Lecture 12

Even more CUDA memory

Histogram

Sorting on GPU

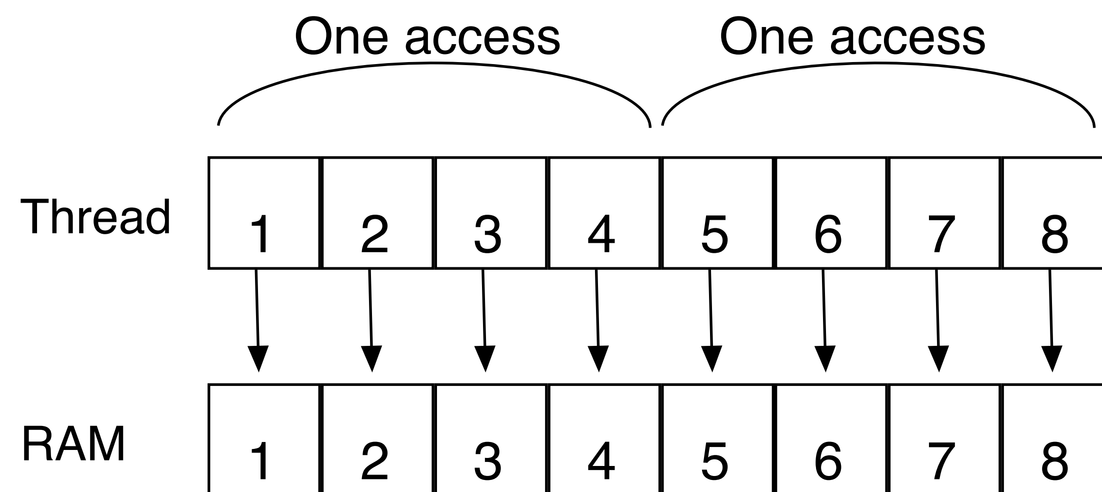
Other problems

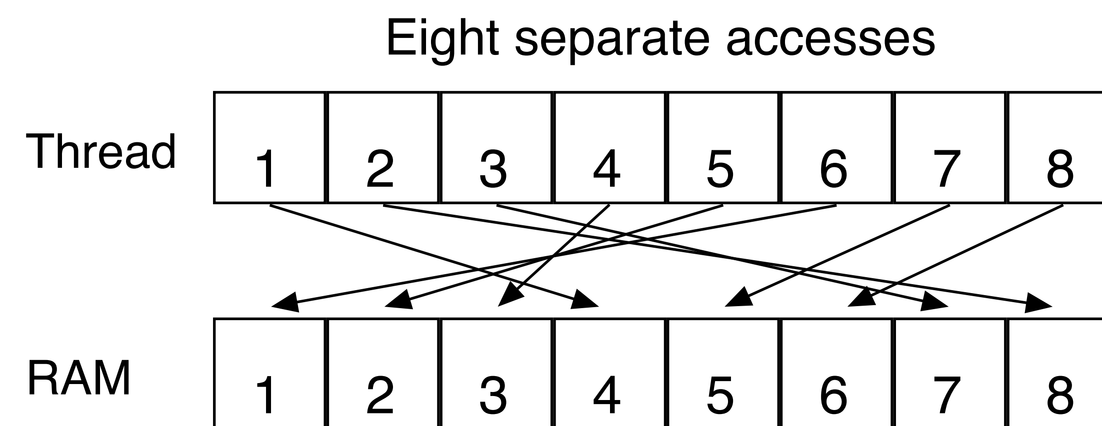
Last time

- Coalescing
- Constant memory
- Texture memory
- Reduction

Coalescing - access in order

Example: Assume that we can get 4 data items per transaction.





Good!

Bad!

Reduction - many to few

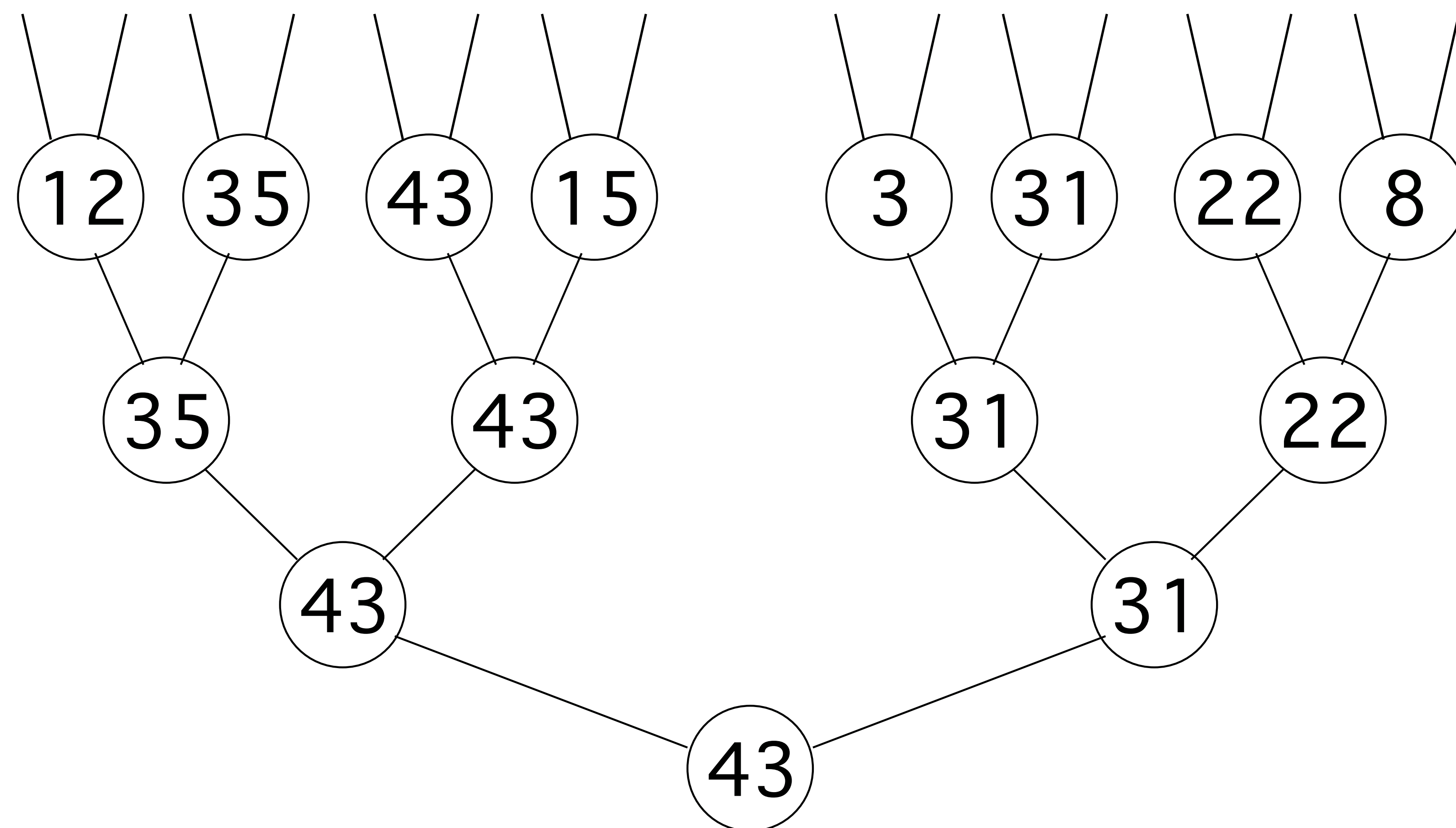
Problems that are tricky to run in parallel.

Acceleration can be limited or nonexistant for small datasets.

Find miniumum, maximum, average...

Tree-based reduction

Break down the problem in small parallel parts.



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!

