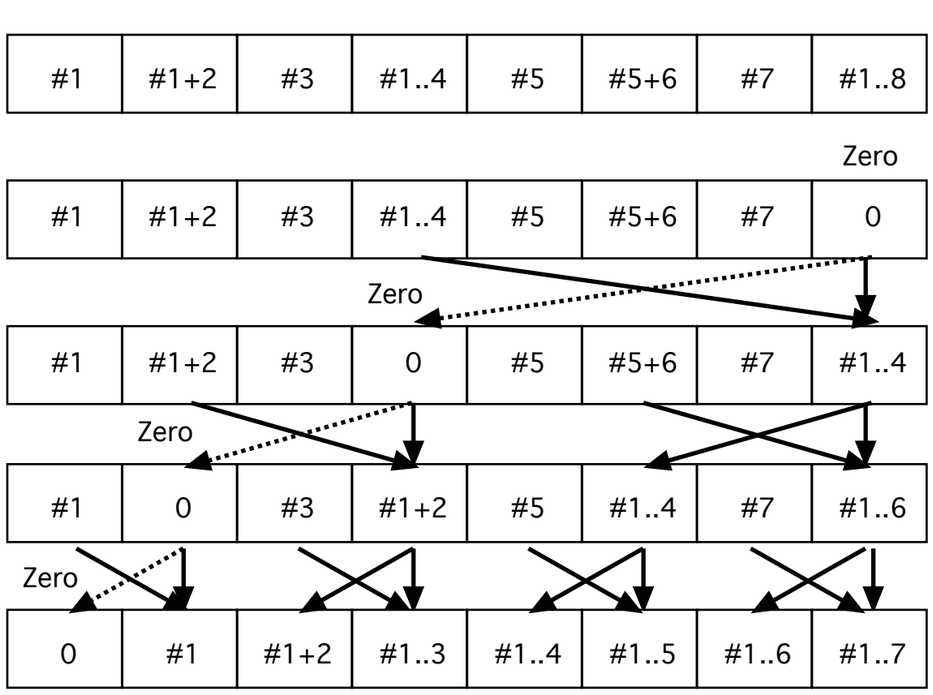
Use parallel prefix sum to create a look-up table for addressing. (Kessler 1.6.3)

Computes sum of all previous items.

Parallel prefix sum

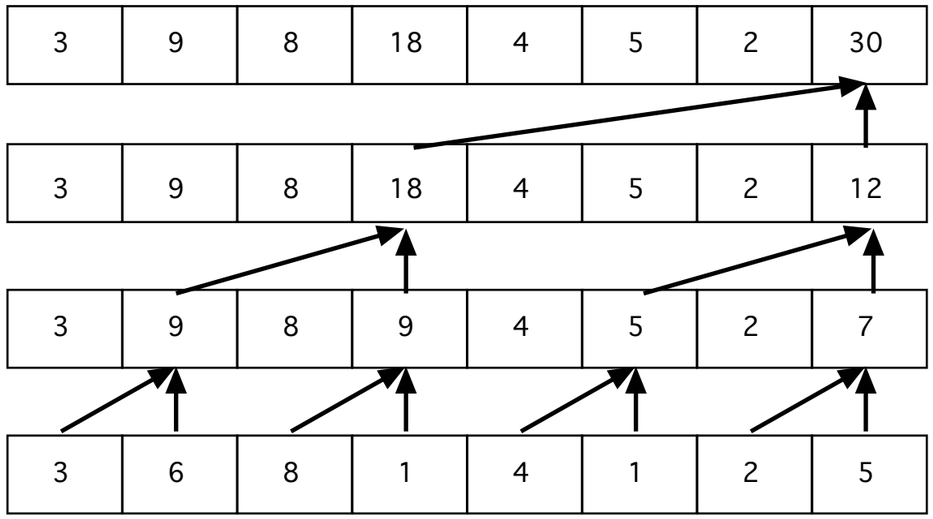
Similar to reduction but full output.

