CS 193G

Lecture 3: CUDA Threads & Atomics

ATOMICS

The Problem

- How do you do global communication?
- Finish a grid and start a new one

Global Communication

- Finish a kernel and start a new one
- All writes from all threads complete before a kernel finishes

```
step1<<<grid1,blk1>>>(...);
// The system ensures that all
// writes from step1 complete.
step2<<<grid2,blk2>>>(...);
```

Global Communication

- Would need to decompose kernels into before and after parts

Race Conditions

- Or, write to a predefined memory location
  - Race condition! Updates can be lost

Race Conditions

```
threadId:0                          threadId:1917
      // vector[0] was equal to 0
vector[0] += 5;                     vector[0] += 1;
...                                 ...
a = vector[0];                      a = vector[0];
```

- What is the value of `a` in thread 0?
- What is the value of `a` in thread 1917?

Race Conditions

- Thread 0 could have finished execution before 1917 started
- Or the other way around
- Or both are executing at the same time

Race Conditions

- Answer: not defined by the programming model, can be arbitrary

Atomics

- CUDA provides atomic operations to deal with this problem

Atomics

- An atomic operation guarantees that only a single thread has access to a piece of memory while an operation completes
- The name atomic comes from the fact that it is uninterruptable
- No dropped data, but ordering is still arbitrary
- Different types of atomic instructions
- `atomic{Add, Sub, Exch, Min, Max, Inc, Dec, CAS, And, Or, Xor}`
- More types in fermi

Example: Histogram

```
// Determine frequency of colors in a picture
// colors have already been converted into ints
// Each thread looks at one pixel and increments
// a counter atomically
__global__ void histogram(int* color,
                          int* buckets)
{
  int i = threadIdx.x
        + blockDim.x * blockIdx.x;
  int c = colors[i];
  atomicAdd(&buckets[c], 1);
}
```

Example: Workqueue

```
// For algorithms where the amount of work per item
// is highly non-uniform, it often makes sense for
// to continuously grab work from a queue
__global__
void workq(int* work_q, int* q_counter,
           int* output, int queue_max)
{
  int i = threadIdx.x
        + blockDim.x * blockIdx.x;
  int q_index =
    atomicInc(q_counter, queue_max);
  int result = do_work(work_q[q_index]);
  output[i] = result;
}
```

Atomics

- Atomics are slower than normal load/store
- You can have the whole machine queuing on a single location in memory
- Atomics unavailable on G80!

Example: Global Min/Max (Naive)

```
// If you require the maximum across all threads
// in a grid, you could do it with a single global
// maximum value, but it will be VERY slow
__global__
void global_max(int* values, int* gl_max)
{
  int i = threadIdx.x
        + blockDim.x * blockIdx.x;
  int val = values[i];
  atomicMax(gl_max,val);
}
```

Example: Global Min/Max (Better)

```
// introduce intermediate maximum results, so that
// most threads do not try to update the global max
__global__
void global_max(int* values, int* max,
                int *regional_maxes,
                int num_regions)
{
  // i and val as before …
  int region = i % num_regions;
  if(atomicMax(&reg_max[region],val) < val)
  {
    atomicMax(max,val);
  }
}
```

Global Min/Max

- Single value causes serial bottleneck
- Create hierarchy of values for more parallelism
- Performance will still be slow, so use judiciously
- See next lecture for even better version!

Summary

- Can't use normal load/store for inter-thread communication because of race conditions
- Use atomic instructions for sparse and/or unpredictable global communication
  - See next lectures for shared memory and scan for other communication patterns
- Decompose data (very limited use of single global sum/max/min/etc.) for more parallelism

