CUDA C: RACE CONDITIONS AND ATOMICS

Will Landau, Prof. Jarad Niemi

OUTLINE

- Overview of race conditions and atomics
- CUDA C built-in atomic functions
- Locks and mutex
- Warps

Featured examples:

- `dot_product_atomic_builtin.cu`
- `dot_product_atomic_lock.cu`
- `race_condition.cu`
- `race_condition_fixed.cu`
- `blockCounter.cu`

RACE CONDITIONS AND ATOMICS

Consider the following GPU operation on integer `x`, which is stored in global memory:

```
x++;
```

which tells the GPU to do 3 things...

```
x++;
```

1. Read the value stored in x.
2. Add 1 to the value read in step 1.
3. Write the result back to x.

Say we need threads A and B to increment x. We want:

STEP	EXAMPLE
1. Thread A reads the value in `x`.	`A` reads 7 from `x`.
2. Thread A adds 1 to the value it read.	`A` computes 8.
3. Thread A writes the result back to `x`.	`x <- 8`.
4. Thread B reads the value in `x`.	`B` reads 8 from `x`.
5. Thread B adds 1 to the value it read.	`B` computes 9.
6. Thread B writes the result back to `x`.	`x <- 9`.

but we might get:

STEP	EXAMPLE
Thread A reads the value in `x`.	`A` reads 7 from `x`.
Thread B reads the value in `x`.	`B` reads 7 from `x`.
Thread A adds 1 to the value it read.	`A` computes 8.
Thread B adds 1 to the value it read.	`B` computes 8.
Thread A writes the result back to `x`.	`x <- 8`.
Thread B writes the result back to `x`.	`x <- 8`.

EXAMPLE: race_condition.cu

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <cuda_runtime.h>

__global__ void colonel(int *a_d){
  *a_d += 1;
}

int main(){

  int a = 0, *a_d;

  cudaMalloc((void**) &a_d, sizeof(int));
  cudaMemcpy(a_d, &a, sizeof(int), cudaMemcpyHostToDevice);

  float    elapsedTime;
  cudaEvent_t start, stop;
  cudaEventCreate(&start);
  cudaEventCreate(&stop);
```

```
    cudaEventRecord( start, 0 );

    colonel<<<1000,1000>>>(a_d);

    cudaEventRecord( stop, 0 );
    cudaEventSynchronize( stop );
    cudaEventElapsedTime( &elapsedTime, start, stop );
    cudaEventDestroy( start );
    cudaEventDestroy( stop );
    printf("GPU Time elapsed: %f seconds\n", elapsedTime/1000.0);


    cudaMemcpy(&a, a_d, sizeof(int), cudaMemcpyDeviceToHost);

    printf("a = %d\n", a);
    cudaFree(a_d);

}
```

```
[landau@impact1 race_condition]$ make
nvcc race_condition.cu -o race_condition
[landau@impact1 race_condition]$ ./race_condition
GPU Time elapsed: 0.000148 seconds
a = 88
[landau@impact1 race_condition]$
```

Since we started with `a` at 0, we should have gotten `a` $= 1000 \cdot 1000 = 1{,}000{,}000$.

Race Condition: A computational hazard that arises when the results of a program depend on the timing of uncontrollable events, such as threads.

Atomic Operation: A command that is executed one thread at a time, thus avoiding a race condition.

To avoid the race condition in our example, we atomically add 1 to `*a_d`:

```
atomicAdd( a_d, 1 );
```

instead of using `*a_d += 1;`

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <cuda_runtime.h>

__global__ void colonel(int *a_d){
  atomicAdd( a_d, 1 );
}

int main(){

  int a = 0, *a_d;

  cudaMalloc((void**) &a_d, sizeof(int));
  cudaMemcpy(a_d, &a, sizeof(int), cudaMemcpyHostToDevice);

  float    elapsedTime;
  cudaEvent_t start, stop;
  cudaEventCreate(&start);
  cudaEventCreate(&stop);
  cudaEventRecord( start, 0 );

  colonel<<<1000,1000>>>(a_d);
```

```
    cudaEventRecord( stop, 0 );
    cudaEventSynchronize( stop );
    cudaEventElapsedTime( &elapsedTime, start, stop );
    cudaEventDestroy( start );
    cudaEventDestroy( stop );
    printf("GPU Time elapsed: %f seconds\n", elapsedTime/1000.0);


    cudaMemcpy(&a, a_d, sizeof(int), cudaMemcpyDeviceToHost);

    printf("a = %d\n", a);
    cudaFree(a_d);

}
```

```
[landau@impact1 race_condition_fixed]$ make
nvcc race_condition_fixed.cu -arch sm_20 -o race_condition_fixed
[landau@impact1 race_condition_fixed]$ ./race_condition_fixed
GPU Time elapsed: 0.014850 seconds
a = 1000000
[landau@impact1 race_condition_fixed]$
```

We got the right answer, but the execution time was 100 times longer? Why?

THINGS TO NOTE

- The code slowed down because that the additions happened sequentially instead of simultaneously.
- If you're using any of the above functions in your code, compile with the flag, `-arch sm_20` as above. (Atomics support floating point operations only for CUDA "compute capability" 2.0. and above.)