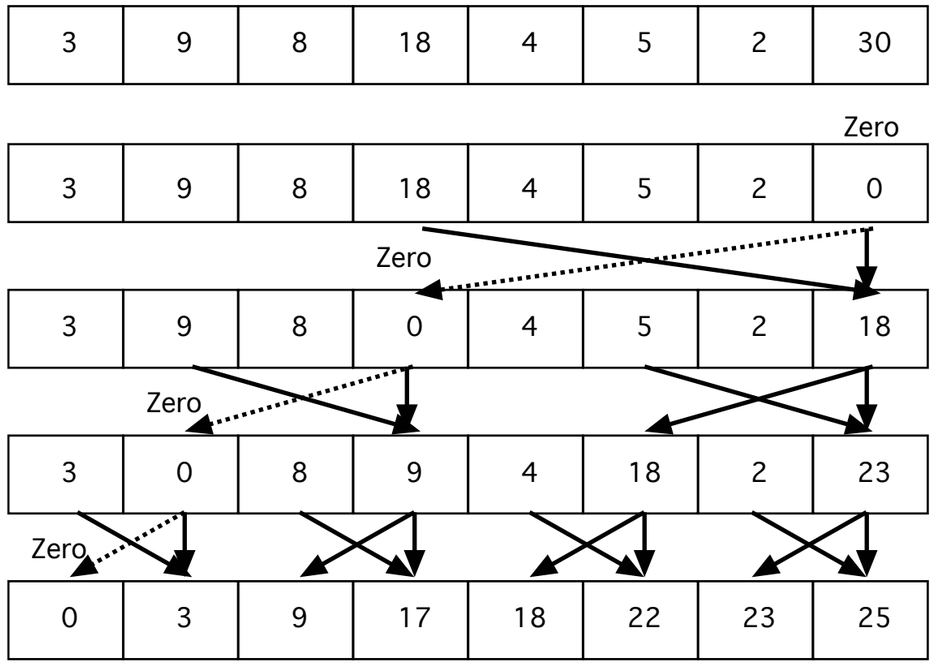




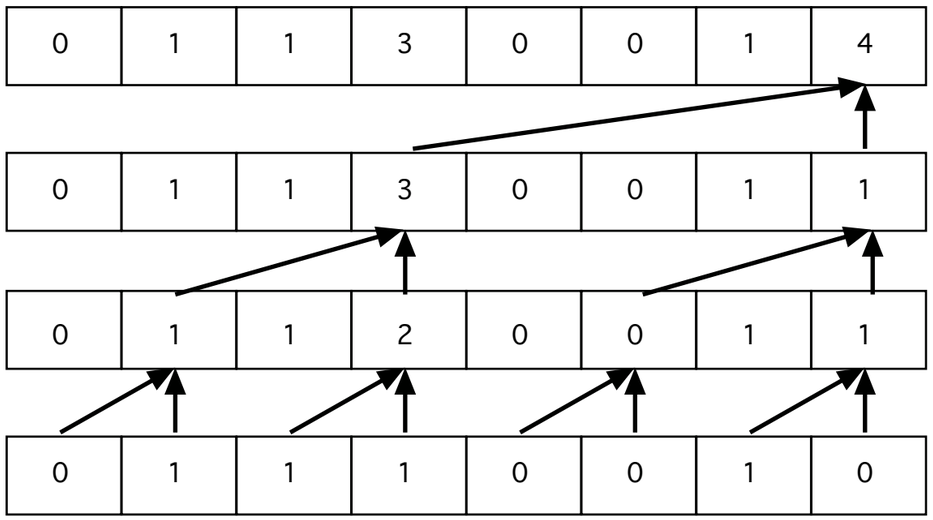
Parallel prefix sum

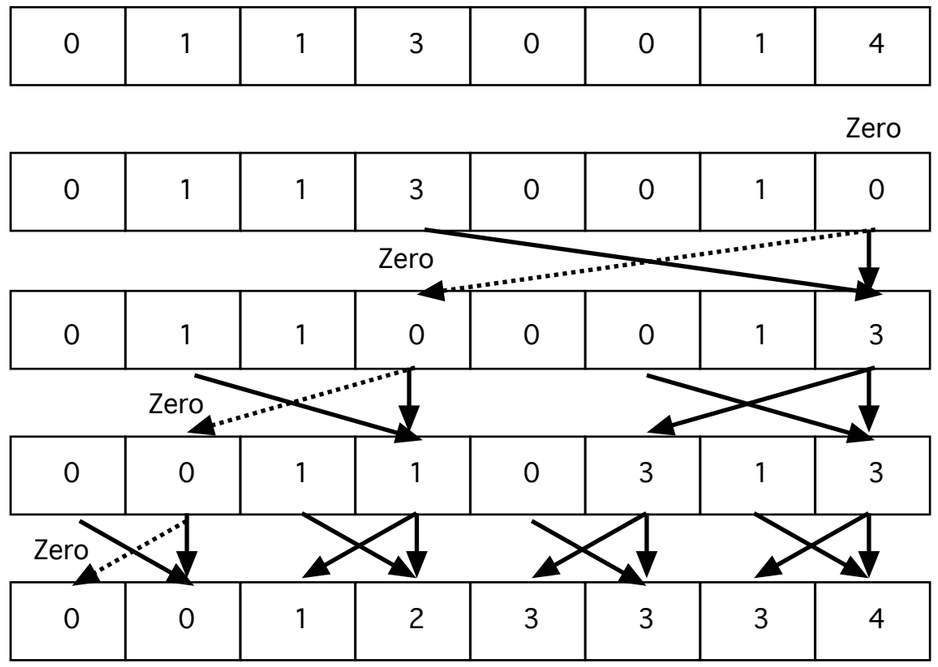
Example





For sorting: Binary parallel prefix sum





Parallel prefix sum on GPU

- **No reason to use few threads. Use as many as you have output items.**

- **Multiple kernel runs to adapt to problem size variation.**

- **As described above, non-coalesced. Pack intermediate values for coalescing. If using shared memory, risk of bank conflicts. [Capannini]**

