PERFORMANCE MEASUREMENT AND ATOMICS IN CUDA C

Will Landau, Prof. Jarad Niemi

EVENTS: MEASURING PERFORMANCE ON THE GPU

Event: a time stamp for the GPU.

Use events to measure the amount of time the GPU spends on a task.

```
int main(){
  float   elapsedTime;
  cudaEvent_t start, stop;

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

  // SOME GPU KERNEL YOU WANT TIMED HERE

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

  cudaEventDestroy( start );
  cudaEventDestroy( stop );

  printf("GPU Time elapsed: %f\n", elapsedTime);
}
```

The variable, `elapsedTime`, is the GPU time spent on the task. You can now print it any way you like.

Note: only GPU elapsed time is measured, not CPU time.

GPU time and CPU time must be measured separately.

MEASURING CPU TIME

```
#include <stdio.h>
#include <time.h>

int main(){

  clock_t start = clock();

  // SOME CPU CODE YOU WANT TIMED HERE

  float elapsedTime = ((double)clock() - start) /
                                 CLOCKS_PER_SEC;

  printf("CPU Time elapsed: %f\n", elapsedTime);
}
```

RACE CONDITIONS AND ATOMICS

Consider the following GPU operation on integer **`x`**:

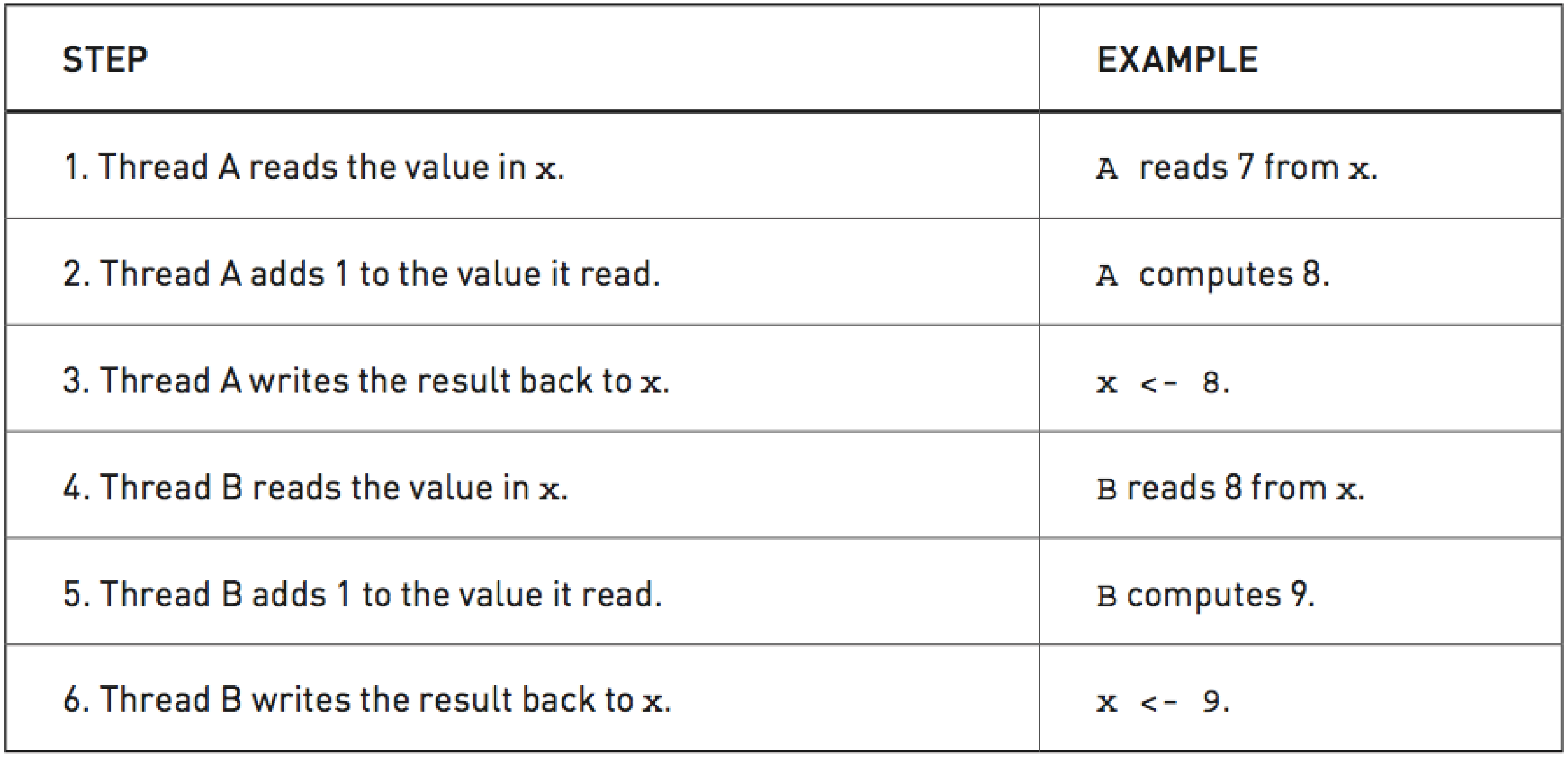
```
x++;
```

which tells the GPU to do 3 things...

```
x++;
```

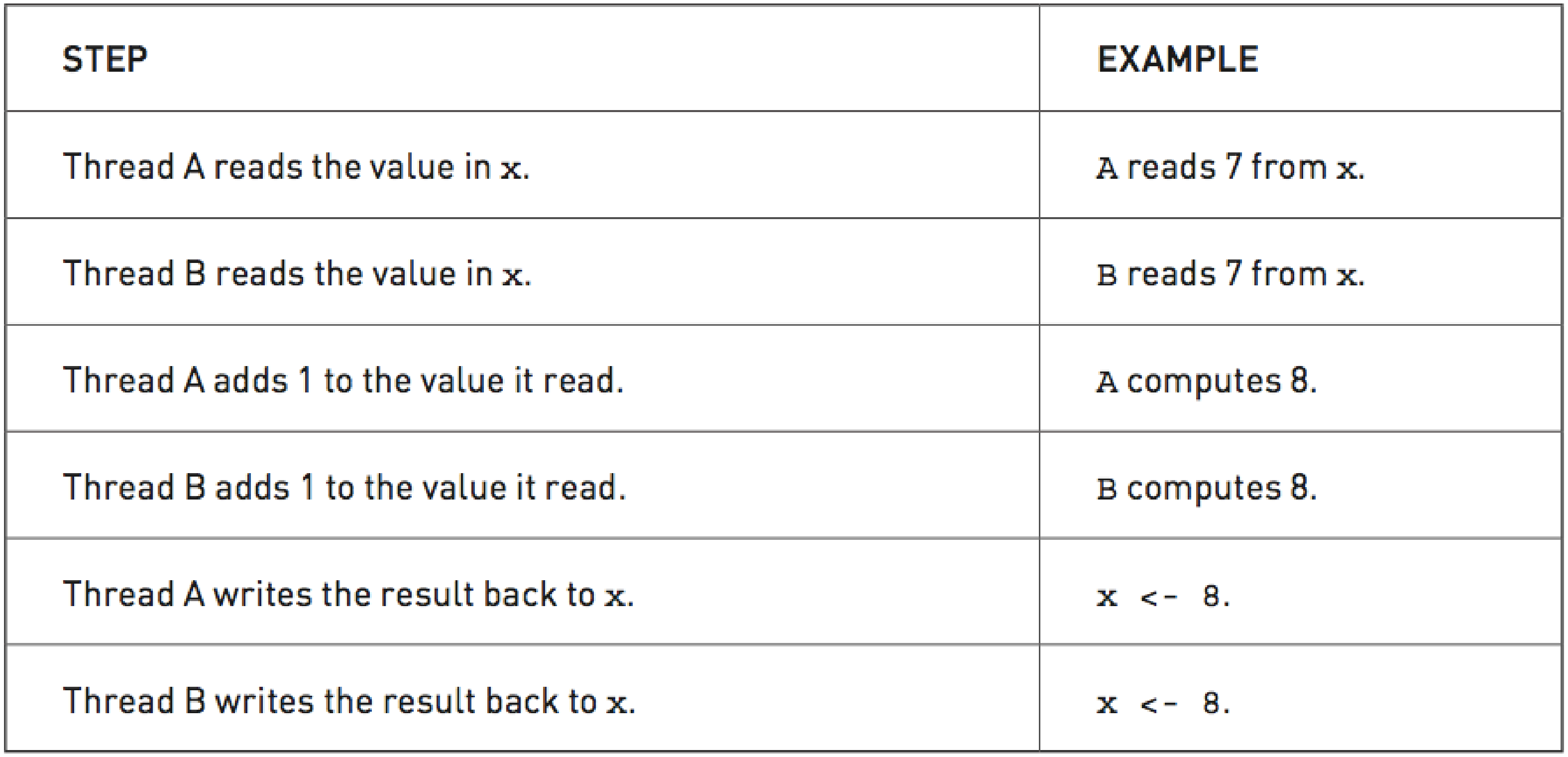
1. Read the value stored in x.
2. Add 1 to the value read in step 1.
3. Write the result back to x.

