NTNU | Norwegian University of Science and Technology

Thread Lightly

Bart Iver van Blokland

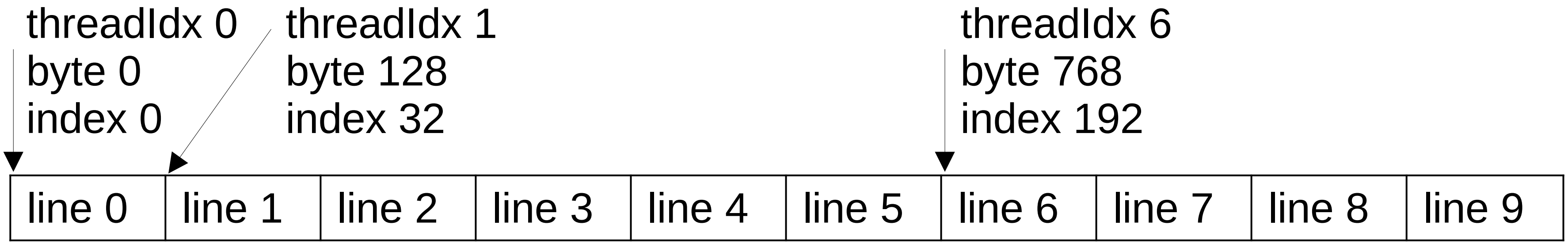
Today

- **Repetition: performance pitfalls and collective instructions**
- Now you’re thinking with warps
- Useful stuff
- Demonstration: CUDA profiling tools

Coalesced memory reads

- Even if you read only a single byte, its containing cache line must be loaded into the core in its entirety

```
__global__ void kernel(int* array, int n) {
    int value = array[32 * threadIdx.x];
    // do stuff with value here
}
```