Thus, QuickSort is not impossible, but more complex than before.

Note:

GPUs have Compare-And-Swap atomics!

GPUs favor massive numbers of threads. One thread per comparison is more than OK!

Implementations available. Example:

https://sourceforge.net/projects/cuda-quicksort/

See also Kessler Ch 2

Recursion

GPUs can't do recursion efficiently… or can they?

Since Kepler we have *concurrent kernels*

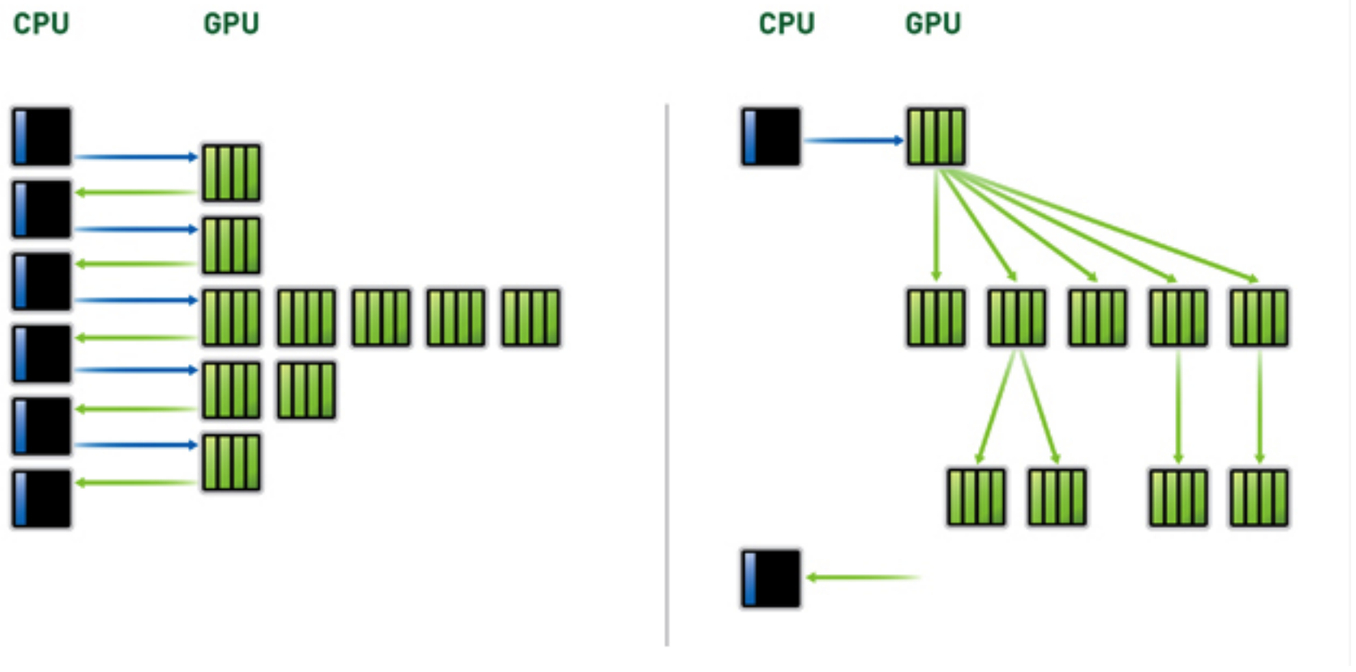
Not only a matter of launching kernels from CPU!

