CPSC/ECE 4780/6780

General-Purpose Computation on Graphical Processing Units (GPGPU)

Lecture 8: Atomics

Recap of Last Lecture

- What is reduction? And parallel reduction?
- What is iterative pairwise implementation?
- Tips to improve parallel reduction
  - Use shared memory rather than global memory
  - Reduce warp divergence by rearranging the array index of each thread to force neighboring threads to perform the addition
  - Unrolling loops by reducing the frequency of branches and loop maintenance instructions
    - Unroll the last warp manually
    - Complete unrolling with template functions to reduce branch overhead

Race Conditions

- Race conditions arise when 2+ threads attempt to access the same memory location concurrently and at least one access is write
- Programs with race conditions may produce unexpected, seemingly arbitrary results

A Simple Race Condition Example

- Let int *x point to global memory. *x+1 happens in 3 steps:
  - Step 1: Read the value in *x
  - Step 2: Add 1 to the value read in step 1
  - Step 3: Write the result back to *x

Read-modify-write operation

- If we want two parallel threads A and B to both increment *x, we want something like:
  - Thread A reads the value, say 7, from *x
  - Thread A adds 1 to the value 7 it read to make 8
  - Thread A writes its result 8 back to *x
  - Thread B reads the value 8 from *x
  - Thread B adds 1 to its value 8 it read to make 9
  - Thread B writes the result 9 back to *x

A Simple Race Condition Example

- But since the threads are parallel, there are many other orderings of these steps that produce the wrong value, we may end up with something like:
  - Thread A reads the value, say 7, from *x
  - Thread B reads the value, 7, from *x
  - Thread A adds 1 to the value 7 it read to make 8
  - Thread A writes its result 8 back to *x
  - Thread B adds 1 to its value 7 it read to make 8
  - Thread B writes the result 8 back to *x

Incorrect!

A Simple Race Condition Example

```
__global__ void increment(int *d_x) {
  *d_x += 1;
}

int main(){
  int x = 0, *d_x;

  cudaMalloc((void**) &d_x, sizeof(int));
  cudaMemcpy(d_x, &x, sizeof(int), cudaMemcpyHostToDevice);

  increment<<<1000,1000>>>(d_x);

  cudaMemcpy(&x, d_x, sizeof(int), cudaMemcpyDeviceToHost);

  printf("x = %d\n", x);
  cudaFree(d_x);
}
```

Should get x = 1000x1000=1,000,000

```
[11:43:40] jin6@titan1:~/CUDA/RaceCondition [56] nvcc -Wno-deprecated-gpu-targets raceCondition.cu
[11:43:56] jin6@titan1:~/CUDA/RaceCondition [57] ./a.out
x = 46
[11:43:58] jin6@titan1:~/CUDA/RaceCondition [58] ./a.out
x = 44
[11:44:00] jin6@titan1:~/CUDA/RaceCondition [59] ./a.out
x = 45
```

Atomics

- Atomic operation: an operation that forces otherwise parallel threads into a bottleneck, executing the operation one at a time
  - Ensures uninterruptable read-modify-write memory operation
  - Serializes contentious updates from multiple threads
  - Enables co-ordination among >1 threads
  - Limited to specific functions and data sizes

Back to Simple Race Condition Example

- In increment(), replace

```
*d_x += 1;
```

with an atomic function,

```
atomicAdd(d_x, 1);
```

to fix the race condition issue

Fixed Simple Race Condition Example

```
__global__ void increment(int *d_x) {
  atomicAdd(d_x, 1);
}

int main(){
  int x = 0, *d_x;

  cudaMalloc((void**) &d_x, sizeof(int));
  cudaMemcpy(d_x, &x, sizeof(int), cudaMemcpyHostToDevice);

  increment<<<1000,1000>>>(d_x);

  cudaMemcpy(&x, d_x, sizeof(int), cudaMemcpyDeviceToHost);

  printf("x = %d\n", x);
  cudaFree(d_x);
}
```

```
[13:05:30] jin6@titan1:~/CUDA/RaceCondition [69] nvcc  -Wno-deprecated-gpu-targets raceConditionFixed.cu
[13:06:28] jin6@titan1:~/CUDA/RaceCondition [70] ./a.out
x = 1000000
```