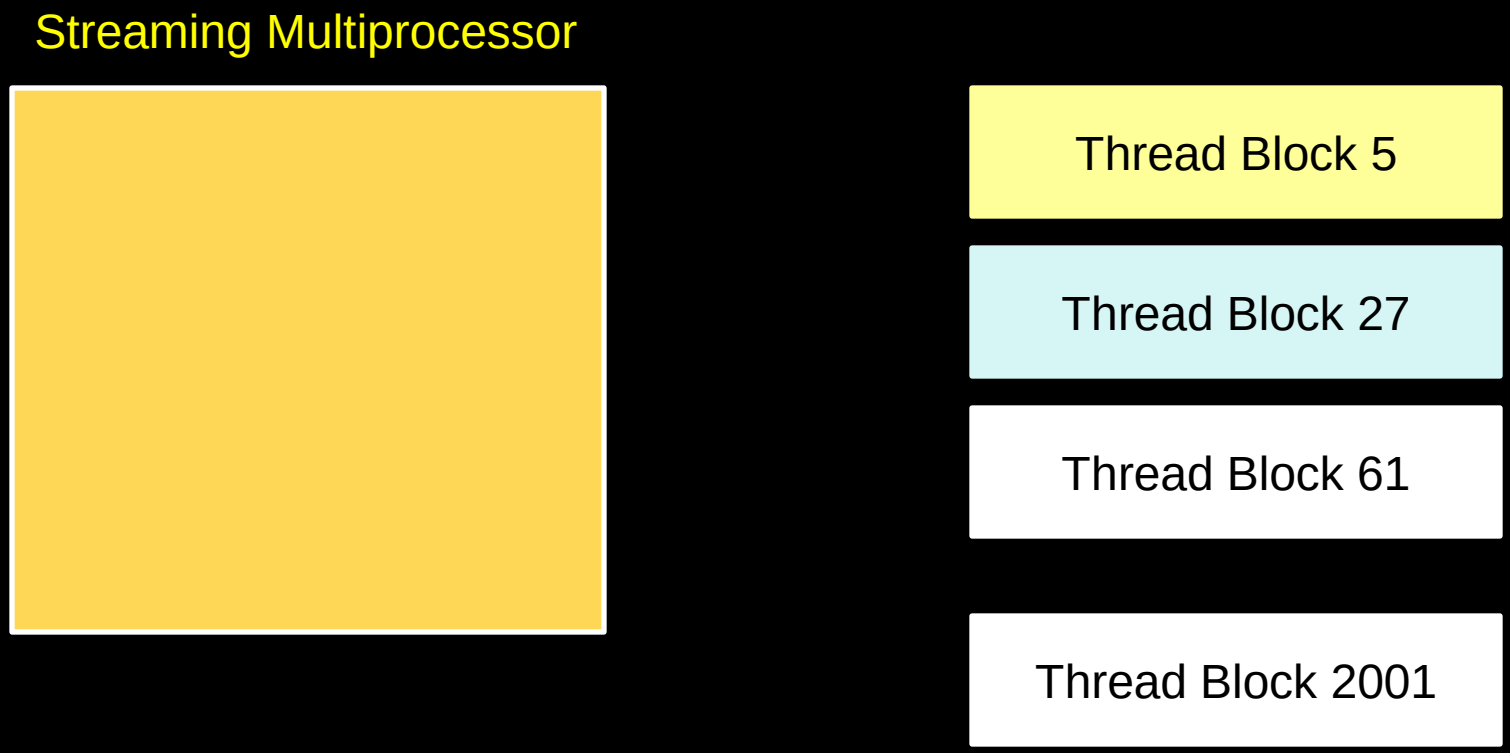
Questions?

