Parallel Computing with GPUs

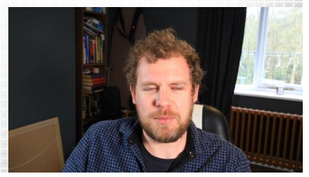
Warp Level CUDA and Atomics Part 1 – Warp Scheduling and Divergence

The University Of Sheffield.

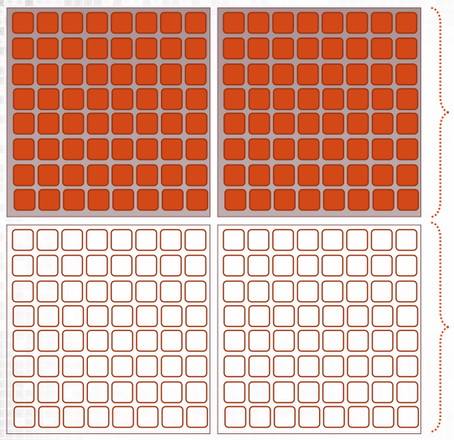
Dr Paul Richmond
http://paulrichmond.shef.ac.uk/teaching/COM4521/

This Lecture (learning objectives)

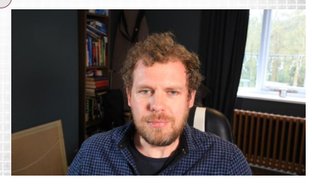
- Warp Scheduling and Divergence
  - Summarise thread block scheduling
  - Examine warp scheduling
  - Define warp divergence and explore the impact via examples



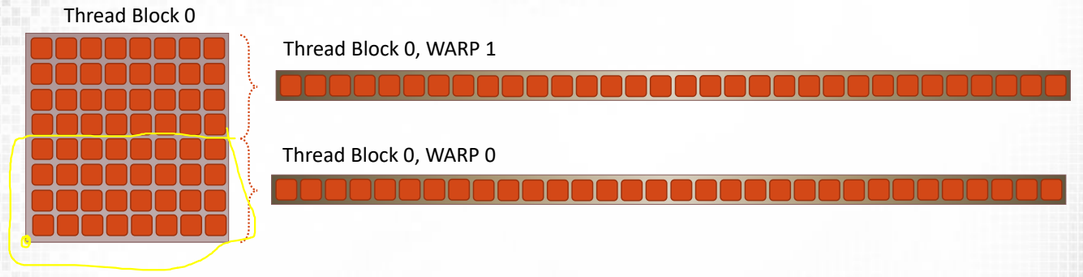
Thread Block Scheduling



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



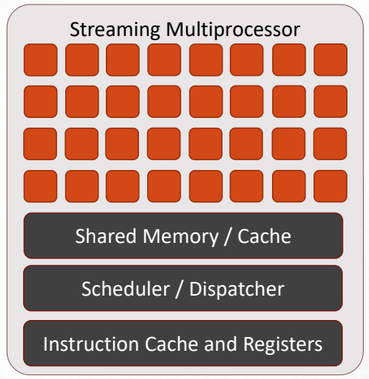
Thread Block Scheduling



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

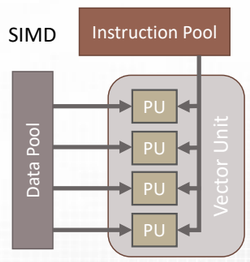
Warp Scheduling

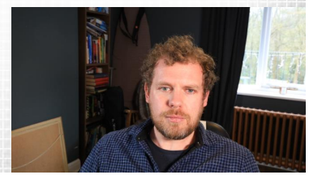
- Zero overhead to swap warps (warp scheduling)
  - Warps contain only threads from a single thread block
  - Warps can be swapped with warps from different blocks assigned to the same streaming multi processor
  - At any one time only one warp has operations being executed
    - Memory movement happens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



Warps (pre Volta)

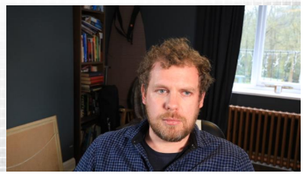
- Execution of GPU instructions is always in groups of threads called **warps**
- A warp has a single program counter (pre-volta)
- Within a warps threads always execute the same instruction (SIMD/SIMT)
- What happens if code within a warp has different control flow?
  - **Branch Divergence**





Divergent Threads

- All threads in warp execute the same instruction
  - 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