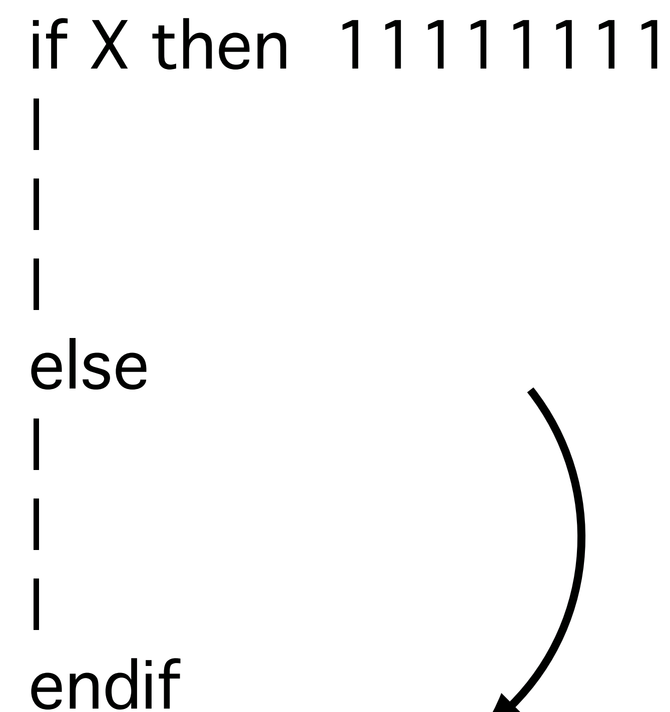
Can not be avoided within warps if a single thread gets a different result from others. Can be avoided if all threads in warp take same branch

Divergent warp

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

Non-divergent warp

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



Reduction in shared

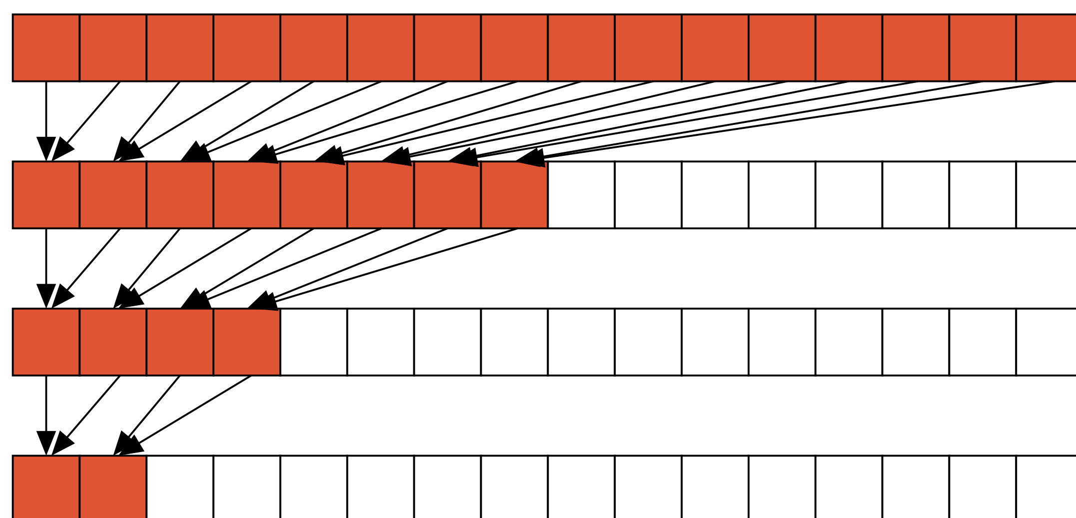
Opposite case to global memory.

In-place is possible.

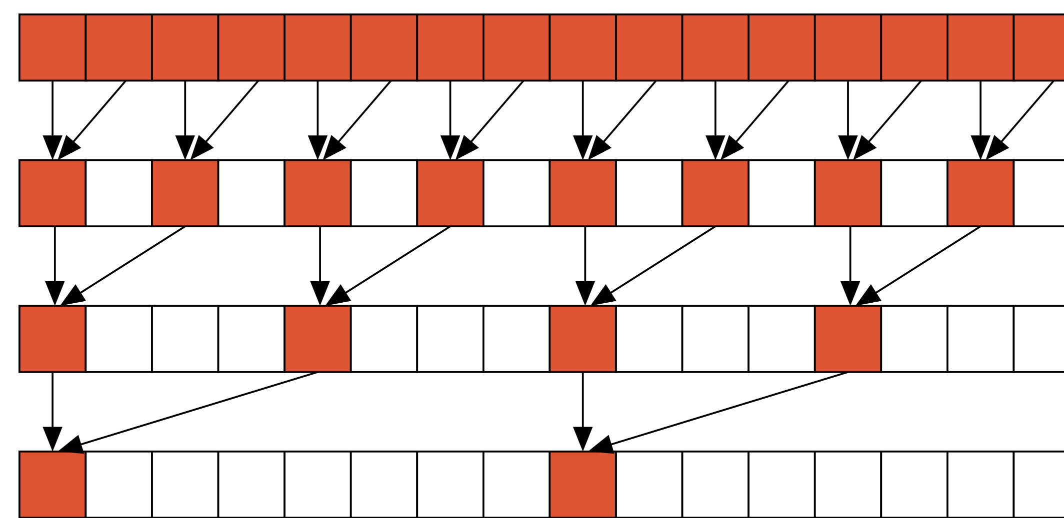
Make sure that threads get idle in consecutive groups.

Entire warp idle - low cost.

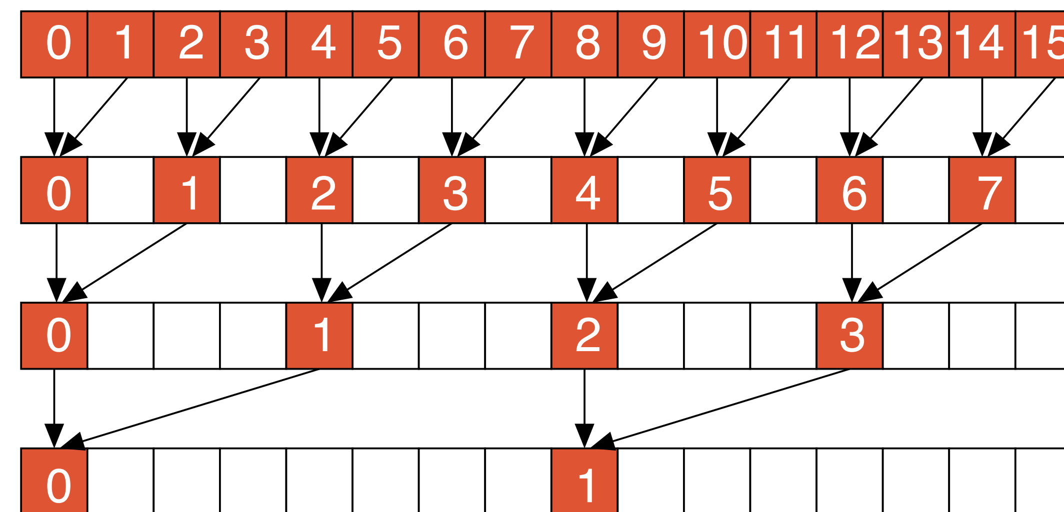
OK in shared:



Not in-place



In-place



Sequential threads - same warps!

Approach for reduction in shared - good opportunity for very fast reduction!

Picture removed

Lecture questions

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

2) Bitonic merge sort is $N\log^2 N$ and QuickSort is $N\log N$. Why they will still have similar complexity on the GPU?

3) What is the reason to use pinned memory?

4) What problem does atomics solve?

CUDA clones

Projects to run CUDA code on other platforms (not complete list):

- mcuda
- Ocelot
- Zluda

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!

Copy to GPU or copy to __managed__, same thing.

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

Atomic operations

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

Available in CUDA from Compute model 1.1 (which means everywhere).

Specify compute model with

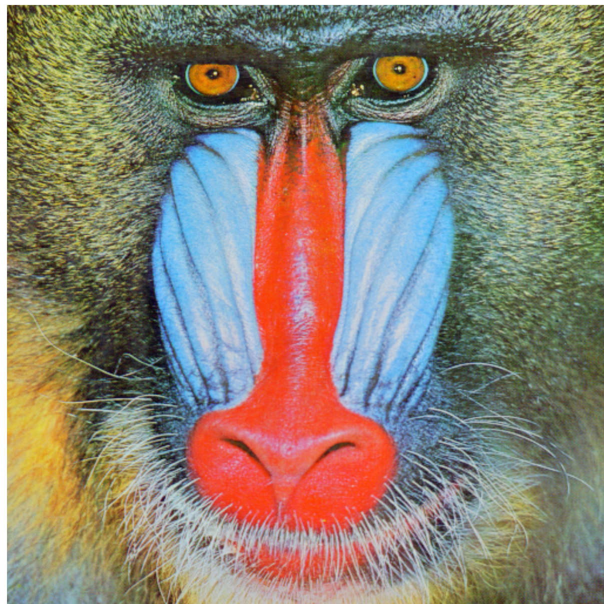
-arch compute_11

but you probably don't have to. (I didn't.)