LIST OF CUDA C BUILT-IN ATOMIC FUNCTIONS

atomicAdd()
atomicSub()
atomicMin()
atomicMax()
atomicInc()
atomicDec()
atomicExch()
atomicCAS()

For documentation, refer to the CUDA C Programming Guide (http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf).

```
int atomicAdd(int* address, int val);
unsigned int atomicAdd(unsigned int* address,
                       unsigned int val);
unsigned long long int atomicAdd(unsigned long long int* address,
                                 unsigned long long int val);
float atomicAdd(float* address, float val);
```

reads the 32-bit or 64-bit word **old** located at the address **address** in global or shared memory, computes **(old + val)**, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns **old**.

```
int atomicSub(int* address, int val);
```

```
unsigned int atomicSub(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)**, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns **old**.

```
int atomicExch(int* address, int val);
unsigned int atomicExch(unsigned int* address,
                        unsigned int val);
unsigned long long int atomicExch(unsigned long long int* address,
                                  unsigned long long int val);
float atomicExch(float* address, float val);
```

reads the 32-bit or 64-bit word **old** located at the address **address** in global or shared memory and stores **val** back to memory at the same address. These two operations are performed in one atomic transaction. The function returns **old**.

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

reads the 32-bit word **old** located at the address **address** in global or shared memory, computes the minimum of **old** and **val**, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns **old**.

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

reads the 32-bit word **old** located at the address **address** in global or shared memory, computes the maximum of **old** and **val**, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns **old**.

```
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**.

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

reads the 32-bit word **old** located at the address **address** in global or shared memory, computes **(((old == 0) | (old > val)) ? val : (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**.

```
int atomicCAS(int* address, int compare, int val);
unsigned int atomicCAS(unsigned int* address,
                       unsigned int compare,
                       unsigned int val);
unsigned long long int atomicCAS(unsigned long long int* address,
                                 unsigned long long int compare,
                                 unsigned long long int val);
```

reads the 32-bit or 64-bit word **old** located at the address **address** in global or shared memory, computes **(old == compare ? val : old)**, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns **old** (Compare And Swap).

```
int atomicAnd(int* address, int val);
unsigned int atomicAnd(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)**, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns **old**.

```
int atomicOr(int* address, int val);
unsigned int atomicOr(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)**, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns **old**.

```
int atomicXor(int* address, int val);
unsigned int atomicXor(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)**, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns **old**.

EXAMPLE: THE ATOMIC DOT PRODUCT

$$(a_0,\ a_1, \ldots,\ a_{15}) \bullet (b_0,\ b_1, \ldots,\ b_{15}) = a_0b_0 + a_1b_1 + \cdots + a_{15}b_{15}$$

The basic workflow is:

- Pass vectors `a` and `b` to the GPU.
- Give each block a sub vector of `a` and the analogous subvector of `b`.

  For the case of 16-element vectors, let there be 2 blocks and 4 threads per block.

Block 0 works on:

$$(a_0, a_1, a_2, a_3,\ a_8, a_9, a_{10}, a_{11})$$
$$(b_0, b_1, b_2, b_3,\ b_8, b_9, b_{10}, b_{11})$$

Block 1 works on:

$$(a_4, a_5, a_6, a_7,\ a_{12}, a_{13}, a_{14}, a_{15})$$
$$(b_4, b_5, b_6, b_7,\ b_{12}, b_{13}, b_{14}, b_{15})$$

- Within each block, compute a vector of partial sums of pairwise products:

Block 0:

$$\texttt{cache} = (a_0 \cdot b_0 + a_8 \cdot b_8,\ a_1 \cdot b_1 + a_9 \cdot b_9\ \ldots,\ a_3 \cdot b_3 + a_{11} \cdot b_{11})$$

Block 1:

$$\texttt{cache} = (a_4 \cdot b_4 + a_{12} \cdot b_{12},\ a_5 \cdot b_5 + a_{13} \cdot b_{13},\ \ldots,\ a_7 \cdot b_7 + a_{15} \cdot b_{15})$$

where `cache` is an array in shared memory.

- Within each block, compute the pairwise sum of `cache` and write it to `cache[0]`. In our example:

  Block 0:

  $$\texttt{cache[0]} = a_0 \cdot b_0 + \cdots + a_3 \cdot b_3 + a_8 \cdot b_8 + \cdots + a_{11} \cdot b_{11}$$

  Block 1:

  $$\texttt{cache[0]} = a_4 \cdot b_4 + \cdots + a_7 \cdot b_7 + a_{12} \cdot b_{12} + \cdots + a_{15} \cdot b_{15}$$

SKIP THESE STEPS:

- Fill a new vector in global memory, `partial_c`, with these partial dot products:

  `partial_c` = (`block 0 cache[0]`, `block 1 cache[0]`)

- Return to the CPU and compute the linear sum of `partial_c` and write it to `partial_c[0]`. Then:

  `partial_c[0]` $= a_0 \cdot b_0 + a_1 \cdot b_1 + \cdots + a_{15} \cdot b_{15}$

INSTEAD:

- Within `dot()`, ATOMICALLY ADD each block's copy of `cache[0]` to the integer `c`. Then:

$$\texttt{c} = a_0 \cdot b_0 + a_1 \cdot b_1 + \cdots + a_{15} \cdot b_{15}$$

dot_product_atomic_builtin.cu