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.`

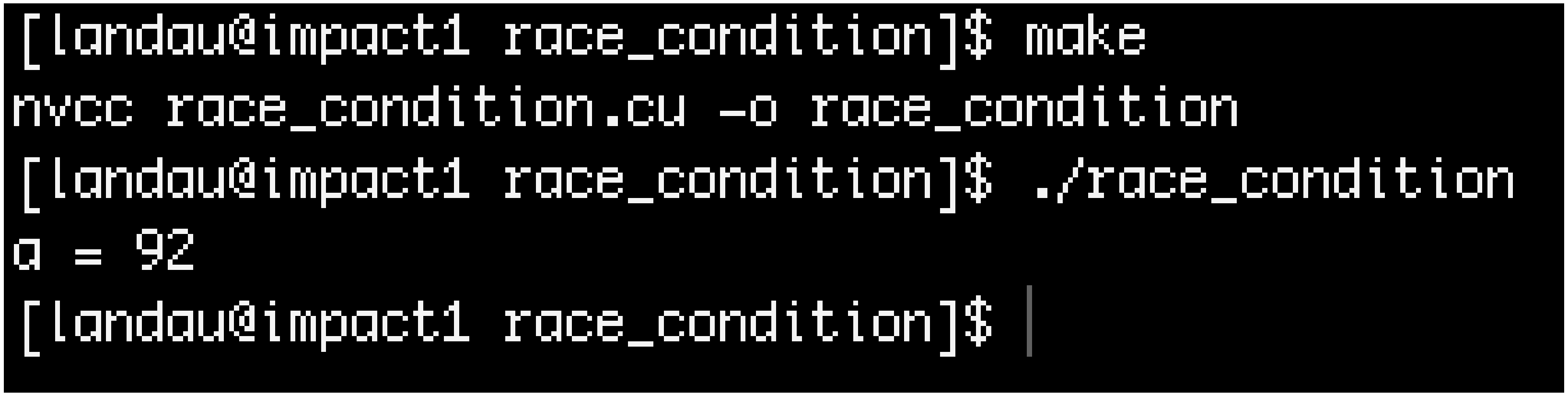
but we might get:



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);

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

  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
a = 92
[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.

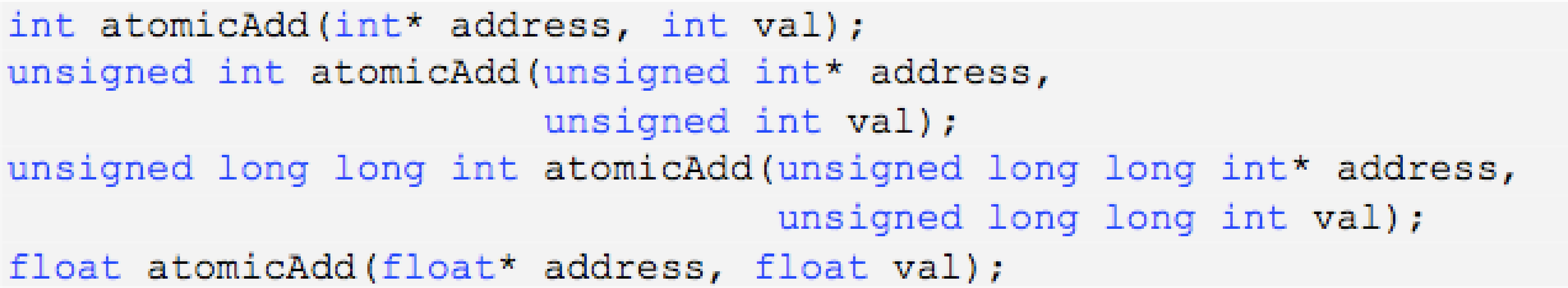
To avoid the race condition in our example, we atomically add 1 to x:

```
atomicAdd( &x, 1 );
```

instead of using `x++;`

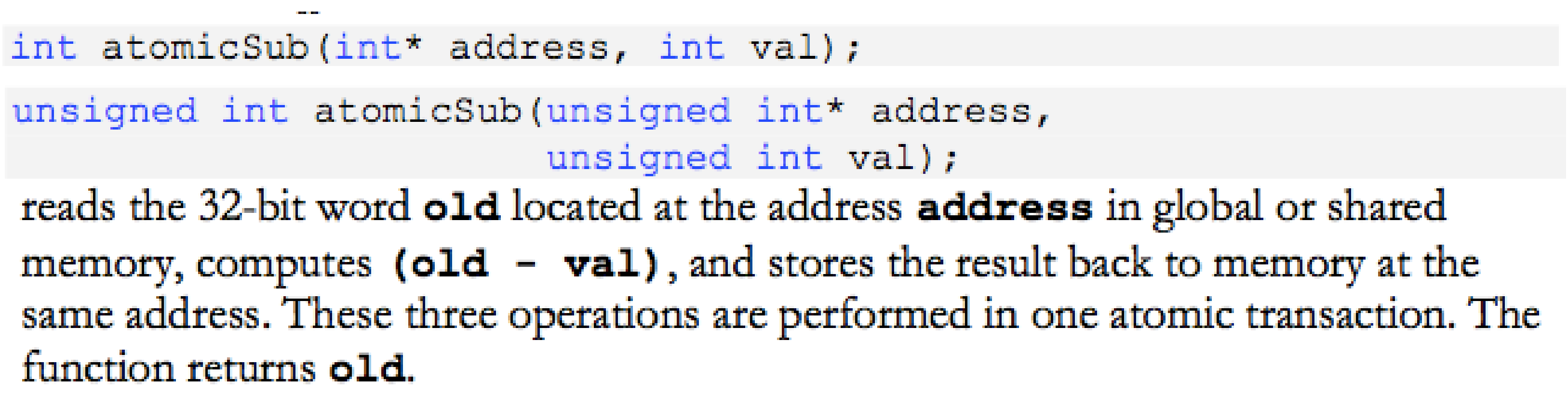
LIST OF CUDA C BUILT-IN ATOMIC FUNCTIONS

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



```
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);
```