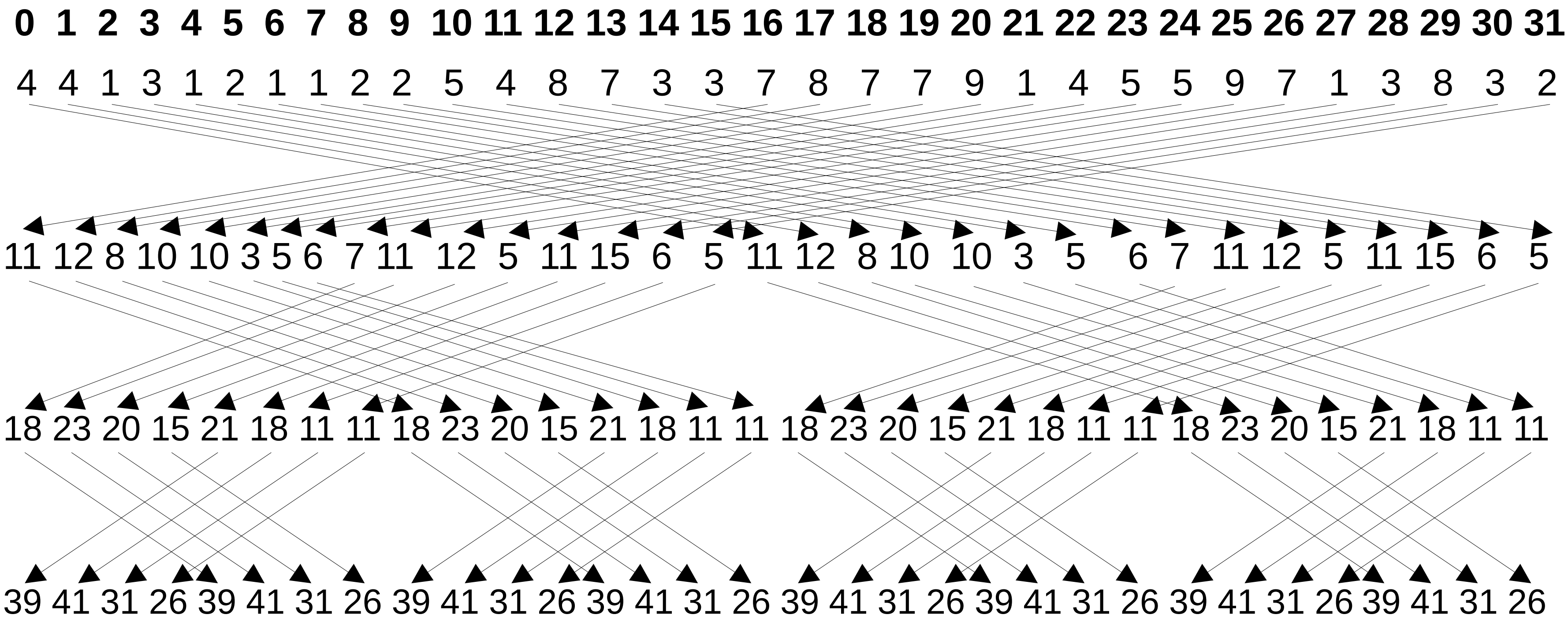


Shuffle instructions

Four variants:

- Read from any thread (index specified by srcLane):
  ```
  T __shfl_sync(unsigned mask, T var, int srcLane);
  ```
- Read from thread (laneid – delta):
  ```
  T __shfl_up_sync(unsigned mask, T var, unsigned int delta);
  ```
- Read from thread (laneid + delta):
  ```
  T __shfl_down_sync(unsigned mask, T var, unsigned int delta);
  ```
- Read from thread (laneid XOR laneMask):
  ```
  T __shfl_xor_sync(unsigned mask, T var, int laneMask);
  ```

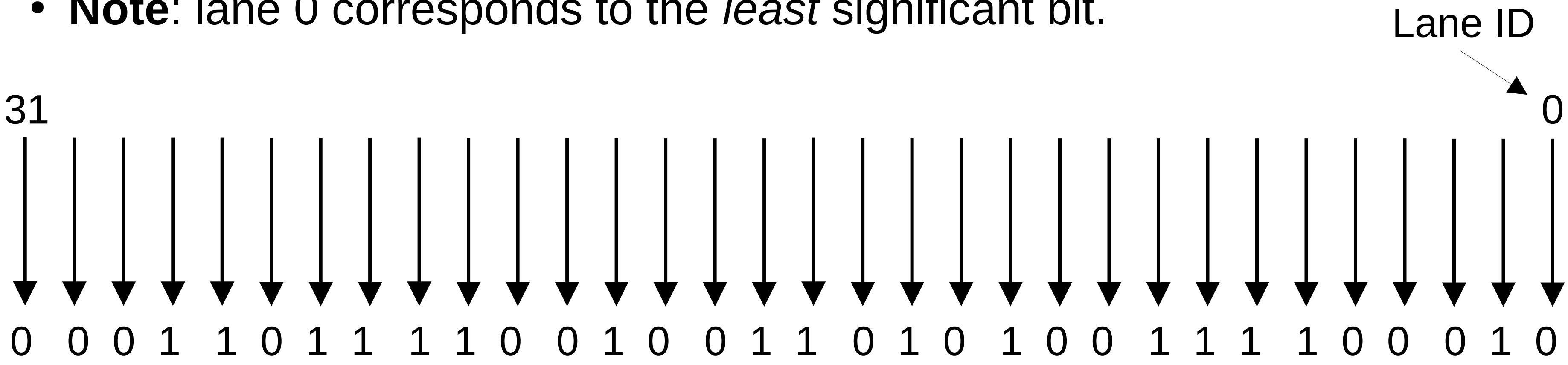
Shuffle instructions




```
sum += __shfl_xor_sync(__activemask(), sum, 4);
```