A kernel can spawn new kernels!

Do recursion by spawning new kernels!

Concurrent kernels, Dynamic Parallelism

Less work for the CPU to manage the computation.



Recursion can look like this:

```
__global__ void quicksort(int *data, int left, int right)
{
    int nleft, nright;
    cudaStream_t s1, s2;

    // Partitions data based on pivot of first element.
    // Returns counts in nleft & nright
    partition(data+left, data+right, data[left], nleft, nright);

    // If a sub-array needs sorting, launch a new grid for it.
    // Note use of streams to get concurrency between sub-sorts
    if(left < nright) {
        cudaStreamCreateWithFlags(&s1, cudaStreamNonBlocking);
        quicksort<<< ..., s1 >>>(data, left, nright);
    }
    if(nleft < right) {
        cudaStreamCreateWithFlags(&s2, cudaStreamNonBlocking);
        quicksort<<< ..., s2 >>>(data, nleft, right);
    }
}

__host__ void launch_quicksort(int *data, int count)
{
    quicksort<<< ... >>>(data, 0, count-1);
}
```

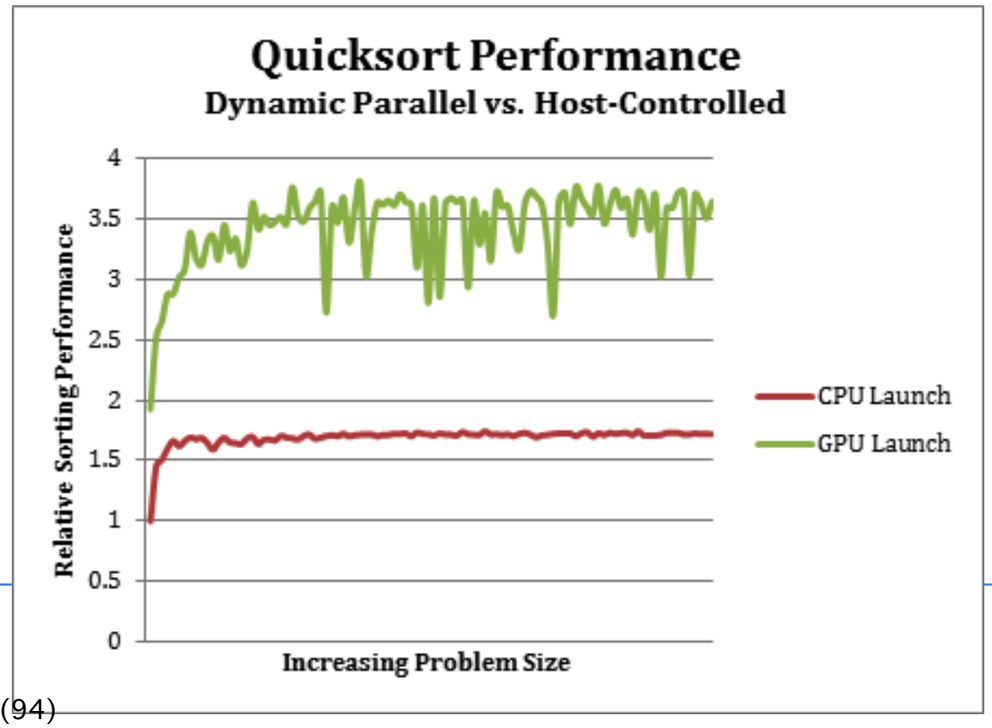
But… does this really do a good job on partitioning?

