Parallel Computing with GPUs: Warp Level CUDA and Atomics

Dr Mozhgan Kabiri Chimeh

http://mkchimeh.staff.shef.ac.uk/teaching/COM4521

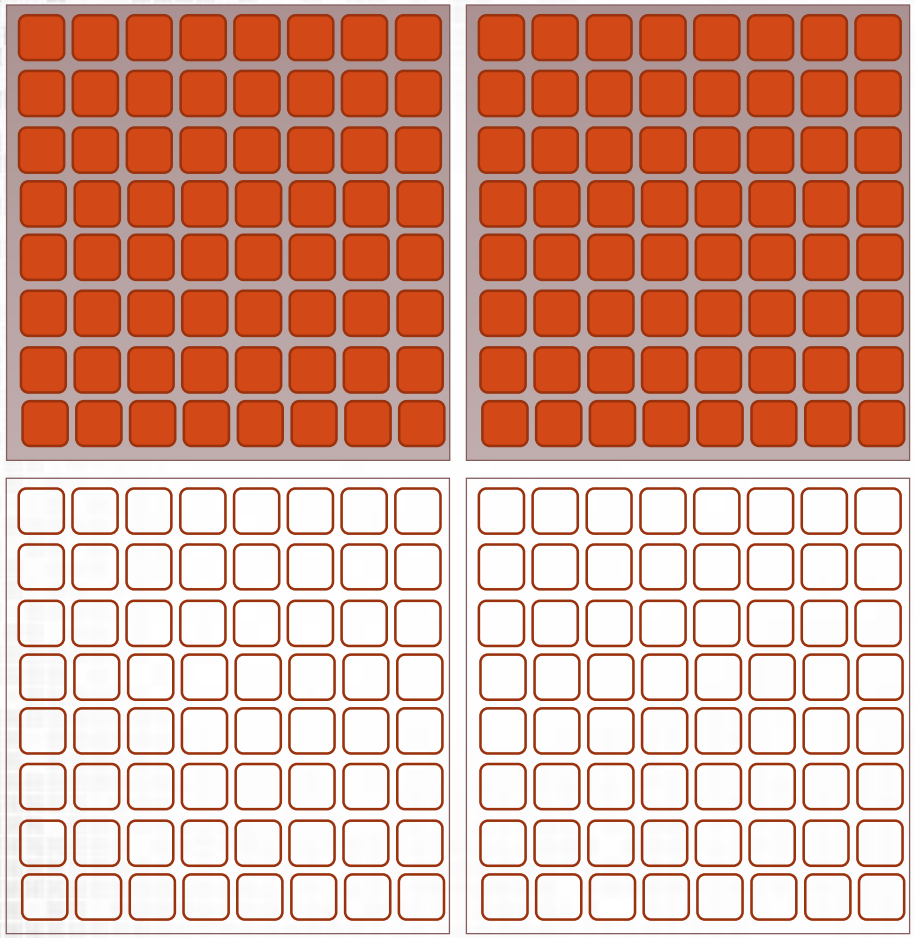
Last Teaching Week

- We learnt about shared memory
  - Very powerful for block level computations
  - Excellent for improving performance by reducing memory bandwidth
  - User controlled caching and needs careful consideration for bank conflicts and boundary conditions
- Memory coalescing: Vital for good memory bandwidth performance
  - Need to be aware of cache usage and line size
- Occupancy can be changed by modifying block sizes, registers and shared memory usage
- This week:
  - How exactly are warps scheduled?
  - Can we program at the warp level?
  - What mechanisms are there for communication between threads?

Overview

- Warp Scheduling & Divergence
- Atomics
- Warp Operations

Thread Block Scheduling



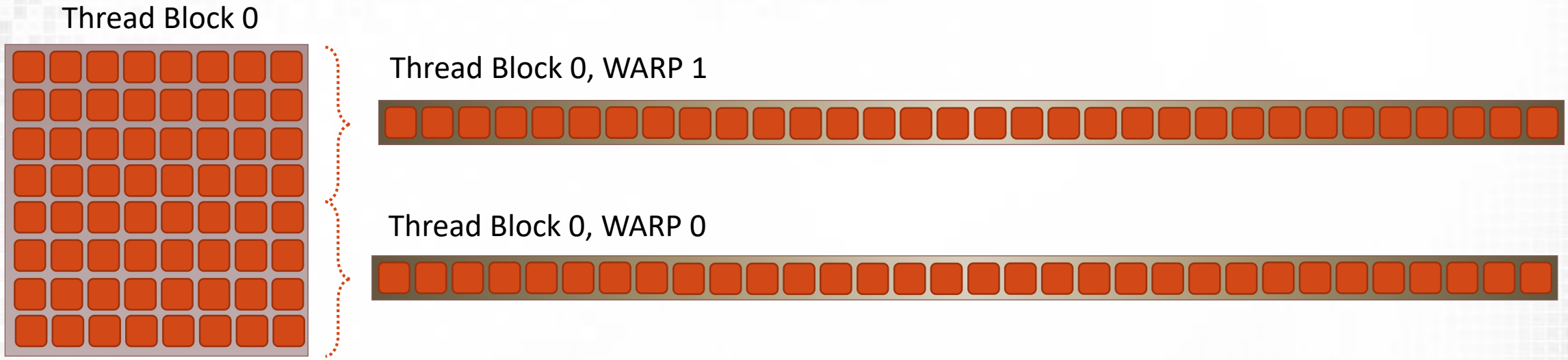
Multiple blocks scheduled to Streaming Multiprocessor

Streaming Multiprocessor 1

Streaming Multiprocessor 2

- No guarantee of block ordering on SMPs
- Hardware will schedule blocks to a SMP as soon as necessary resources are available