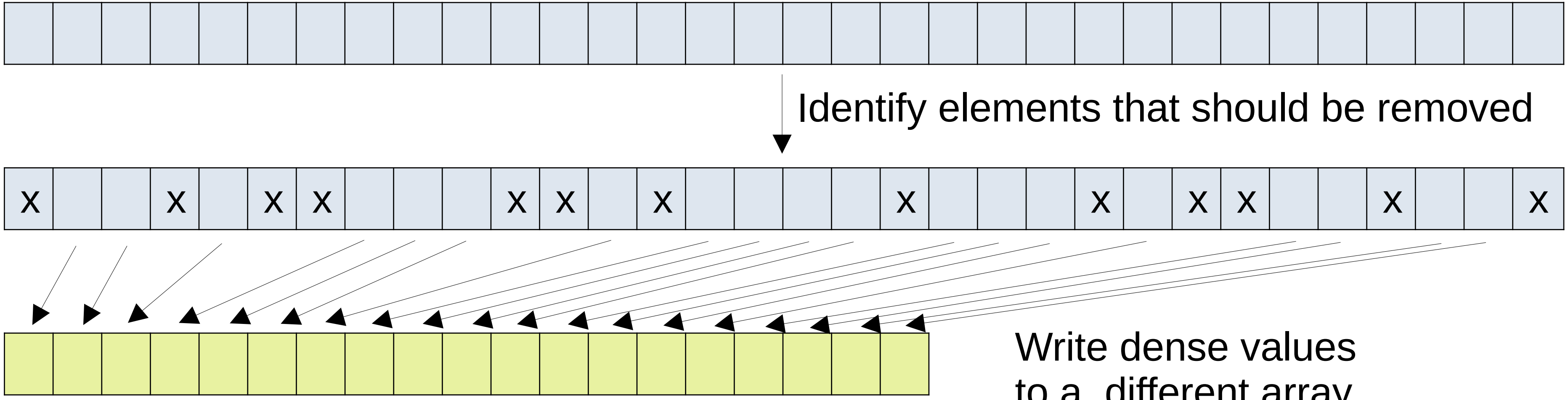
Warp Voting: ballot instruction

- Each thread in the warp sets one bit in a 32-bit integer
- Bit index corresponds to the lane index
- Only active threads vote
- **Note**: lane 0 corresponds to the *least* significant bit.



Warp voting use cases

- Stream filtering



Today

- Repetition: performance pitfalls and collective instructions
- **Now you're thinking with warps**
- Useful stuff
- Demonstration: CUDA profiling tools

Putting it all together: prefix sum

- Each element in the list becomes the sum of all elements up to that point
  - For example, the prefix sum of the sequence:

    11, 9, 4, 19, 16, 12, 3, 15, 11, 14

    is:

    11, 20, 24, 43, 59, 71, 74, 89, 100, 114

- Computing linearly requires fewest computations, but parallel implementation can be much faster

  CPU: 20.7s

  GPU: 7.86ms

Putting it all together: prefix sum

- Let's start with one warp:

```
float value = array[threadIdx.x];
for(int delta = 1; delta < 32; delta *= 2) {
    float sum = __shfl_up_sync(value, delta);
    if(threadIdx.x >= delta) {
        value += sum;
    }
}
```

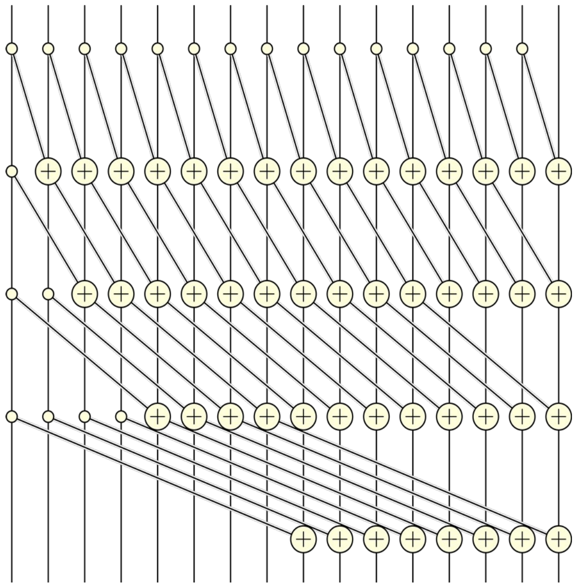
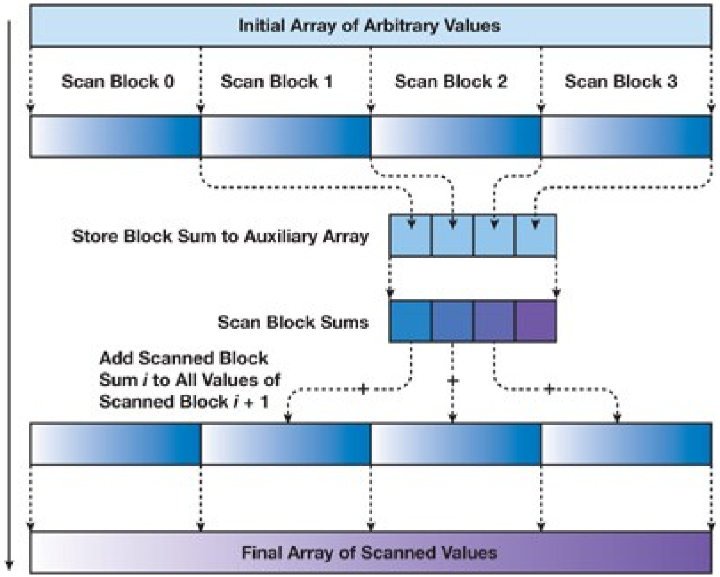