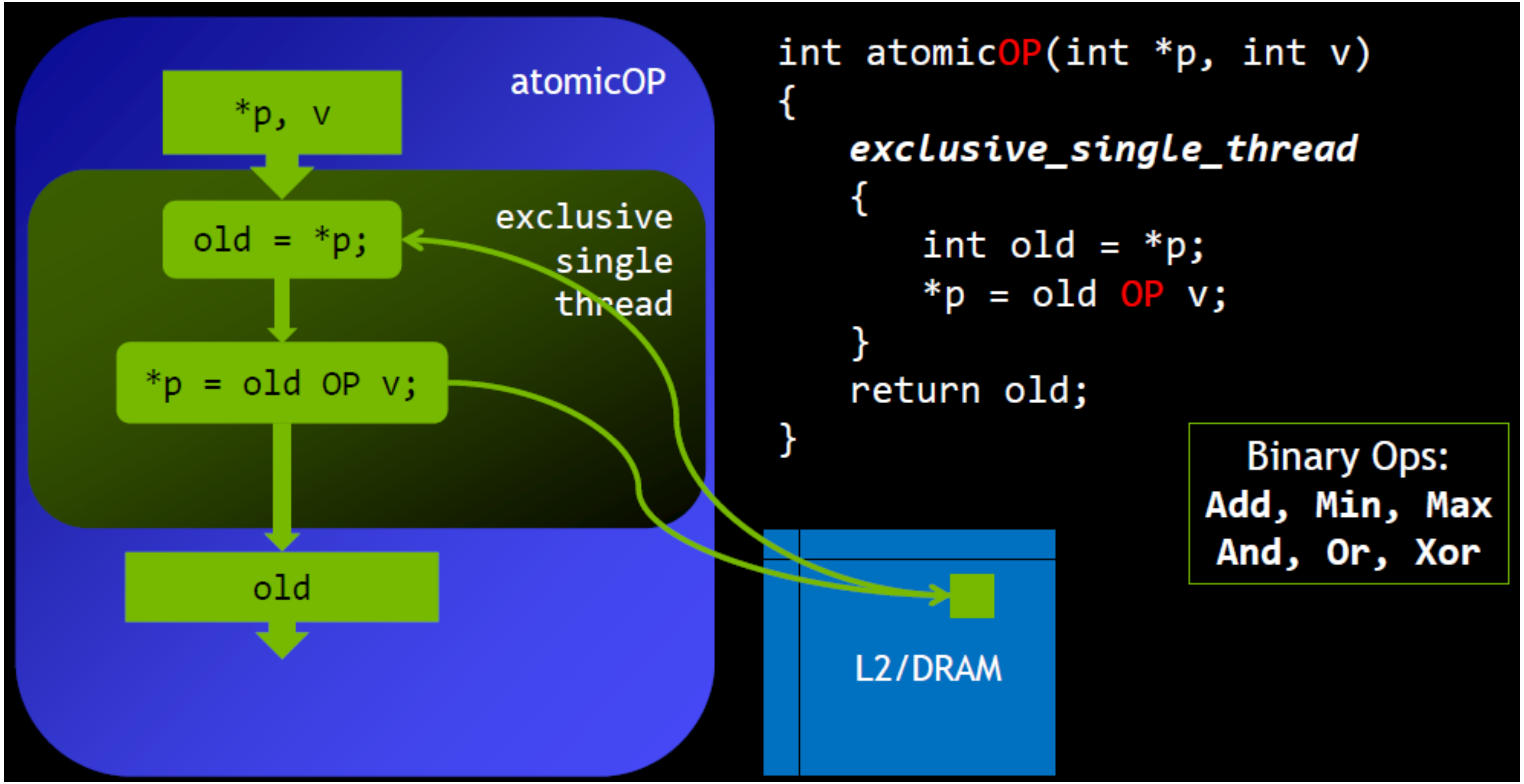
Atomic Functions

- Atomic functions perform read-modify-write operations on data residing in global and shared memory
  - Atomic functions guarantee that only one thread may access a memory location while the operation completes
  - Order in which threads get to write is not specified though
- Synopsis of arithmetic/logical atomic function atomicOp(a, b) is typically