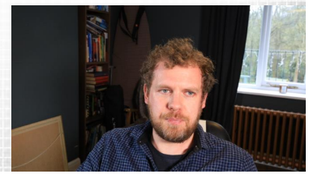


Divergent and Coherent Code

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

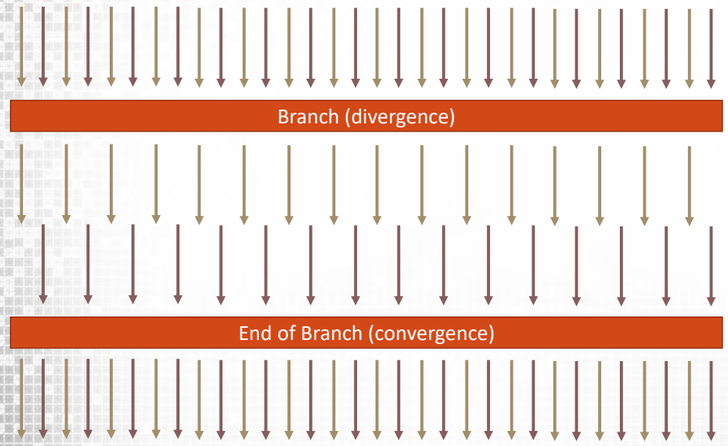
```
__global__ void b_kernel()
{
    if (threadIdx.x % 2)
        //something
    else
        //something else
}
```

- Which is coherent?
- Which is divergent?

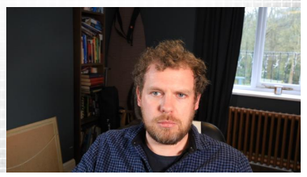


Divergence Example

All warps

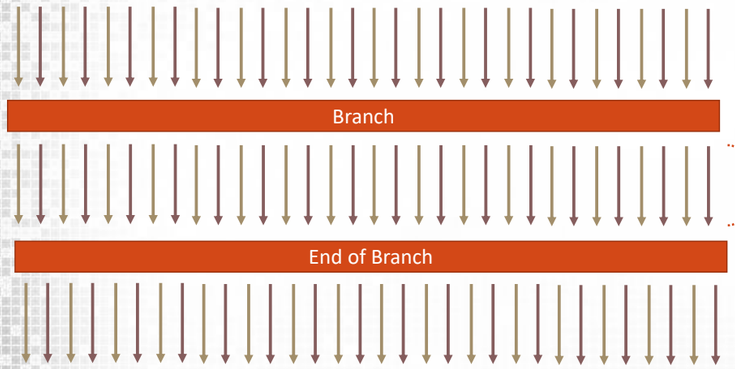



```
__global__ void a_kernel()
{
    if (threadIdx.x % 2)
        //something
    else
        //something else
}
```

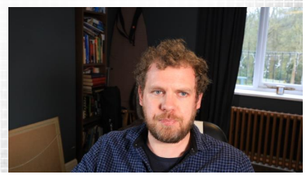


Divergence Example Alternative

Warp 0

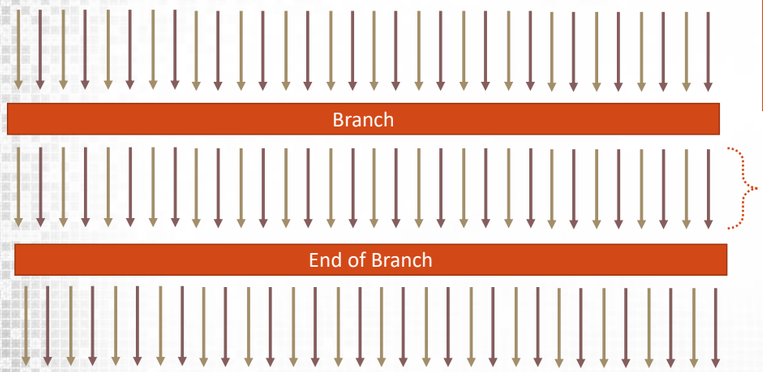



```
__global__ void a_kernel()
{
    if (blockIdx.x % 2)
        //something
    else
        //something else
}
```

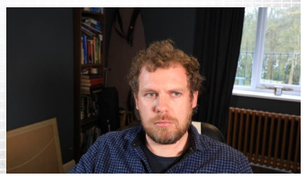


Divergence Example Alternative

Warp 1 (assuming thread block size of 32)

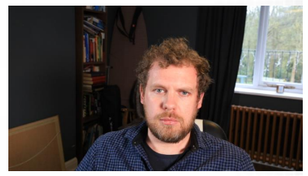



```
__global__ void a_kernel()
{
    if (blockIdx.x % 2)
        //something
    else
        //something else
}
```



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!