SM EXECUTION & DIVERGENCE

How an SM executes threads

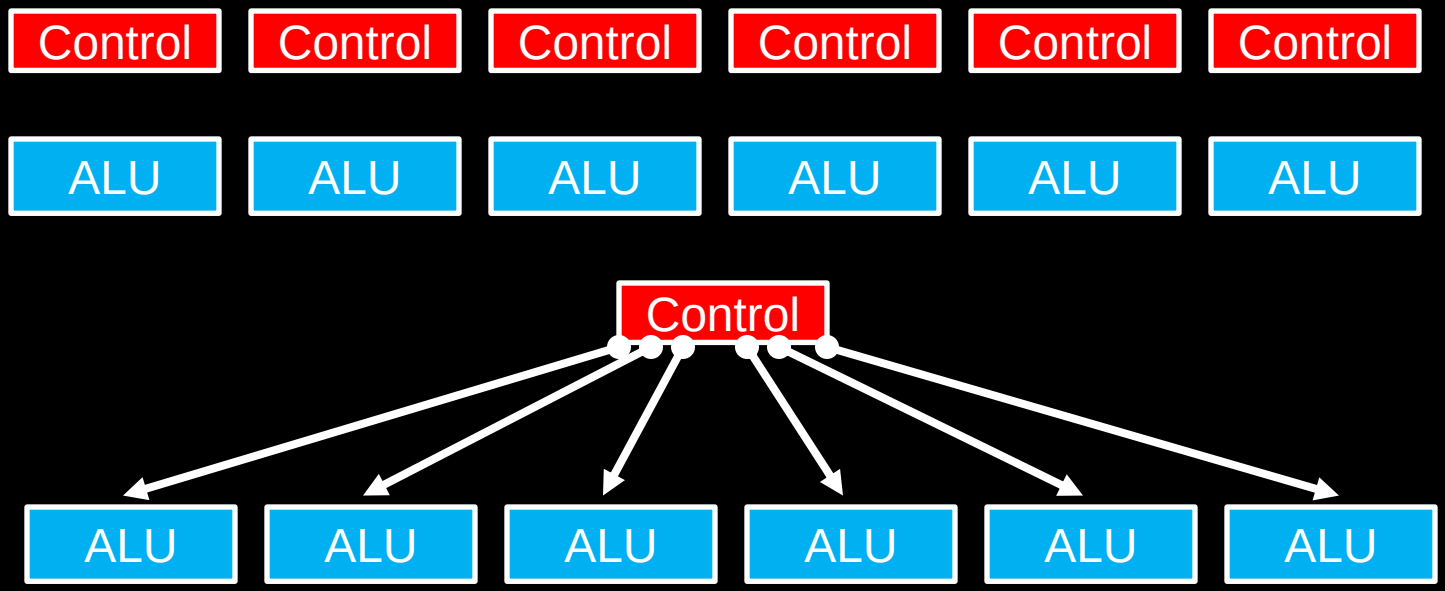
- Overview of how a Stream Multiprocessor works
- SIMT Execution
- Divergence

Scheduling Blocks onto SMs



- **HW Schedules thread blocks onto available SMs**
  - **No guarantee of ordering among thread blocks**
  - **HW will schedule thread blocks as soon as a previous thread block finishes**

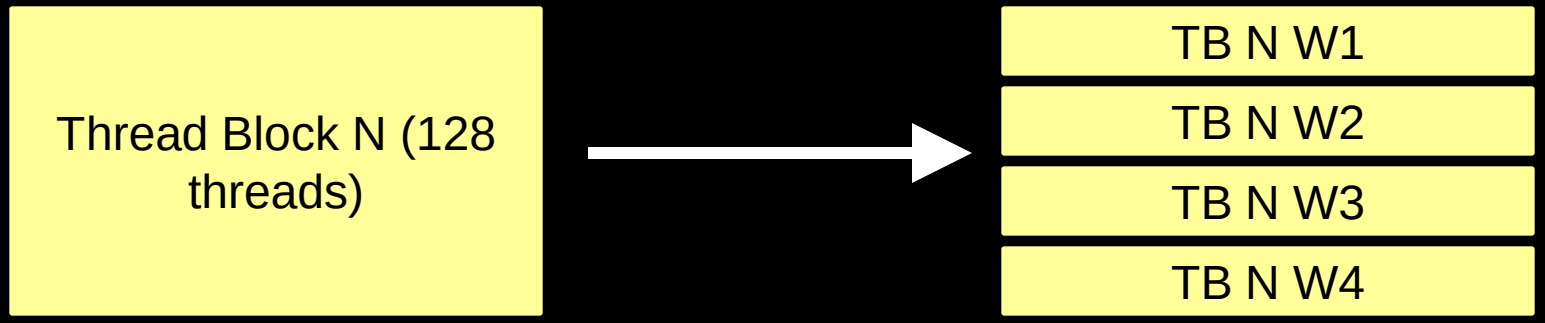
Warps



- A **warp** = 32 threads launched together
  - Usually, execute together as well