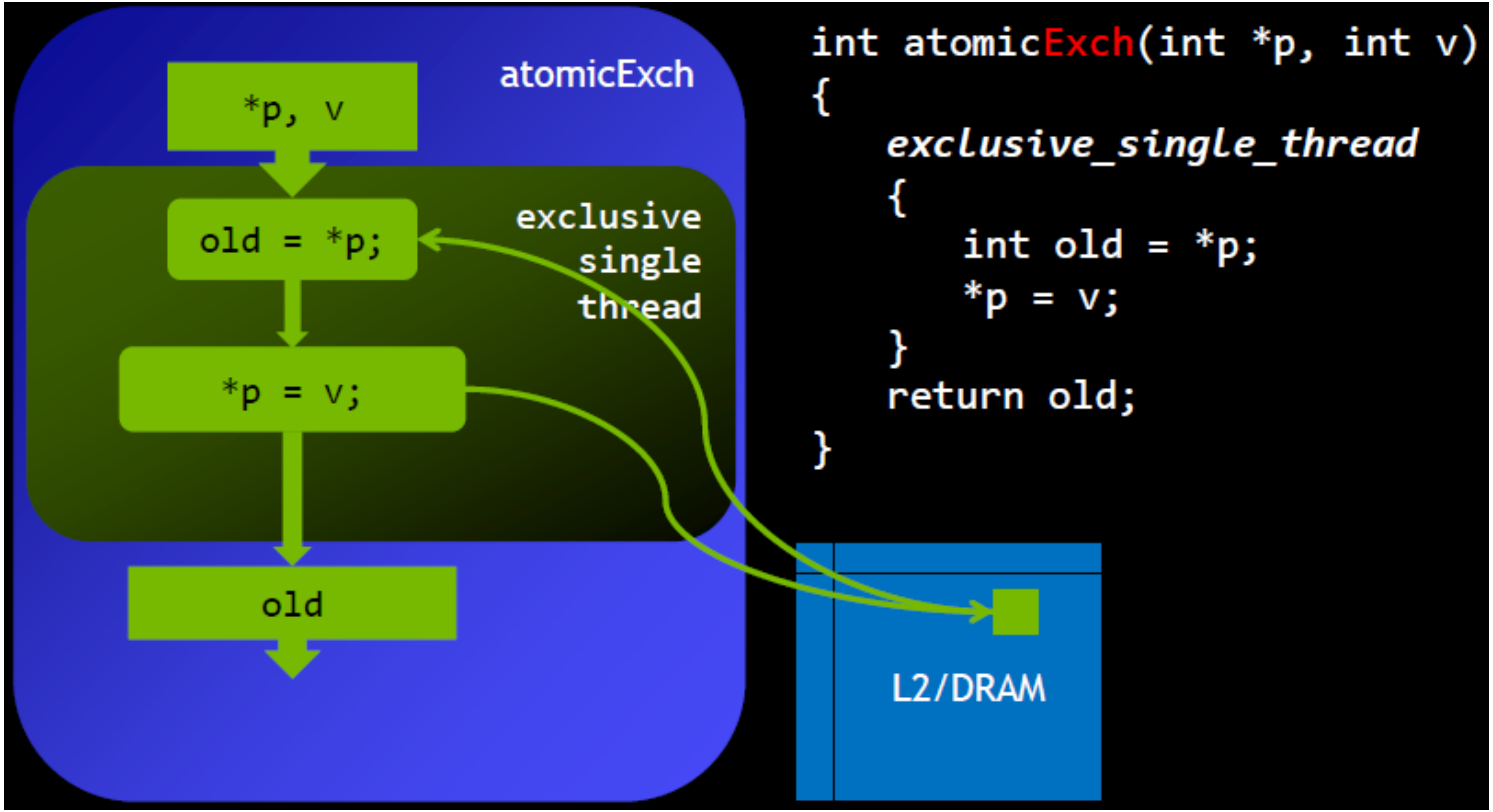
```
t1 = *a;           // read
t2 = t1 OP (*b);  // modify
*a = t2;           // write
return t1;
```

