NTNU | Norwegian University of Science and Technology

Shuffle Shenanigans

Bart Iver van Blokland

Today

- **Repetition: thread structure and limits**
- Performance pitfalls
- Collective instructions

Thread structure

- A grid consists of blocks
- Blocks consist of threads
- Groups of 32 threads within a block represent a warp
  - All instructions are executed grouped as a warp
- Block coordinates are «flattened» into warps

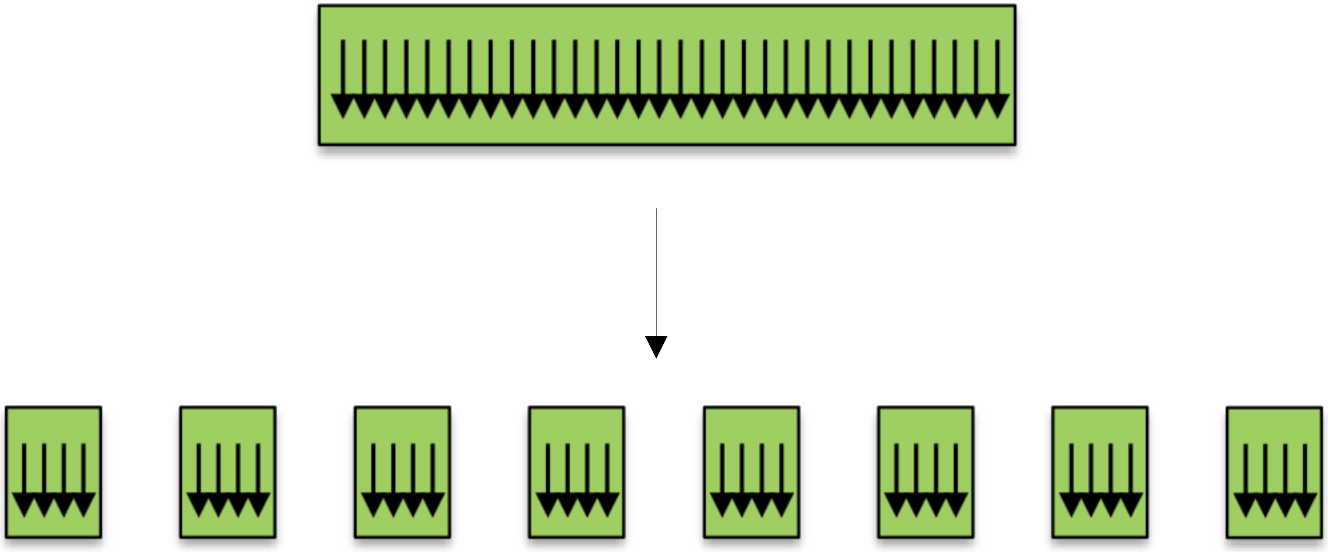
0, 0	1, 0	2, 0	3, 0	4, 0	5, 0	6, 0	7, 0	8, 0
0, 1	1, 1	2, 1	3, 1	4, 1	5, 1	6, 1	7, 1	8, 1
0, 2	1, 2	2, 2	3, 2	4, 2	5, 2	6, 2	7, 2	8, 2
0, 3	1, 3	2, 3	3, 3	4, 3	5, 3	6, 3	7, 3	8, 3
0, 4	1, 4	2, 4	3, 4	4, 4	5, 4	6, 4	7, 4	8, 4

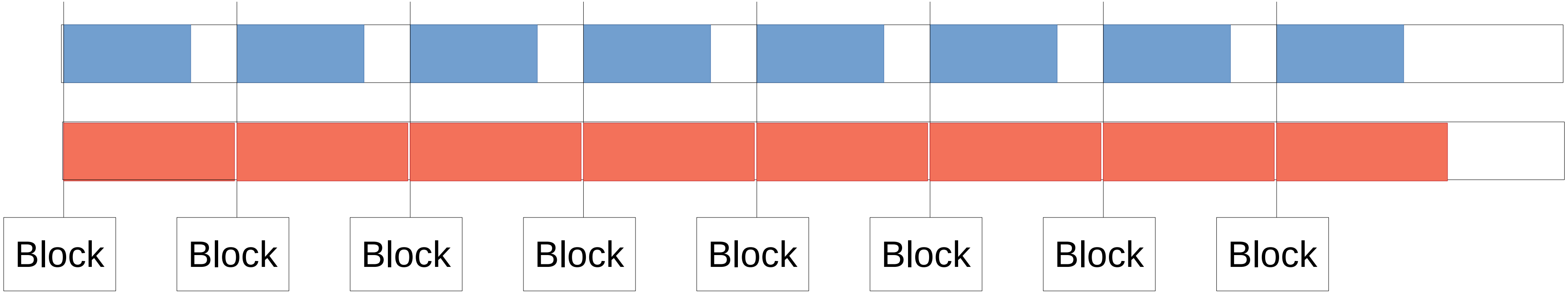
Warp 1

Warp 2

Group Partitioning

- Create smaller subgroups from larger ones
- Similar capabilities and interface to the larger groups





- Limit 1: Number of threads per block: 1024
- Limit 2: Number of blocks per SM: 24
- Limit 3: Number of warps per SM: 48 (1536 threads)
- Limit 4: Number of registers per thread: 255
- Limit 5: Available shared memory: up to 100Kb
- Limit 6: register requirements limit the number of warps that can be executed simultaneously in an SM
- Limit 7: blocks have a constant number of warps and cannot be partially allocated to an SM
- Limit 8: shared memory required by each block limits the number of warps

Tip:

The CUDA page on wikipedia has a good overview over device-specific features and limits

https://en.wikipedia.org/wiki/CUDA

Technical Specification [edit]

Technical specifications	Compute capability (version)																			
	1.0	1.1	1.2	1.3	2.x	3.0	3.2	3.5	3.7	5.0	5.2	5.3	6.0	6.1	6.2	7.0	7.2	7.5	8.0	8.6
Maximum number of resident grids per device (concurrent kernel execution, can be lower for specific devices)	1				16		4	32				16	128	32	16	128	16			
Maximum dimensionality of grid of thread blocks	2				3															
Maximum x-dimension of a grid of thread blocks	65535					$2^{31} - 1$														
Maximum y-, or z-dimension of a grid of thread blocks	65535																			
Maximum dimensionality of thread block	3																			
Maximum x- or y-dimension of a block	512				1024															
Maximum z-dimension of a block	64																			
Maximum number of threads per block	512				1024															
Warp size	32																			
Maximum number of resident blocks per multiprocessor	8					16				32								16	32	
Maximum number of resident warps per multiprocessor	24		32		48	64												32	64	
Maximum number of resident threads per multiprocessor	768		1024		1536	2048												1024	2048	
Number of 32-bit regular registers per multiprocessor	8 K		16 K		32 K	64 K			128 K	64 K										
Number of 32-bit uniform registers per multiprocessor	No																	2 K[73][74]		
Maximum number of 32-bit registers per thread block	8 K		16 K		32 K	64 K	32 K	64 K				32 K	64 K		32 K	64 K				
Maximum number of 32-bit regular registers per thread	124				63		255													
Maximum number of 32-bit uniform registers per warp	No																	63[75][76]		
Amount of shared memory per multiprocessor (out of overall shared memory + L1 cache, where applicable)	16 KiB				16 / 48 KiB (of 64 KiB)	16 / 32 / 48 KiB (of 64 KiB)			80 / 96 / 112 KiB (of 128 KiB)	64 KiB	96 KiB	64 KiB		96 KiB	64 KiB	0 / 8 / 16 / 32 / 64 / 96 KiB (of 128 KiB)		32 / 64 KiB (of 96 KiB)	0 / 8 / 16 / 32 / 64 / 100 / 132 / 164 KiB (of 192 KiB)	0 / 8 / 1[illegible] 32 / 64 / [illegible] KiB (of [illegible] KiB)
Maximum amount of shared memory per thread block	16 KiB				48 KiB											96 KiB	48 KiB	64 KiB	163 KiB	99 Ki[illegible]
Number of shared memory banks	16				32															
Amount of local memory per thread	16 KiB				512 KiB															
Constant memory size accessible by CUDA C/C++ (1 bank, PTX can access 11 banks, SASS can access 18 banks)	64 KiB																			
Cache working set per multiprocessor for constant memory	8 KiB												4 KiB	8 KiB						
Cache working set per multiprocessor for texture memory	16 KiB per TPC			24 KiB per TPC	12 KiB	12 – 48 KiB[77]				24 KiB	48 KiB	32 KiB[78]	24 KiB	48 KiB	24 KiB	32 – 128 KiB		32 – 64 KiB	28 – 192 KiB	28 – 128
Maximum width for 1D texture reference bound to a CUDA array	8192				65536								131072							
Maximum width for 1D texture reference bound to linear memory	2^{27}												2^{28}	2^{27}		2^{28}	2^{27}			