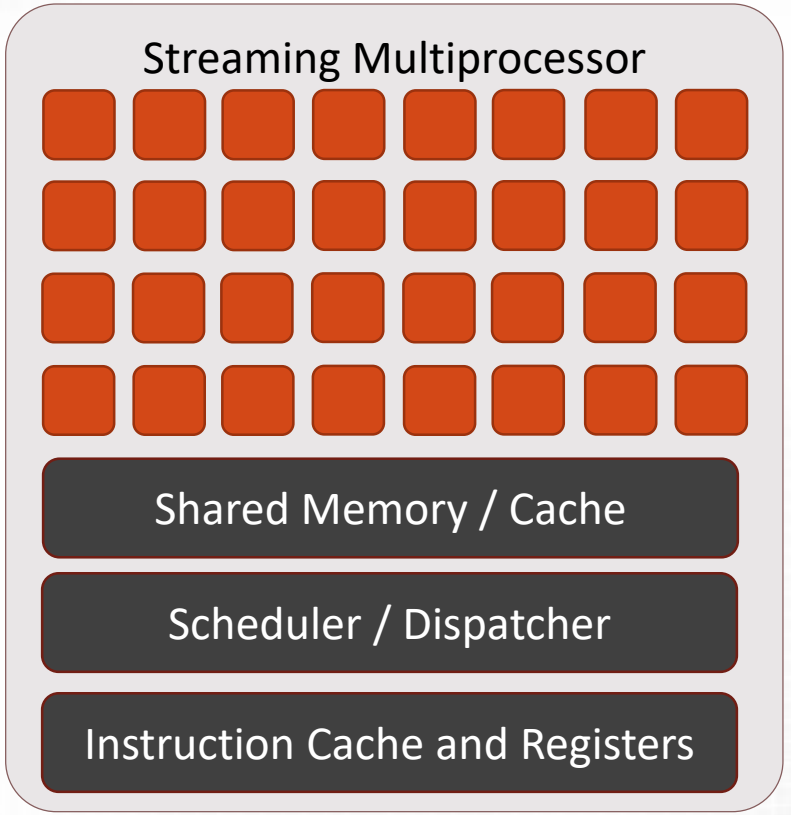
Thread Block Scheduling



- Each thread block is mapped to one or more warps
- 2D blocks are split into warps first by x index then y then z

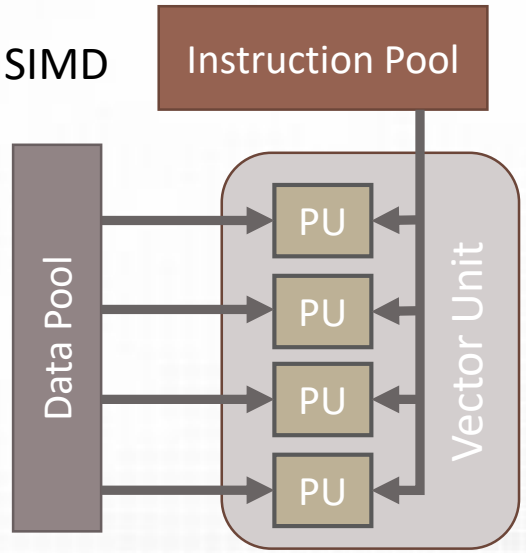
Warp Scheduling

- Zero overhead to swap warps (warp scheduling)
  - Warps contain only threads from a single thread block
  - Warps can be swapped with warps from different blocks assigned to the same streaming multi processor
  - At any one time only one warp has operations being executed
    - Memory movement happens in background



Warps and SIMD

- Execution of GPU instructions is always in groups of threads called **warps**
- Within a warp execution on the hardware follows the SIMD execution model
  - The view outside of a warp is SIMT
- What happens if code within a warp has different control flow?
  - **Branch Divergence**



Divergent Threads

- All threads must follow SIMD model
  - Multiple code branch paths must be evaluated
  - Not all threads will be active during code execution
  - Coherence = all threads following the same path

- How to avoid divergence
  1. Avoid conditional code
  2. **Especially** avoid conditional code based on `threadIdx`

- Fully coherent code can still have branches
  - BUT all threads in the warp follow the same path

Coherent Code

```
__global__ void a_kernel()
{
    if (blockIdx.x % 2)
        //something
    else
        //something else
}
```

```
__global__ void b_kernel()
{
    if (threadIdx.x % 2)
        //something
    else
        //something else
}
```

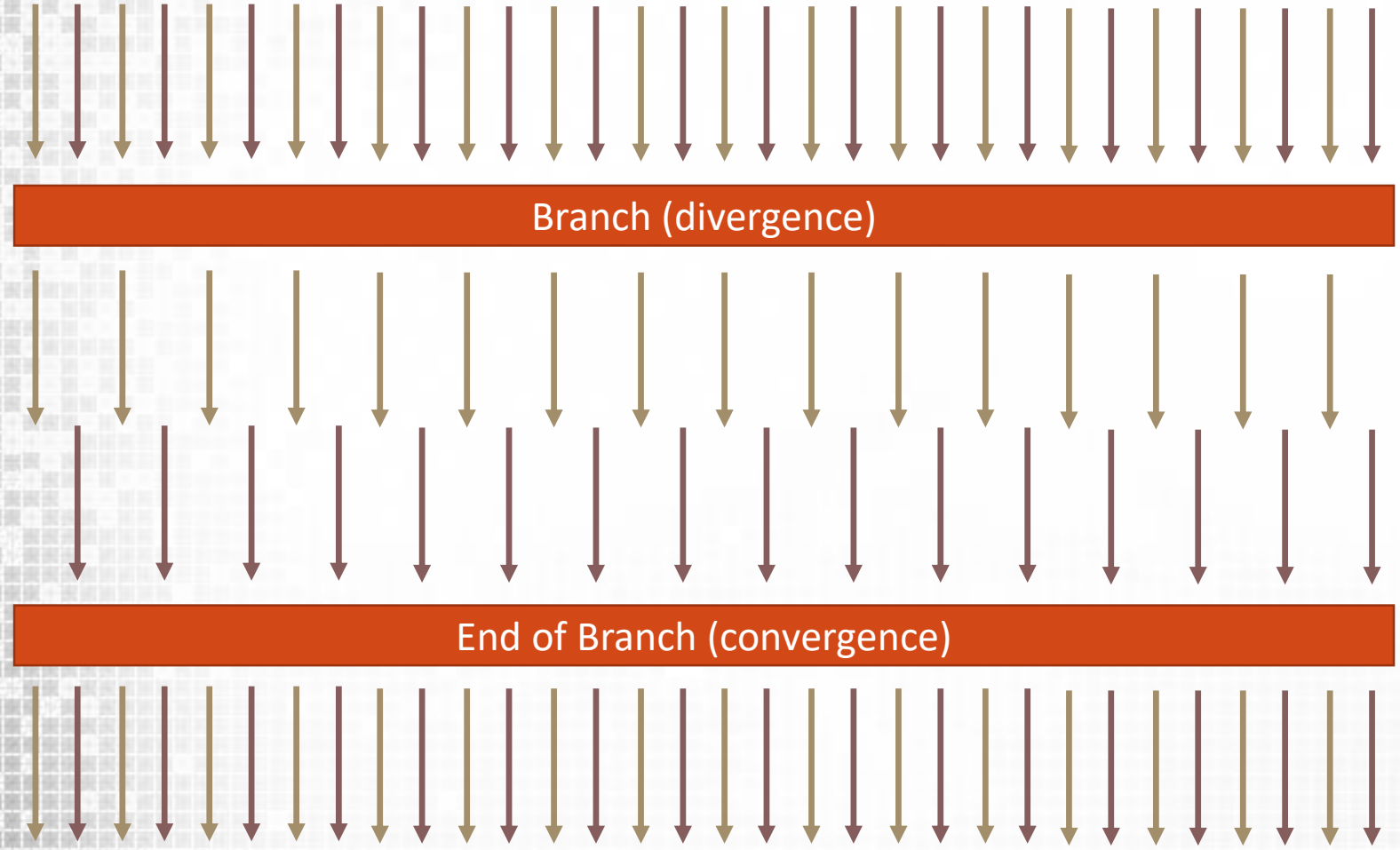
- Which is coherent?
- Which is divergent?