- The hardware ensures that all statements are executed atomically without interruption by any other atomic functions

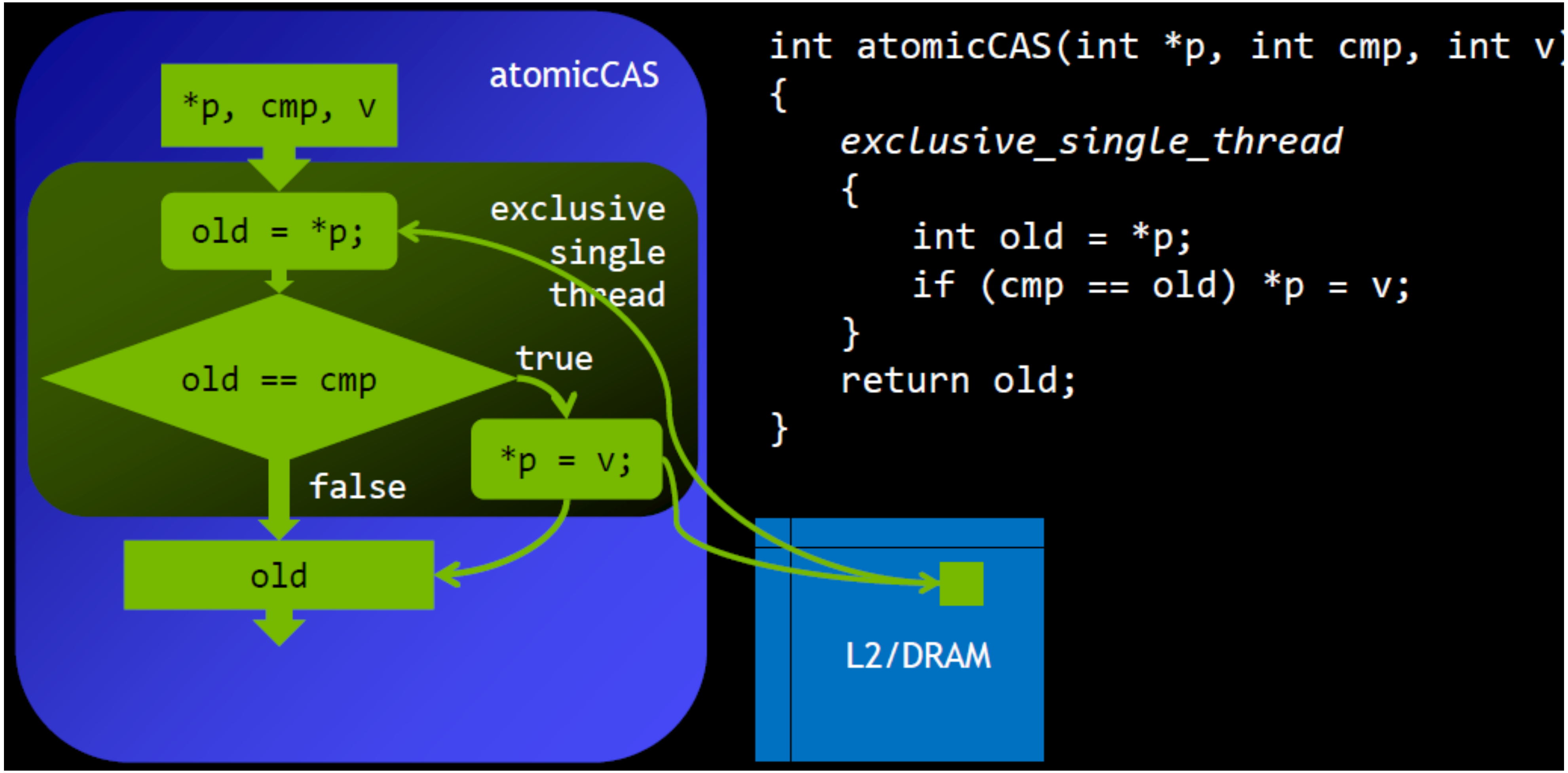
Arithmetic/Logical Atomic Operation



Overwriting Atomic Operation



Compare-and-Swap Operation



CUDA Built-in Atomic Functions

- With CUDA **compute capability 2.0** or above, you have:
  - atomicAdd(), atomicSub(), atomicInc(), atomicDec()
  - atomicAnd(), automicOr(), atomicXor()
  - atomicExch()
  - atomicMin(), atomicMax()
  - atomicCAS()
- For documentation, refer to the CUDA C Programming Guide

Compute Capability

- NVIDIA refers to the supported features of a GPU as its compute capability

```
cudaDeviceProp  prop;

int count;
HANDLE_ERROR( cudaGetDeviceCount( &count ) );
for (int i=0; i< count; i++) {
    HANDLE_ERROR( cudaGetDeviceProperties( &prop, i ) );
    printf( "   --- General Information for device %d ---\n", i );
    printf( "Name:  %s\n", prop.name );
    printf( "Compute capability:  %d.%d\n", prop.major, prop.minor );
```

```
   --- General Information for device 0 ---
Name:  GeForce GTX TITAN Black
Compute capability:  3.5
Clock rate:  980000
Device copy overlap:  Enabled
Kernel execution timeout :  Enabled
   --- Memory Information for device 0 ---
Total global mem:  6376390656
Total constant Mem:  65536
Max mem pitch:  2147483647
Texture Alignment:  512
   --- MP Information for device 0 ---
Multiprocessor count:  15
Shared mem per mp:  49152
Registers per mp:  65536
Threads in warp:  32
Max threads per block:  1024
Max thread dimensions:  (1024, 1024, 64)
Max grid dimensions:  (2147483647, 65535, 65535)
```

