SYNCHRONIZATION AND ATOMIC OPERATIONS

SYNCHRONIZATION

Race Conditions [1/2]

- Race conditions arise when 2+ threads attempt to access the same memory location concurrently and at least one access is a write.

```
// race.cu
__global__ void race(int* x)
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  *x = i;
}

// main.cpp
int x;
race<<<1,128>>>(d_x);
cudaMemcpy(&x, d_x, sizeof(int), cudaMemcpyDeviceToHost);
```

SYNCHRONIZATION

Race Conditions [2/2]

- Programs with race conditions may produce unexpected, seemingly arbitrary results
  - Updates may be missed, and updates may be lost

```
// race.cu
__global__ void race(int* x)
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  *x = *x + 1;
}

// main.cpp
int x;
race<<<1,128>>>(d_x);
cudaMemcpy(&x, d_x, sizeof(int), cudaMemcpyDeviceToHost);
```

SYNCHRONIZATION

Synchronization

- Accesses to shared locations need to be correctly synchronized (coordinated) to avoid race conditions
- In many common shared memory multithreaded programming models, one uses coordination objects such as locks to synchronize accesses to shared data
- CUDA provides several scalable synchronization mechanisms, such as efficient barriers and atomic memory operations.
- In general, always most efficient to design algorithms to avoid synchronization whenever possible.

SYNCHRONIZATION

Synchronization

- Assume thread T1 reads a value defined by thread T0

```
// update.cu
__global__ void update_race(int* x, int* y)
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  if (i == 0) *x = 1;
  if (i == 1) *y = *x;
}

// main.cpp
update_race<<<1,2>>>(d_x, d_y);
cudaMemcpy(&y, d_y, sizeof(int), cudaMemcpyDeviceToHost);
```

- Program needs to ensure that thread T1 reads location after thread T0 has written location.

SYNCHRONIZATION

Synchronization within Block

NVIDIA

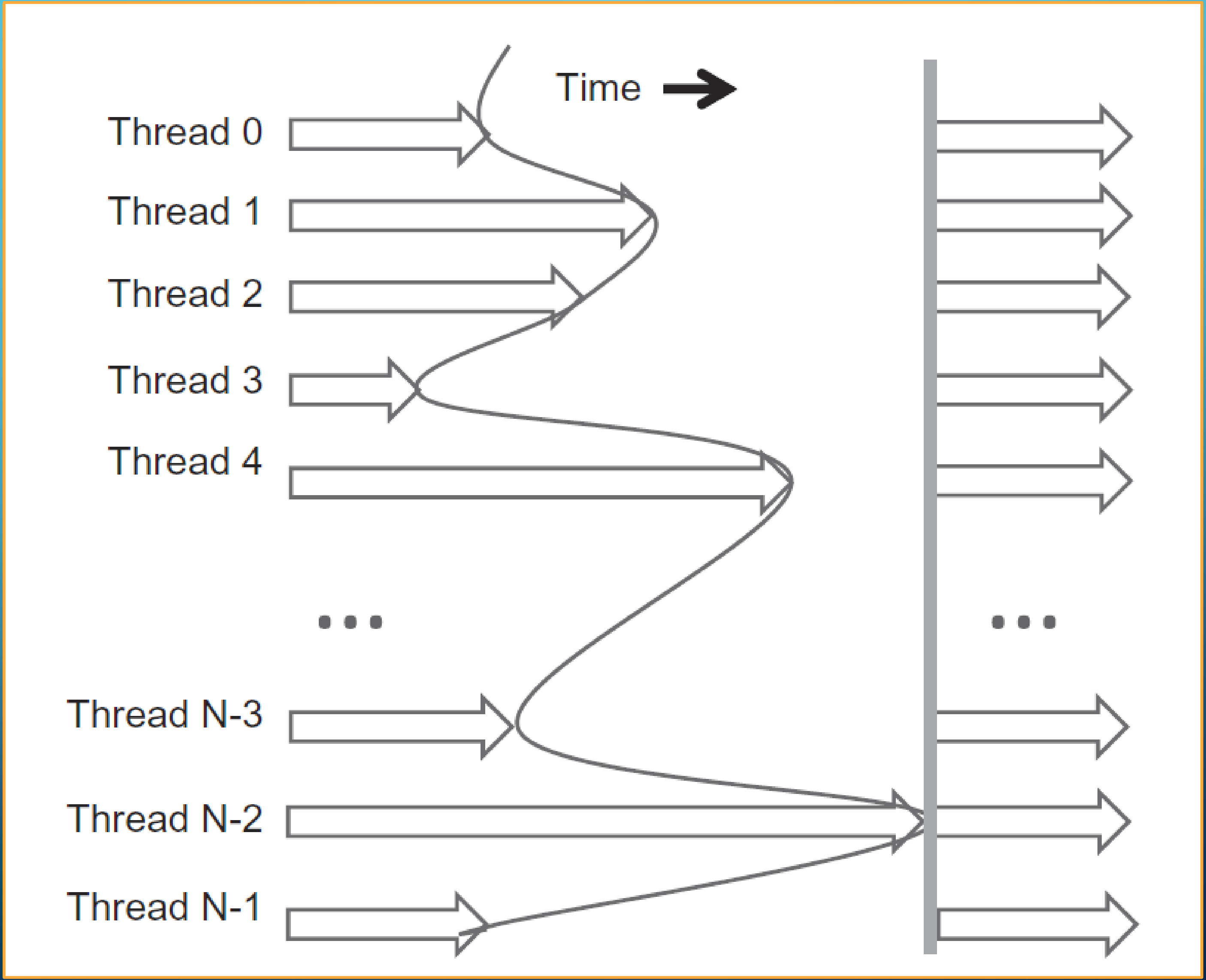
- Threads in same block: can use __synchthreads() to specify synchronization point that orders accesses

```
// update.cu
__global__ void update(int* x, int* y)
{
   int i = threadIdx.x + blockDim.x * blockIdx.x;
   if (i == 0) *x = 1;
   __syncthreads();
   if (i == 1) *y = *x;
}

// main.cpp
update<<<1,2>>>(d_x, d_y);
cudaMemcpy(&y, d_y, sizeof(int), cudaMemcpyDeviceToHost);
```