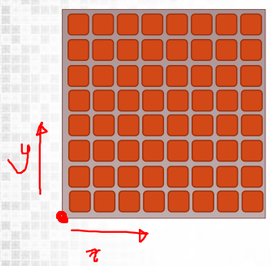


```
__global__ void a_kernel(int *a)
{
    int a = a[threadIdx.x + blockIdx.x*blockDim.x]
    if (a==0)
        //code for case 0
    else if (a==1)
        //code for case 1
    else if (a==2)
        //code for case 2
    else if (a==3)
        //code for case 3
}
```

```
__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

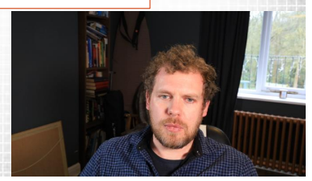
Thread Block 0




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

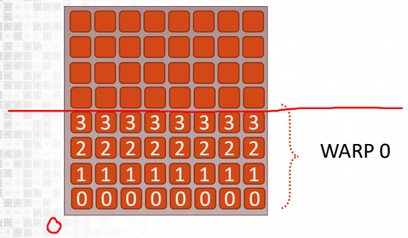
```
__global__ void b_kernel()
{
    if (threadIdx.y / 4)
        //something
    else
        //something else
}
```

- How many ways of divergence?



2D blocks and divergence

Thread Block 0 – showing `threadIdx.y`



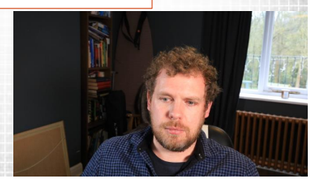

```
__global__ void a_kernel()
{
    if (threadIdx.y % 2)
        //something
    else
        //something else
}
```

2

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

0

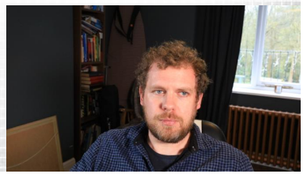
- How many ways of divergence?



Summary

- Warp Scheduling and Divergence
  - Summarise thread block scheduling
  - Examine warp scheduling
  - Define warp divergence and explore the impact via examples

- Next Lecture: Advanced Divergence (and Volta)

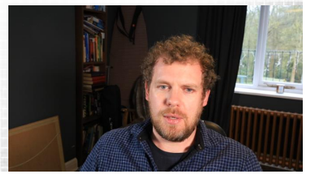


Parallel Computing with GPUs

Warp Level CUDA and Atomics
Part 2 – Advanced Divergence

The University Of Sheffield.

Dr Paul Richmond
http://paulrichmond.shef.ac.uk/teaching/COM4521/



This Lecture (learning objectives)

- Advanced Divergence
  - Compare divergence with predication
  - Present changes to instruction operation in Volta+ hardware
  - Explain the impact of Volta hardware changes on scheduling

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

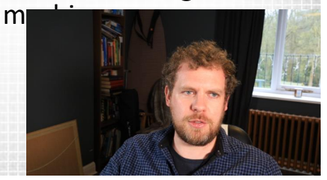
CUDA C