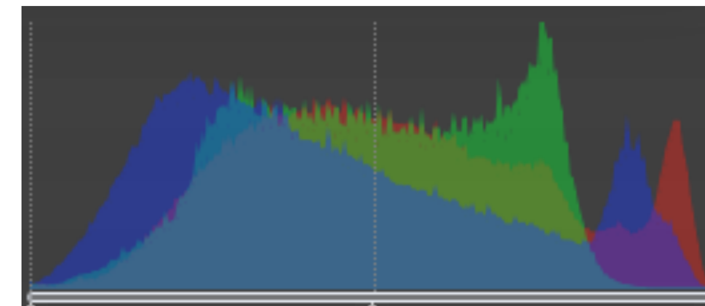
Example: Histogram

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 in parallel

Each thread reads a[threadIdx]

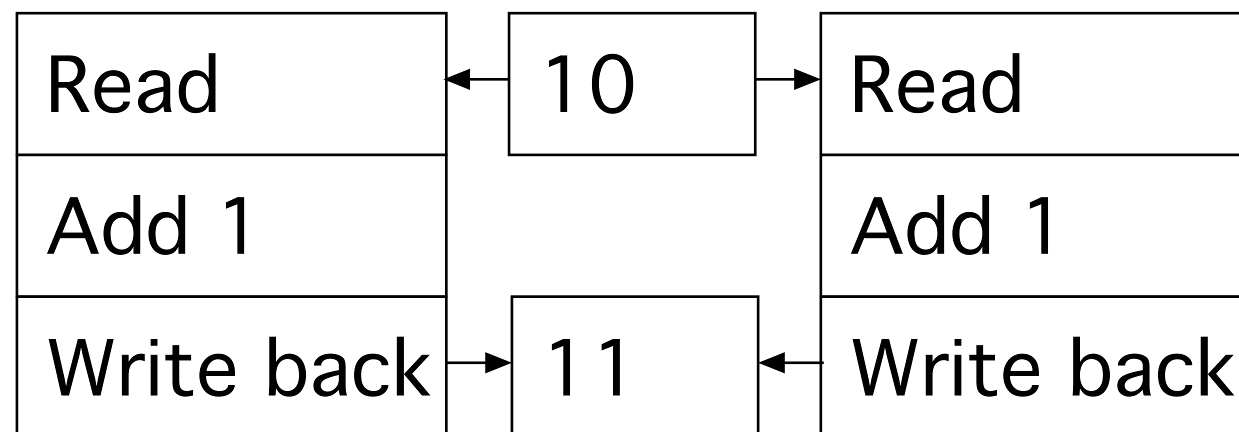
and perform h[a[threadIdx]] = h[a[threadIdx]] + 1

...not

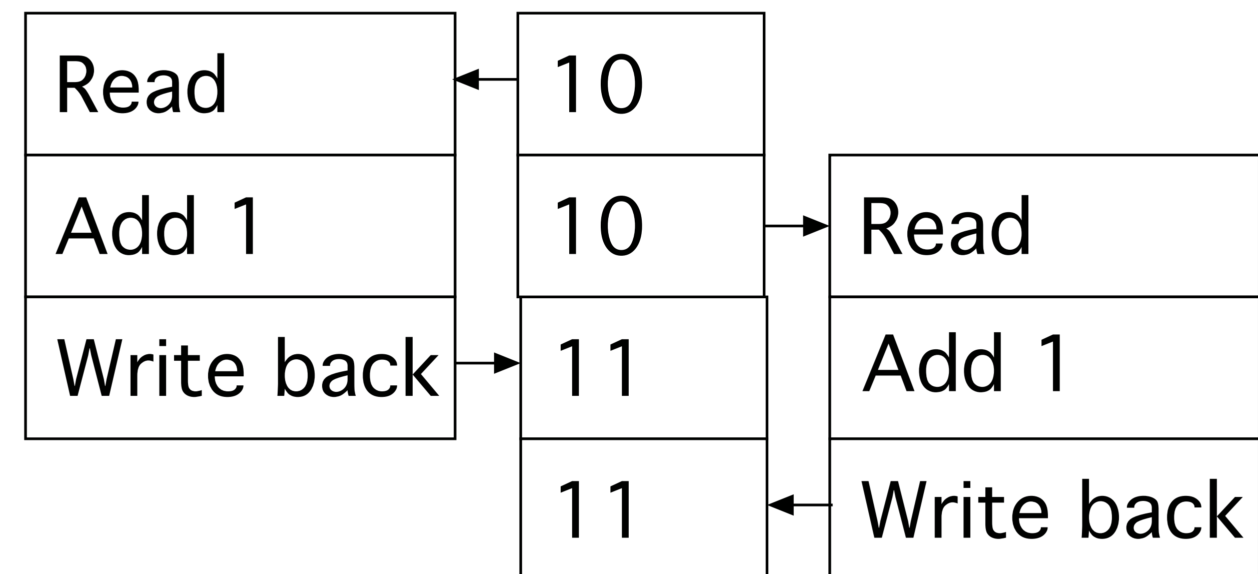
Histogram memory conflicts

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

Supported for both global and shared memory.

But it comes for a cost!

Slower than other operations

Simpler but slower than reduction solutions!

...except that shared memory is fast!

How would I do histograms?

Atomics are fairly OK... 256 lanes of parallelism.

Split to parts for separate blocks.

Produce one histogram for each part.

Merge result.

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: Split to blocks, compute per block in shared!

Atomic conclusions

Simplifies some operations

Serializes conflicting operations

Can hurt performance! Use with care!

Atomics to shared memory much faster!

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()

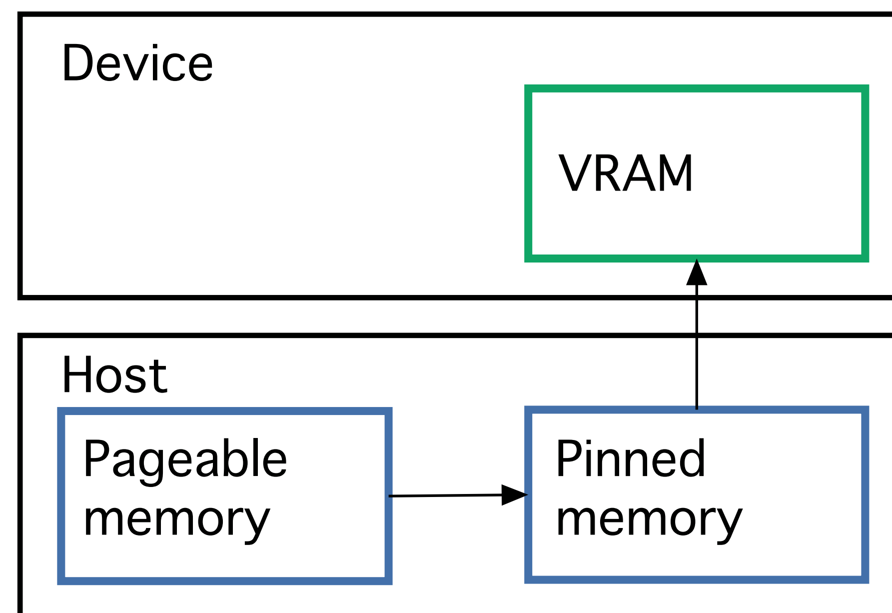
Allocated page-locked memory! Fixed physical location!