Today

- Repetition: thread structure and limits
- **Performance pitfalls**
  - **Memory**
  - Suboptimal launch parameters
  - Thread divergence
  - Register spilling
  - PCIe bandwidth
- Collective instructions

Today

- Repetition: thread structure and limits
- **Performance pitfalls**
  - **Memory**
  - Suboptimal launch parameters
  - Thread divergence
  - Register spilling

Memory

Usually the primary bottleneck of a kernel

- **Common issues:**
  - **Non-coalesced memory reads**
  - Atomic write contention
- Common tool: struct of arrays
- Common tool: shared memory
- Memory tips

Coalesced memory reads

- All modern high performance processors have memory systems built around cache lines
  - CPU: usually 64 bytes per line
  - GPU: usually 128 bytes per line (32 threads/warp x 4 bytes)

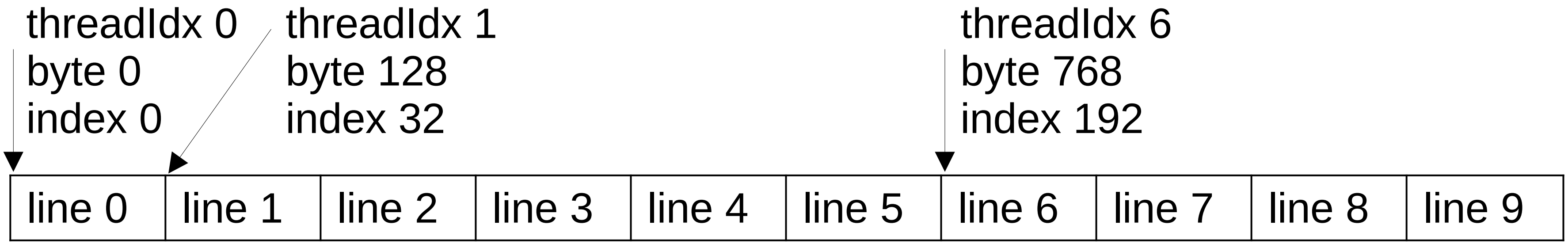
- Memory allocated using cudaMalloc() is always "aligned"; byte 0 of the allocated region is at the start of a cache line

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

Coalesced memory reads

- Kicker: 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
}
```



Coalesced memory reads

- Double whammy:
  - We need to load in one cache line per thread
    - More memory traffic
    - More wait time for each thread
  - We only use 4 bytes from every 128 byte cache line
    - Bad cache utilisation
    - Bad bandwidth utilisation

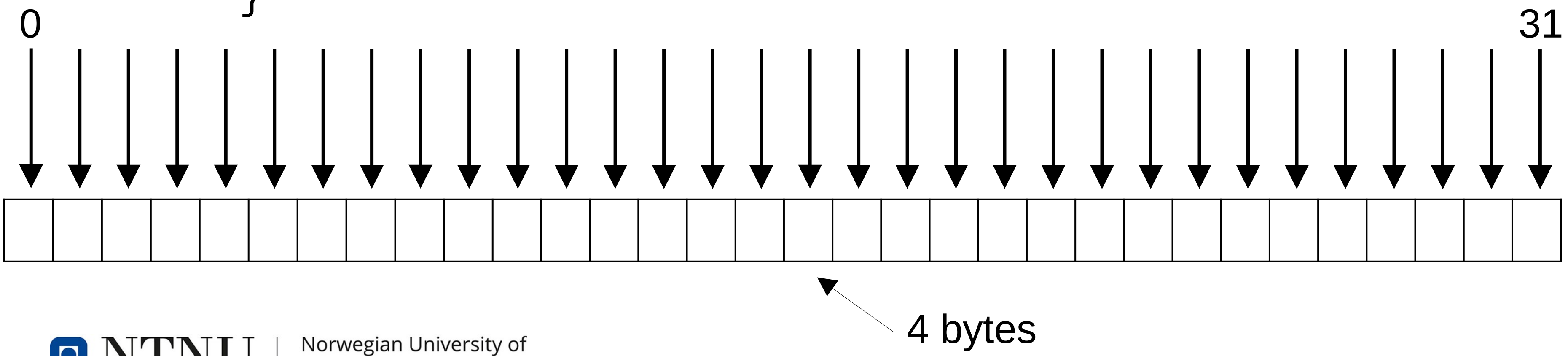
```
__global__ void kernel(int* array, int n) {
    int value = array[32 * threadIdx.x];
    // do stuff with value here
}
```

Coalesced memory reads

This kernel reads optimally:

- 1 cache line read per warp,
- Each thread reads 4 bytes
- 100% bandwidth utilisation
- These reads are called “coalesced”

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



Coalesced memory reads

Quiz question: how many cache lines are loaded in per warp?

```
__global__ void kernel1(int* array, int n) {
    int value = array[threadIdx.x + 1];
    // do stuff with value here
}

__global__ void kernel2(int* array, int n) {
    int value = array[blockIdx.x + 1];
    // do stuff with value here
}

__global__ void kernel3(int* array, int n) {
    int value = array[threadIdx.x + 1] - array[threadIdx.x];
    // do stuff with value here
}
```

Memory

- Common issues:
  - Coalesced memory reads
  - **Atomic write contention**
- Common tool: struct of arrays
- Common tool: shared memory
- Memory tips

Atomic write contention

- When multiple threads attempt to perform an atomic operation, their effects are serialised
  - Threads have to wait for their turn
  - Better solution: do a reduction within the warp, then have a single thread (e.g. thread 0) do the atomic operation

```
__global__ void kernel(int* array, int* oddCount) {
    int value = array[threadIdx.x];
    if(value % 2 == 1) {
        atomicAdd(oddCount, 1);
    }
}
```

Memory

- Common issues:
  - Coalesced memory reads
  - Atomic write contention
- **Common tool: struct of arrays**
- Common tool: shared memory
- Memory tips

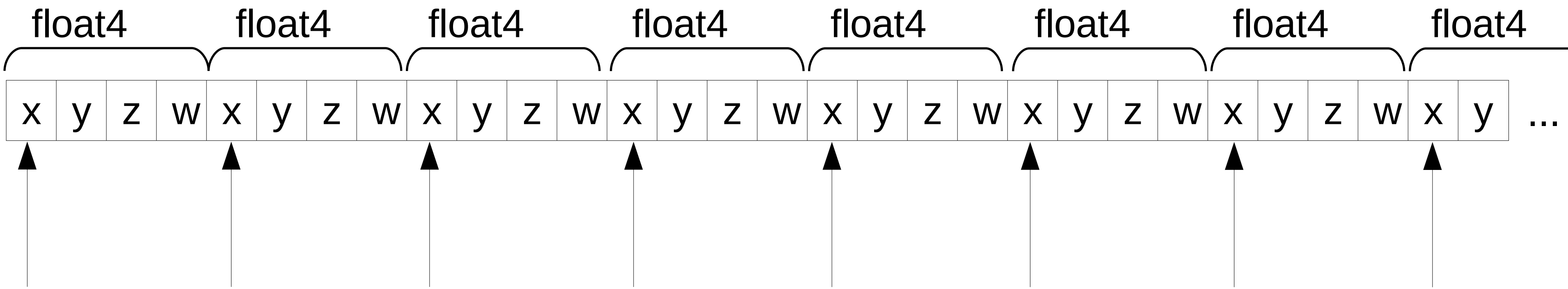
Common tool: struct of arrays

- Problem: misaligned reads often come from structs