The University Of Sheffield.

NVIDIA GPU RESEARCH CENTER

Divergence Example

All warps

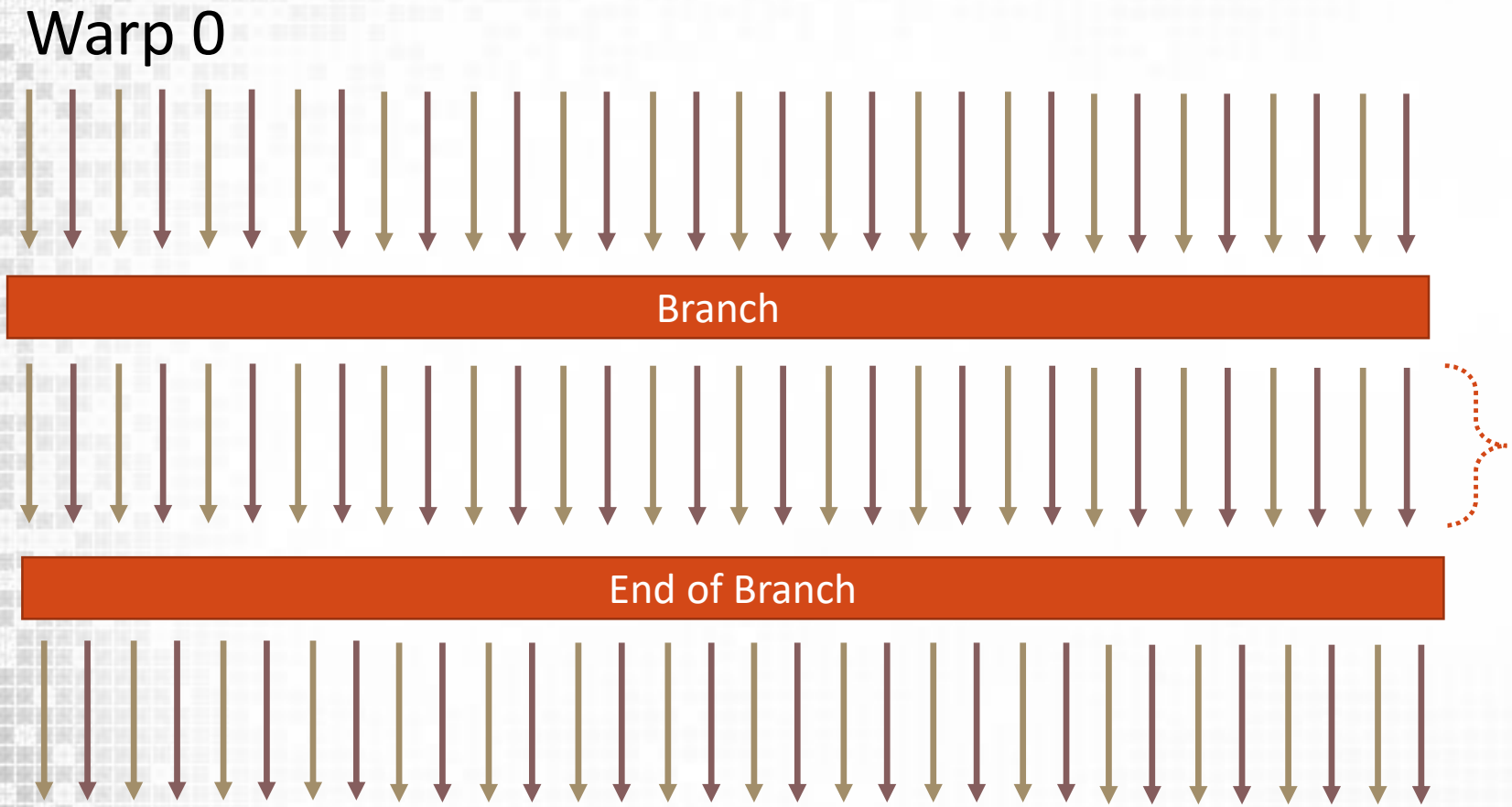



```
__global__ void a_kernel()
{
    if (threadIdx.x % 2)
        //something
    else
        //something else
}
```

```
if (threadIdx.x % 2)
```

```
else
```

