Parallel Computing with GPUs

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

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)

```
__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 need for assignment to a variable when using the atomic functions

❑ 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;
    }
  }
}
```

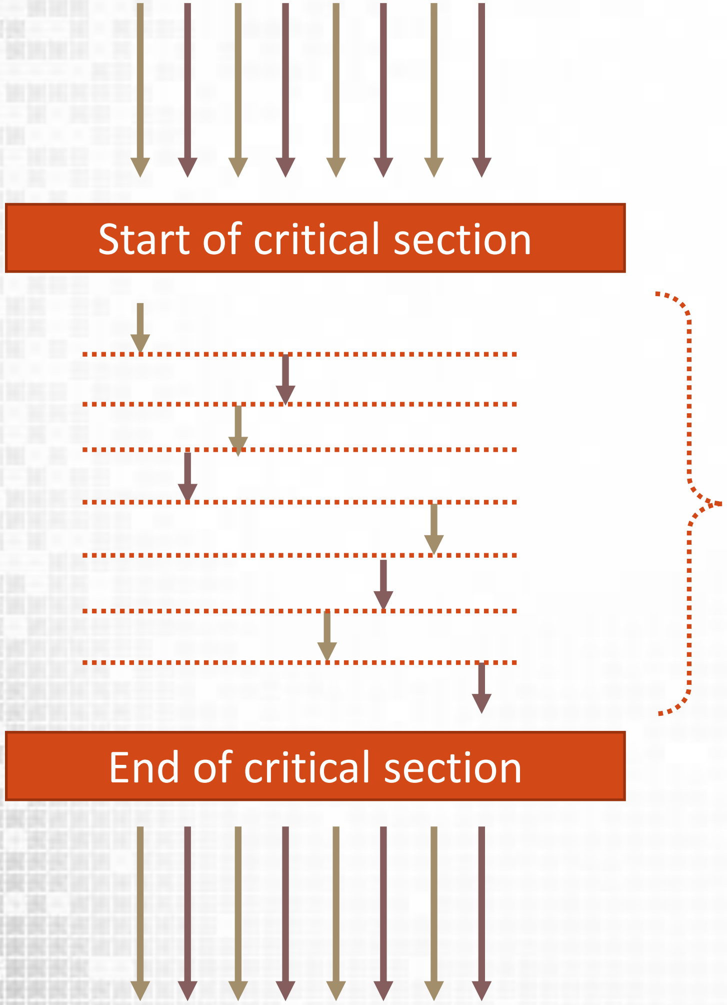
```
int atomicCAS(int* address, int compare, int val)
```

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

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

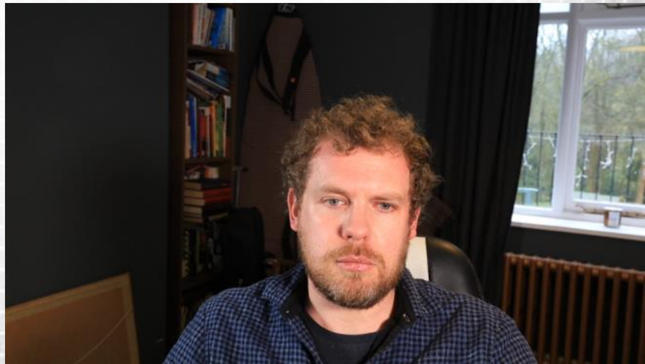
Returning the old value at the address

Serialisation



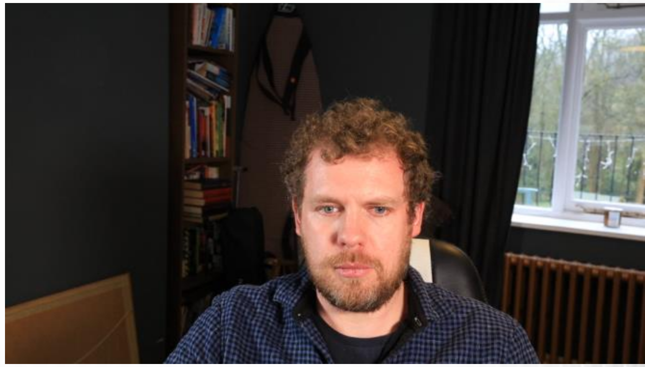
- ❑ What happens to performance when using atomics?