Pinned memory

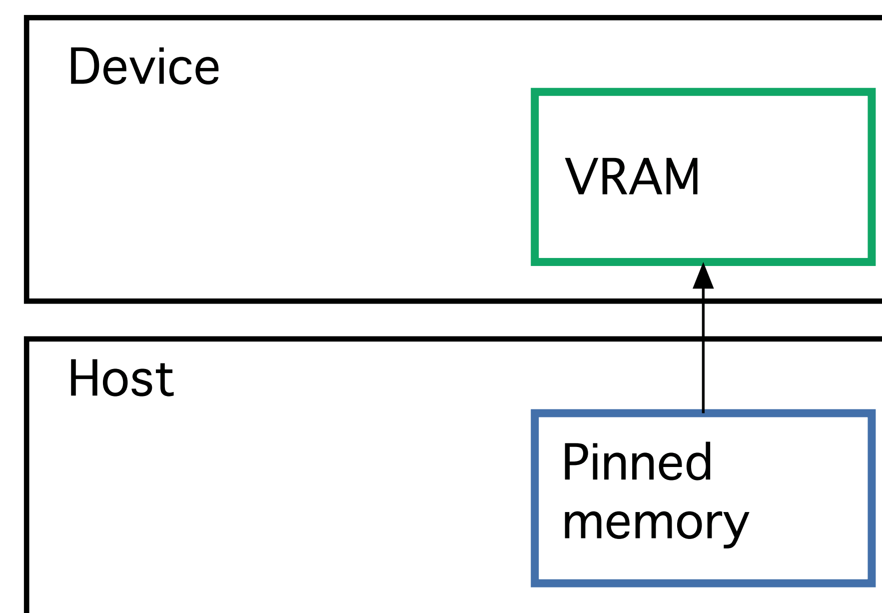
Page-locked memory is a limited resource!

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

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

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!

cuda-gdb

Variant of the GDB debugger

Allows breakpoints and single-stepping CUDA kernels!

Sorting on GPUs

Some not-so-good sorting approaches

Bitonic sort

QuickSort

Concurrent kernels and recursion

Adapt algorithms to parallel execution

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)

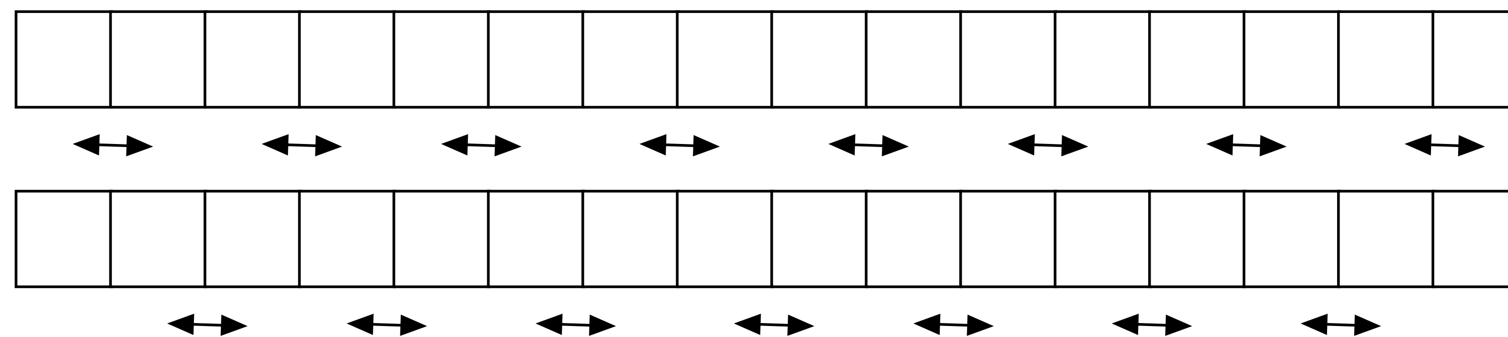
Bubble sort, parallel version

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



Even phase

Odd phase

$O(n^2)$

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

Perfect for sorting many small sets but not one large!

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

Rank sort

Count number of items that are smaller

Values must be unique!

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!

Just as exercise

Rank sort optimization

Everybody want to know what rank they have.

They all need to compare to everything.

For each block of N threads

Split memory in chunks of N

Read chunk shared, one per thread

Synchronize

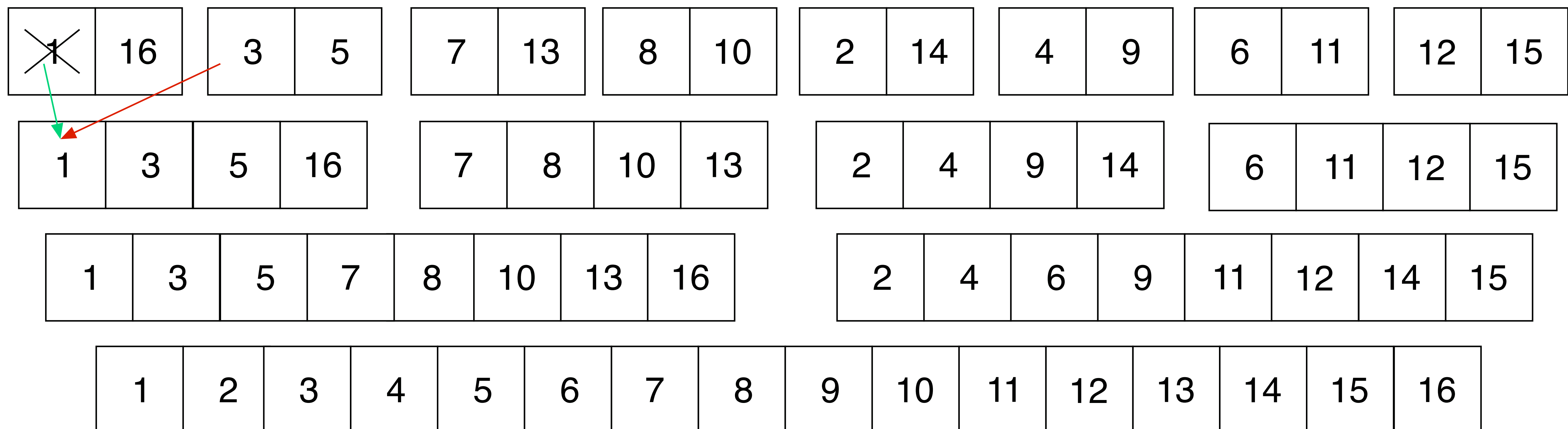
Read through chunk in shared

Writing result is conflict free

Merge sort

Very good sorting algorithm, no worst case.

Sorts bigger and bigger, merges by comparing from end



Compare left end to left end, move lowest to output

Suitable for GPU?

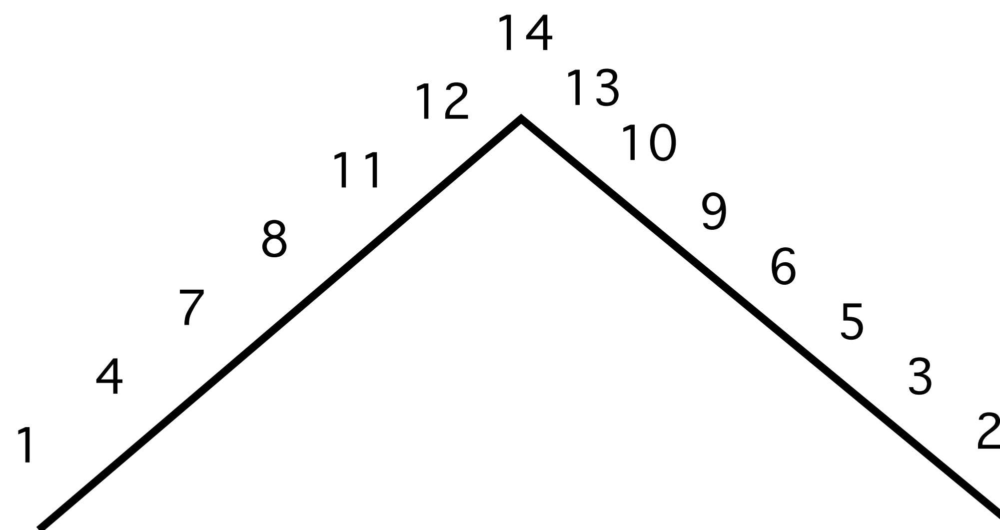
Early parts, very much.

Later parts, less and less!

Last part full size, N/2 vs N/2, can not be done in parallel!

Bitonic merge sort

Bitonic set: Two monotonic parts in different direction.