Divergence Example Alternative

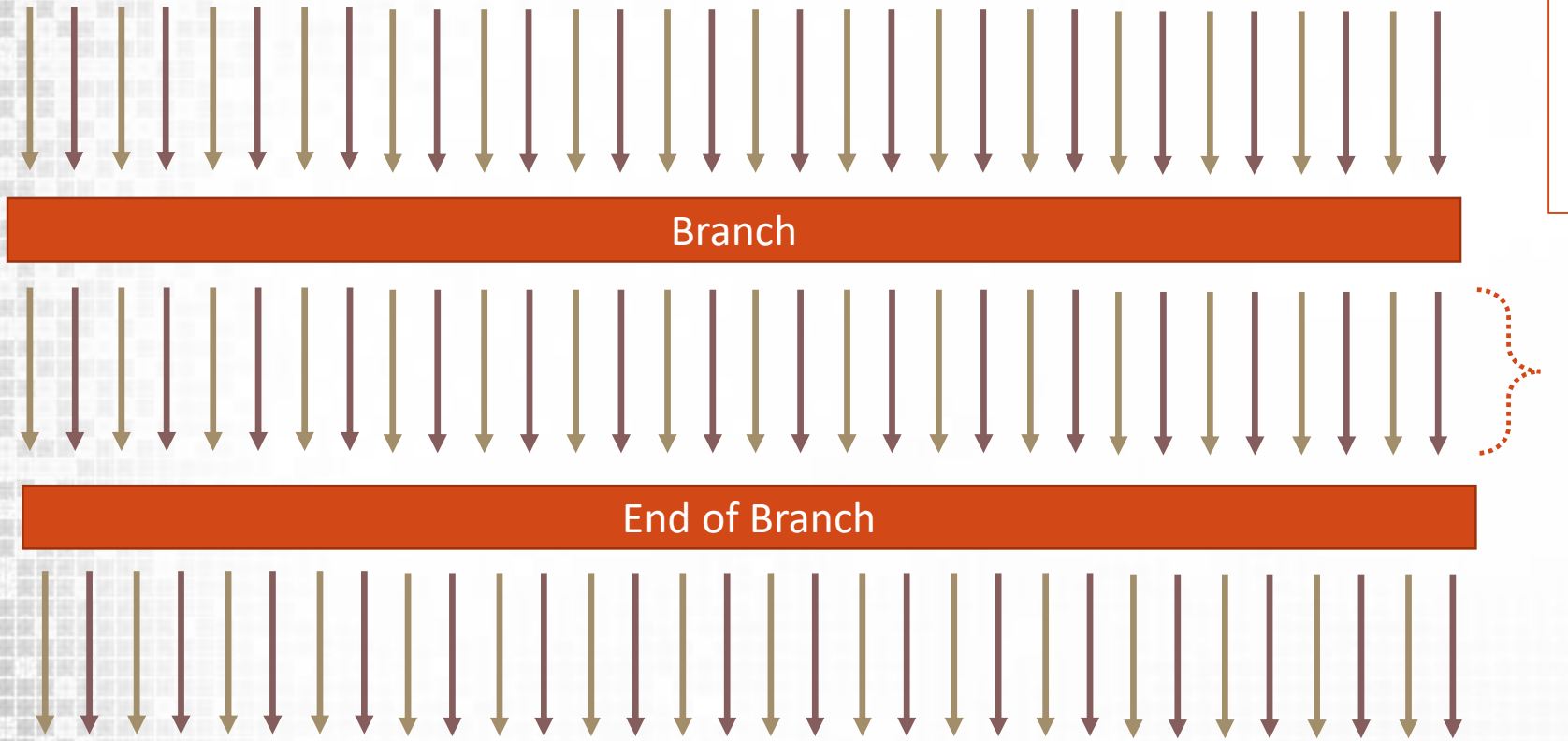



```
__global__ void a_kernel()
{
    if (blockIdx.x % 2)
        //something
    else
        //something else
}
```

```
if (blockIdx.x % 2)
```

Divergence Example Alternative

Warp 1 (assuming thread block size of 32)




```
__global__ void a_kernel()
{
    if (blockIdx.x % 2)
        //something
    else
        //something else
}
```

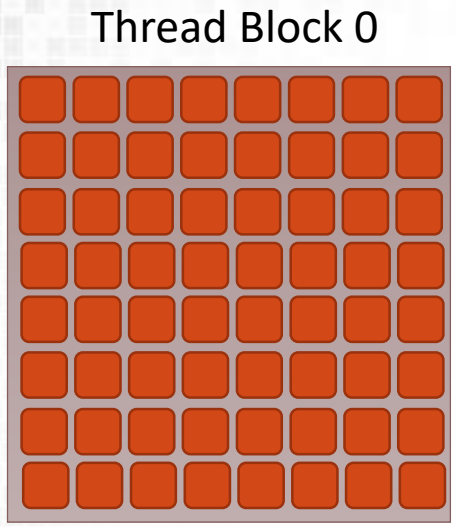
Levels of divergence

- Divergent code can be classified by how many "ways" it diverges.
  - E.g. the following examples are 4-way divergent (and functionally equivalent)
- If a warp has 32-way divergence this will have a **massive** impact on performance!

```
__global__ void a_kernel(int *a)
{
    int a = a[threadIdx.x + blockIdx.x*blockDim.x]
    if (a==0)
        //code for case 0
    else if (a==1)
        //code for case 1
    else if (a==2)
        //code for case 2
    else if (a==3)
        //code for case 3
}
```

```
__global__ void a_kernel(int *a)
{
    int a = a[threadIdx.x + blockIdx.x*blockDim.x]
    switch (a){
      case(0):
        //code for case 0 with break
      case(1):
        //code for case 1 with break
      case(2)
        //code for case 2 with break
      case(3)
        //code for case 3 with break
    }
}
```

2D blocks and divergence



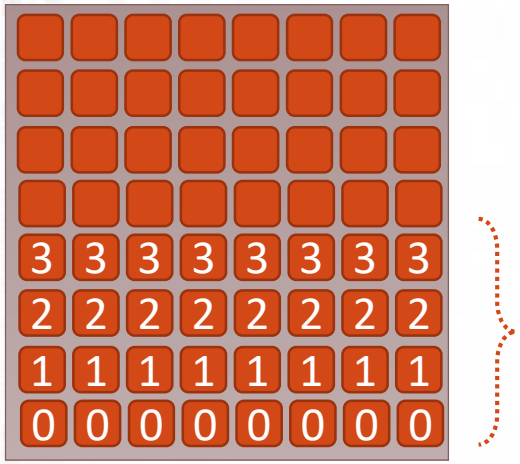

```
__global__ void a_kernel()
{
    if (threadIdx.y % 2)
        //something
    else
        //something else
}
```

```
__global__ void b_kernel()
{
    if (threadIdx.y / 4)
        //something
    else
        //something else
}
```

- How many ways of divergence?