- **Important:** all threads within the block must reach the __synchthreads() statement

SYNCHRONIZATION



ATOMIC OPERATIONS

Introduction to Atomics

- Atom memory operations (atomic functions) are used to solve all kinds of synchronization and coordination problems in parallel computer systems.

- General concept is to provide a mechanism for a thread to update a memory location such that the update appears to happen atomically (without interruption) with respect to other threads.

- This ensures that all atomic updates issued concurrently are performed (often in some unspecified order) and that all threads can observe all updates.

ATOMIC OPERATIONS

Atomic Functions [1/3]

- **Atomic functions perform read-modify-write operations on data residing in global and shared memory**

```
//example of int atomicAdd(int* addr, int val)
__global__ void update(unsigned int* x)
{
   int i = threadIdx.x + blockDim.x * blockIdx.x;
   int j = atomicAdd(x, 1);    // j = *x; *x = j + i;
}

// main.cpp
int x = 0;
cudaMemcpy(d_x, x, cudaMemcpyHostToDevice);
update<<<1,128>>>;
cudaMemcpy(&x, d_x, cudaMemcpyHostToDevice);
```

- **Atomic functions guarantee that only one thread may access a memory location while the operation completes**

ATOMIC OPERATIONS

Atomic Functions [2/3]

NVIDIA

- Synopsis of atomic function `atomicOP(a,b)` is typically

```
t1 = *a;        // read
t2 = t1 OP b;   // modify
*a = t2;        // write
return t;
```

- The hardware ensures that all statements are executed atomically without interruption by any other atomic functions.
- The atomic function returns the initial value, not the final value, stored at the memory location.

ATOMIC OPERATIONS

Atomic Functions [3/3]

- The name atomic is used because the update is performed atomically: it cannot be interrupted by other atomic updates.
- The order in which concurrent atomic updates are performed is not defined, and may appear arbitrary.
- However, none of the atomic updates will be lost.
- Many different kinds of atomic operations
  - Add (add), Sub (subtract), Inc (increment), Dec (decrement)
  - And (bit-wise and), Or (bit-wise or) , Xor (bit-wise exclusive or)
  - Exch (Exchange)
  - Min (Minimum), Max (Maximum)
  - Compare-and-Swap

ATOMIC OPERATIONS

Histogram Example

```
// Compute histogram of colors in an image
//
//  color - pointer to picture color data
//  bucket - pointer to histogram buckets, one per color
//

__global__ void histogram(int n, int* color, int* bucket)
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  if (i < n)
  {
    int c = colors[i];
    atomicAdd(&bucket[c], 1);
  }
}
```

ATOMIC OPERATIONS

Performance Notes

- Atomics are slower than normal accesses (loads, stores)
- Performance can degrade when many threads attempt to perform atomic operations on a small number of locations
- Possible to have all threads on the machine stalled, waiting to perform atomic operations on a single memory location.

REFERENCES

- https://developer.nvidia.com/cuda-education