```
#include "../common/book.h"
#include "../common/lock.h"

#define imin(a,b) (a<b?a:b)

// NOTE: COMPILE LIKE THIS:
//nvcc dot_product_atomic.cu -arch sm_20 -o dot_product_atomic

const int N = 32 * 1024 * 1024;
const int threadsPerBlock = 256;
const int blocksPerGrid =
  imin( 32, (N+threadsPerBlock-1) / threadsPerBlock );

__global__ void dot( Lock lock, float *a,
                     float *b, float *c ) {

  __shared__ float cache[threadsPerBlock];
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  int cacheIndex = threadIdx.x;

  float temp = 0;
  while (tid < N) {
```

```
        temp += a[tid] * b[tid];
        tid += blockDim.x * gridDim.x;
}

// set the cache values
cache[cacheIndex] = temp;

// synchronize threads in this block
__syncthreads();

// for reductions, threadsPerBlock must be a power of 2
// because of the following code
int i = blockDim.x/2;
while (i != 0) {
  if (cacheIndex < i)
    cache[cacheIndex] += cache[cacheIndex + i];
  __syncthreads();
  i /= 2;
}

// Here's where we use atomic addition:
if (cacheIndex == 0) {
        atomicAdd(c, cache[0]);
}
```

```
}

int main( void ) {
  float *a, *b, c = 0;
  float *dev_a, *dev_b, *dev_c;

  // allocate memory on the CPU side
  a = (float*)malloc( N*sizeof(float) );
  b = (float*)malloc( N*sizeof(float) );

  // allocate the memory on the GPU
  HANDLE_ERROR( cudaMalloc( (void**)&dev_a,
                            N*sizeof(float) ) );
  HANDLE_ERROR( cudaMalloc( (void**)&dev_b,
                            N*sizeof(float) ) );
  HANDLE_ERROR( cudaMalloc( (void**)&dev_c,
                            sizeof(float) ) );

  // fill in the host memory with data
  for (int i=0; i<N; i++) {
    a[i] = i;
    b[i] = i*2;
  }
```

```
// copy the arrays 'a' and 'b' to the GPU
HANDLE_ERROR( cudaMemcpy( dev_a, a, N*sizeof(float),
                          cudaMemcpyHostToDevice ) );
HANDLE_ERROR( cudaMemcpy( dev_b, b, N*sizeof(float),
                          cudaMemcpyHostToDevice ) );
HANDLE_ERROR( cudaMemcpy( dev_c, &c, sizeof(float),
                          cudaMemcpyHostToDevice ) );

Lock lock;
dot<<<blocksPerGrid,threadsPerBlock>>>( lock, dev_a,
                                        dev_b, dev_c );

// copy c back from the GPU to the CPU
HANDLE_ERROR( cudaMemcpy( &c, dev_c,
                          sizeof(float),
                          cudaMemcpyDeviceToHost ) );

#define sum_squares(x) (x*(x+1)*(2*x+1)/6)
printf( "Does GPU value %.6g = %.6g?\n", c,
         2 * sum_squares( (float)(N - 1) ) );

// free memory on the GPU side
cudaFree( dev_a );
cudaFree( dev_b );
```

```
    cudaFree( dev_c );