2D blocks and divergence

Thread Block 0 – showing `threadIdx.y`




```
__global__ void a_kernel()
{
    if (threadIdx.y % 2)
        //something
    else
        //something else
}
```

2

```
__global__ void b_kernel()
{
    if (threadIdx.y / 4)
        //something
    else
        //something else
}
```

0

- How many ways of divergence?

Branching vs. Predication

- Predication is an optional guard that can be applied to machine instructions
  - A predicate is set in predicate registers (virtual registers)
  - Predicates are unique to each thread
- Depending on the predicate value the instruction can be conditionally executed
  - `NOP` otherwise
- How does this differ to branching?
  - No labels or change in program counter
  - Smaller more compact code
    - Less operations = better performance

Branching code

CUDA C

```
int a = 0;

if (i < n)
  a = 1;
else
  a = 2;
```

PTX ISA

```
        mov.s32 a, 0;           //a=0
        setp.lt.s32 p, i, n;    //p=(i<n)
@!p     bra A_FALSE;
A_TRUE:                         //if true
        mov.s32 a, 1;           //a=1
        bra A_END;
A_FALSE:                        //if false
        mov.s32 a, 2;           //a=2
A_END:
        ...
```

- Consider the following branching code…
- Code is PTX ISA
  - A low-level parallel thread execution virtual machine and instruction set architecture (ISA) for CUDA
  - Independent of NVIDIA GPU architecture
  - Used to generate native target architecture machine instructions

Branching code using predicate

CUDA C

```
int a = 0;

if (i < n)
  a = 1;
else
  a = 2;
```

PTX ISA (compiler optimised)

```
        mov.s32 a, 0;           //a=0
        setp.lt.s32 p, i, n;    //p=(i<n)
@p      mov.s32 a, 1;           //a=1
@!p     mov.s32 a, 2;           //a=2
```

CUDA C (improved)

```
int a = 0;
a = (i < n)? 1: 2;
```

PTX ISA (improved)

```
        mov.s32 a, 0;           //a=0
        setp.lt.s32 p, i, n;    //p=(i<n)
        selp a, 1, 2, p         //a=(p)?1:2
```

- Consider the following branching code…
- In this case the predicate can be used to reduce the number of instructions
- The compiler is good at balancing branching and predication
- Can hint to the compiler by using ternary operators

- ❑Warp Scheduling & Divergence
- ❑Atomics
- ❑Warp Operations

What is wrong with the following

```
__global__ void max_kernel(int *a)
{
  __shared__ int max;

  int my_local = a[threadIdx.x + blockIdx.x*blockDim.x];

  if (my_local > max)
    max = my_local;
}
```

- More than one thread may try to modify max at the same time
  - Race condition

```
__global__ void max_kernel(int *a)
{
  __shared__ int max;

  int my_local = a[threadIdx.x + blockIdx.x*blockDim.x];

  if (my_local > max)
    max = my_local;
}
```

Atomics

- Atomics are used to ensure correctness when concurrently reading and writing to a memory location (global or shared)

```
__global__ void max_kernel(int *a)
{
  __shared__ int max;

  int my_local = a[threadIdx.x + blockIdx.x*blockDim.x];

  if (my_local > max)
    max = atomicMax(&max, my_local);
}
```

- No race condition
- Function *supported* in (some) hardware
  - Support varies depending on which memory is used (global, shared etc.)

Atomic Functions and Locks

- An atomic function
  - Must guarantee that an operation can complete without interference from any other thread
  - Does not provide any guarantee of ordering or provide any synchronisation
- How can we implement critical sections?

```
__device__ int lock = 0;

__global__ void kernel() {
  bool need_lock = true;
  // get lock
  while (need_lock) {
    if (atomicCAS(&lock, 0, 1)==0) {
      //critical code section
      atomicExch(&lock, 0);
      need_lock = false;
    }
  }
}
```

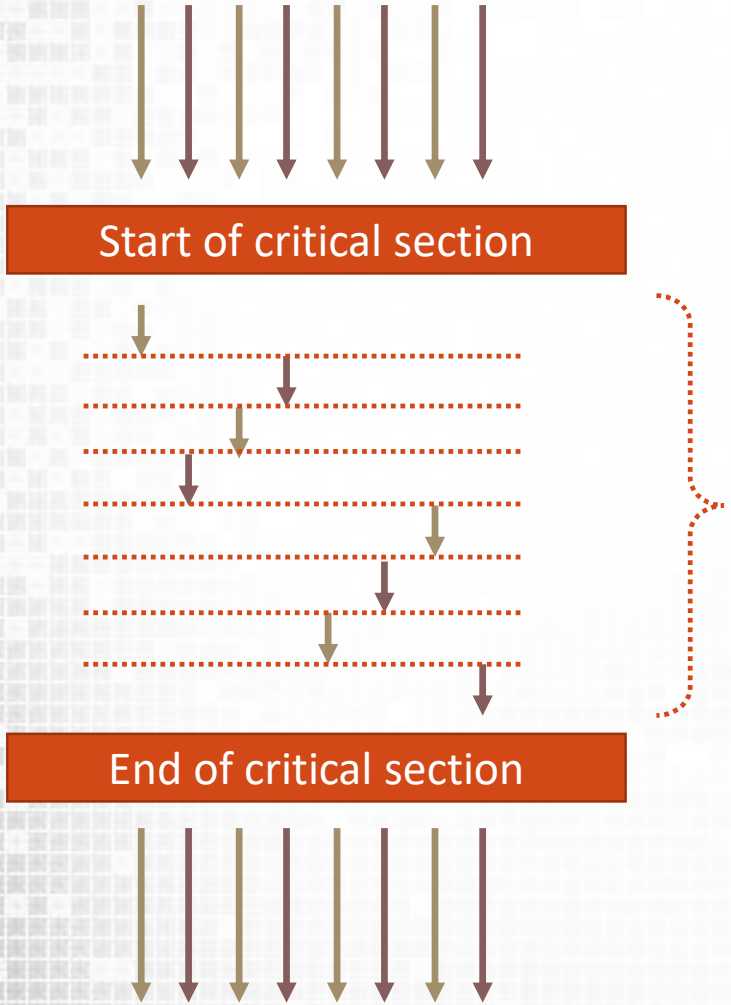
```
int atomicCAS(int* address, int compare, int val)
```