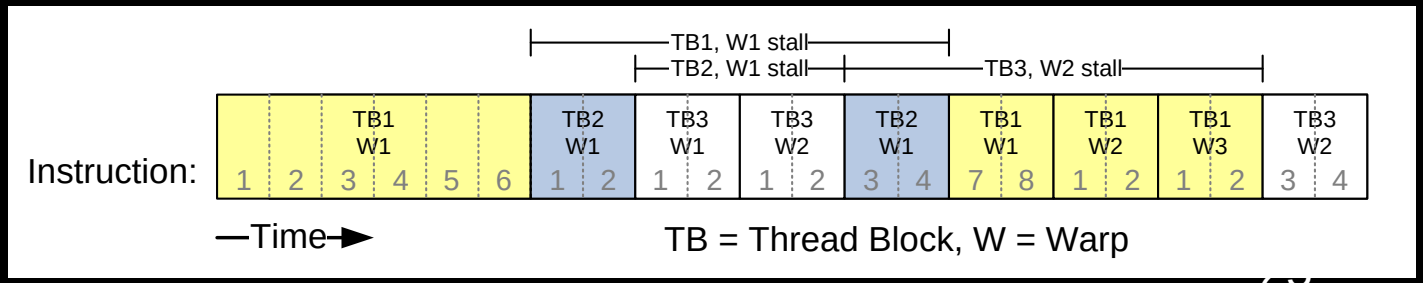
Mapping of Thread Blocks

- Each thread block is mapped to one or more warps
- The hardware schedules each warp independently



Thread Scheduling Example

- SM implements zero-overhead warp scheduling
  - At any time, only one of the warps is executed by SM *
  - Warps whose next instruction has its inputs ready for consumption are eligible for execution
  - Eligible Warps are selected for execution on a prioritized scheduling policy
  - All threads in a warp execute the same instruction when selected