    // free memory on the CPU side
    free( a );
    free( b );
}
```

OUTPUT

```
[landau@impact1 dot_product_atomic_builtin]$ nvcc
  dot_product_atomic_builtin.cu -arch sm_20 -o dot
[landau@impact1 dot_product_atomic_builtin]$ ./dot
Does GPU value 2.76217e+22 = 2.76217e+22?
[landau@impact1 dot_product_atomic_builtin]$
```

Again, if you're using any of the above functions in your code, compile with the flag, `-arch sm_20` as above.

LOCKS

Lock: a mechanism in parallel computing that forces a block of code to be executed atomically.

mutex: short for "mutual exclusion", the idea behind locks: while a thread is running code inside a lock, it blocks all other threads from running the code.

lock.h:

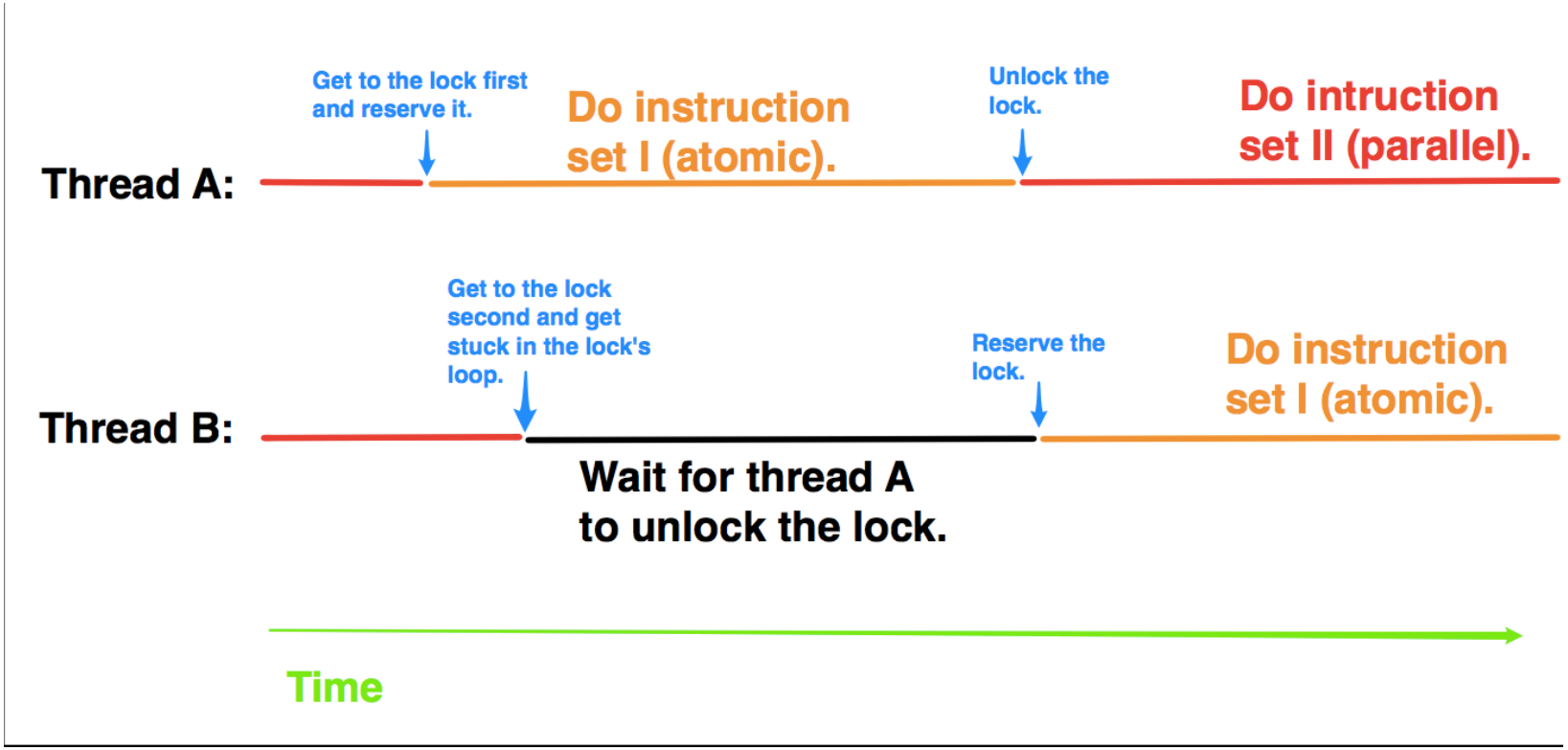
```
struct Lock {
    int *mutex;
    Lock( void ) {
        int state = 0;
        HANDLE_ERROR( cudaMalloc( (void**)& mutex,
                                  sizeof(int) ) );
        HANDLE_ERROR( cudaMemcpy( mutex, &state, sizeof(int),
                                  cudaMemcpyHostToDevice ) );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

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

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

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

Now, let's look at the lock function:

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

In pseudo-code:

```
__device__ void lock( void ) {
  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
```


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

dot_product_atomic_lock.cu: THE ATOMIC DOT PRODUCT WITH LOCKS

```
#include "../common/book.h"
#include "../common/lock.h"

#define imin(a,b) (a<b?a:b)

// NOTE: COMPILE LIKE THIS:
//nvcc dot_product_atomic.cu -arch sm_20 -o dot_product_atomic

const int N = 32 * 1024 * 1024;
const int threadsPerBlock = 256;
const int blocksPerGrid =
  imin( 32, (N+threadsPerBlock-1) / threadsPerBlock );

__global__ void dot( Lock lock, float *a,
                     float *b, float *c ) {

  __shared__ float cache[threadsPerBlock];
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  int cacheIndex = threadIdx.x;
```

```
float temp = 0;
while (tid < N) {
      temp += a[tid] * b[tid];
      tid += blockDim.x * gridDim.x;
}

// set the cache values
cache[cacheIndex] = temp;

// synchronize threads in this block
__syncthreads();

// for reductions, threadsPerBlock must be a power of 2
// because of the following code
int i = blockDim.x/2;
while (i != 0) {
  if (cacheIndex < i)
    cache[cacheIndex] += cache[cacheIndex + i];
  __syncthreads();
  i /= 2;
}

// Here's where locks come in:
```

```
    if (cacheIndex == 0) { lock.lock();
            *c += cache[0];
            lock.unlock();
    }
}

int main( void ) {
    float *a, *b, c = 0;
    float *dev_a, *dev_b, *dev_c;

    // allocate memory on the CPU side
    a = (float*)malloc( N*sizeof(float) );
    b = (float*)malloc( N*sizeof(float) );

    // allocate the memory on the GPU
    HANDLE_ERROR( cudaMalloc( (void**)&dev_a,
                              N*sizeof(float) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_b,
                              N*sizeof(float) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_c,
                              sizeof(float) ) );

    // fill in the host memory with data
    for (int i=0; i<N; i++) {
```

```
    a[i] = i;
    b[i] = i*2;
}

// copy the arrays 'a' and 'b' to the GPU
HANDLE_ERROR( cudaMemcpy( dev_a, a, N*sizeof(float),
                          cudaMemcpyHostToDevice ) );
HANDLE_ERROR( cudaMemcpy( dev_b, b, N*sizeof(float),
                          cudaMemcpyHostToDevice ) );
HANDLE_ERROR( cudaMemcpy( dev_c, &c, sizeof(float),
                          cudaMemcpyHostToDevice ) );

Lock lock;
dot<<<blocksPerGrid,threadsPerBlock>>>( lock, dev_a,
                                        dev_b, dev_c );

// copy c back from the GPU to the CPU
HANDLE_ERROR( cudaMemcpy( &c, dev_c,
                          sizeof(float),
                          cudaMemcpyDeviceToHost ) );

#define sum_squares(x) (x*(x+1)*(2*x+1)/6)
printf( "Does GPU value %.6g = %.6g?\n", c,
         2 * sum_squares( (float)(N - 1) ) );
```

```
    // free memory on the GPU side
    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );

    // free memory on the CPU side
    free( a );
    free( b );
}
```