Histogram Example

- Given a data set that consists of some set of elements, a histogram represents a count of the frequency of each element
- Let’s compute histogram of colors in an image
- Colors have already been converted into ints
- __global__void histogram(int* colors, int* buckets)

Bug?

```
// 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* colors, int* buckets)
{
  int i= threadIdx.x+ blockDim.x* blockIdx.x;
  int c = colors[i];

  buckets[c]++;
}
```

Bug

```
// 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* colors, int* buckets)
{
  int i= threadIdx.x+ blockDim.x* blockIdx.x;
  int c = colors[i];

  buckets[c]++;
}
```

Race condition!
Multiple threads race to update buckets[c]!

Solution – atomicAdd()

```
// 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* colors, int* buckets)
{
   int i= threadIdx.x+ blockDim.x* blockIdx.x;
   int c = colors[i];

   atomicAdd(&buckets[c], 1);
}
```

To ensure only one thread can update buckets[c] at a time!

Work Queue Example

```
// For algorithms where the amount of work per item
// is highly non-uniform, it often makes sense
// to continuously grab work from a queue
__device__ int do_work(int x)
{
  return f(x-1) + f(x) + f(x+1);
}

__global__ void process_work_q(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;
}
```

atomicInc()

```
unsigned int atomicInc(unsigned int* address,
                       unsigned int val);
```

reads the 32-bit word **old** located at the address **address** in global or shared memory, computes **((old >= val) ? 0 : (old+1))**, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns **old**.