```
struct float4 {
    float x;
    float y;
    float z;
    float w;
};

float4 vertex;
vertex.x = vertexArray[threadIndex].x;
vertex.y = vertexArray[threadIndex].y;
vertex.z = vertexArray[threadIndex].z;
vertex.w = vertexArray[threadIndex].w;
```

Common tool: struct of arrays

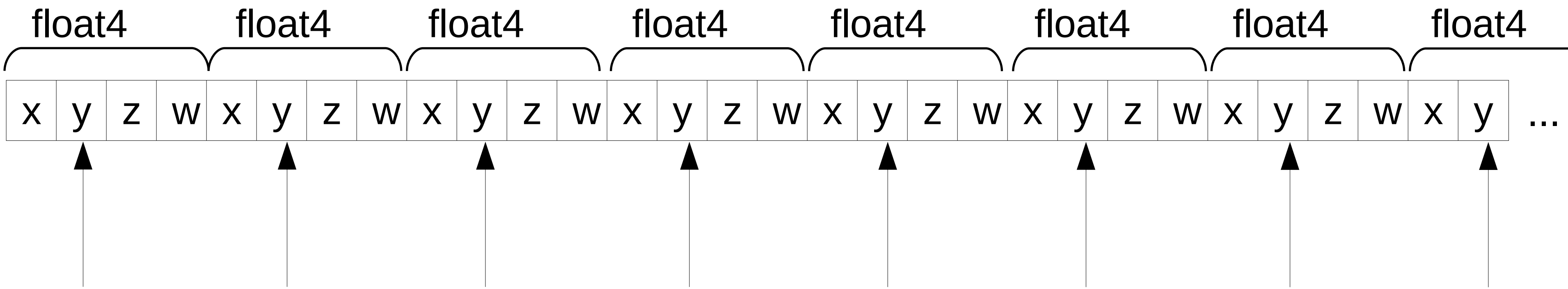
- Problem: misaligned reads often come from structs




```
float4 vertex;
vertex.x = vertexArray[threadIndex].x;
vertex.y = vertexArray[threadIndex].y;
vertex.z = vertexArray[threadIndex].z;
vertex.w = vertexArray[threadIndex].w;
```

Common tool: struct of arrays

- Problem: misaligned reads often come from structs

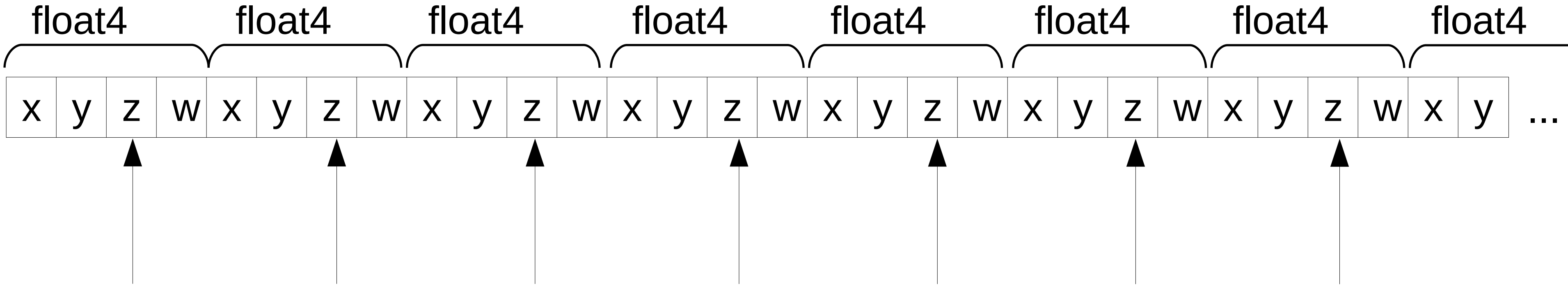



```
float4 vertex;
vertex.x = vertexArray[threadIndex].x;
vertex.y = vertexArray[threadIndex].y;
vertex.z = vertexArray[threadIndex].z;
vertex.w = vertexArray[threadIndex].w;
```

NTNU | Norwegian University of Science and Technology

Common tool: struct of arrays

- Problem: misaligned reads often come from structs




```
float4 vertex;
vertex.x = vertexArray[threadIndex].x;
vertex.y = vertexArray[threadIndex].y;
vertex.z = vertexArray[threadIndex].z;
vertex.w = vertexArray[threadIndex].w;
```

Each read operation only uses 25% of the cache lines loaded into the core

Common tool: struct of arrays

- Solution: create a separate array for each member variable such that all data is stored together

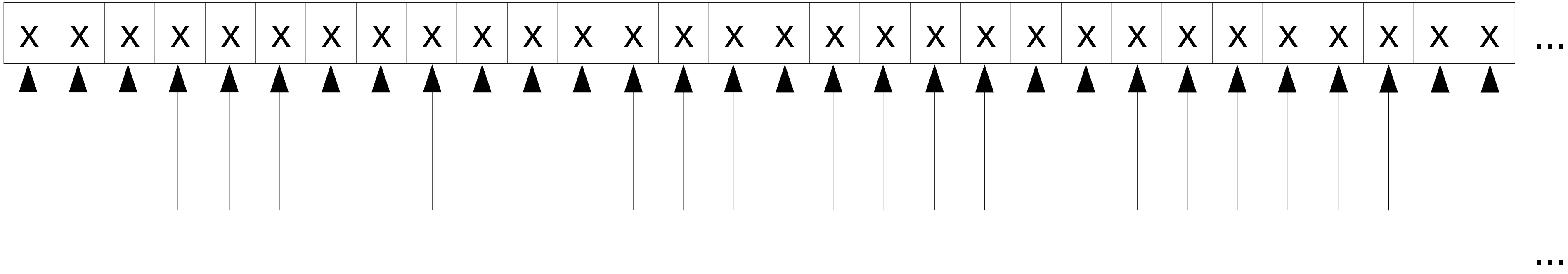
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	...
y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	...
z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	...
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	...