OUTPUT

```
[landau@impact1 dot_product_atomic_lock]$ nvcc
   dot_product_atomic_lock.cu -arch sm_20 -o dot
[landau@impact1 dot_product_atomic_lock]$ ./dot
Does GPU value 2.76217e+22 = 2.76217e+22?
[landau@impact1 dot_product_atomic_lock]$
```

NOTE: we still need `-arch sm_20` because the lock relies on built-in atomic functions.

ANOTHER EXAMPLE: COUNTING THE NUMBER OF BLOCKS

Compare these two kernels, both of which attempt to count the number of spawned blocks:

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

__global__ void blockCounter1( Lock lock, int *nblocks ){
  if(threadIdx.x == 0){
    lock.lock();
    *nblocks = *nblocks + 1;
    lock.unlock();
  }
}
```

Which one gives us the correct answer?
Which one is faster?

blockCounter.cu

```
#include "../common/lock.h"
#define NBLOCKS_TRUE 512
#define NTHREADS_TRUE 512 * 2

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

__global__ void blockCounter1( Lock lock, int *nblocks ){
  if(threadIdx.x == 0){
    lock.lock();
    *nblocks = *nblocks + 1;
    lock.unlock();
  }
}
```

```
int main(){
  int nblocks_host, *nblocks_dev;
  Lock lock;
  float elapsedTime;
  cudaEvent_t start, stop;

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


  //blockCounterUnlocked:

  nblocks_host = 0;
  cudaMemcpy( nblocks_dev, &nblocks_host, sizeof(int), cudaMemcpyHostToDevice );

  cudaEventCreate(&start);
  cudaEventCreate(&stop);
  cudaEventRecord( start, 0 );

  blockCounterUnlocked<<<NBLOCKS_TRUE, NTHREADS_TRUE>>>(nblocks_dev);

  cudaEventRecord( stop, 0 );
  cudaEventSynchronize( stop );
  cudaEventElapsedTime( &elapsedTime, start, stop );
```

```
cudaEventDestroy( start );
cudaEventDestroy( stop );

cudaMemcpy( &nblocks_host, nblocks_dev, sizeof(int), cudaMemcpyDeviceToHost );
printf("blockCounterUnlocked <<< %d, %d >>> () counted %d blocks in %f ms.\n",
      NBLOCKS_TRUE,
      NTHREADS_TRUE,
      nblocks_host,
      elapsedTime);


//blockCounter1:

nblocks_host = 0;
cudaMemcpy( nblocks_dev, &nblocks_host, sizeof(int), cudaMemcpyHostToDevice );

cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord( start, 0 );

blockCounter1<<<NBLOCKS_TRUE, NTHREADS_TRUE>>>(lock, nblocks_dev);

cudaEventRecord( stop, 0 );
cudaEventSynchronize( stop );
```

```
  cudaEventElapsedTime( &elapsedTime, start, stop );

  cudaEventDestroy( start );
  cudaEventDestroy( stop );

  cudaMemcpy( &nblocks_host, nblocks_dev, sizeof(int), cudaMemcpyDeviceToHost );
  printf("blockCounter1 <<< %d, %d >>> () counted %d blocks in %f ms.\n",
        NBLOCKS_TRUE,
        NTHREADS_TRUE,
        nblocks_host,
        elapsedTime);

  cudaFree(nblocks_dev);
}
```

```
[landau@impact1 blockCounter]$ nvcc blockCounter.cu  -arch sm_20 -o blockCounter
[landau@impact1 blockCounter]$ ./blockCounter
blockCounterUnlocked <<< 512, 1024 >>> () counted 47 blocks in 0.057920 ms.
blockCounter1 <<< 512, 1024 >>> () counted 512 blocks in 0.636064 ms.
[landau@impact1 blockCounter]$
[landau@impact1 blockCounter]$ ./blockCounter
blockCounterUnlocked <<< 512, 1024 >>> () counted 51 blocks in 0.052288 ms.
blockCounter1 <<< 512, 1024 >>> () counted 512 blocks in 0.638304 ms.
[landau@impact1 blockCounter]$
[landau@impact1 blockCounter]$ ./blockCounter
blockCounterUnlocked <<< 512, 1024 >>> () counted 47 blocks in 0.052096 ms.
blockCounter1 <<< 512, 1024 >>> () counted 512 blocks in 0.640768 ms.
[landau@impact1 blockCounter]$
```

WITH MORE THAN 1 THREAD PER BLOCK, THIS KERNEL WILL MAKE YOUR PROGRAM PAUSE INDEFINITELY