```
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 m… 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
```

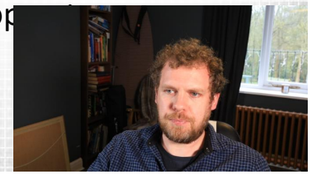
CUDA C (improved)

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

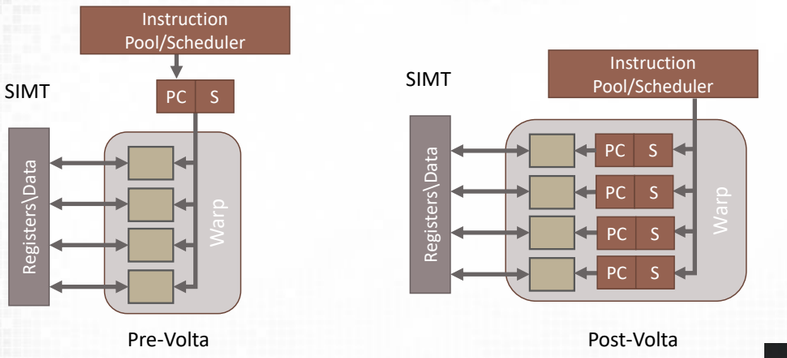
PTX ISA (improved)

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

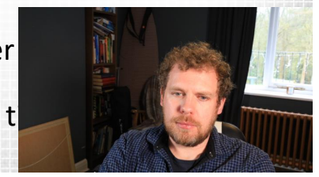
- Consider the following branching code…
- In this case the predicate can be used to reduce the number of instructions
- The compiler is good at balancing branching and predication
- Can hint to the compiler by using ternary op…



Volta+ hardware changes

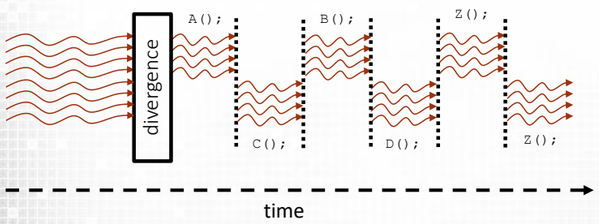


- Volta hardware gives each thread its own Program Counter Counter (S)
- Warps still only able to **execute** a single instruction at one t

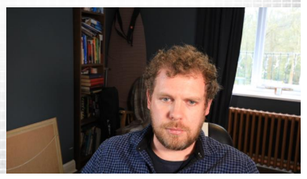


Volta Execution

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

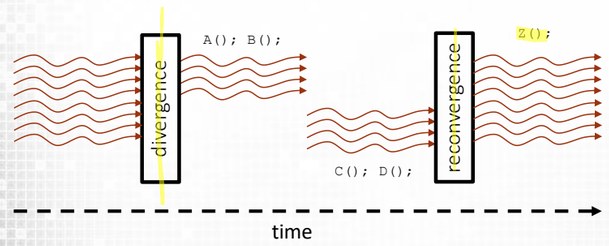


- Scheduler can yield execution to any thread at sub warp level
- Latency in functions can be masked by switching to active threads in other parts of the warp.

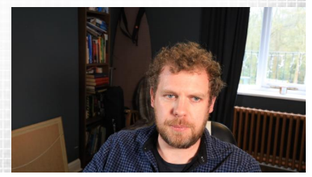


Pre-Volta Execution

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

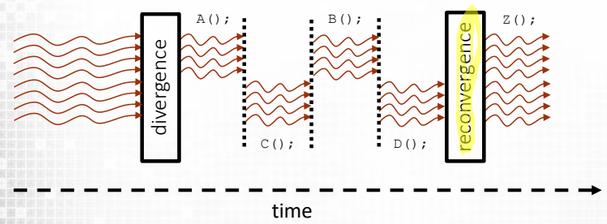


- Divergent threads only reconverge after branch evaluation
- Latency in A or B must be masked by *other* warps

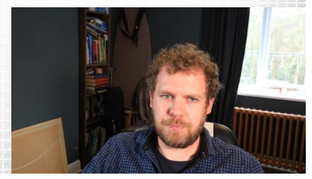


Warp Synchronisation

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



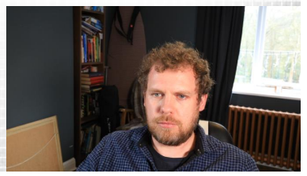
- The compiler will converge warps automatically
- Warps can be forced to reconverge
- Helpful to ensure greater effeciancy



Summary

- Advanced Divergence
  - Compare divergence with predication
  - Present changes to instruction operation in Volta+ hardware
  - Explain the impact of Volta hardware changes on scheduling

- Next Lecture: Atomics and Warp Operations



Parallel Computing with GPUs

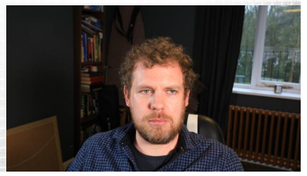
Warp Level CUDA and Atomics Part 3 – Atomics and Warp Operations

The University Of Sheffield.

Dr Paul Richmond
http://paulrichmond.shef.ac.uk/teaching/COM4521/

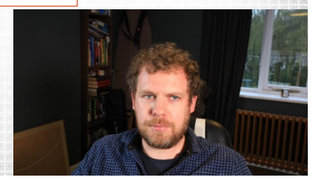
This Lecture (learning objectives)

- Atomics
  - Present GPU atomic operations
  - Demonstrate performance of GPU atomics and locks
- Warp Operations
  - Present warp shuffle operations for communication of threads within a warp
  - Give examples of warp operations such as sum and warp voting.



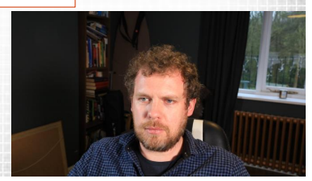
What is wrong with the following