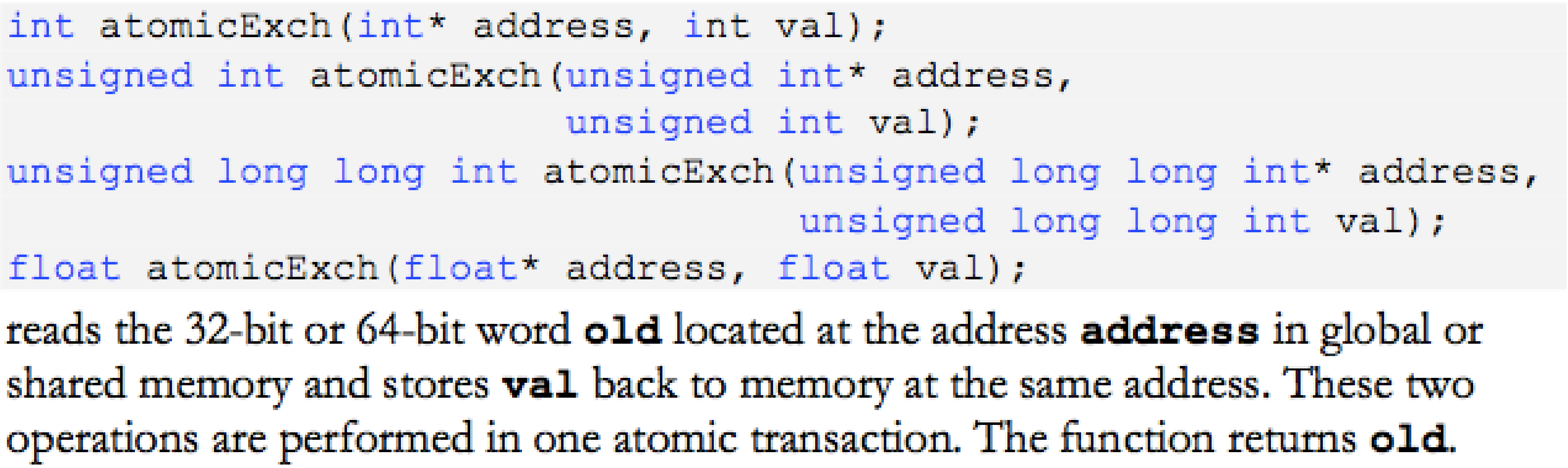
reads the 32-bit or 64-bit word **`old`** located at the address **`address`** in global or shared memory, computes **`(old + val)`**, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns **`old`**.



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

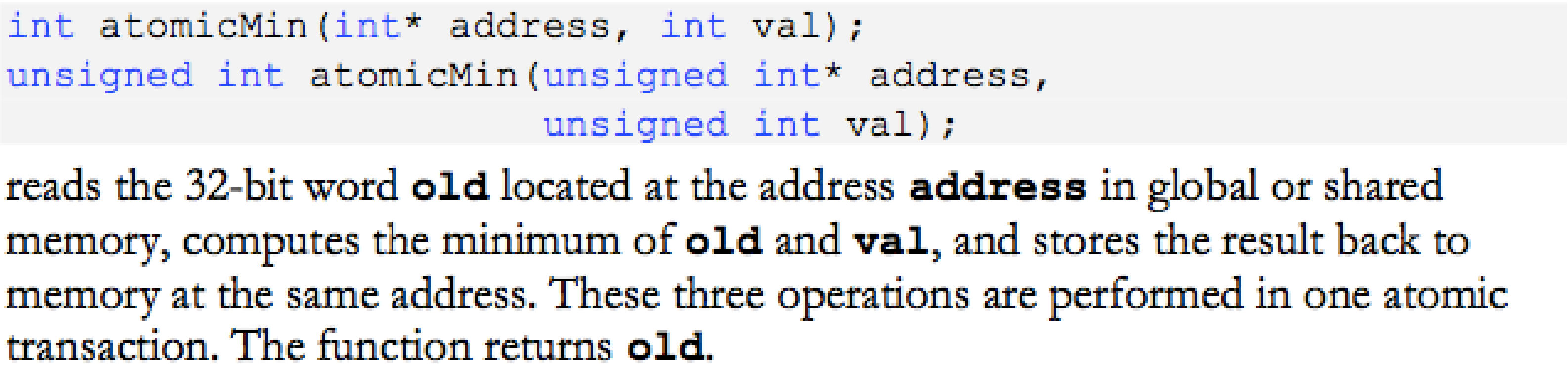
```
unsigned int atomicSub(unsigned int* address,
                       unsigned int val);
```

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`**.