```
__global__ void blockCounter2( Lock lock, int *nblocks ){
  lock.lock();
  if(threadIdx.x == 0){
    *nblocks = *nblocks + 1 ;
  }
  lock.unlock();
}
```

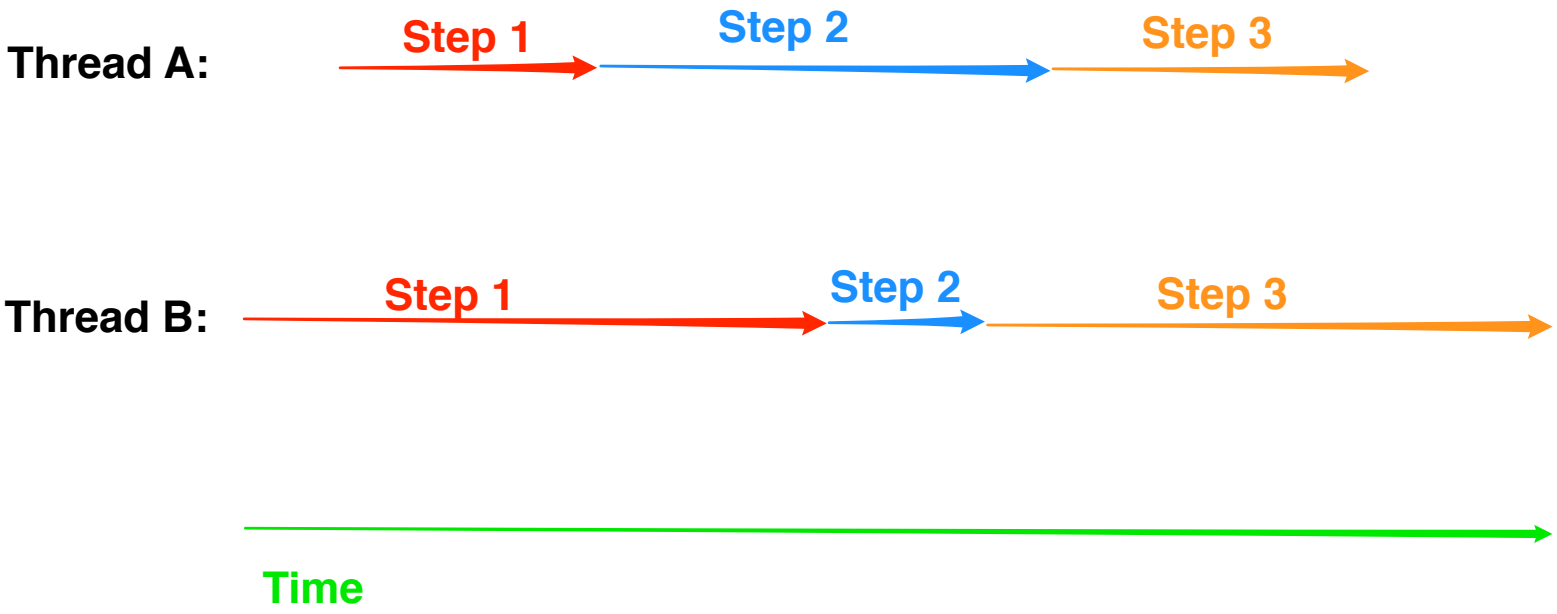
WHY? BECAUSE OF WARPS!

Each block is divided into groups of 32 threads called warps.

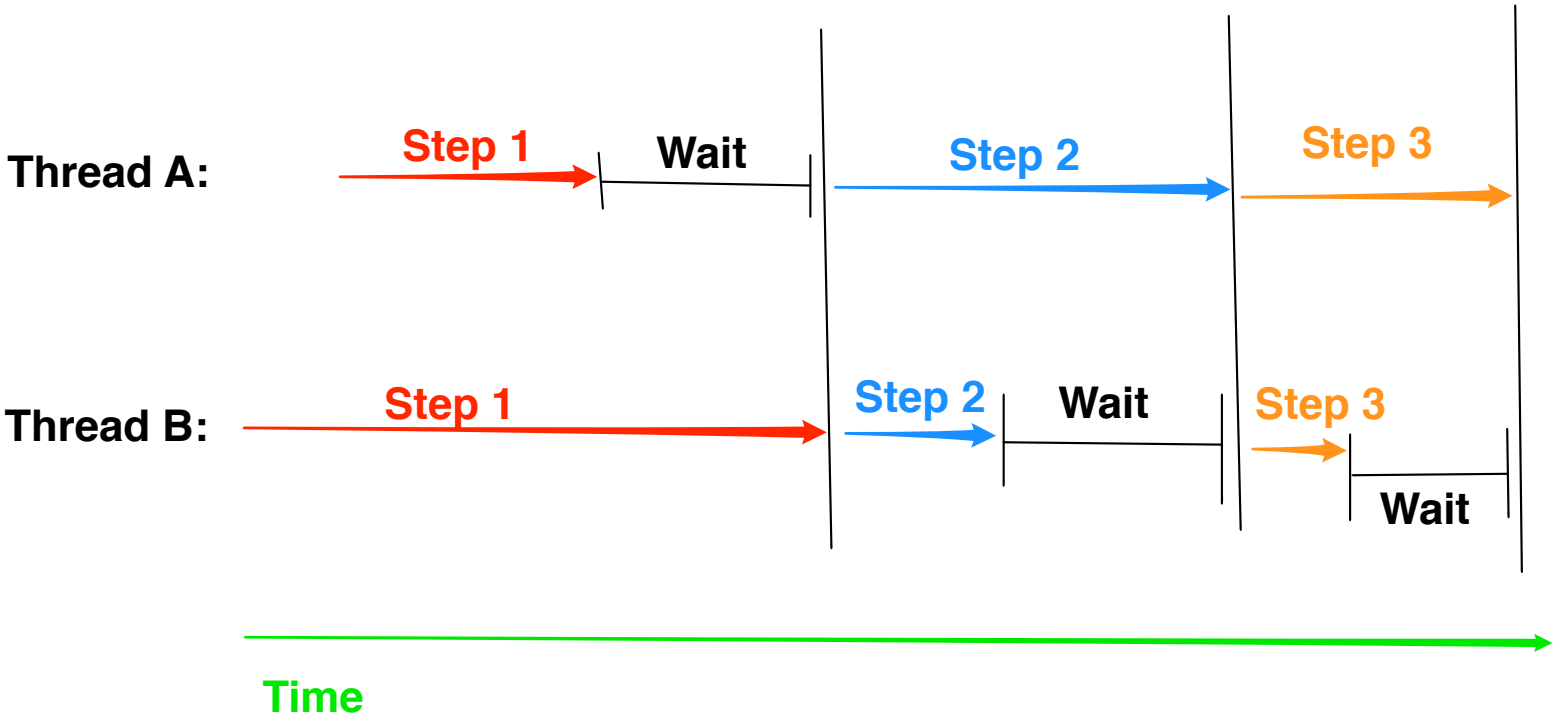
Warp: a group of 32 threads that execute together in lockstep: that is, all threads in the warp synchronize after every single step.

Imagine that a warp is saturated with calls to `__synchThreads()`.

Threads in different warps:



Threads in the same warp:



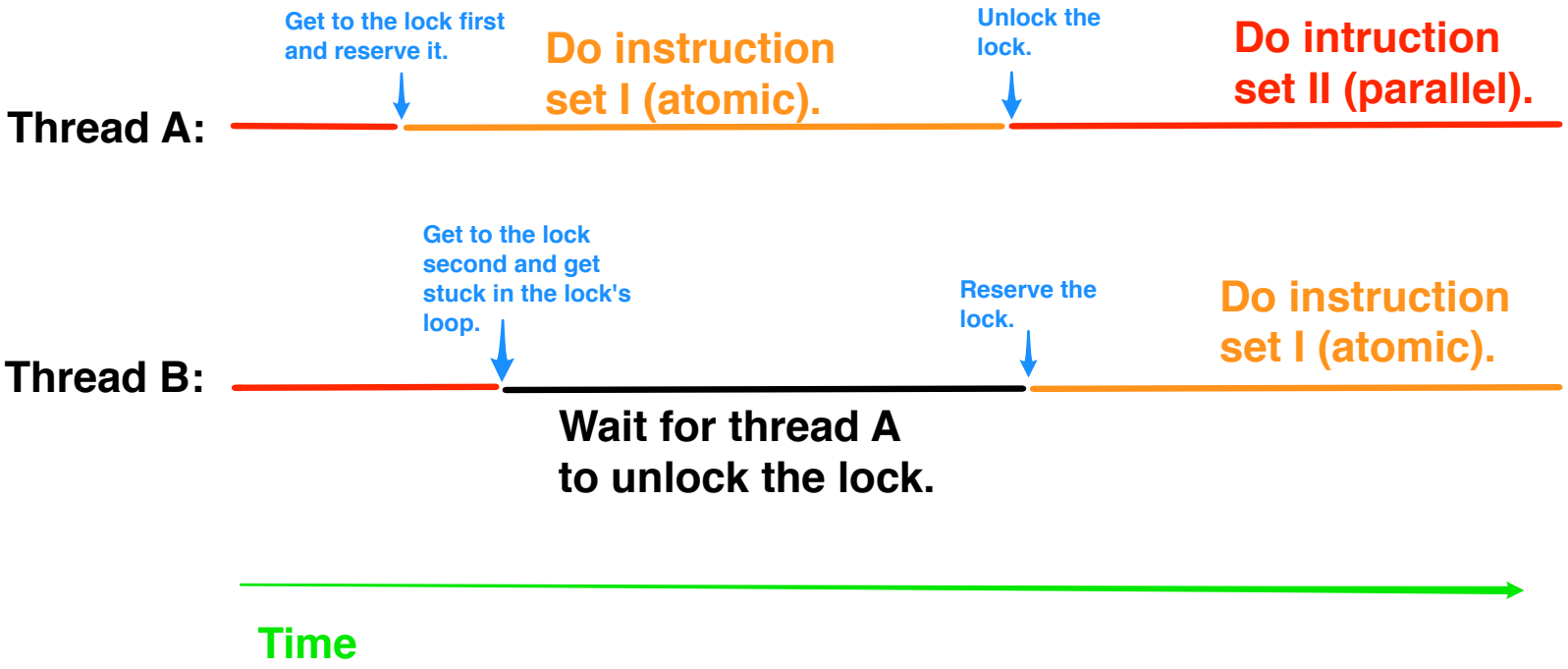