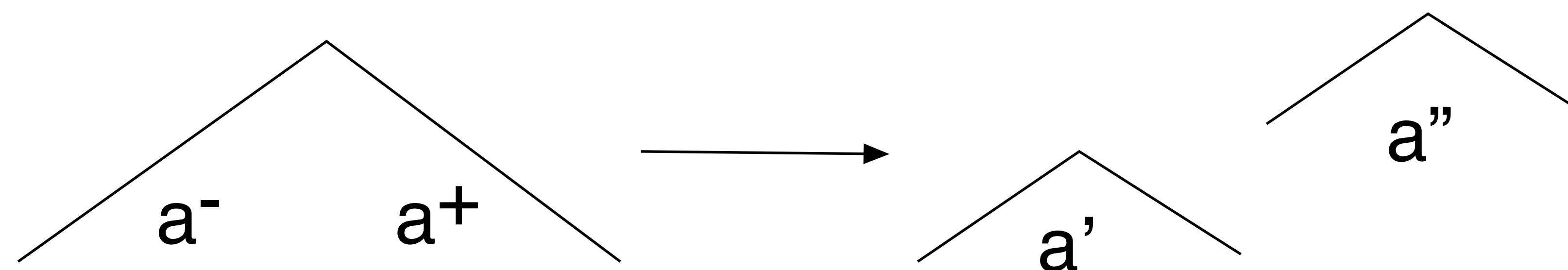
Bitonic merge sort

(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

7	1	1	1	1	1	1
1	7	3	3	3	2	2
8	8	8	7	4	4	3
3	3	7	8	2	3	4
5	5	5	6	6	6	5
6	6	6	5	5	5	6
2	4	4	4	7	7	7
4	2	2	2	8	8	8

Bitonic sort of smaller parts

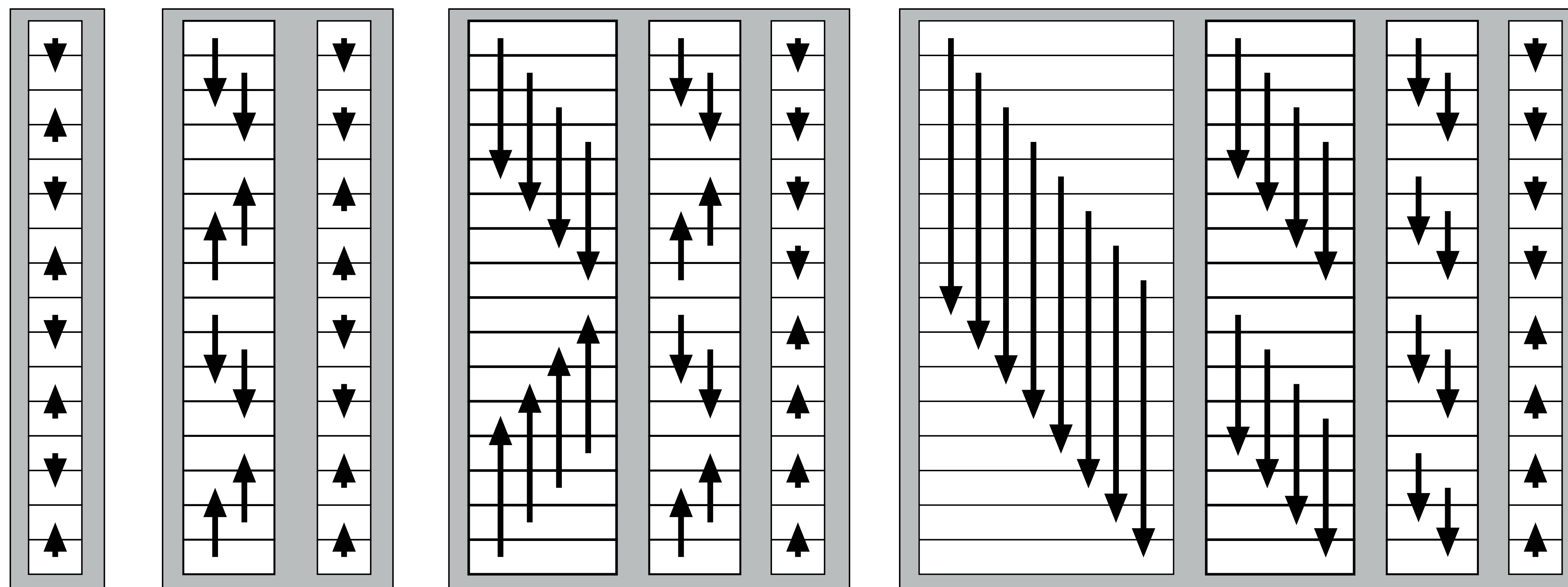
Reverse parts (bitonic merge)

Bitonic sort of main part

Reverse parts (bitonic merge)

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	3	2	2
1	3	12	12	9	9	8	8	4	4	3
3	1	15	15	8	8	9	7	2	3	4
16	9	13	16	16	12	12	6	6	6	5
9	16	16	13	13	13	13	5	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 features

- 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.

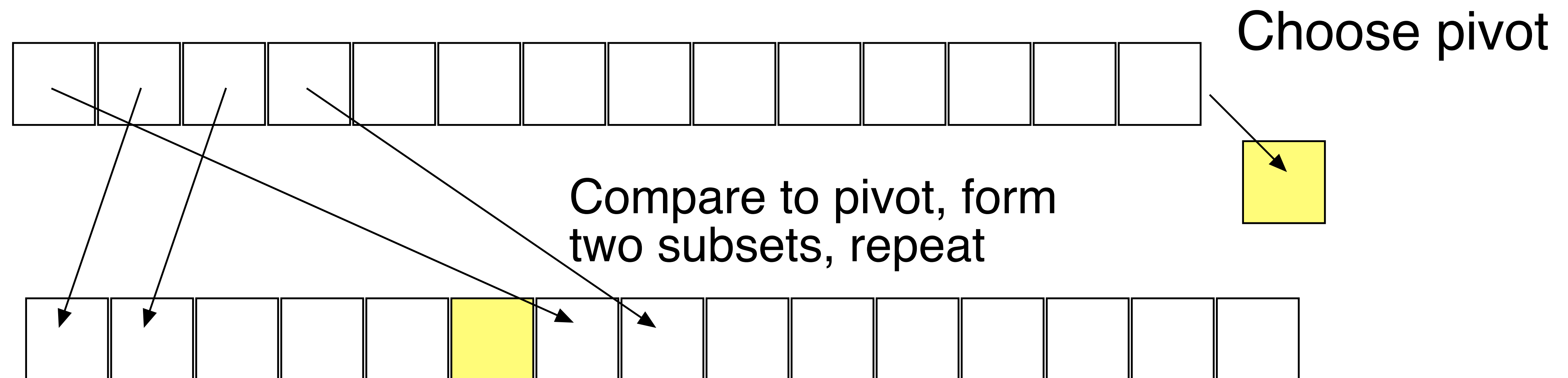
Big leaps (>number of threads/block): No synchronization possible between blocks!

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…

Picture removed

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

Picture removed

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!

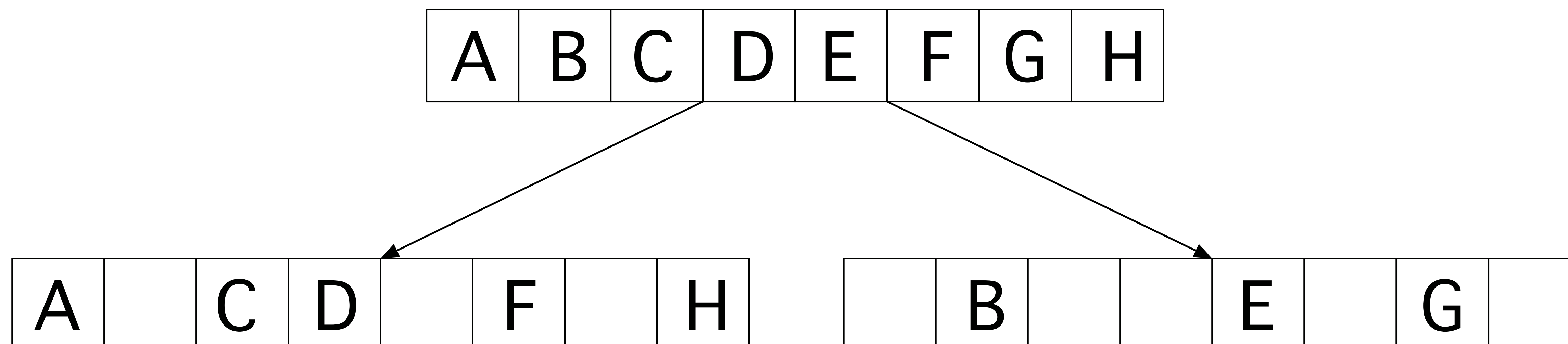
Parallel partitioning 1: Atomic fetch & increment.
(GPUs have atomics!)