Performance Notes

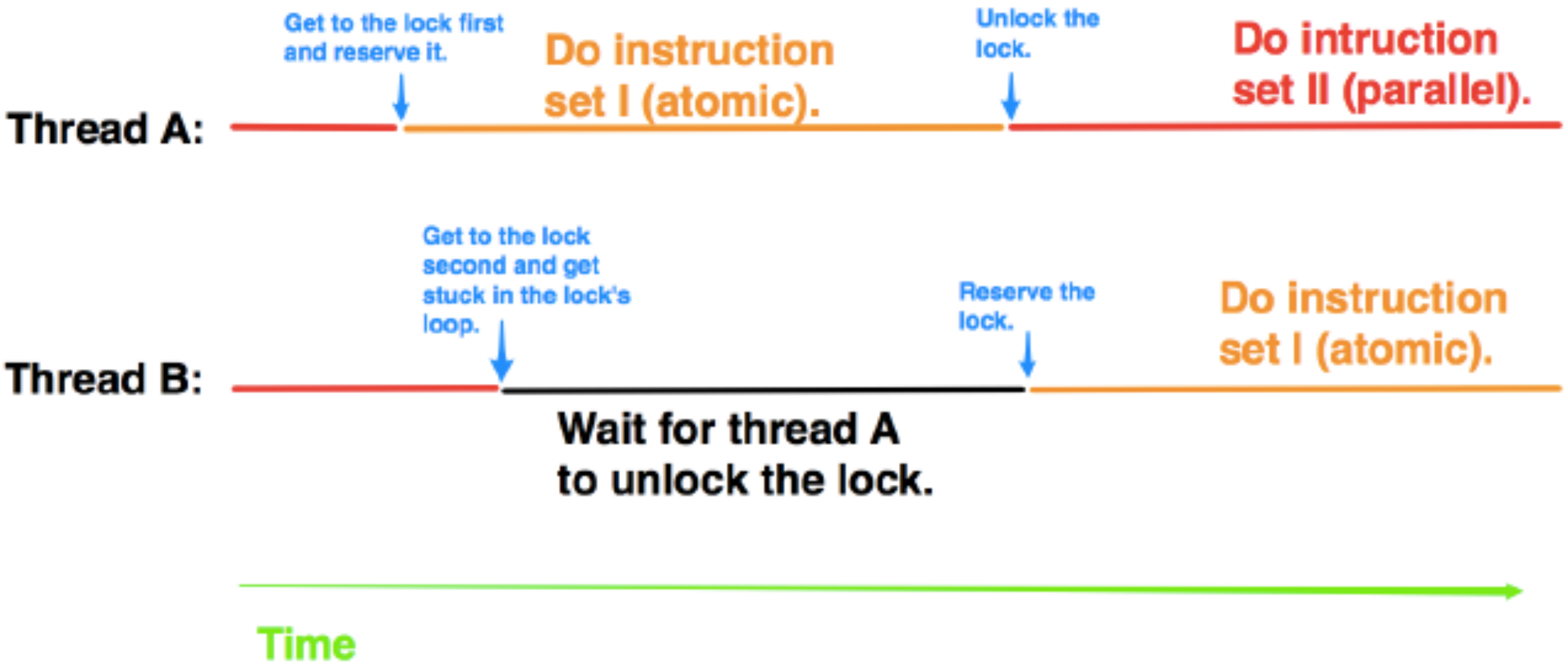
- Atomics are convenient to use, but come at a typically high efficiency loss
  - Atomics are slower than normal accesses (loads, stores)
  - Performance can degrade when many threads attempt to perform atomic operations on a small number of locations
  - Possible to have all threads on the machine stalled, waiting to perform atomic operations on a single memory location