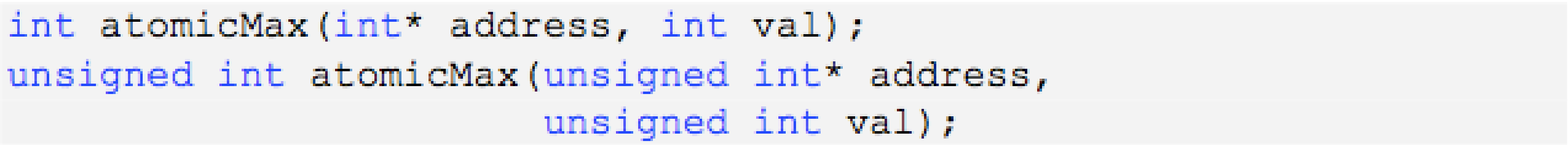
```
int atomicExch(int* address, int val);
unsigned int atomicExch(unsigned int* address,
                        unsigned int val);
unsigned long long int atomicExch(unsigned long long int* address,
                                  unsigned long long int val);
float atomicExch(float* address, float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



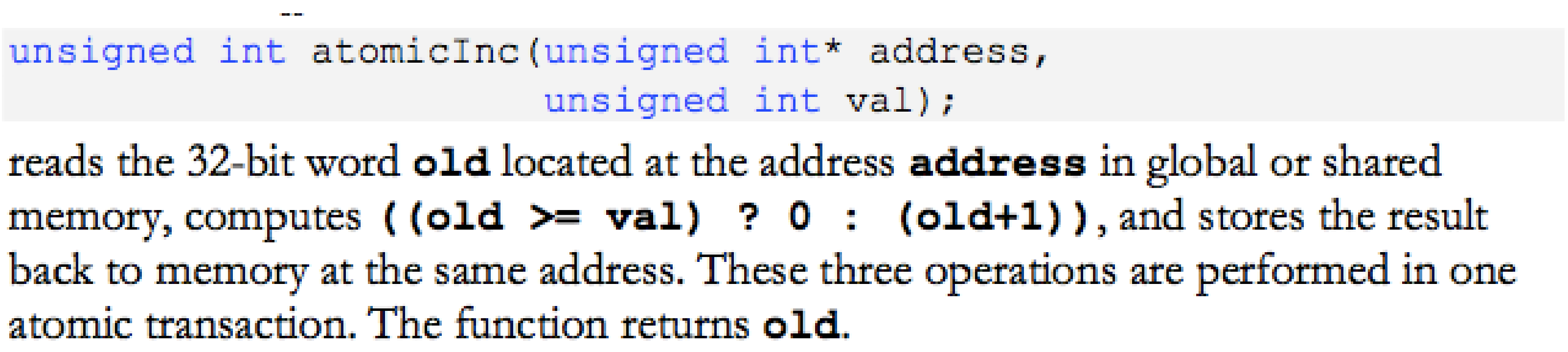
```
int atomicMin(int* address, int val);
unsigned int atomicMin(unsigned int* address,
                       unsigned int val);
```

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);
```

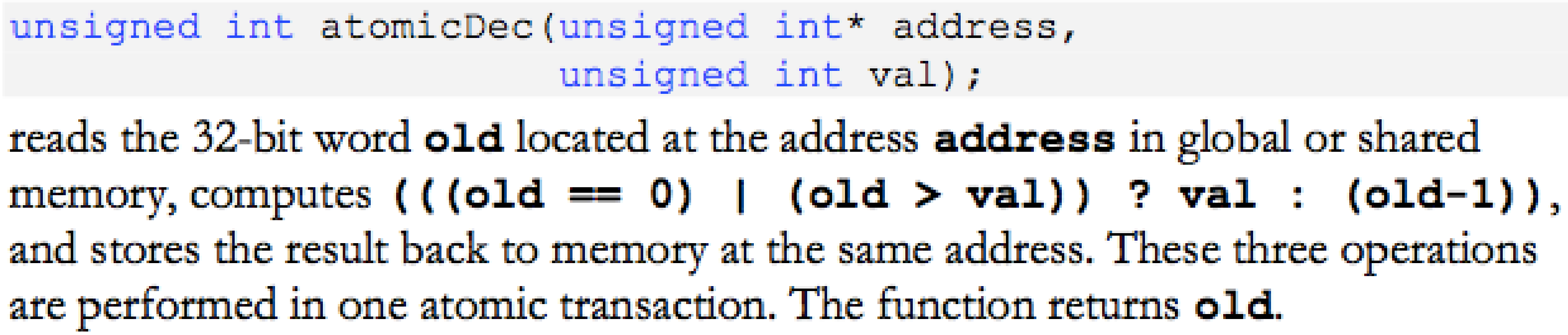
reads the 32-bit word **old** located at the address **address** in global or shared memory, computes the maximum of **old** and **val**, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns **old**.



--

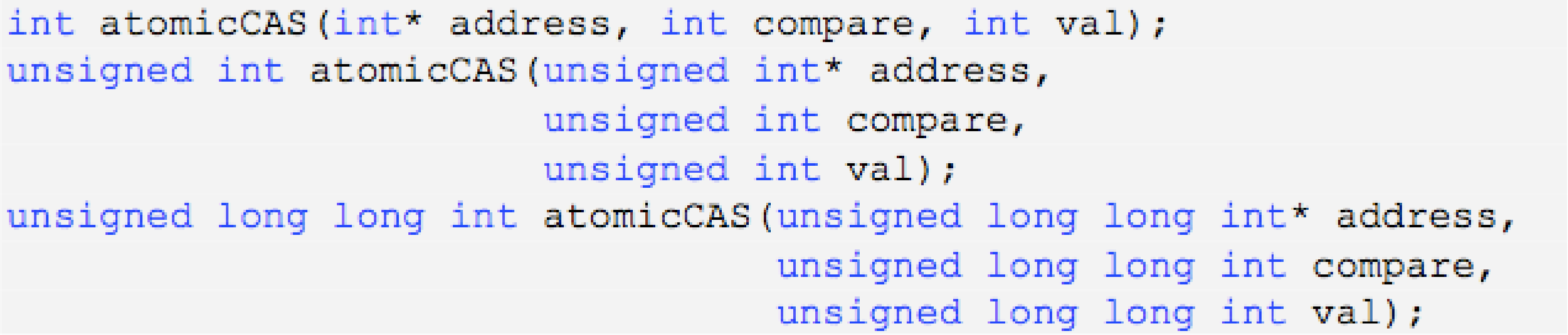
```
unsigned int atomicInc(unsigned int* address,
                       unsigned int val);
```