```
struct arrayOfFloat4 {
    float* x;
    float* y;
    float* z;
    float* w;
};
```

```
// instead of:
float value = array[i].x;

// write:
float value = array.x[i];
```

Common tool: struct of arrays

- Solution: create a separate array for each member variable such that all data is stored together



Now reading the field utilises the available bandwidth perfectly!

This trick also works for the CPU

```
// instead of:
float value = array[i].x;

// write:
float value = array.x[i];
```

Memory

- Usually the primary bottleneck of a kernel
- Common issues:
  - Coalesced memory reads
  - Atomic write contention
- Common tool: struct of arrays
- **Common tool: shared memory**
- Memory tips

Reducing memory latency with shared memory

- Shared memory: user managed L1 cache
  - Roughly 100x faster than main memory
  - Can be used to communicate between warps in a block
  - Allocated on a per-block basis

- Use for:
  - Storing intermediate results
  - Storing data you will reuse many times
  - Exchange values between warps in a block

Reducing memory latency with shared memory

- Problems:
  - Number of blocks resident in the SM is in part determined by the amount of shared memory each block requires
  - Be aware of race conditions

Reducing memory latency with shared memory

```
__global__ void kernel(float* buffer, int length, int* total) {
    __shared__ int count = 0;
    __syncthreads();
    for(int i = threadIdx.x; i < length; i += blockDim.x) {
        if(buffer[i] > 50) {
            atomicAdd(&count, 1);
        }
    }
    __syncthreads();
    if(threadIdx.x == 0) {
        *total = count;
    }
}
```

Memory

- Usually the primary bottleneck of a kernel
- Common issues:
  - Coalesced memory reads
  - Atomic write contention
- Common tool: struct of arrays
- Common tool: shared memory
- **Memory tips**

Memory

- Vectorised loads
  - If your data type is exactly 4, 8, or 16 bytes, the memory system guarantees your data is loaded in one operation
  - Can avoid struct of arrays when you know it is one of these sizes
- Use smaller data types
  - For example: use short instead of int
  - All data that does not need to go through the memory system reduces the load on it

Today

- Repetition: thread structure and limits
- **Performance pitfalls**
  - Memory
  - **Suboptimal launch parameters**
  - **Thread divergence**
  - Register spilling
  - PCIe bandwidth
- Collective instructions

Today

- Repetition: thread structure and limits
- **Performance pitfalls**
  - Memory
  - Suboptimal launch parameters
  - Thread divergence
  - **Register spilling**
  - PCIe bandwidth
- Collective instructions

Register spilling

- A given kernel will require a certain number of registers to run. The compiler determines how many are needed.

- Register spilling: temporarily write some register values to memory
  - Upside: can run more threads simultaneously
  - Downside: more memory transactions

- The compiler will spill registers when it believes it will improve overall performance

Register spilling

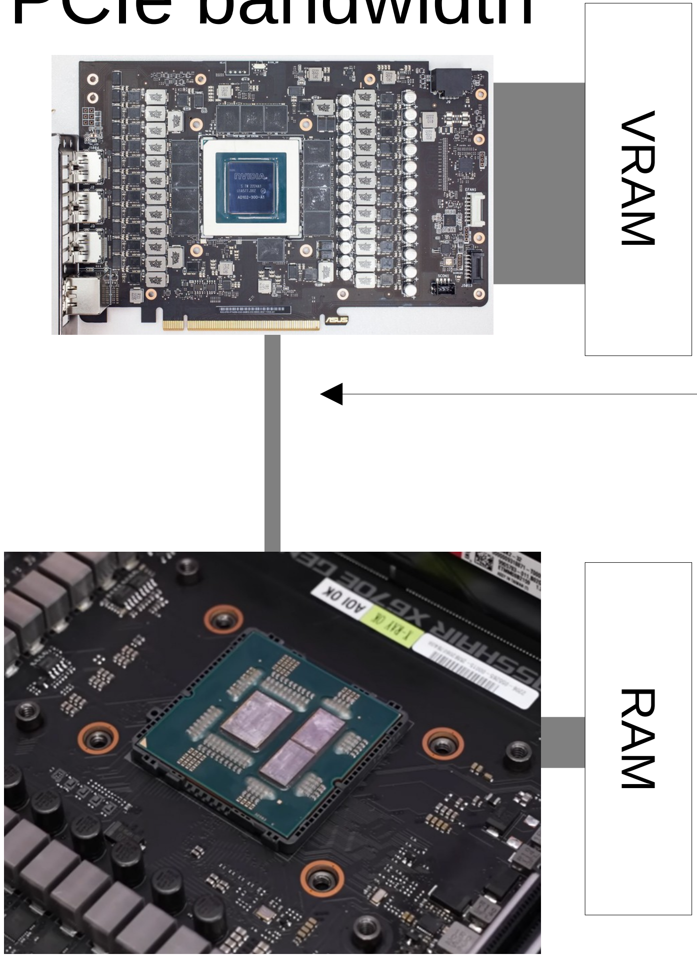
- How to resolve:
  - Usually difficult to resolve once it occurs
  - Identify parts of your kernel where many variables must be kept simultaneously.
    This can for example be caused by:
    - Nested function calls (recursive calls are a red flag)
    - Loops
  - Consider recomputing values if doing so is not very expensive
  - Cut fields from structs that are unrelated to your kernel

Today

- Repetition: thread structure and limits
- **Performance pitfalls**
  - Memory
  - Suboptimal launch parameters
  - Thread divergence
  - Register spilling
  - **PCIe bandwidth**
- Collective instructions

PCIe bandwidth

- Data copied from and to GPU memory needs to go over the PCI Express bus.
- PCIe is slow compared to system and GPU memory (RAM and VRAM)
- Avoid passing too much data back and forth



VRAM:
1.0 TB/s

PCI Express bus
Gen 3 x16: 15.8 GB/s
Gen 4 x16: 31.5 GB/s

RAM:
102.4 GB/s

Note: RAM and VRAM numbers vary between CPU and GPU models

Performance pitfalls

- Miscellaneous: do not use double
  - Only supported in hardware on enterprise GPUs
  - Slowdown of 32x on consumer cards (all double precision operations are emulated in software)
  - Often still slower than single precision operations on enterprise cards (~2x, but varies across generations)

Performance pitfalls

- The most important thing when tackling performance problems is:

Performance pitfalls

- The most important thing when tackling performance problems is:

MEASURE

(and find a better algorithm)

Today

- Repetition: thread structure and limits
- Performance pitfalls
- **Collective instructions**
  - **Shuffle instructions**
  - Shuffle instructions: example use cases
  - Warp voting
  - Warp voting: example use cases
  - Warp reductions

Shuffle instructions

- Exchange of values within a single warp
- Each thread provides one value
- Each thread reads a value provided by one of the threads

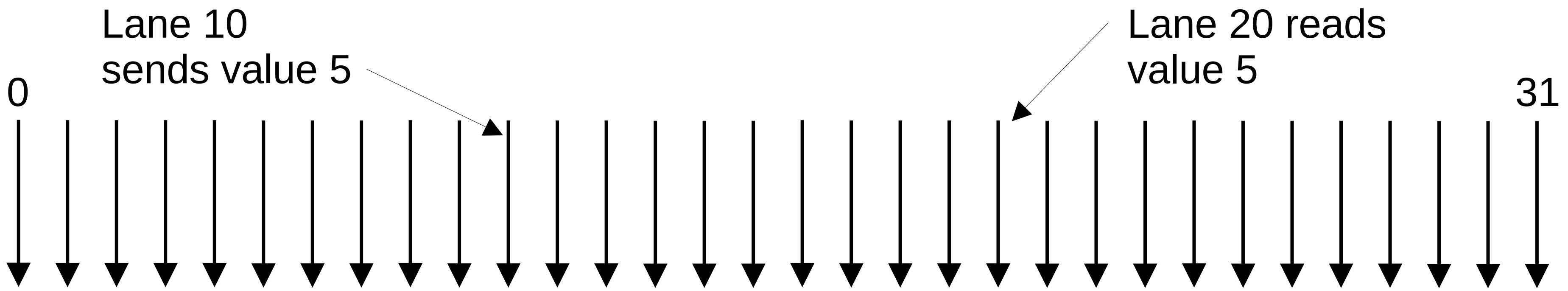
```
T __shfl_sync(unsigned mask, T var, int srcLane);
```

Usually `__activemask()` (refers to `mask`)

Where T is one of:

```
int, unsigned int,
long, unsigned long,
long long, unsigned long long,
float, double
```

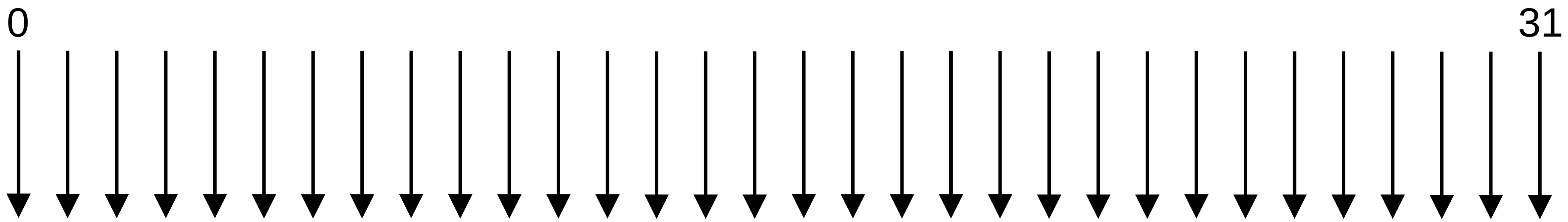
Shuffle instructions



- Example:

```
T __shfl_sync(unsigned mask, T var, int srcLane);

// Thread 10 executes (sends value 5):
int out = __shfl_sync(__activemask(), 5, 3);
// Thread 20 executes (receives value 5):
int out = __shfl_sync(__activemask(), 17, 10);
// Value of out in thread 20: 5
```

Shuffle instructions



- Threads **must** be in the same block
- Threads **must** be in the same warp or cooperative group up to 32 threads in size

- Extremely cheap instruction
  - Exact same behaviour would be extremely expensive on the CPU

- Reading from a thread that is not participating is undefined behaviour

- The _sync adjective implies that participating threads are first synchronised

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

- Read from thread (laneid XOR laneMask):
```
T __shfl_xor_sync(unsigned mask, T var, int laneMask);
```

  - XOR flips a bit if one of the operands is 1
    - laneMask effectively specifies which bits to flip
  - Applying XOR twice gives you back your original value

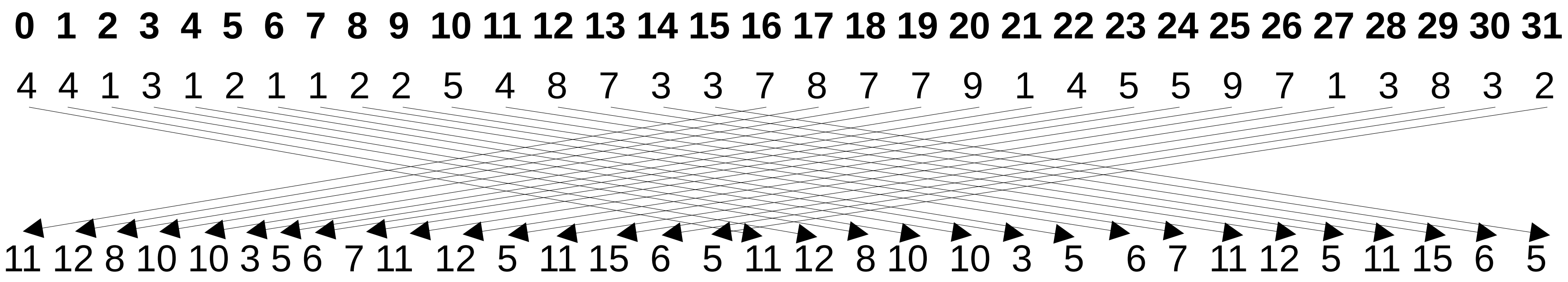
→ Becomes an exchange between two threads

Shuffle instructions

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**
4	4	1	3	1	2	1	1	2	2	5	4	8	7	3	3	7	8	7	7	9	1	4	5	5	9	7	1	3	8	3	2

```
int valueToSum = /* ... */;
```

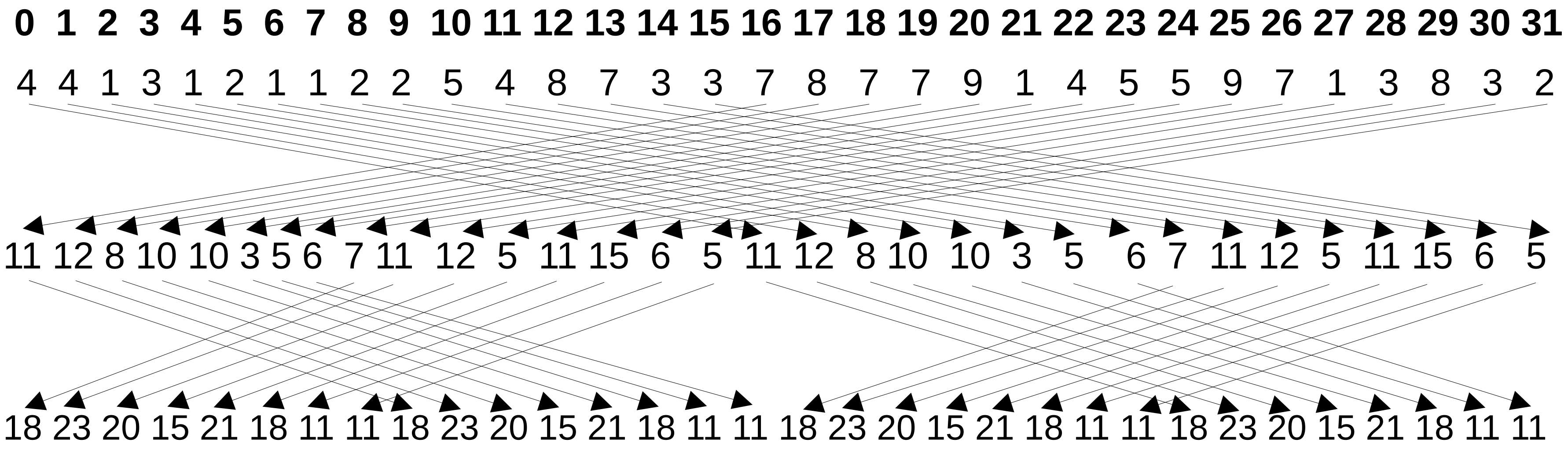
Shuffle instructions