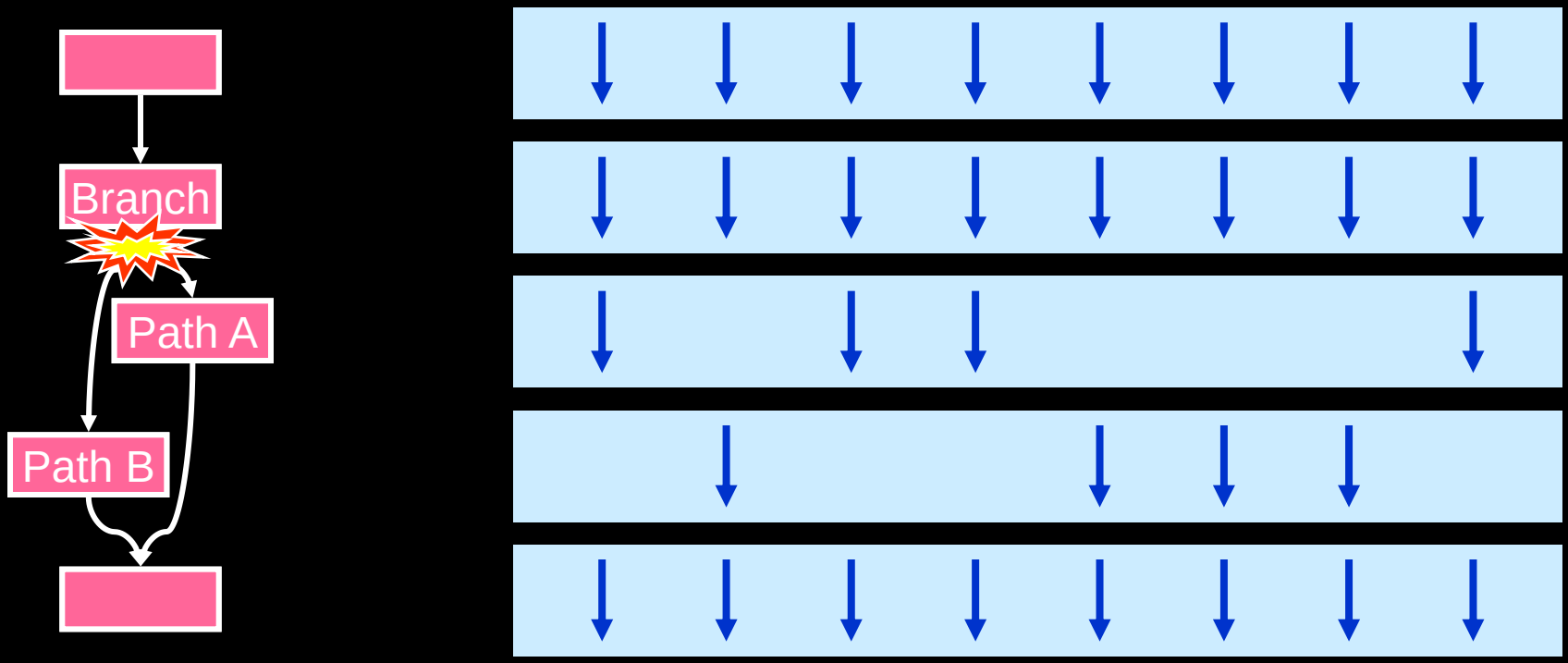


Control Flow Divergence

- What happens if you have the following code?

```
if(foo(threadIdx.x))
{
  do_A();
}
else
{
  do_B();
}
```