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`**.



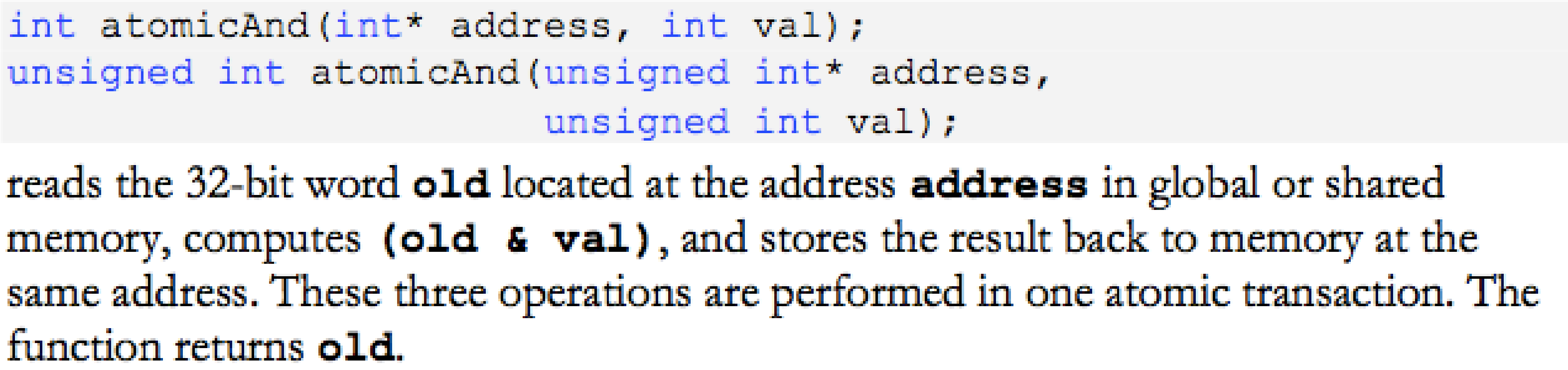
```
unsigned int atomicDec(unsigned int* address,
                       unsigned int val);
```

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**.