Performs the following in a single atomic transaction (atomic instruction)

```
*address=(*address==compare)? val : *address;
```

Returning the old value at the address

Serialisation



- What happens to performance when using atomics?
- In the case of the critical section example
  - This is serialised for each thread accessing the shared value
- For the atomic CAS instruction access to the shared lock variable is serialised
  - This is true of any atomic function or instruction in CUDA

CUDA Atomic Functions / Instructions

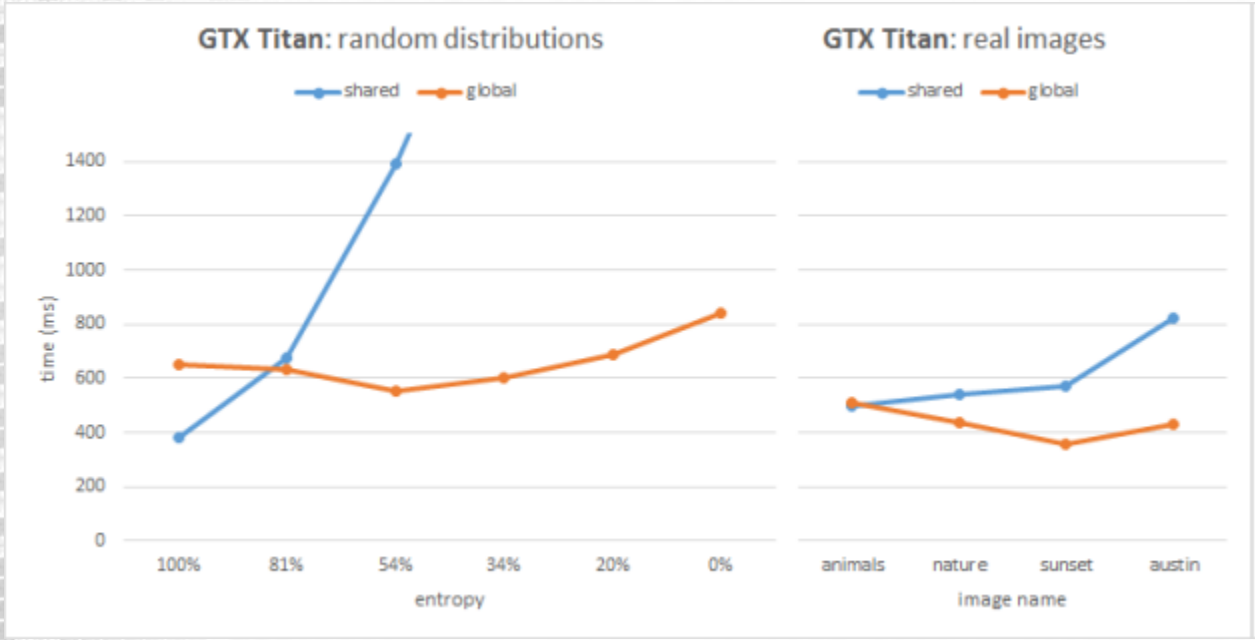
- In addition to `atomicCAS` the following atomic functions/instructions are available
  - Addition/subtraction
    - E.g. `int atomicAdd(int* address, int val)` – add `val` to integer at `address`
  - Exchange
    - Exchange a value with a new value
  - Increment/Decrement
  - Minimum and Maximum
- Variants of atomic functions
  - Floating point versions require Compute 2.0
  - 64 bit integer and double versions available in Pascal (Compute 6.0)
  - See docs: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions

Shared vs Global Atomics

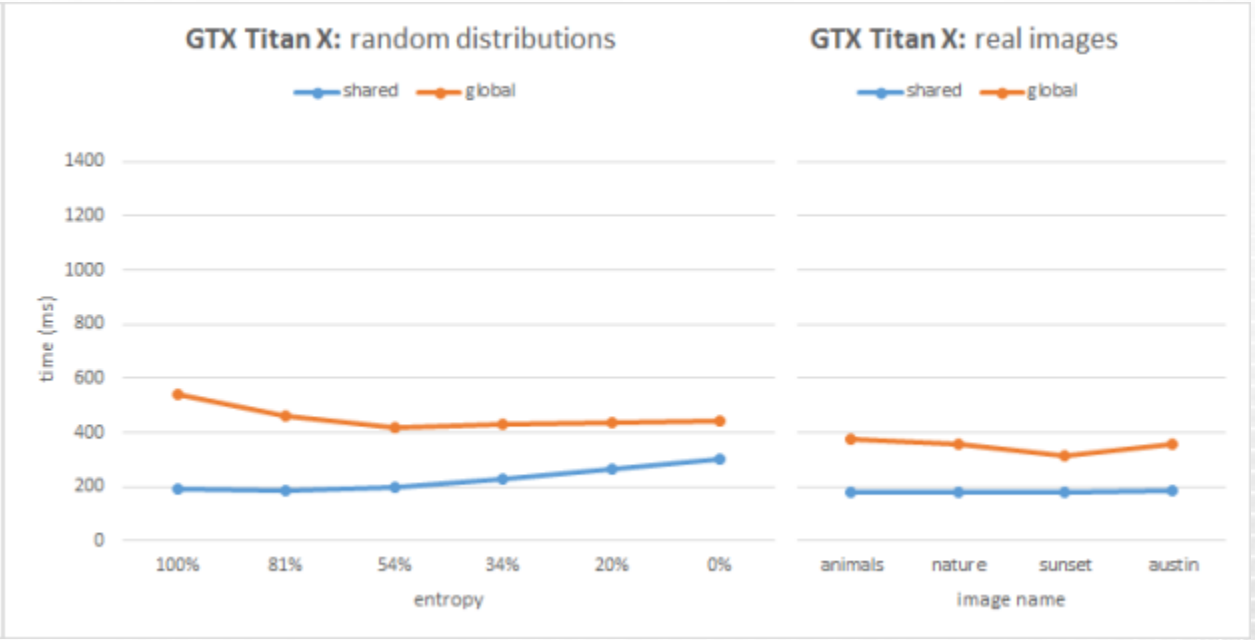
- Global Atomics
  - Fermi: Atomics are not cached and are hence very slow
  - Kepler and Maxwell: both use L2 caching of global atomics
- Shared Memory Atomics
  - Fermi and Kepler: No hardware support for SM atomics
    - Emulated using locks in software
    - Poor when there is high contention
    - Sometimes worse than global atomics
  - Maxwell+: Hardware supported SM atomics
    - Much improved performance