```
int valueToSum = /* ... */;
sum += __shfl_xor_sync(__activemask(), valueToSum, 16);
```

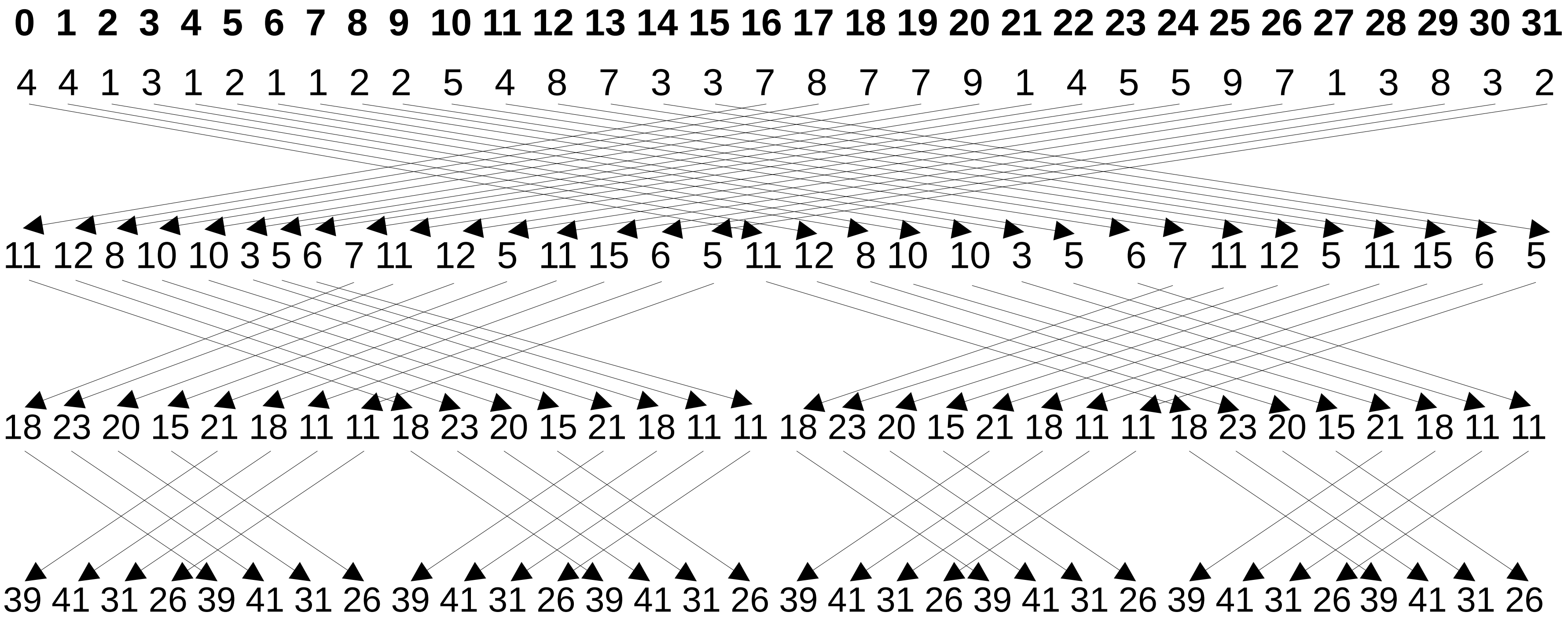
Here: `0xFFFFFFFF`

Shuffle instructions



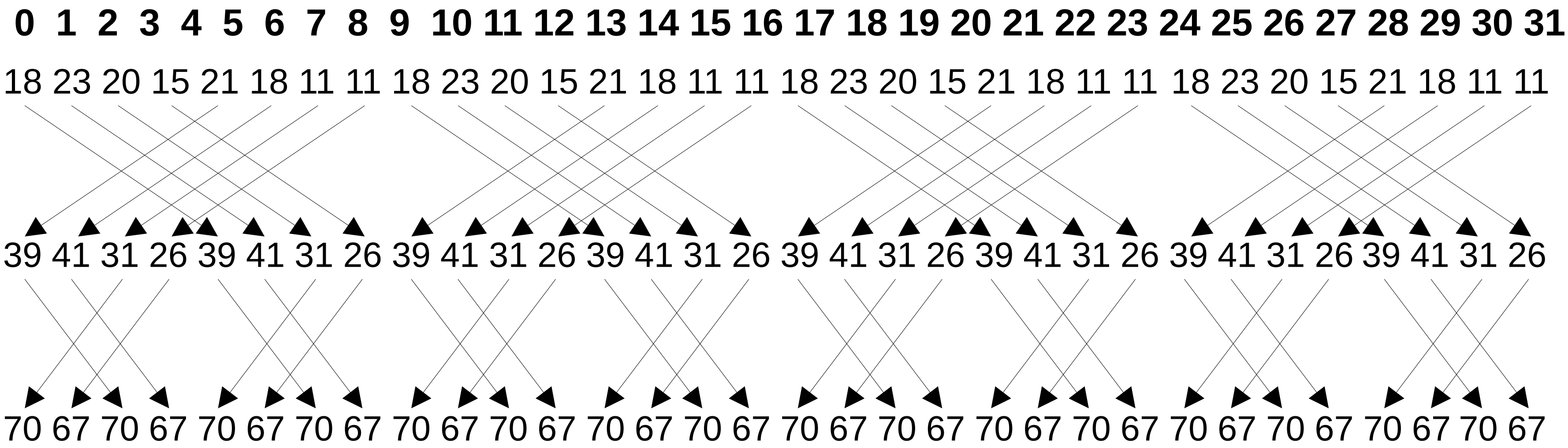

```
sum += __shfl_xor_sync(__activemask(), sum, 8);
```

Shuffle instructions



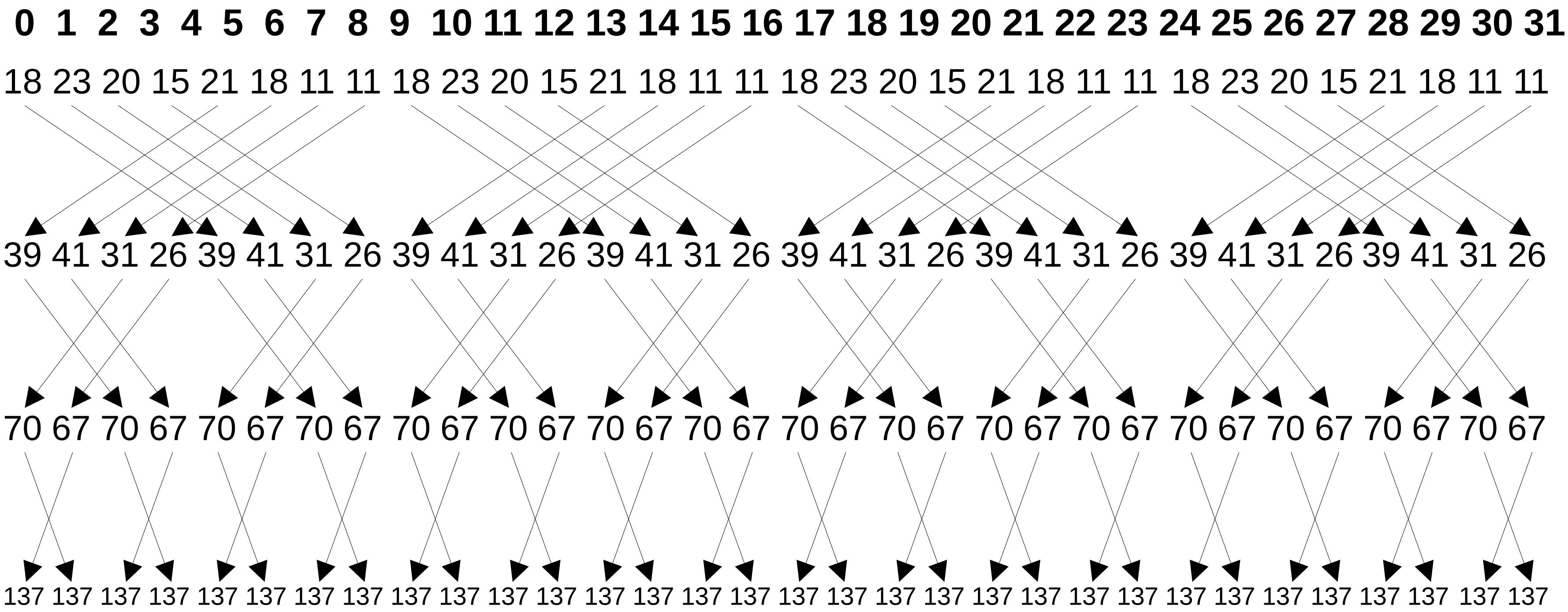