Image by Scott Pakin - Own work, CC BY-SA 4.0

Putting it all together: prefix sum

- Approach: use 3 kernels



Kernel 1: compute partial sums for all elements in each block

Kernel 2: compute partial sums at a block level (can reuse kernel 1)

Kernel 3: add partial block sums to each element within the block

Phase 1: prefix sum in block

4	6	3	8	9	3	8	5	8	3	3	8	9	1	6	4	5	8	8	4	2

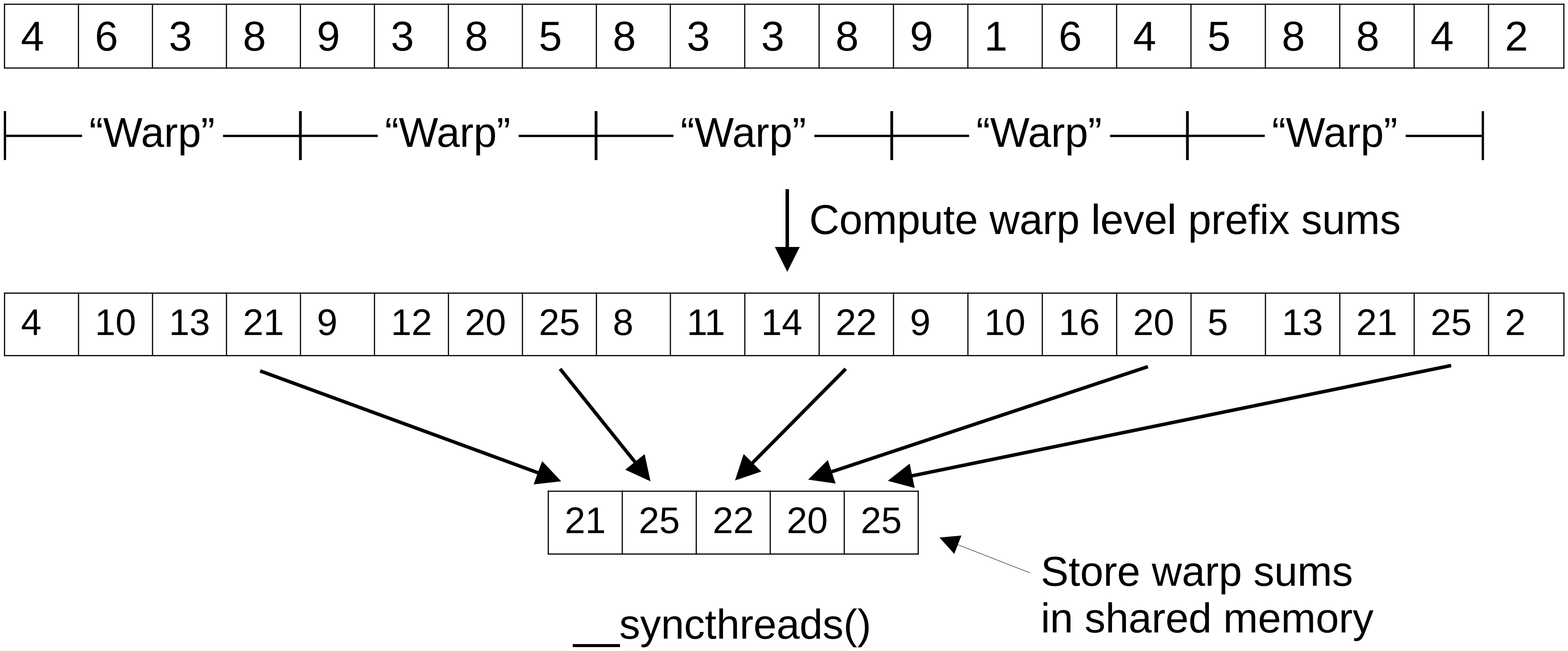
“Warp” | “Warp” | “Warp” | “Warp” | “Warp”