Local vs Global Atomics

Kepler



Maxwell



- Image histogram example
  - Accumulation of colour values for images
  - Entropy: measure of the level of disorder (lower entropy == higher contention)
  - https://devblogs.nvidia.com/parallelforall/gpu-pro-tip-fast-histograms-using-shared-atomics-maxwell/

The University Of Sheffield.

NVIDIA GPU RESEARCH CENTER

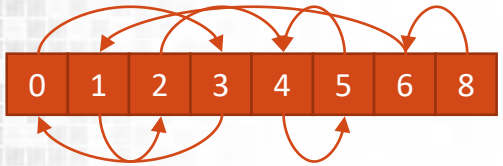
- ❑Warp Scheduling & Divergence
- ❑Atomics
- ❑Warp Operations

Warp Shuffle

- For moving/comparing data between threads in a block it is possible to use Shared Memory (SM)
- For moving/comparing data between threads in a warp (known as lanes in this context) it is possible to use a *warp shuffle* (SHFL)
  - Direct exchange of information between two threads
    - Can replace atomics
    - Should never depend on conditional execution!
  - Does not require SM
    - Always faster than SM equivalent
  - Implicit synchronisation (no need for `__syncthreads`)
    - EXCEPT on Volta hardware
  - Works by allowing threads to read another threads registers
  - Available on Kepler and Maxwell

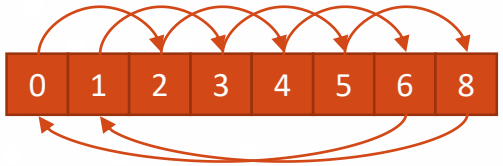
Shuffle Variants

`__shfl()`



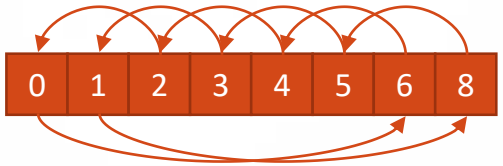
Shuffled between any two index threads

`__shfl_up()`



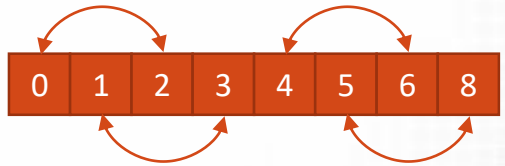
Shuffles to n^{th} right neighbour wrapping indices (in this case n=2)

`__shfl_down()`



Shuffles to n^{th} left neighbour wrapping indices (in this case n=2)

`__shfl_xor()`



Butterfly (XOR) exchange shuffle pattern

Shuffle function arguments

- `int __shfl(int var, int srcLane, int width=warpSize);`
  - Direct copy of `var` in `srcLane`
- `int __shfl_up(int var, unsigned int delta, int width=warpSize);`
- `int __shfl_down(int var, unsigned int delta, int width=warpSize);`
  - delta is the n step used for shuffling
- `int __shfl_xor(int var, int laneMask, int width=warpSize);`
  - Source lane determined by bitwise XOR with `laneMask`

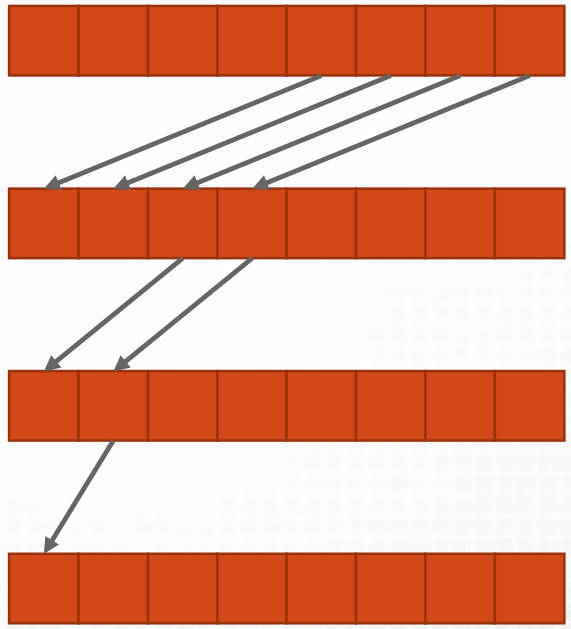
- Optional width argument
  - Must be a power of 2 and less than or equal to warp size
  - If smaller than warp size each subsection acts independently (own wrapping)
- All functions available as float and half versions
  - http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#warp-shuffle-functions