```
sum += __shfl_xor_sync(__activemask(), sum, 4);
```

Shuffle instructions




```
sum += __shfl_xor_sync(__activemask(), sum, 2);
```

Shuffle instructions




```
sum += __shfl_xor_sync(__activemask(), sum, 1);
```

Shuffle instructions

```
__device__ int warpReductionSum(int threadValue) {
    int sum = __shfl_xor_sync(__activemask(), threadValue, 16);
    sum += __shfl_xor_sync(__activemask(), sum, 8);
    sum += __shfl_xor_sync(__activemask(), sum, 4);
    sum += __shfl_xor_sync(__activemask(), sum, 2);
    sum += __shfl_xor_sync(__activemask(), sum, 1);
    return sum;
}
```

Today

- Repetition: thread structure and limits
- Performance pitfalls
- **Collective instructions**
  - Shuffle instructions
  - **Shuffle instructions: example use cases**
  - Warp voting
  - Warp voting: example use cases
  - Warp reductions

Shuffle instruction applications

- Warp-level reductions

  The XOR «butterfly» reduction we saw before

Shuffle instruction applications

- Warp-level reductions
- Broadcasting

Reserve a block of 32 entries in a buffer:

```
int startIndex = -1;
if(threadIdx.x == 0) {
    startIndex = atomicAdd(&nextBufferIndex, 32);
}
int index = __shfl_sync(__activemask(), startIndex, 0);
buffer[index + threadIdx.x] = usefulComputations();
```

All threads read from index 0

Shuffle instruction applications

- Warp-level reductions
- Broadcasting
- Bandwidth saving

```
for(int i = threadIdx.x; i < length; i += 32) {
    float value = buffer[i];
    float nextValue = buffer[i + 1];

    outputBuffer[i] = nextValue - value;
}
```

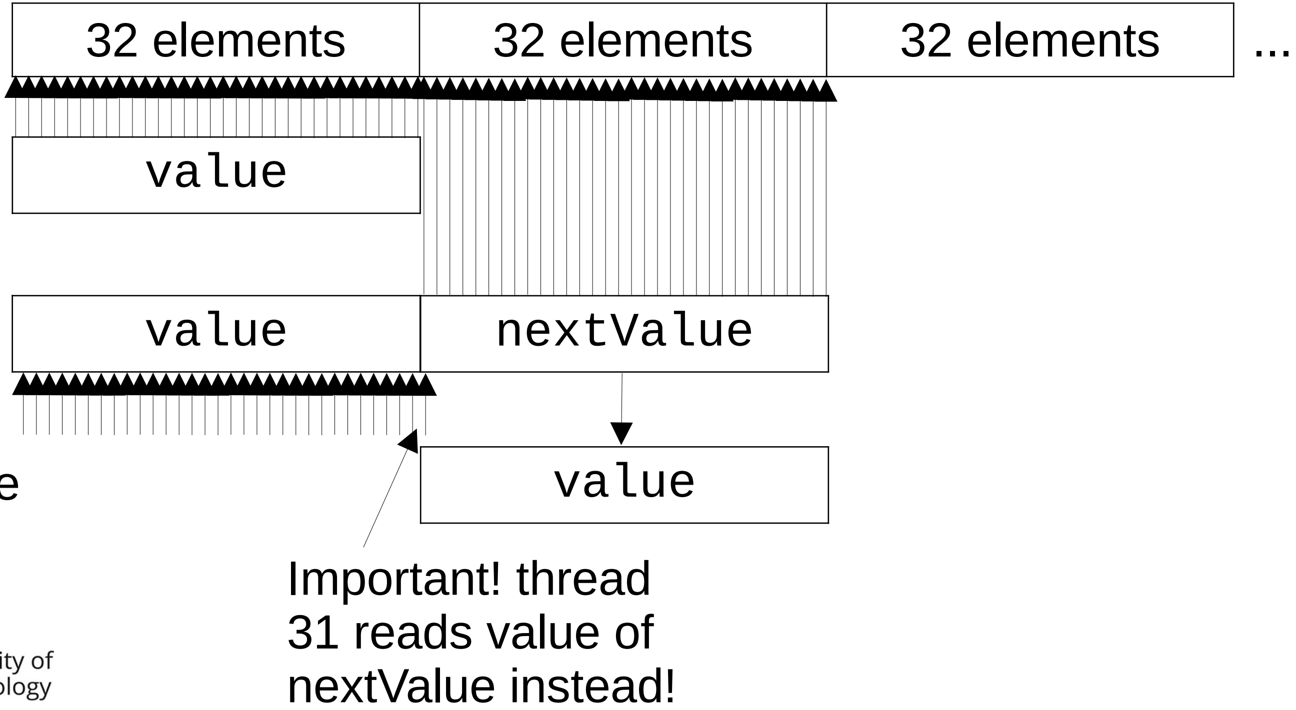
Neighbouring thread has this value! (points to `float nextValue = buffer[i + 1];`)

Shuffle instruction applications

- Bandwidth saving
  - Plan of attack

1. Read first block

For each block in buffer:
2. Read next block
3. Shuffle neighbour value
4. assign nextValue to value



NTNU | Norwegian University of Science and Technology

Shuffle instruction applications

- Bandwidth saving

```
float value = buffer[threadIdx.x];
for(int i = threadIdx.x; i < length; i += 32) {
    float nextBufferValue = buffer[i + 32];
    float valueToSend = (threadIdx.x == 0)
                              ? nextBufferValue : value;
    int laneToRead = (threadIdx.x == 31)
                           ? 0 : threadIdx.x + 1;
    float nextValue = __shfl_sync(__activemask(),
                              valueToSend, laneToRead);

    outputBuffer[i] = nextValue – value;
    value = nextBufferValue;
}
```

TODO: bounds check

This **halves** required bandwidth compared to the original version!

Shuffle instruction applications

- Warp-level reductions
- Broadcasting
- Bandwidth saving
- Broadcast + bandwidth saving