↓ Compute warp level prefix sums

4	10	13	21	9	12	20	25	8	11	14	22	9	10	16	20	5	13	21	25	2

- We’re launching one thread per element in the input array

Phase 1: prefix sum in block



Phase 1: prefix sum in block

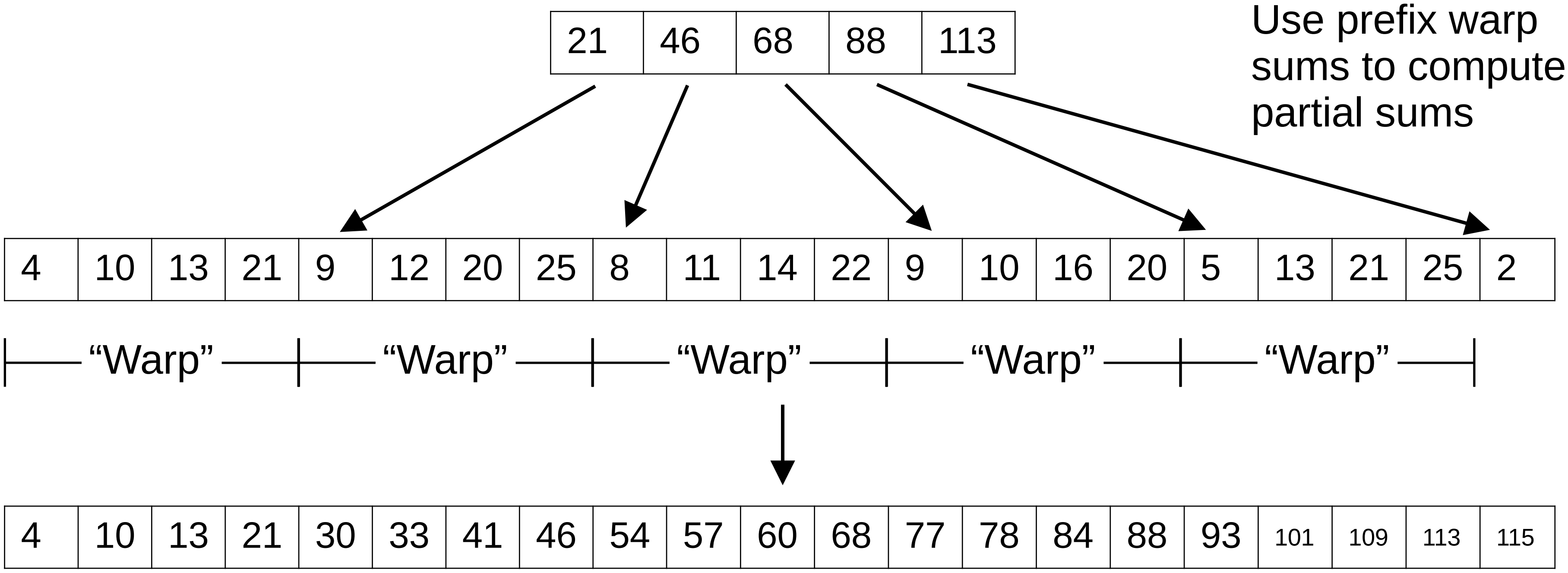
21	25	22	20	25

↓ Compute prefix sum for warp sums

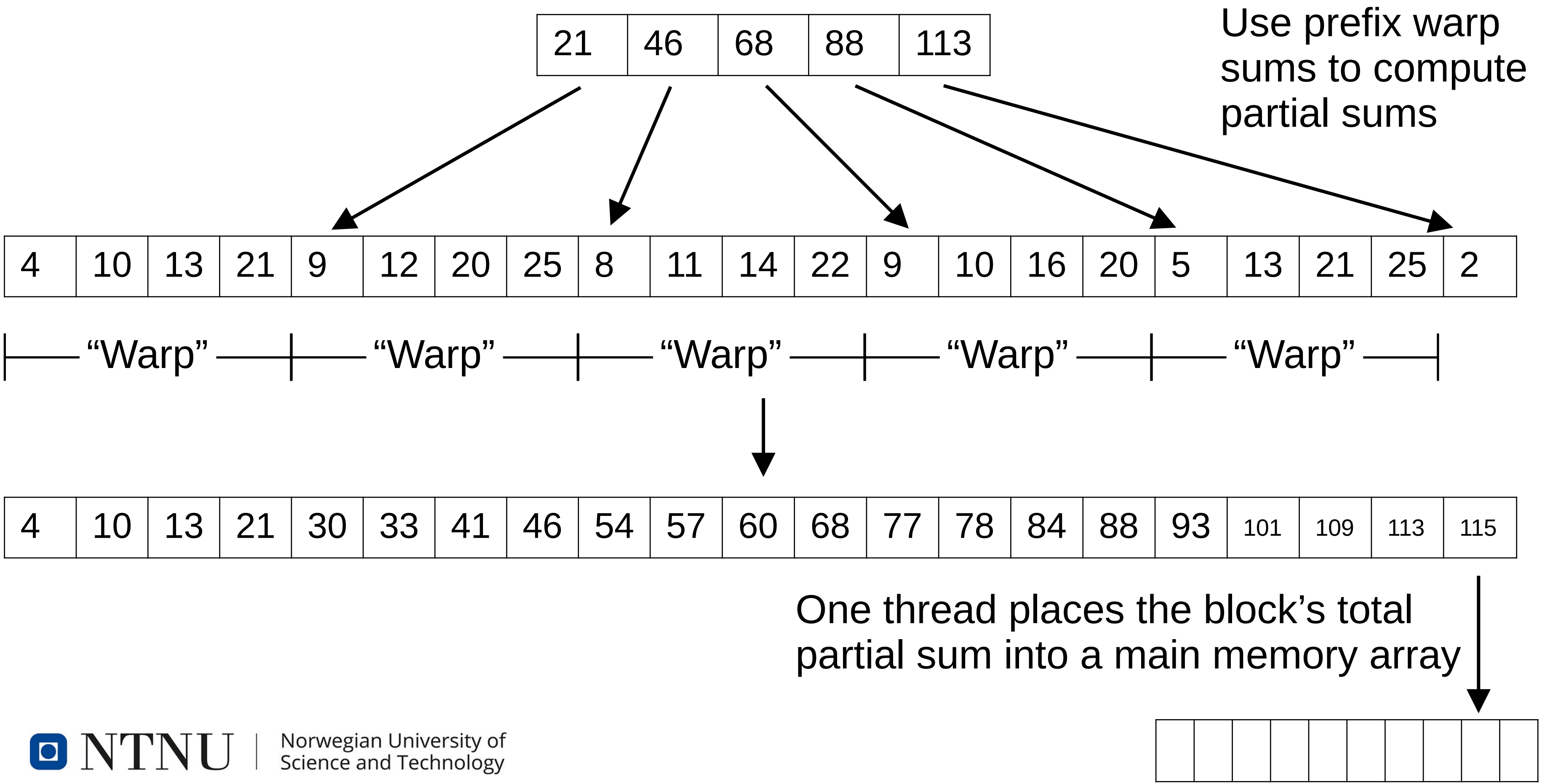
21	46	68	88	113

__syncthreads()

Phase 1: prefix sum in block

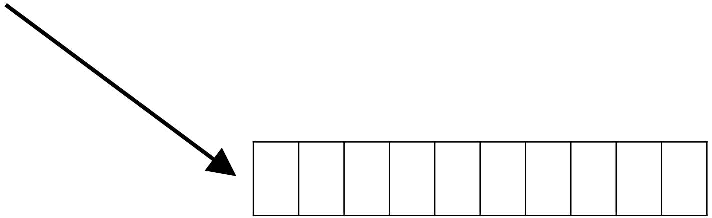


Phase 1: prefix sum in block



Phase 2: prefix sum of block sums

Repeat the first kernel
for the block sum array



Phase 3: prefix sum of block sums

- We now know the total sum up until the first element of each block

- We have also computed the prefix sums within each block

→ Final step: add the total sum to the block to each element

		38							

4	10	13	21	30	33	41	46	54	57	60	68	77	78	84	88	93	101	109	113	115

↓

42	48	51	59	68	71	79	84	92	95	98	106	115	116	122	126	131	139	147	151	153

Now what?

- Now that the kernel works, it's time to pull out the profiler and determine the optimal launch parameters