- ❑ 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

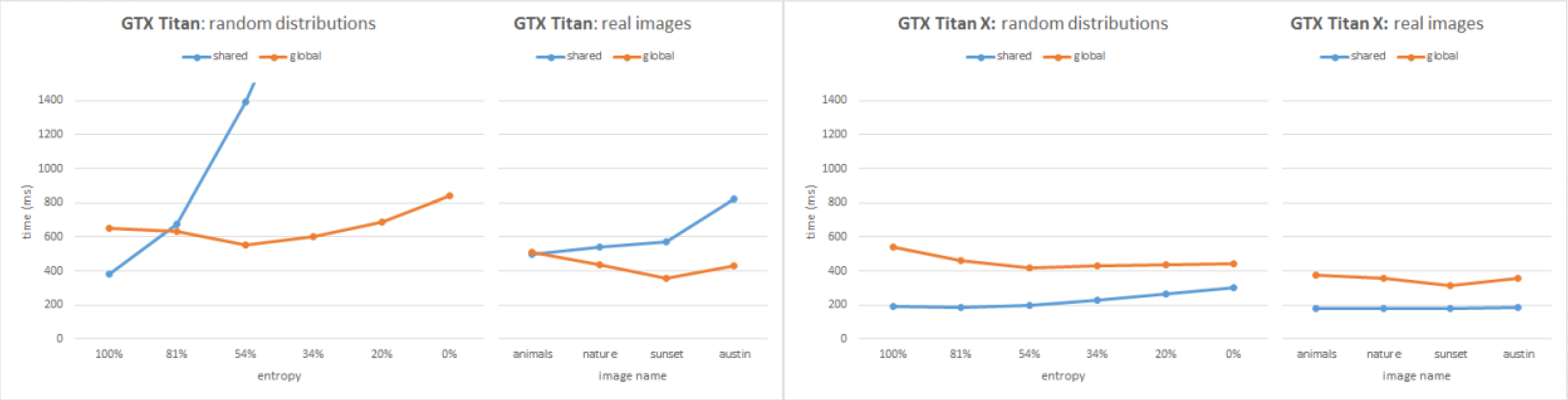
- In addition to `atomicCAS` the following atomic functions/instructions are available
  - Addition/subtraction
    - E.g. `int atomicAdd(int* address, int val)` – add `val` to integer at `address`
  - Exchange
    - Exchange a value with a new value
  - Increment/Decrement
  - Minimum and Maximum
- 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