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

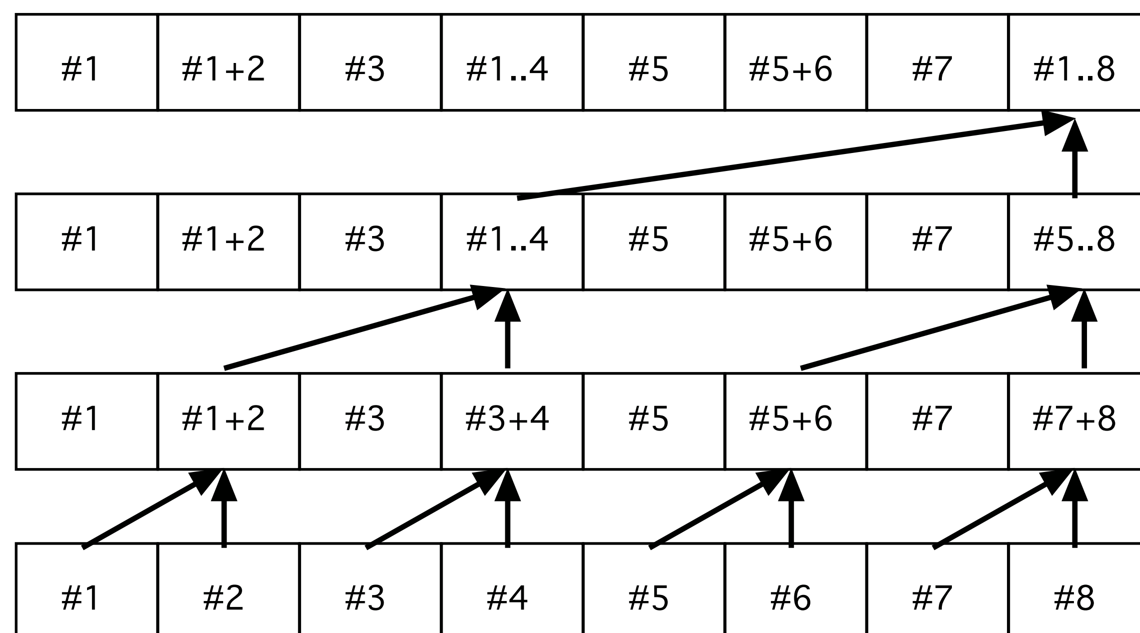
Use *parallel prefix sum* to create a look-up table for addressing.

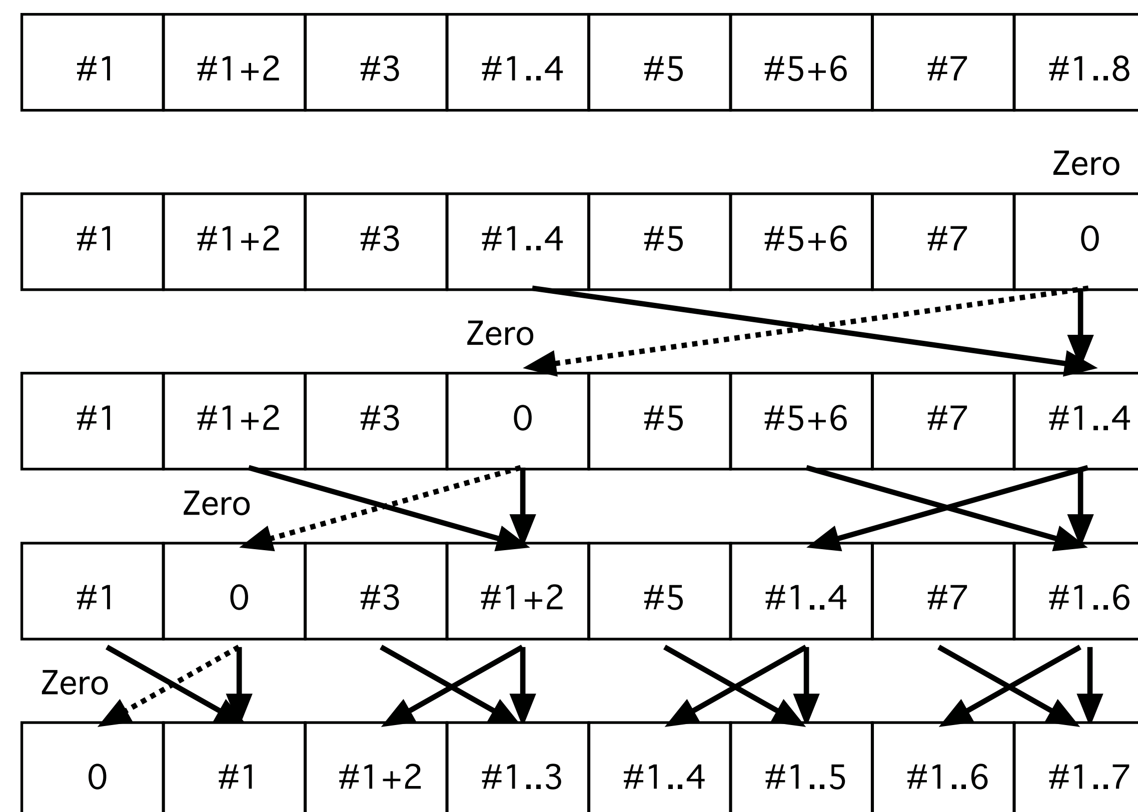
Computes sum of all previous items.

Takes logN steps to perform.

Parallel prefix sum

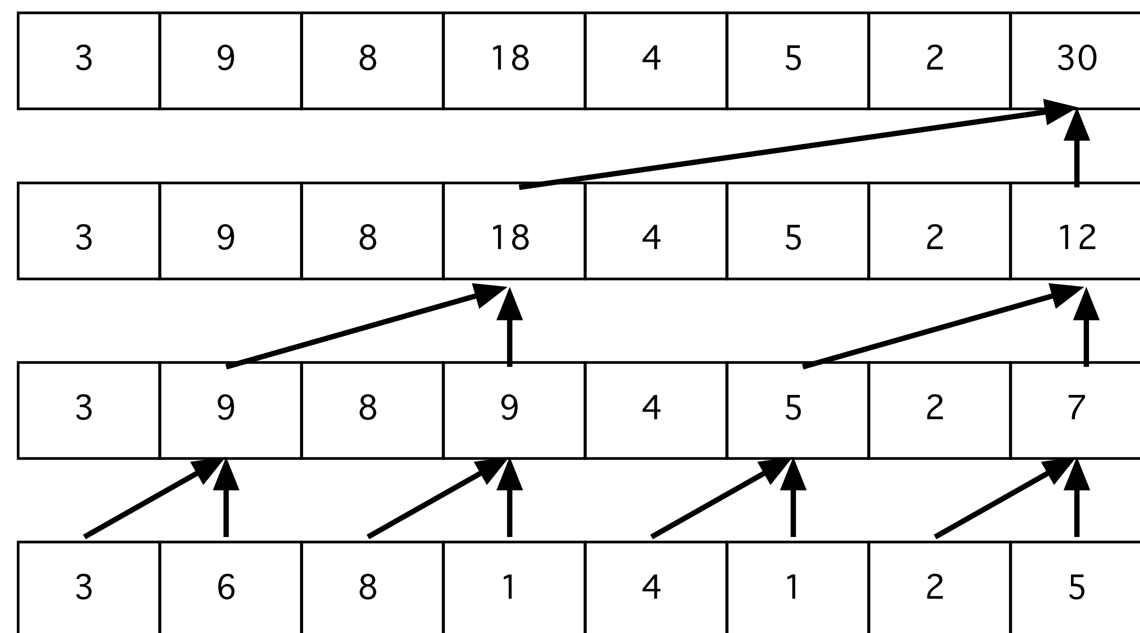
Similar to reduction but full output.