  - In this case only block dimensions
  - We cannot adjust the amount of shared memory as it depends on the block size

Today

- Repetition: performance pitfalls and collective instructions
- Now you're thinking with warps
- **Useful stuff**
  - **Built-in types**
  - Miscellaneous functions
- Demonstration: CUDA profiling tools

Built-in types

- Many calculations involve multidimensional coordinates
- CUDA natively has types available for storing these

Type	Signed version	Unsigned version
`char`	`char[1-4]`	`uchar[1-4]`
`short`	`short[1-4]`	`ushort[1-4]`
`int`	`int[1-4]`	`uint[1-4]`
`long`	`long[1-4]`	`ulong[1-4]`
`long long`	`longlong[1-4]`	`ulonglong[1-4]`
`float`	`float[1-4]`	`uchar[1-4]`
`double`	`double[1-4]`	`uchar[1-4]`

Example:

```
uint3 a;
a.x = 5;
```

Built-in types: half precision

- CUDA also supports 16-bit float (half precision)
- Usually handled in pairs (type: `__half2`)
- Convert to and from with functions
- Can do arithmetic as usual, but this time applies on 2 values at the same time
- Should mostly be used for normalised values (-1 to 1)
- Saves bandwidth

Today

- Repetition: performance pitfalls and collective instructions
- Now you're thinking with warps
- **Useful stuff**
  - Built-in types
  - **Miscellaneous functions**
- Demonstration: CUDA profiling tools

Miscellaneous functions

- CUDA has a number of utility functions available
  - Either extremely efficient or implemented in hardware
  - Builtin functions can be recognised by their __ prefix

Useful functions

Integer operations:

```
// Compute the sum of absolute differences: |x – y| + z
unsigned int __sad(int x, int y, int z);

// Compute the average of two integers
// Guaranteed to not overflow
unsigned int __uhadd(unsigned int x, unsigned int y);
```

Useful functions

Floating point operations:

```
// The remainder of a/b
float remainder(float a, float b);

// Computes sqrt(a*a + b*b + c*c)
float norm3df(float a, float b, float c);

// Calculate a x y + z
// Useful because it only rounds at the end
float ___fmaf_rz(float a, float y, float z);
```

Useful functions

Bit manipulation:

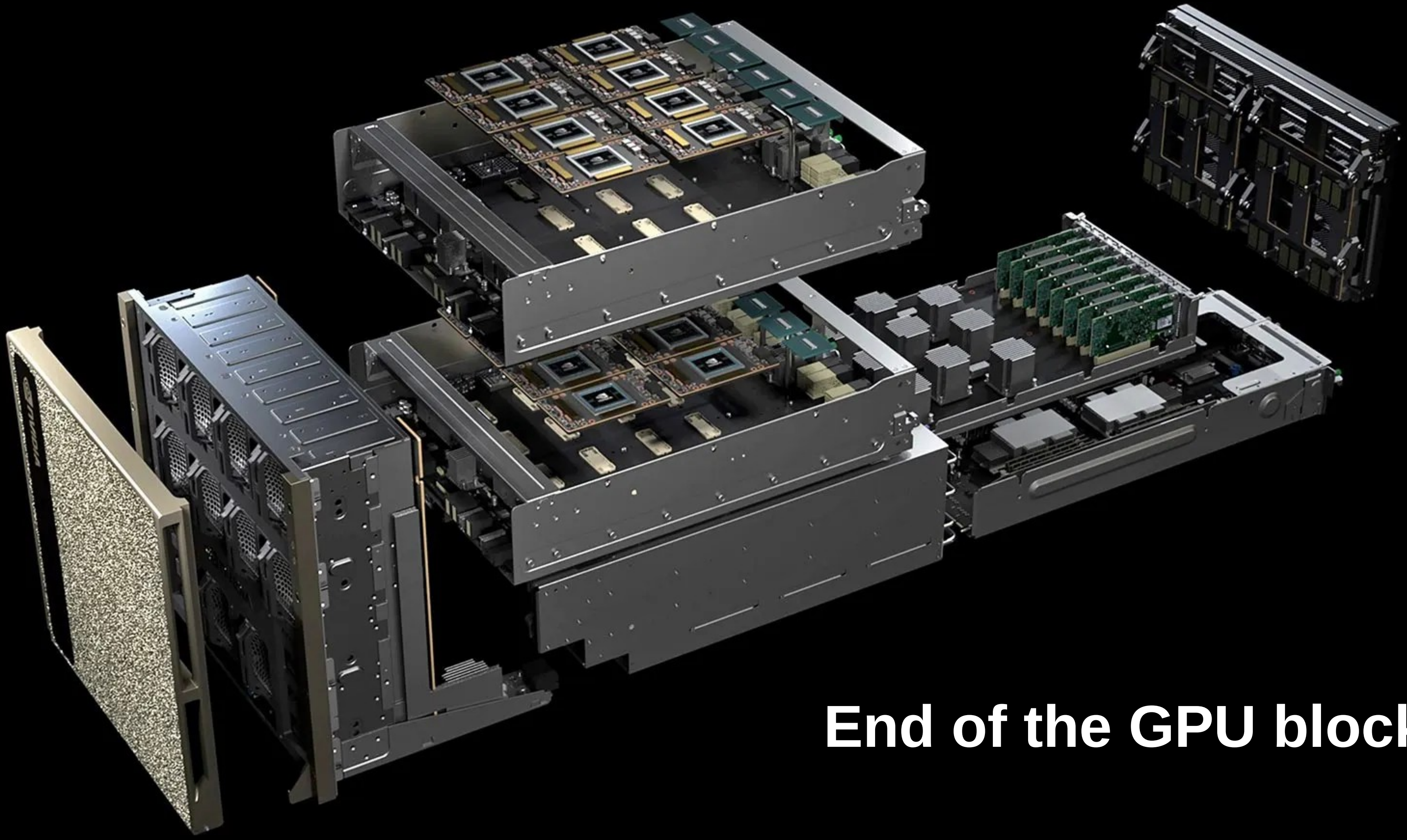
```
// Find the index with the first bit set to 1
int __ffs(int x);

// Count leading zeroes (starting from the least significant bit)
int __clz(int x);

// Find the position of the {offset}th bit set to 1
// Counting starts at index base
unsigned int __fns(unsigned int x, unsigned int base, int offset);
```

Today

- Repetition: performance pitfalls and collective instructions
- Now you’re thinking with warps
- Useful stuff
- **Demonstration: CUDA profiling tools**

IMHO

- GPU computing is cool because:
  - More predictable than a CPU because threads are not executed out of order
  - Optimal performance is highly dependent on knowing the details of the underlying architecture
  - Cheap cooperation between threads means you tend to work with threads in groups, which poses really interesting modelling challenges
  - When your code is well optimised, it can run orders of magnitude faster than on a CPU



End of the GPU block