Atomic Locks

- Lock: a mechanism in parallel computing that forces an entire segment of code to be executed atomically
- Mutex
  - “mutual exclusion”, the principle behind locks
  - While a thread is running code inside a lock, it shuts all the other threads out of the lock

```
__global__ void kernel(void) {
  Lock mylock;
  // some parallel code
  mylock.lock();
  // some sequential code
  mylock.unlock();
  // some parallel code
}
```

The Concept


Get to the lock first and reserve it.
Do instruction set I (atomic).
Unlock the lock.
Do intruction set II (parallel).
Thread A:
Get to the lock second and get stuck in the lock's loop.
Reserve the lock.
Do instruction set I (atomic).
Thread B:
Wait for thread A to unlock the lock.
Time

Implementation of Lock Function

```
void lock( void ) {
   if( *mutex == 0 ) {
   *mutex = 1; //store a 1 to lock
   }
}
```

Problem: what happens if another threads writes a 1 to the mutex after our thread has read the value to be 0?

```
__device__ void lock( void ) {
   while( atomicCAS( mutex, 0, 1 ) != 0 );
}
```

In pseudocode:

```
__device void lock(){
  repeat{
    do atomically{

      if(mutex == 0){
        mutex = 1;
        return_value = 0;
      }

      else if(mutex == 1){
        return_value = 1;
      }
    } // do atomically

    if(return_value == 0)
      exit loop;

  } // repeat
} // lock
```

Struct Lock

```
struct Lock {
  int *mutex;
  Lock( void ) {
    int state = 0;
    HANDLE_ERROR( cudaMalloc( (void**)& mutex, sizeof(int) ) );
    HANDLE_ERROR( cudaMemcpy( mutex, &state, sizeof(int), cudaMemcpyHostToDevice ) );
  }

  ~Lock( void ) {
    cudaFree( mutex );
  }

  __device__ void lock( void ) {
    while( atomicCAS( mutex, 0, 1 ) != 0 );
  }

  __device__ void unlock( void ) {
   atomicExch( mutex, 0 );
  }
};
```

Example: Counting the Number of Blocks

- Compare the two kernels:

```
__global__ void blockCounterUnlocked(int *nblocks) {
  if (threadIdx.x == 0) {
    *nblocks = *nblocks + 1;
  }
}
```

```
__global__ void blockCounterLocked(Lock lock, int *nblocks) {
  if (threadIdx.x == 0) {
    lock.lock();
    *nblocks = *nblocks + 1;
    lock.unlock();
  }
}
```

blockCount.cu main()

```
int main(){
  int nblocks, *d_nblocks;
  Lock lock;

  cudaMalloc((void**) &d_nblocks, sizeof(int));

  // blockCounterUnlocked
  nblocks = 0;
  cudaMemcpy(d_nblocks, &nblocks, sizeof(int), cudaMemcpyHostToDevice);

  blockCounterUnlocked<<<512,1024>>>(d_nblocks);

  cudaMemcpy(&nblocks, d_nblocks, sizeof(int), cudaMemcpyDeviceToHost);

  printf("blockCountUnlocked counted %d blocks\n", nblocks);

  // blockCounterLocked
  nblocks = 0;
  cudaMemcpy(d_nblocks, &nblocks, sizeof(int), cudaMemcpyHostToDevice);

  blockCounterLocked<<<512,1024>>>(lock, d_nblocks);

  cudaMemcpy(&nblocks, d_nblocks, sizeof(int), cudaMemcpyDeviceToHost);

  printf("blockCountLocked counted %d blocks\n", nblocks);

  cudaFree(d_nblocks);
}
```

```
[16:41:23] jin6@titan1:~/CUDA/Lock [31] ./a.out
blockCountUnlocked counted 31 blocks
blockCountLocked counted 512 blocks
```

Conclusion

- Use atomics when there are race conditions that cannot be handled by normal load/store for reliable inter-thread communication
- Use atomic functions for infrequent, spares, and/or unpredictable global communication
- Use lock when we need to perform arbitrary operations on an associated memory location or data structure
- Use atomics can make performance slow, so use judiciously!