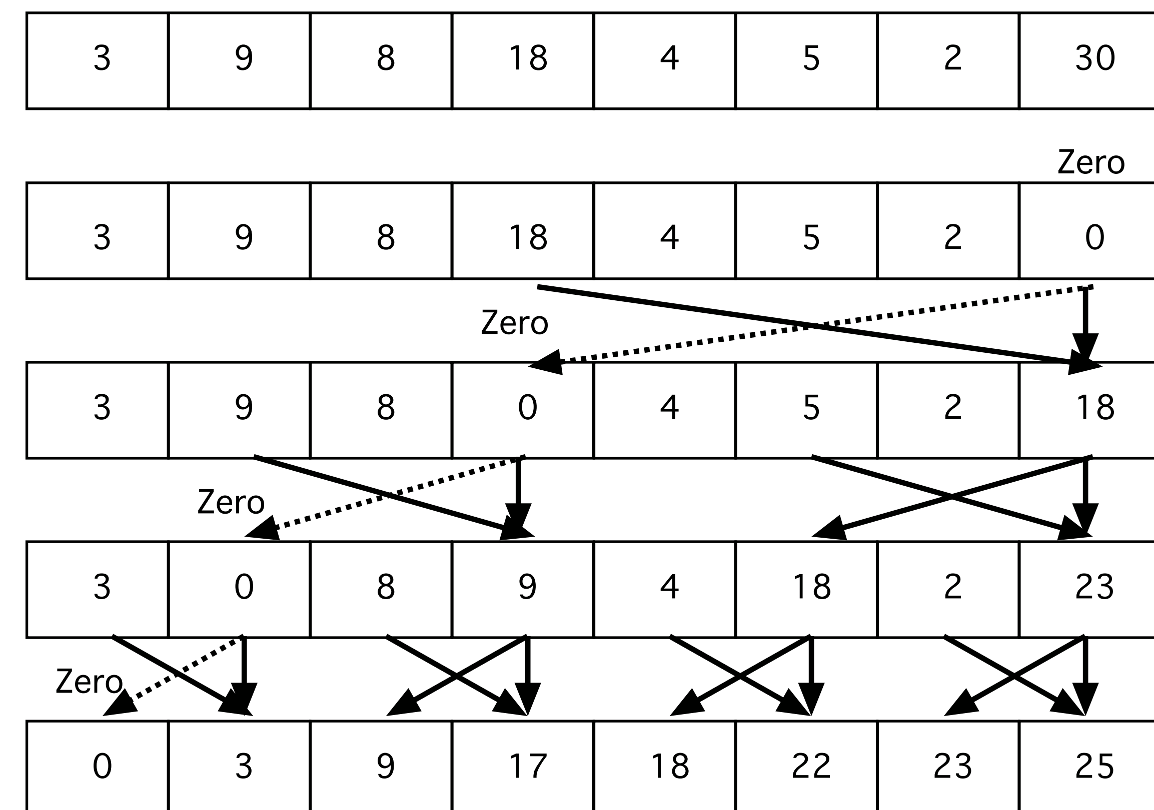




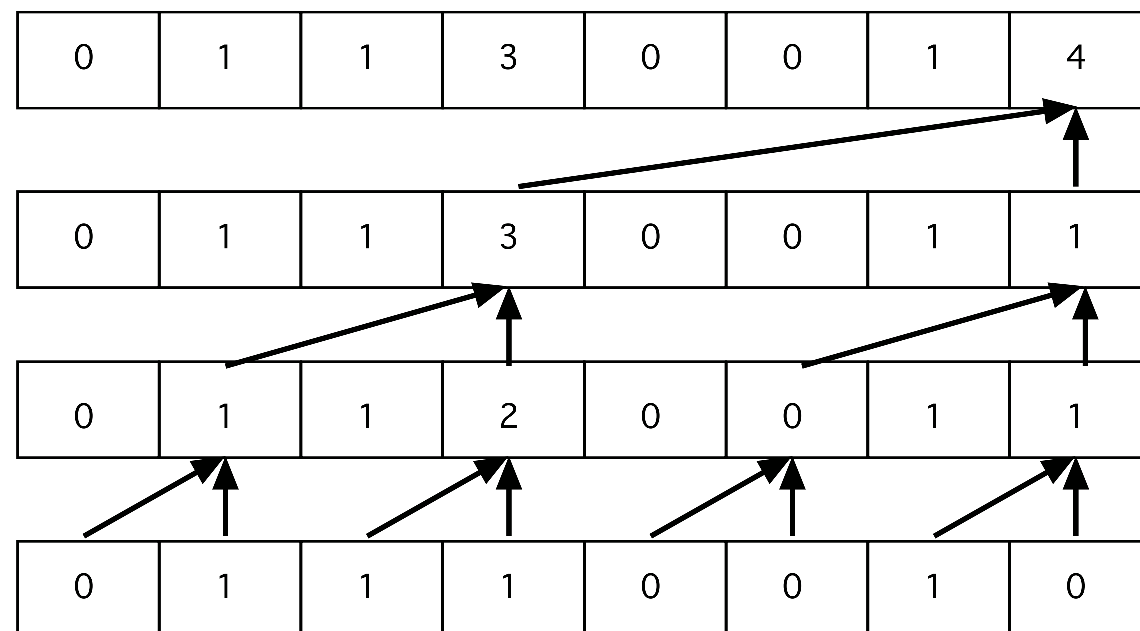
Parallel prefix sum

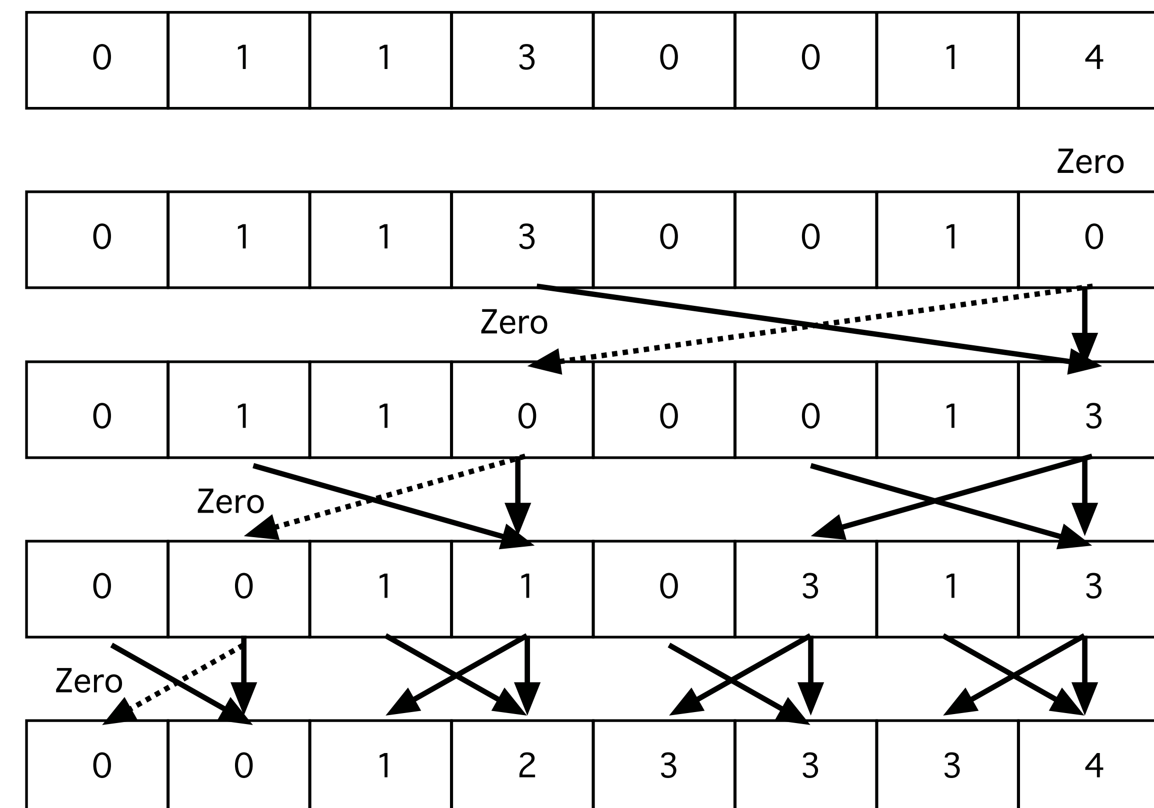
Example





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/

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.