Now, let's look at the lock function again:

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

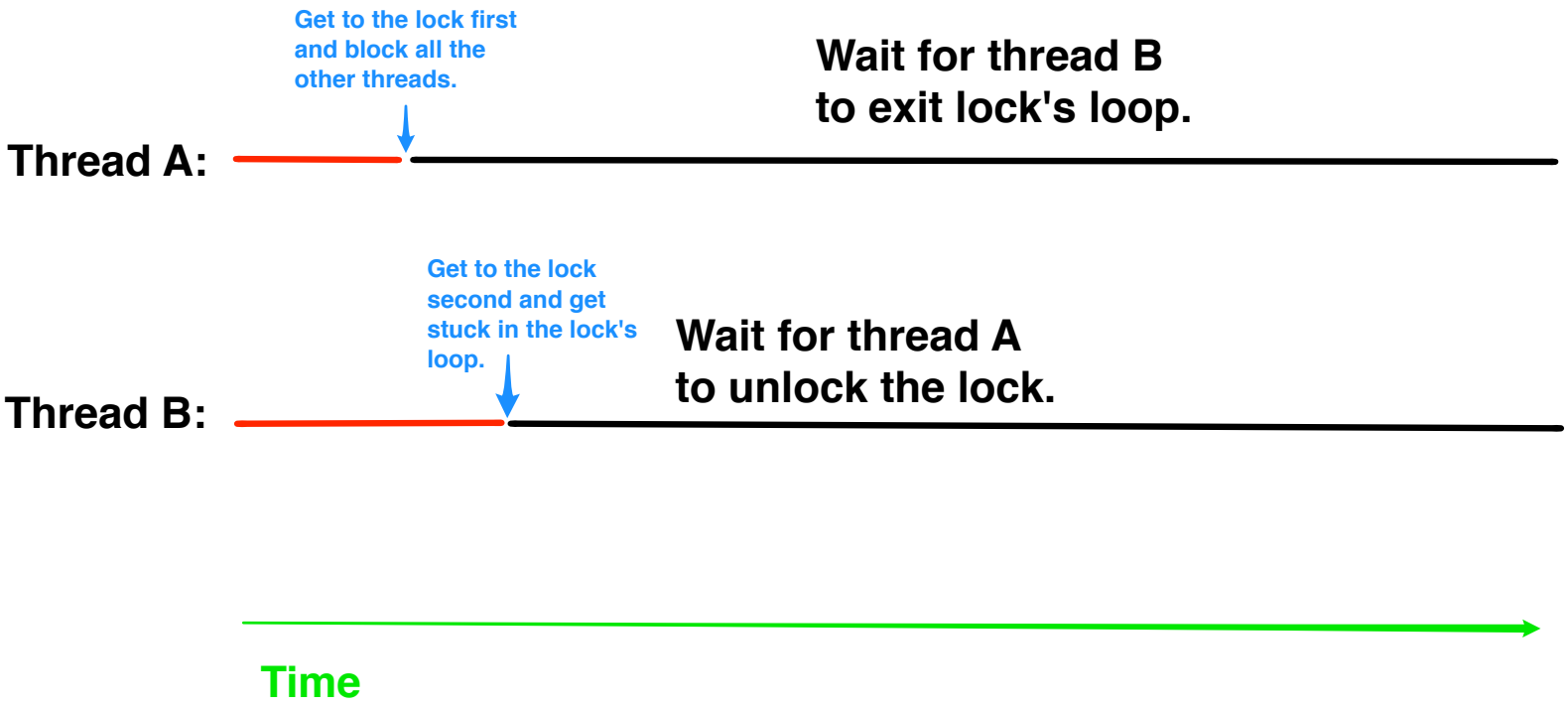
In pseudo-code:

```
__device__ void lock( void ) {
  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
```

Threads in different warps:



Threads in the same warp:



KEEP TO THE SINGLE INSTRUCTION MULTIPLE DATA PARADIGM AS MUCH AS POSSIBLE

- Race conditions are difficult to debug.
- Atomics are sequential, and therefore slow.
- Locks are tricky, and may cause your programs to pause indefinitely.

OUTLINE

- Race conditions and atomics
- CUDA C built-in atomic functions
- Locks and mutex
- Warps

Featured examples:

- `dot_product_atomic_builtin.cu`
- `dot_product_atomic_lock.cu`
- `race_condition.cu`
- `race_condition_fixed.cu`
- `blockCounter.cu`

LECTURE SERIES MATERIALS

These lecture slides, a tentative syllabus for the whole lecture series, and code are available at:

https://github.com/wlandau/gpu.

After logging into you home directory on impact1, type:

```
git clone https://github.com/wlandau/gpu
```

into the command line to download all the course materials.

REFERENCES

NVIDIA CUDA C Programming Guide. Version 3.2. 2010.

J. Sanders and E. Kandrot. *CUDA by Example*. Addison-Wesley, 2010.