```
__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;
}
```



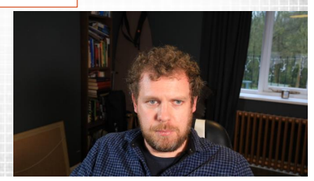
Atomics

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

No need for assignment to a variable when using the atomic functions

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

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

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

- No race condition
- Function supported in *most* hardware
  - Some older generation GPUs lack shared memory and floating point atomic 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;
    }
  }
}
```

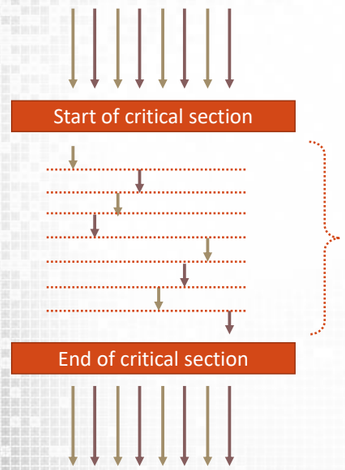
```
int atomicCAS(int* address, int compare, int val)
```

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

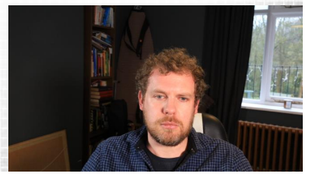
```
*address=(*address==compare)? val : *address;
```

Returning the old value at the address

Serialisation

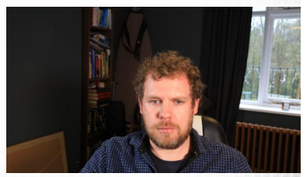


- What happens to performance when using atomics?
- In the case of the critical section example
  - This is serialised for each thread accessing the shared value
- For the atomic CAS instruction access to the shared lock variable is serialised
  - This is true of any atomic function or instruction in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



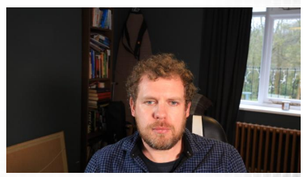
CUDA Atomic Functions / Instructions

- 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
  - 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



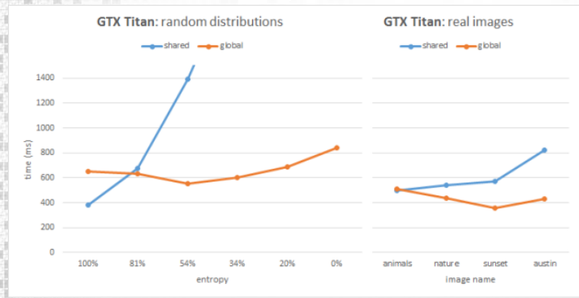
Warp Shuffle

- 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 (use `__syncwarp`)
  - Works by allowing threads to read another threads registers
  - https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#warp-shuffle-functions

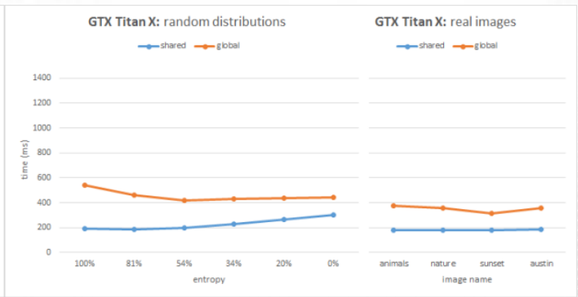


Local vs Global Atomics

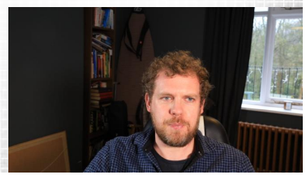
Kepler



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/



Shuffle Variants

`__shfl_sync()`

Shuffled between any two index threads

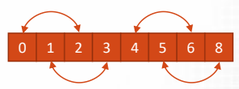
`__shfl_up_sync()`

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

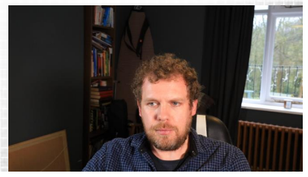
`__shfl_down_sync()`

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

`__shfl_xor_sync()`



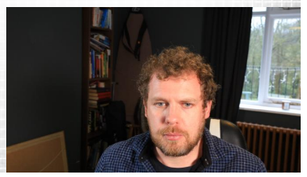
Butterfly (XOR) exchange shuffle pattern