Picture removed

Recursion can look like this:

Picture removed

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)

Picture removed

Recursive CUDA kernels, a significant improvement, powerful option

Many other sorting algorithms exist...
like this one last year:

Picture removed

Other non-trivial algorithms

FFT, Fast Fourier Transform

Distance transform

Fractal Brownian Motion

Fast Fourier Transform

Based on a sequence of "butterflies"

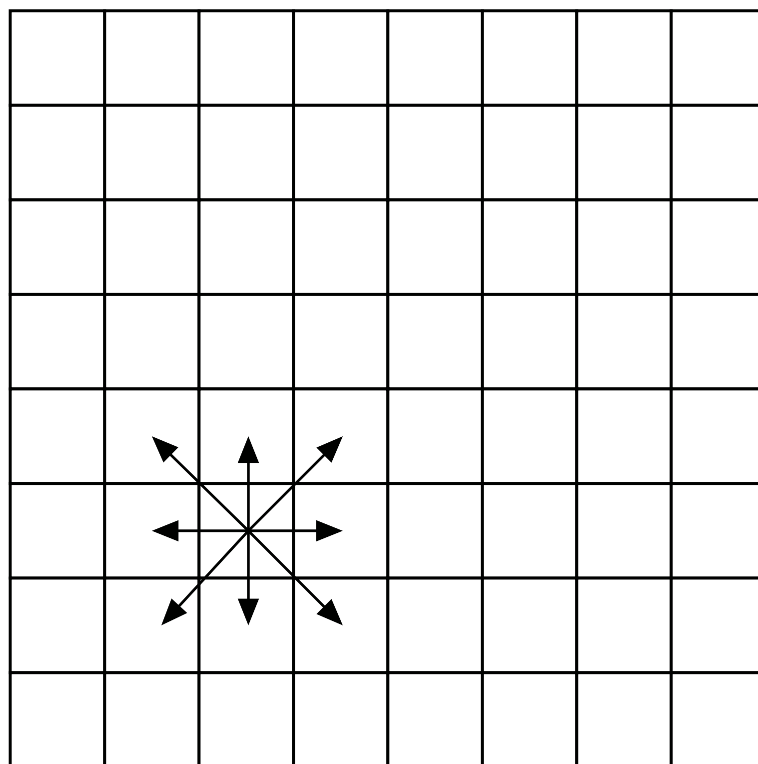
Similarily to Bitonic sort, can be computed several stage in one run for the "smaller" stages

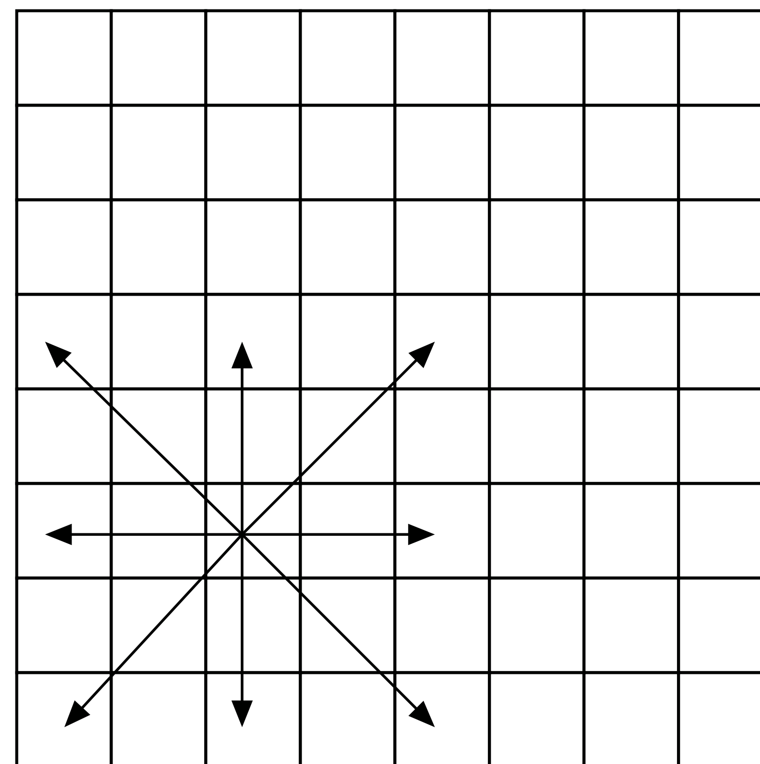
Picture removed

Distance transform

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

Makes "jumps" of various length





Every "jump" needs to be one kernel run!

Fractal Brownian Motion

Used for e.g. realistic looking procedural terrains

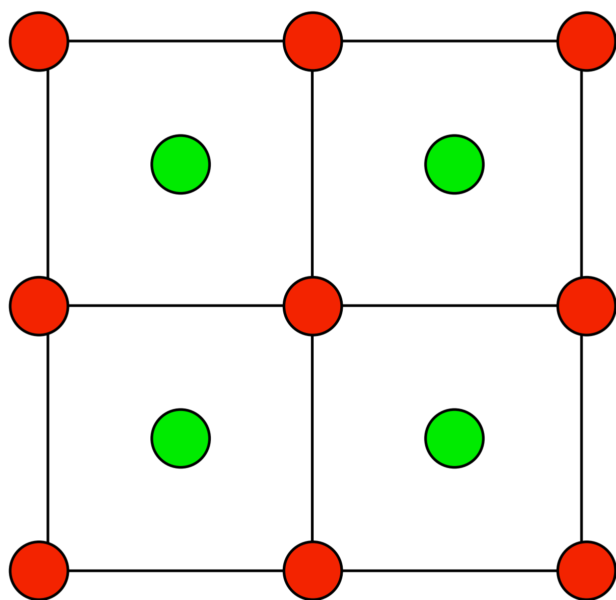
Among other methods:

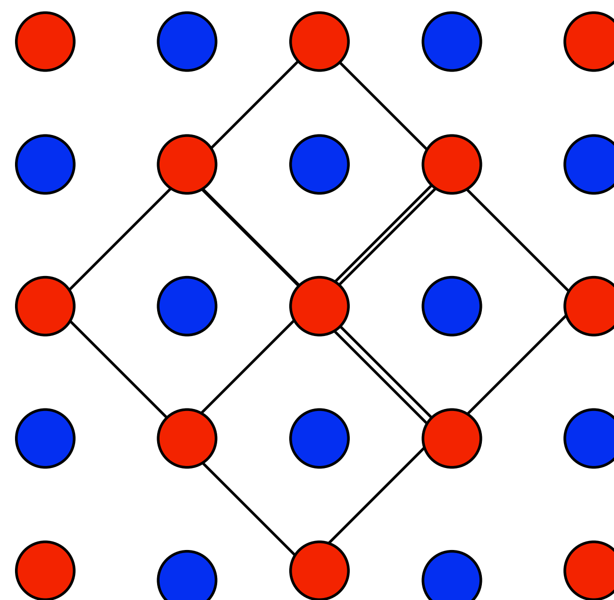
- Diamond-square
- Multi-pass Perlin noise

Diamond-square algorithm

1) Midpoint from corners

2) Edge from corners and midpoints





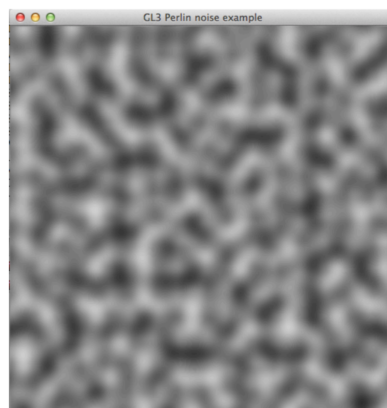
Repeat to desired resolution

Multi-octave Perlin noise

Theoretically slower than Diamond-square

BUT

can be computed by independent threads! One kernel run!



Single octave

FBM 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!