Source: http://blogs.nvidia.com/blog/2012/09/12/how-tesla-k20-speeds-up-quicksort-a-familiar-comp-sci-code/

Advantages

- **Less work for CPU**
- **Less synchronizing (from CPU side)**
- **Easier programming!**

They claim it matters this much (but your milage will vary)



Recursive CUDA kernels, a significant improvement

Other non-trivial algorithms

FFT, Fast Fourier Transform

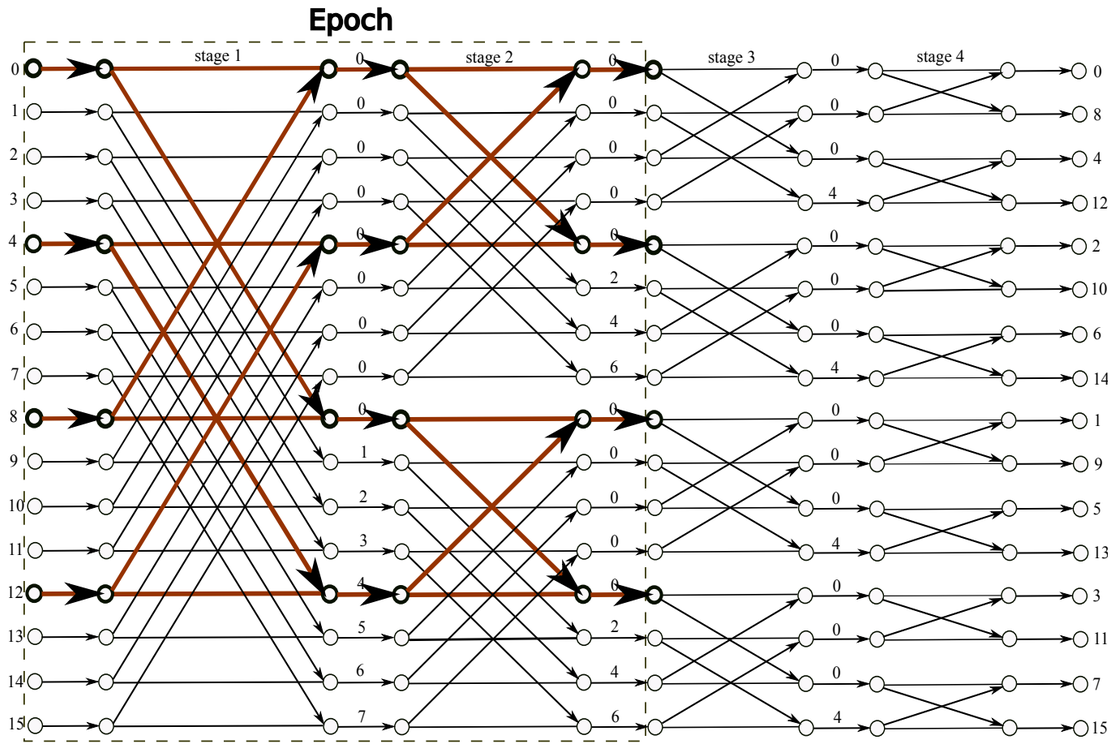
Distance transform

Fractal Brownian Motion

Fast Fourier Transform

Based on a sequence of "butterflies"

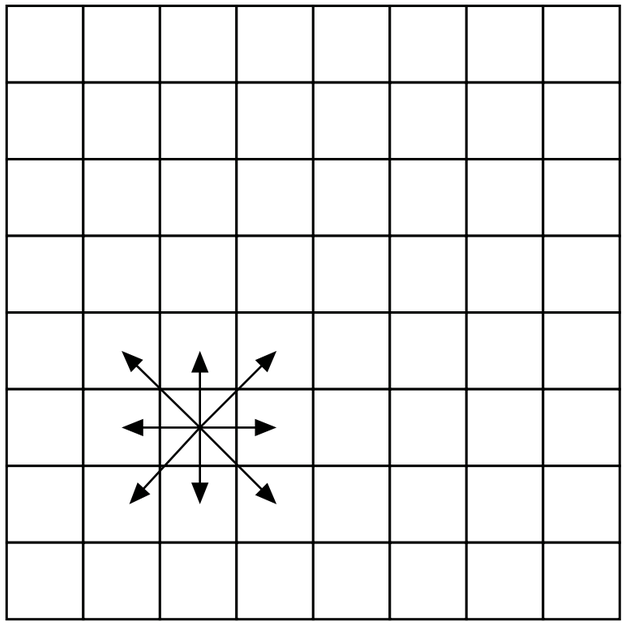
Similarily to Bitonic sort, can be computed several stage in one run for the "smaller" stages