```
for(int i = threadIdx.x; i < length; i += 32) {
    float threadValue = buffer[i];
    for(int j = 0; j < 32; j++) {
        float value = __shfl_sync(0xFFFFFFFF, threadValue, j);
        processValue(value);
    }
}
```

Useful when we want to do work with an entire warp for each element

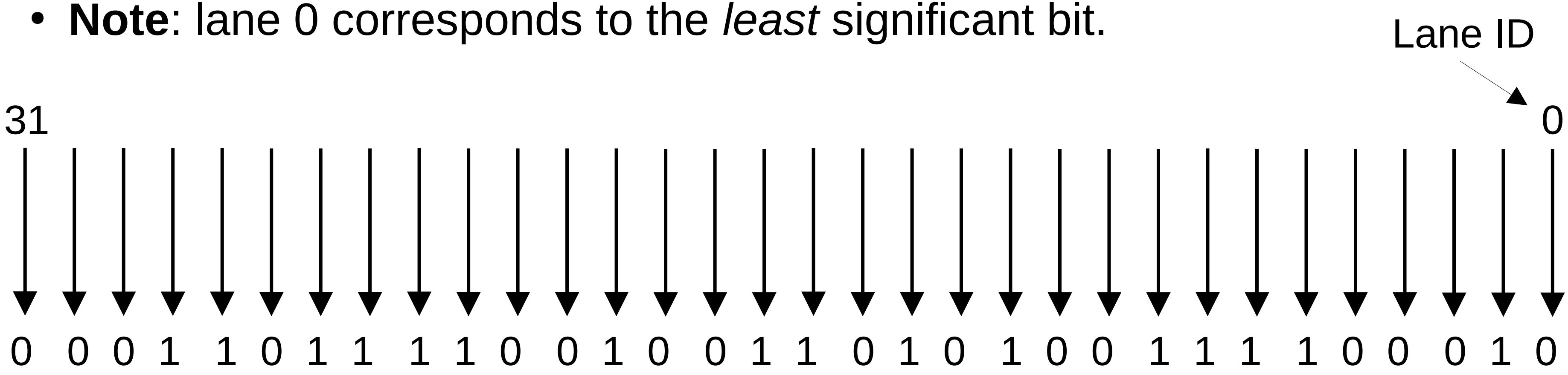
But notice: we are spending registers to accomplish this.

Today

- Repetition: thread structure and limits
- Performance pitfalls
- **Collective instructions**
  - Shuffle instructions
  - Shuffle instructions: example use cases
  - **Warp voting**
  - Warp voting: example use cases
  - Warp reductions

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

- Warp voting instructions:

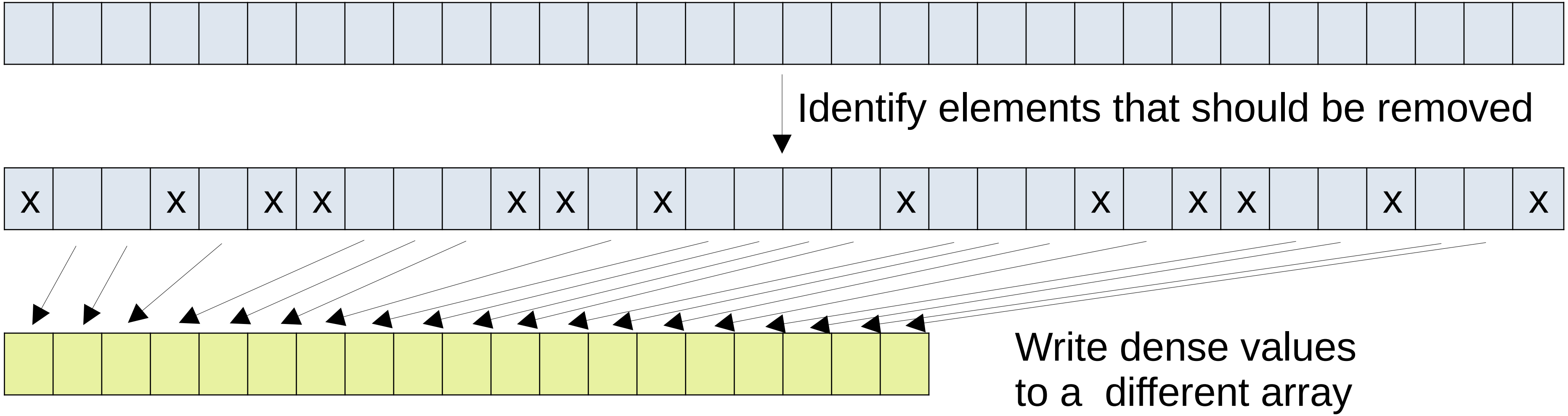
```
// Create a 32-bit integer where each lane sets one bit
unsigned int __ballot_sync(unsigned mask, bool predicate);

// Returns true if all threads vote true
bool __all_sync(unsigned mask, bool predicate);

// Returns true if one thread votes true
bool __any_sync(unsigned mask, bool predicate);


// Useful in conjunction: reverses a 32-bit integer
unsigned int __brev(unsigned int mask);
```

Today

- Repetition: thread structure and limits
- Performance pitfalls
- **Collective instructions**
  - Shuffle instructions
  - Shuffle instructions: example use cases
  - Warp voting
  - **Warp voting: example use cases**
  - Warp reductions
- Now you're thinking with warps
- Useful stuff

Warp voting use cases

- Stream filtering



Warp voting use cases

- Stream filtering

```
for(int i = threadIdx.x; i < length; i += 32) {
    float value = buffer[i];
    bool remove = criterion(value);
    unsigned int mask = __ballot_sync(0xFFFFFFFF, !remove);
    unsigned int sourceThread = __fns(mask, 0, threadIdx.x);
    if(sourceThread == 0xFFFFFFFF) {
        continue;
    }
    float condensedValue = __shfl_sync(0xFFFFFFFF, value, sourceThread);
    outputBuffer[bufferPointer + threadIdx.x] = condensedValue;
    if(threadIdx.x == 0) {
        bufferPointer += __popc(mask);
    }
}
```

Warp voting use cases

- Stream filtering

```
for(int i = threadIdx.x; i < length; i += 32) {
    float value = buffer[i];
    bool remove = criterion(value);
    unsigned int mask = __ballot_sync(0xFFFFFFFF, !remove);
    unsigned int sourceThread = __fns(mask, 0, threadIdx.x);
    if(sourceThread == 0xFFFFFFFF) {
        continue;
    }
    float condensedValue = __shfl_sync(0xFFFFFFFF, value, sourceThread);
    outputBuffer[bufferPointer + threadIdx.x] = condensedValue;
    if(threadIdx.x == 0) {
        bufferPointer += __popc(mask);
    }
}
```

Read value, determine if it should be removed

Warp voting use cases

- Stream filtering

```
for(int i = threadIdx.x; i < length; i += 32) {
    float value = buffer[i];
    bool remove = criterion(value);
    unsigned int mask = __ballot_sync(0xFFFFFFFF, !remove);
    unsigned int sourceThread = __fns(mask, 0, threadIdx.x);
    if(sourceThread == 0xFFFFFFFF) {
        continue;
    }
    float condensedValue = __shfl_sync(0xFFFFFFFF, value, sourceThread);
    outputBuffer[bufferPointer + threadIdx.x] = condensedValue;
    if(threadIdx.x == 0) {
        bufferPointer += __popc(mask);
    }
}
```

Communicate about which values to keep

Warp voting use cases

- Stream filtering

```
for(int i = threadIdx.x; i < length; i += 32) {
    float value = buffer[i];
    bool remove = criterion(value);
    unsigned int mask = __ballot_sync(0xFFFFFFFF, !remove);
    unsigned int sourceThread = __fns(mask, 0, threadIdx.x);
    if(sourceThread == 0xFFFFFFFF) {
        continue;
    }
    float condensedValue = __shfl_sync(0xFFFFFFFF, value, sourceThread);
    outputBuffer[bufferPointer + threadIdx.x] = condensedValue;
    if(threadIdx.x == 0) {
        bufferPointer += __popc(mask);
    }
}
```

We find the nth set bit in the mask, there n is the lane index

Warp voting use cases

- Stream filtering

```
for(int i = threadIdx.x; i < length; i += 32) {
    float value = buffer[i];
    bool remove = criterion(value);
    unsigned int mask = __ballot_sync(0xFFFFFFFF, !remove);
    unsigned int sourceThread = __fns(mask, 0, threadIdx.x);
    if(sourceThread == 0xFFFFFFFF) {
        continue;
    }
    float condensedValue = __shfl_sync(0xFFFFFFFF, value, sourceThread);
    outputBuffer[bufferPointer + threadIdx.x] = condensedValue;
    if(threadIdx.x == 0) {
        bufferPointer += __popc(mask);
    }
}
```

Move values from entire warp to the first n threads, where n is the number of values that should be kept, and write them to a buffer

Today

- Repetition: thread structure and limits
- Performance pitfalls
- **Collective instructions**
  - Shuffle instructions
  - Shuffle instructions: example use cases
  - Warp voting
  - Warp voting: example use cases
  - **Warp reductions**

Warp reductions

- Reductions implemented in hardware
  - Added in RTX 3000 series cards
  - Integer values only. For float use the XOR reduction shown before

```
unsigned __reduce_add_sync(unsigned mask, unsigned value);
unsigned __reduce_min_sync(unsigned mask, unsigned value);
unsigned __reduce_max_sync(unsigned mask, unsigned value);

int __reduce_add_sync(unsigned mask, int value);
int __reduce_min_sync(unsigned mask, int value);
int __reduce_max_sync(unsigned mask, int value);

unsigned __reduce_and_sync(unsigned mask, unsigned value);
unsigned __reduce_or_sync(unsigned mask, unsigned value);
unsigned __reduce_xor_sync(unsigned mask, unsigned value);
```

Today

- Performance pitfalls
- Thread cooperation

Tomorrow

- GPU profiling tools