Local vs Global Atomics

Kepler

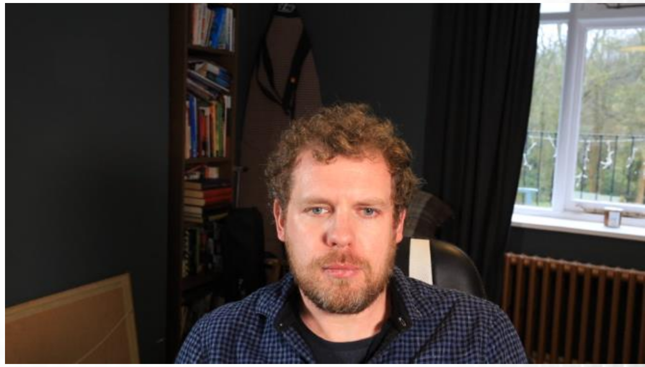
Maxwell



- ❑ 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/

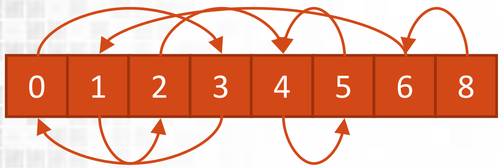
Warp Shuffle

- 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



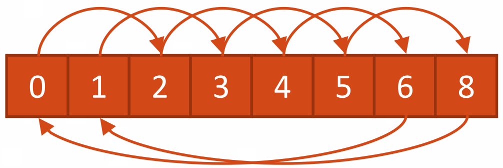
Shuffle Variants

`__shfl_sync()`



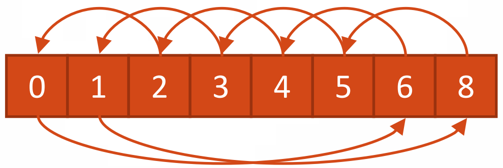
Shuffled between any two index threads

`__shfl_up_sync()`



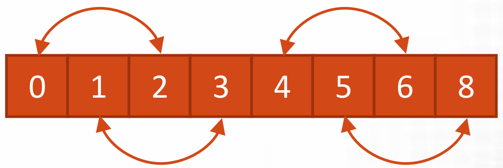
Shuffles to nth right neighbour wrapping indices (in this case n=2)

`__shfl_down_sync()`

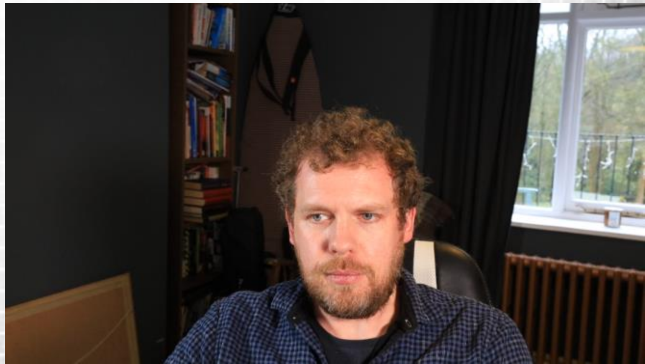


Shuffles to nth left neighbour wrapping indices (in this case n=2)

`__shfl_xor_sync()`



Butterfly (XOR) exchange shuffle pattern



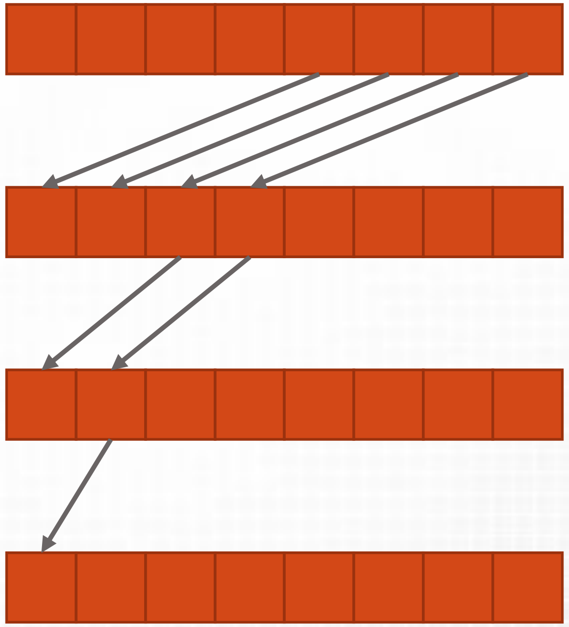
Shuffle function arguments

- `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)
}
```

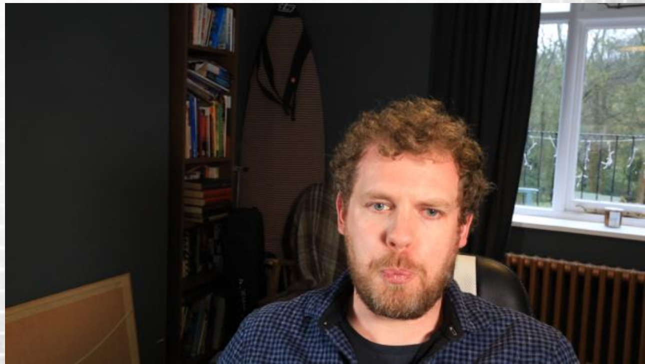


offset==4

offset==2

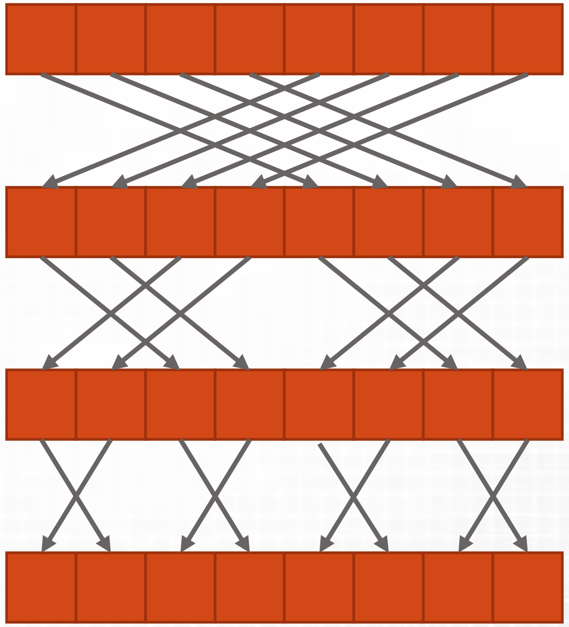
offset==1

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)
}
```