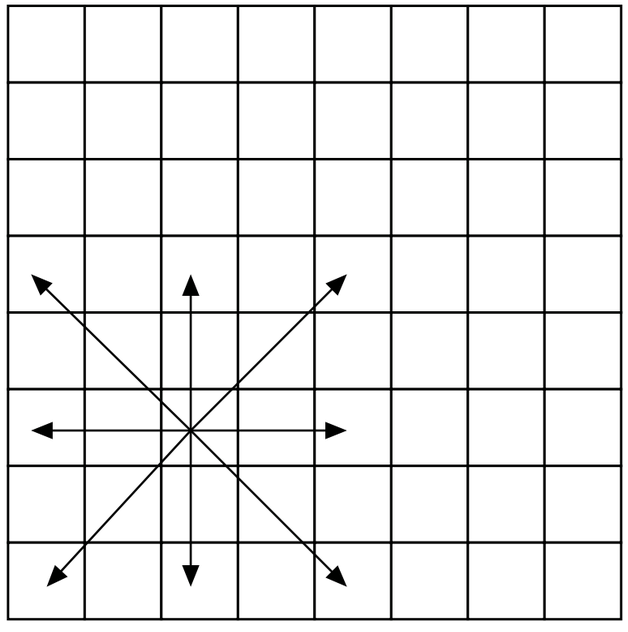


Distance transform

Fast and simple version by Danielsson 1980: "Jump flooding"

Makes "jumps" of various length





Every "jump" need to be one kernel run!

Fractal Brownian Motion

Used for e.g. realistic looking procedural terrains

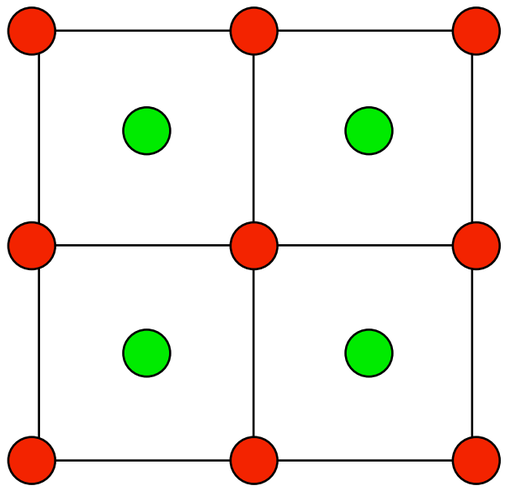
Among other methods:

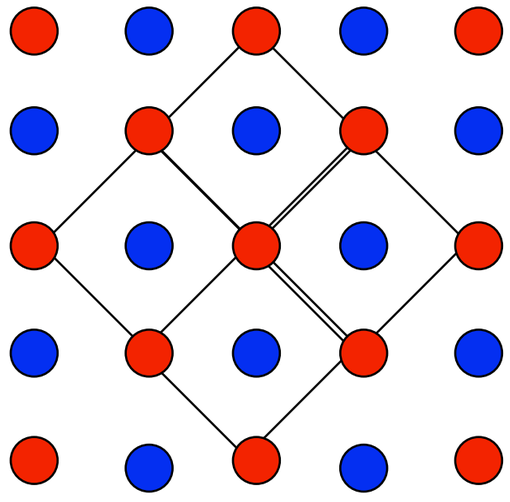
- **Diamond-square**
- **Multi-pass Perlin noise**

Diamond-square algorithm

1) Midpoint from corners

2) Edge from corners and midpoints





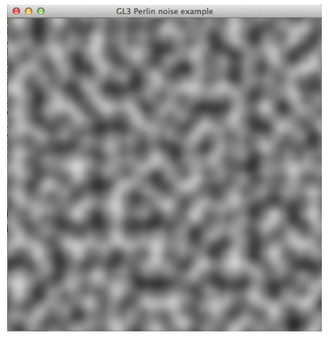
Repeat to desired resolution

Multi-pass Perlin noise

Theoretically slower than Diamond-square

BUT

can be computed by independent threads! One kernel run!



Single octave

Needs log N passes of different frequency

Conclusion

Algorithms with dependency in computed data often need multiple kernel runs.

This is an extra cost!

Does it pay when the computational complexity is lower?

That's all folks!