Control Flow Divergence



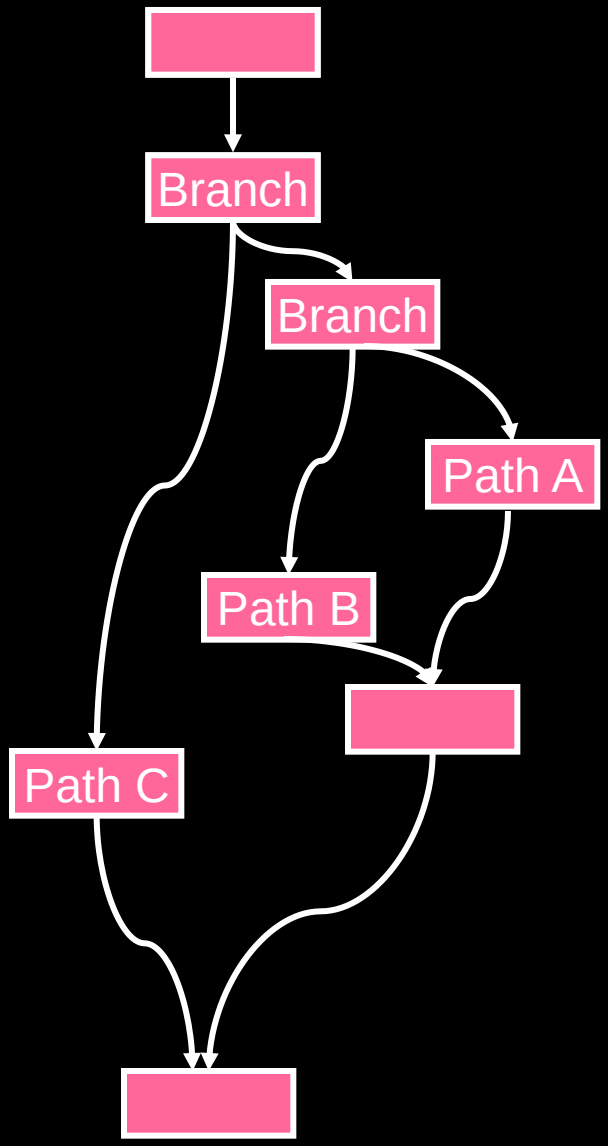
From Fung et al. MICRO ‘07

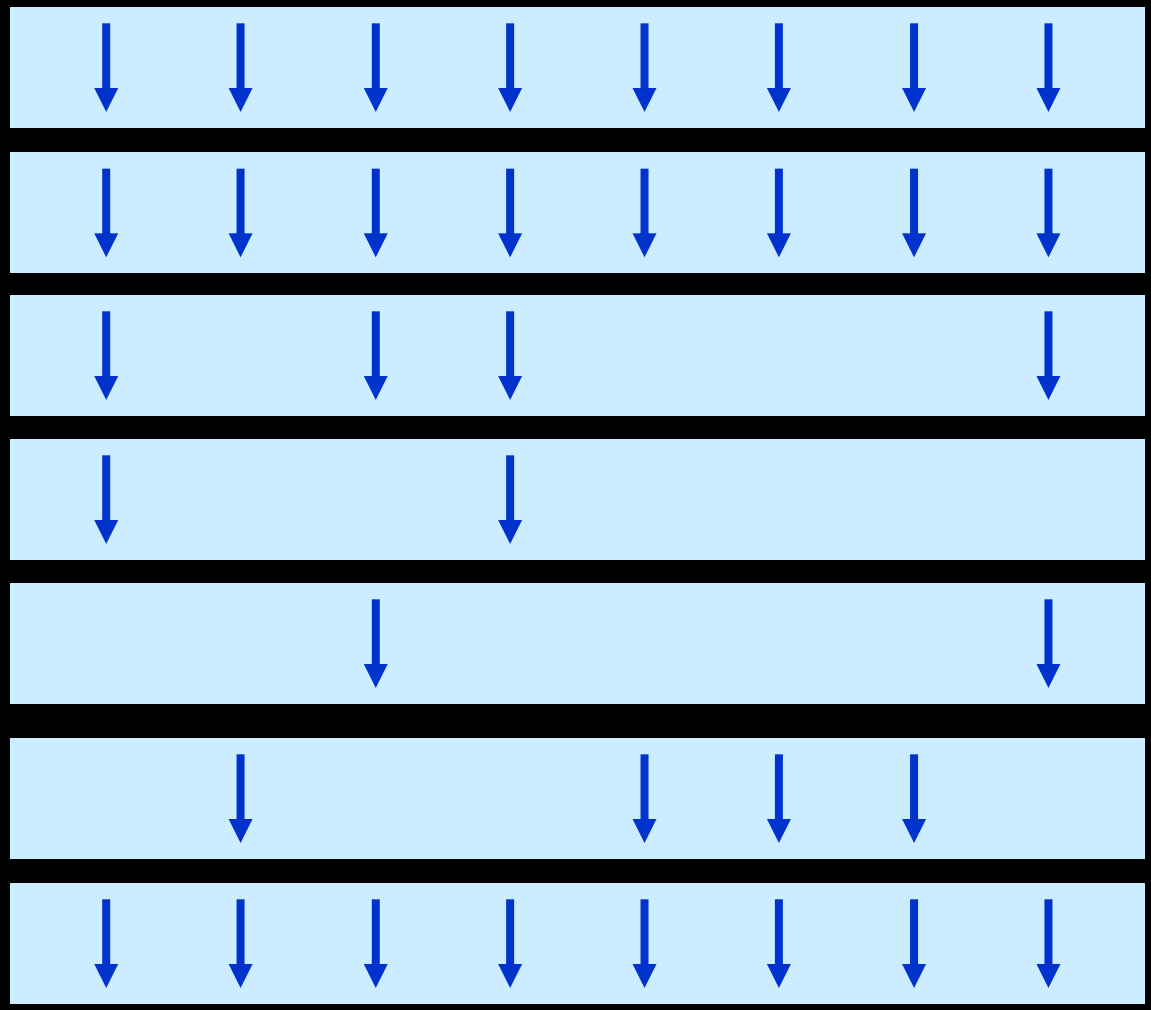
Control Flow Divergence

- Nested branches are handled as well

```
if(foo(threadIdx.x))
{
  if(bar(threadIdx.x))
    do_A();
  else
    do_B();
}
else
  do_C();
```