Shuffle function arguments

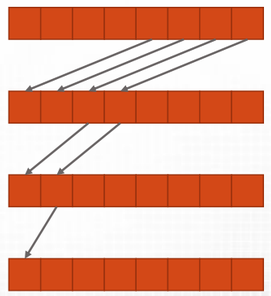
- `T __shfl_sync(unsigned mask, T var, int srcLane, int width=warpSize);`
- `T __shfl_up_sync(unsigned mask, T var, unsigned int delta, int width=warpSize);`
- `T __shfl_down_sync(unsigned mask, T var, unsigned int delta, int width=warpSize);`
  - delta is the n step used for shuffling
- `T __shfl_xor_sync(unsigned mask, T var, int laneMask, int width=warpSize);`
  - Source lane determined by bitwise XOR with `laneMask`

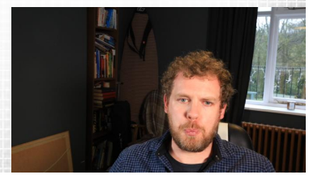
- Mask is a bit mask for the warp to indicate which threads participate
- T can be `int`, `unsigned int`, `long`, `unsigned long`, `long long`, `unsigned long long`, `float` or `double`
- Optional width argument
  - Must be a power of 2 and less than or equal to warp size
  - If smaller than warp size each subsection acts independently (own wrapping)



Shuffle Warp Sum Example (down)

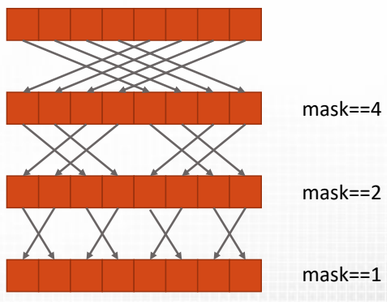
```
__global__ void sum_warp_kernel_shfl_down(int *a)
{
  int local_sum = a[threadIdx.x + blockIdx.x*blockDim.x];

  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)
}
```

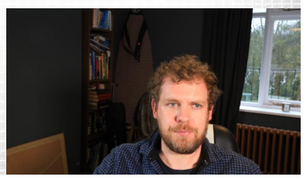




Shuffle Warp Sum Example (xor)

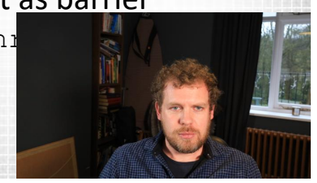
```
__global__ void sum_warp_kernel_shfl_xor(int *a)
{
  int local_sum = a[threadIdx.x + blockIdx.x*blockDim.x];

  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 Voting

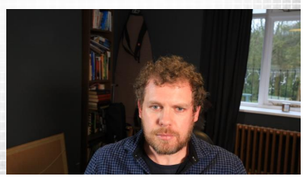
- Warp shuffles allow data to be exchanged between threads in a warp
- Warp voting allows threads to test a condition across all threads in a warp
  - `int all(condition)`
    - True if the condition is met by all threads in the warp
  - `int any(condition)`
    - True is any thread in warp meets condition
  - `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 `syncthr`



Warp Voting Example

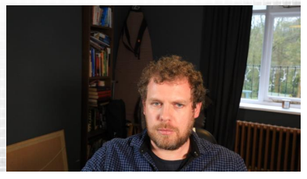
```
__global__ void voteAllKernel(unsigned int *input, unsigned int *result)
{
  int i = threadIdx.x + blockIdx.x*blockDim.x;
  int j = i % WARP_SIZE;

  int vote_result = all(input[i]);

  if (j==0)
    result[i / WARP_SIZE] = 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



Summary

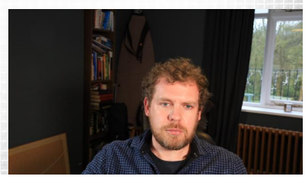
- Atomics
  - Present GPU atomic operations
  - Demonstrate performance of GPU atomics and locks
- Warp Operations
  - Present warp shuffle operations for communication of threads within a warp
  - Give examples of warp operations such as sum and warp voting.



Global Communication Summary

- 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);
```



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/