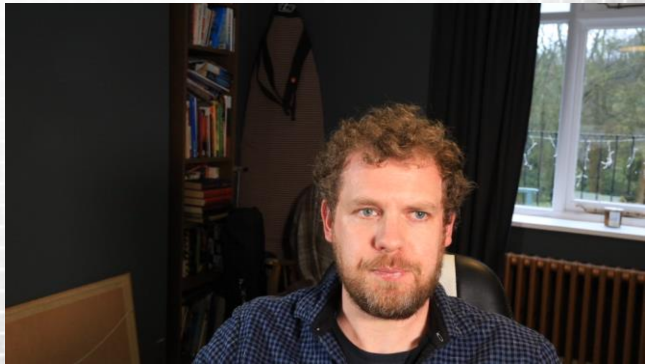


mask==4

mask==2

mask==1

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 nth bit of the return value based on the nth threads condition value
- All warp voting functions are single instruction and act as barrier
  - Only active threads participate, does not block like `synchr`

Warp Voting Example

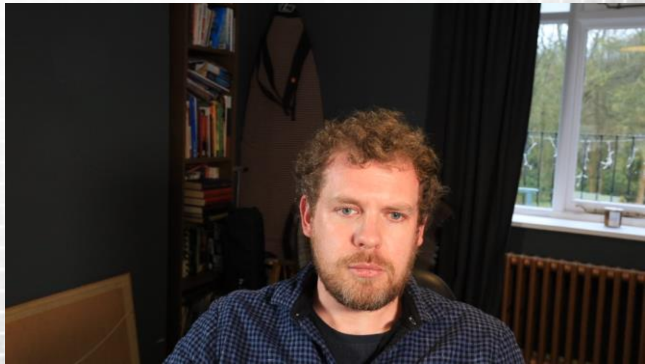
```
__global__ void voteAllKernel(unsigned int *input, unsigned int *result)
{
  int i = threadIdx.x + blockIdx.x*blockDim.x;
  int j = i % WARP_SIZE;

  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

Global Communication Summary

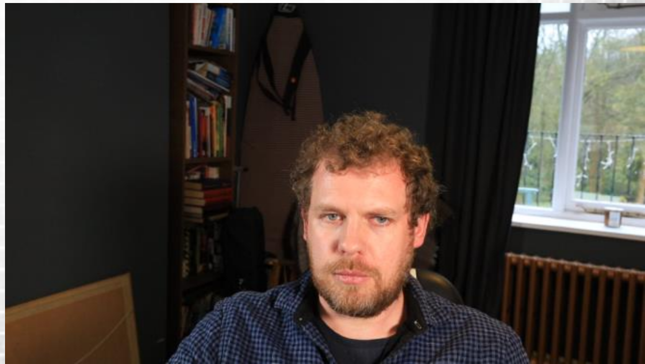
- Shared memory is per thread block
- Shuffles and voting for warp level
- Atomics can be used for some global (grid wide) operations
- What about general global communication?
  - Not possible within a kernel (*except in Volta – not covered*)!
  - Remember a grid may not be entirely in flight on the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



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.



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/