Control Flow Divergence





Control Flow Divergence

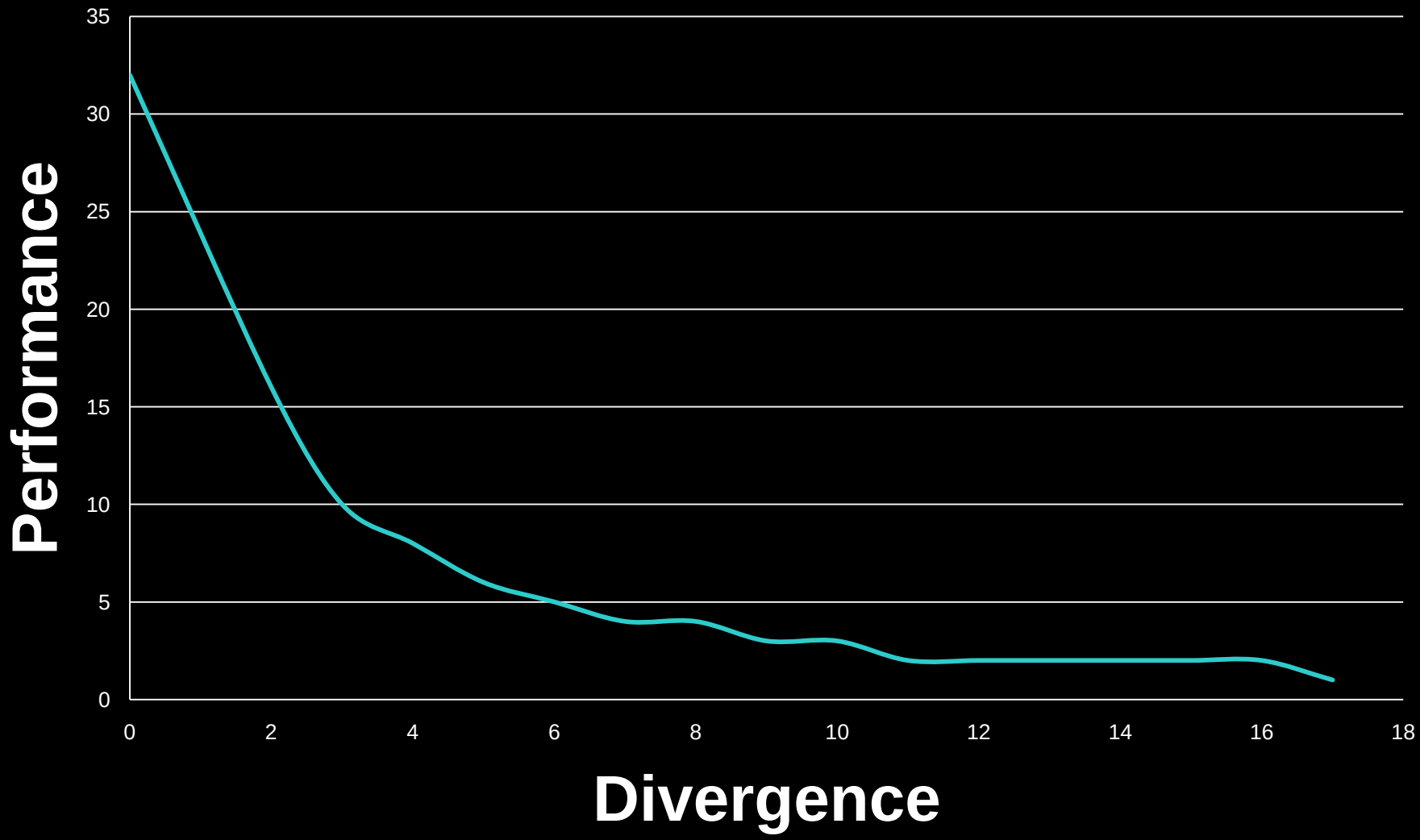
- You don’t have to worry about divergence for correctness (*)
- You might have to think about it for performance
  - Depends on your branch conditions

Control Flow Divergence

- Performance drops off with the degree of divergence

```
switch(threadIdx.x % N)
{
  case 0:
    ...
  case 1:
    ...
}
```

Divergence


Performance
35
30
25
20
15
10
5
0
0
2
4
6
8
10
12
14
16
18
Divergence

Atomics

- `atomicAdd` returns the previous value at a certain address
- Useful for grabbing variable amounts of data from a list

Questions?

Backup

Compare and Swap

```
int compare_and_swap(int* register,
   int oldval, int newval)
{
   int old_reg_val = *register;
   if(old_reg_val == oldval)
      *register = newval;

   return old_reg_val;
}
```

Compare and Swap

- Most general type of atomic
- Can emulate all others with CAS

Locks

- Use very judiciously
- Always include a `max_iter` in your spinloop!
- Decompose your data and your locks