Shuffle Warp Sum Example (down)

```
__global__ void sum_warp_kernel_shfl_down(int *a)
{
  int local_sum = a[threadIdx.x + blockIdx.x*blockDim.x];

  for (int offset = WARP_SIZE / 2; offset>0; offset /= 2)
    local_sum += __shfl_down(local_sum, offset);

  if (threadIdx.x%32 == 0)
    printf("Warp max is %d", local_sum)
}
```



offset==4

offset==2

offset==1

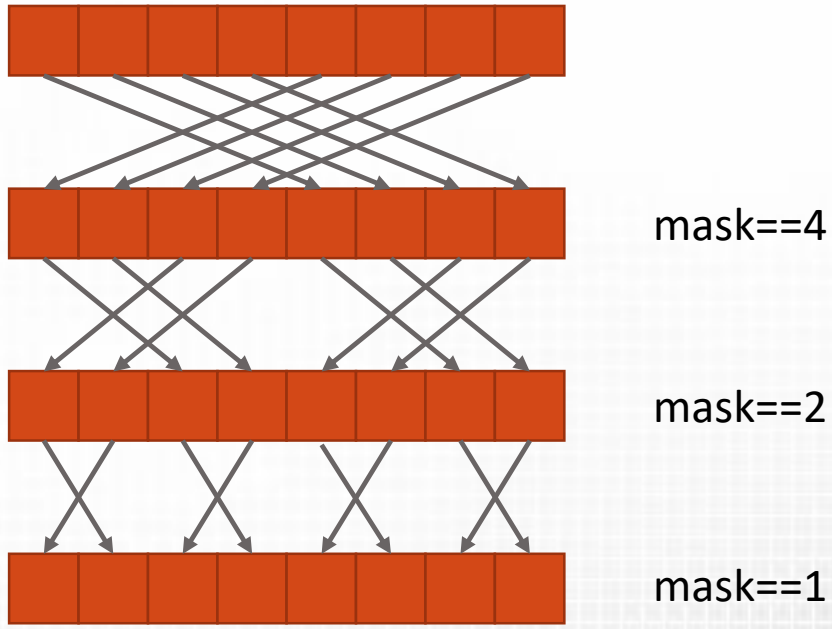
Warp sum in `threadIdx.x%32==0`

Shuffle Warp Sum Example (xor)

```
__global__ void sum_warp_kernel_shfl_xor(int *a)
{
  int local_sum = a[threadIdx.x + blockIdx.x*blockDim.x];

  for (int mask = WARP_SIZE / 2; mask>0; mask /= 2)
    local_sum += __shfl_xor(local_sum, mask);

  if (threadIdx.x%32 == 0)
    printf("Warp max is %d", local_sum)
}
```



Warp sum in all threads

Warp Voting

- Warp shuffles allow data to be exchanged between threads in a warp
- Warp voting allows threads to test a condition across all threads in a warp
  - `int all(condition)`
    - True if the condition is met by all threads in the warp
  - `int any(condition)`
    - True is any thread in warp meets condition
  - `unsigned int ballot(condition)`
    - Sets the n^{th} bit of the return value based on the n^{th} threads condition value
- All warp voting functions are single instruction and act as barrier
  - Only active threads participate, does not block like `syncthreads()`

Warp Voting Example

```
__global__ void voteAllKernel(unsigned int *input, unsigned int *result)
{
  int i = threadIdx.x + blockIdx.x*blockDim.x;
  int j = i % WARP_SIZE;

  int vote_result = all(input[i]);

  if (j==0)
    result[j] = vote_result;
```

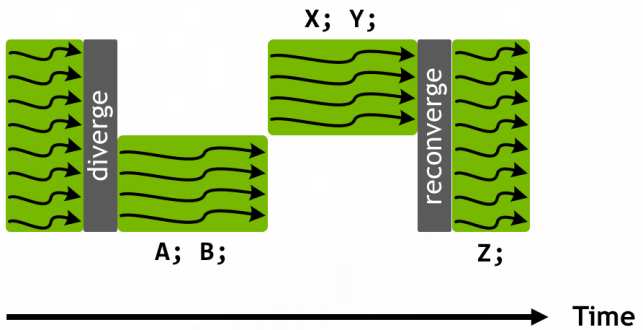
- For each first thread in the warp calculate if all threads in the warp have `true` valued input
- Save the warp vote to a compact array
  - A reduction of factor 32

__shfl_sync

- Volta hardware allows interleaved execution of statements from divergent branches
  - Each thread has its own program counter to allow this

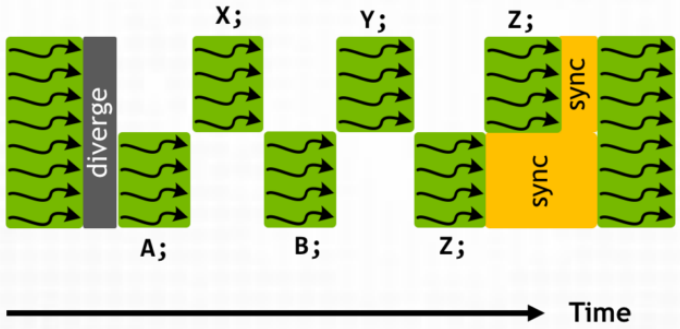
```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```

Pre-Volta hardware




```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
__syncwarp()
```

Volta hardware



- As a result warp operations require a synchronised version
  - CUDA 8:

```
int __shfl(int var, int srcLane, int width=warpSize);
```

  - CUDA 9:
    - `int __shfl(unsigned int mask, int var, int srcLane, int width=warpSize);`
- A mask of `0xFFFFFFFF` will sync whole warp and act like CUDA 8 shuffle
- Allow syncing of units smaller than a warp

The University Of Sheffield.

NVIDIA GPU RESEARCH CENTER

Global Communication

- Shared memory is per thread block
- Shuffles and voting for warp level
- Atomics can be used for some global (grid wide) operations
- What about general global communication?
  - Not possible within a kernel (*except in Volta – not covered*)!
  - Remember a grid may not be entirely in flight on the device
  - Can be enforced by finishing the kernel

```
step1 <<<grid, blk >>>(input, step1_output);
// step1_output can safely be used as input for step2
step2 <<<grid, blk >>>(step1_output, step2_output);
```

Summary

- Warps are the level in which threads are grouped for execution
- Divergent code paths within a warp are very bad for performance
- Warps can communicate directly via warp shuffles and voting
- The performance of warp communication is very fast (single instruction)
- Atomic can be used to allow threads co-operative access to a shared variable
- Atomic performance varies greatly with different architectures

Acknowledgements and Further Reading

- Predication: http://docs.nvidia.com/cuda/parallel-thread-execution/index.html#predicated-execution
- Shuffling: http://on-demand.gputechconf.com/gtc/2013/presentations/S3174-Kepler-Shuffle-Tips-Tricks.pdf
- Volta: https://devblogs.nvidia.com/cuda-9-features-revealed/