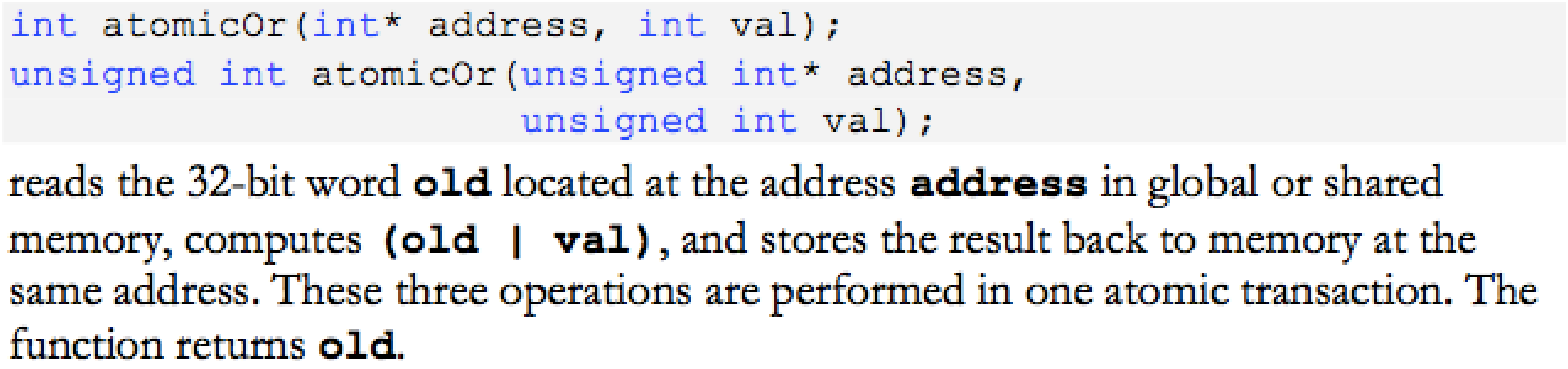
```
int atomicCAS(int* address, int compare, int val);
unsigned int atomicCAS(unsigned int* address,
                       unsigned int compare,
                       unsigned int val);
unsigned long long int atomicCAS(unsigned long long int* address,
                                 unsigned long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



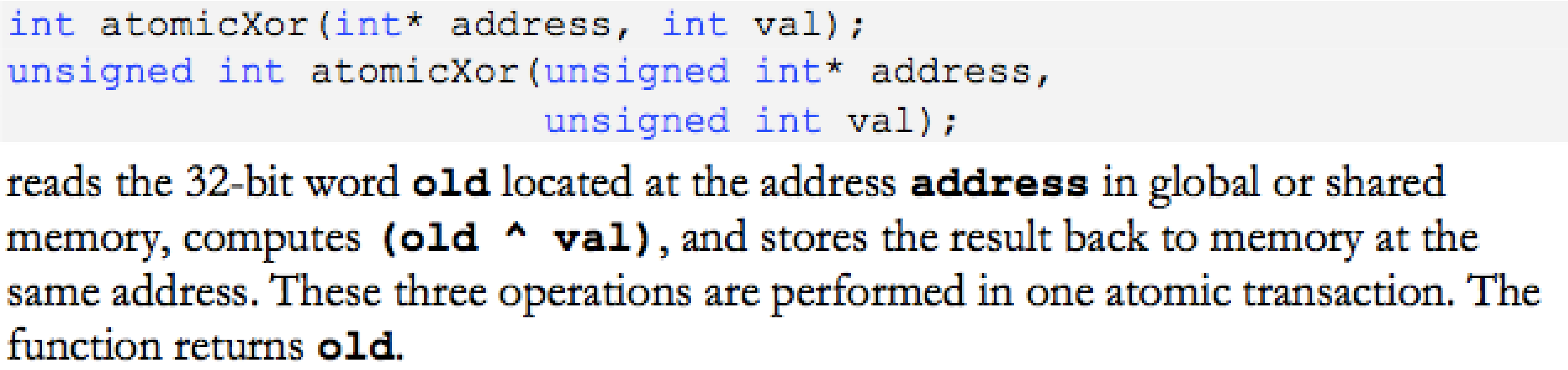
```
int atomicAnd(int* address, int val);
unsigned int atomicAnd(unsigned int* address,
                       unsigned int val);
```

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`**.

NOTE: If you're using any of the above functions in your code, compile with the flag, `-arch sm_20`

Example:

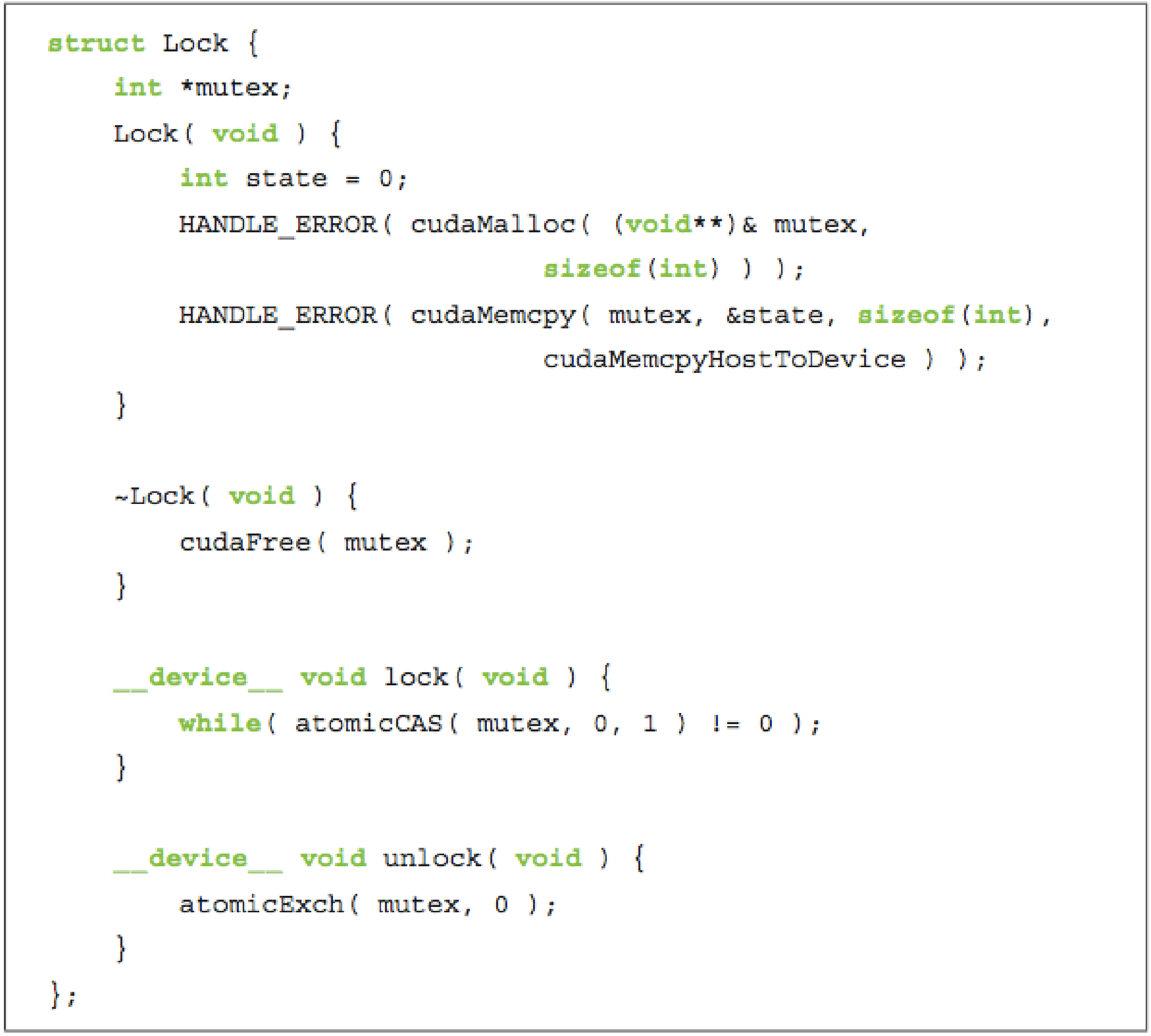
```
nvcc dot_product_atomic.cu -arch sm_20
     -o dot_product_atomic
```

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:



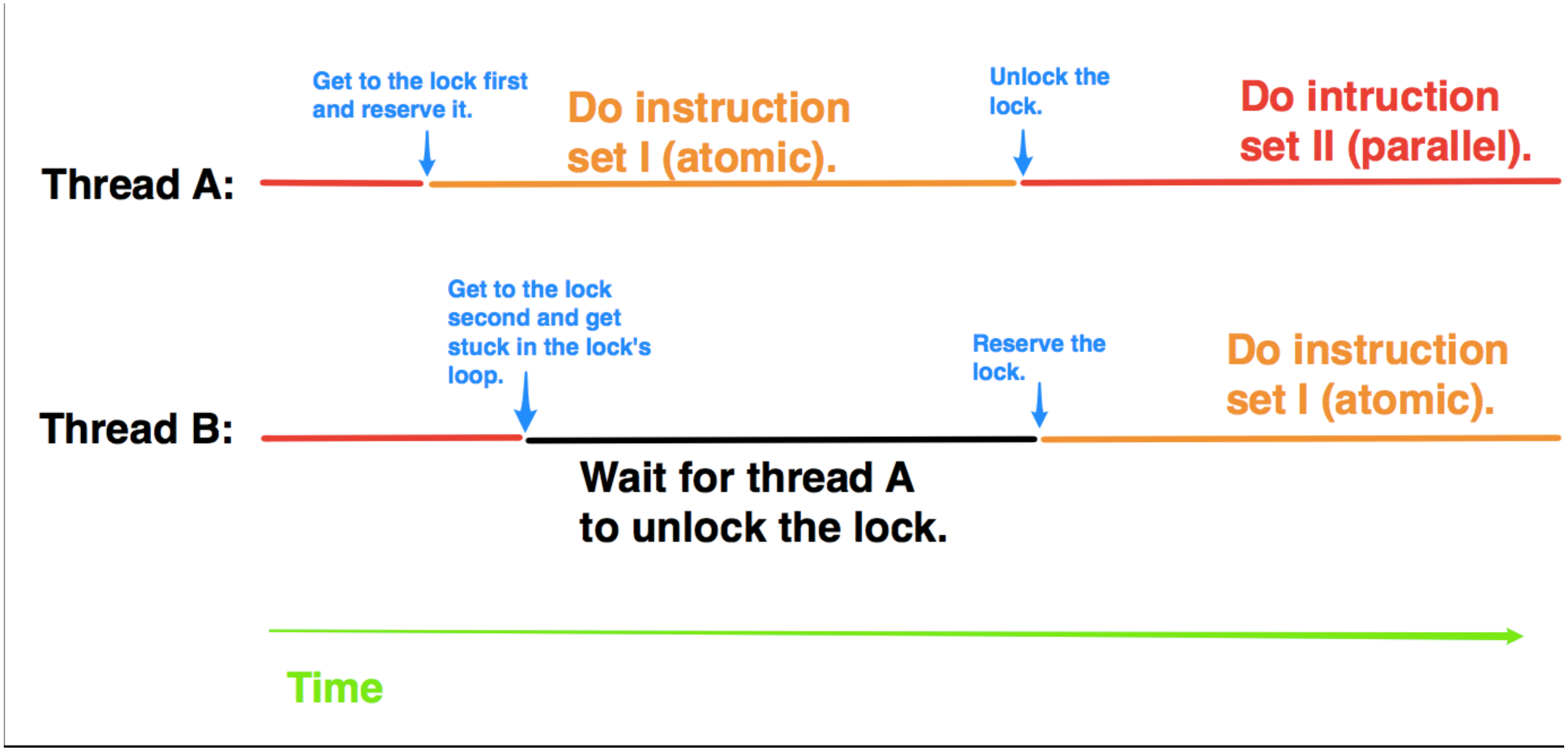
```
struct Lock {
    int *mutex;
    Lock( void ) {
        int state = 0;
        HANDLE_ERROR( cudaMalloc( (void**)& mutex,
                                  sizeof(int) ) );
        HANDLE_ERROR( cudaMemcpy( mutex, &state, sizeof(int),
                                  cudaMemcpyHostToDevice ) );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

    ~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
```


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

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?

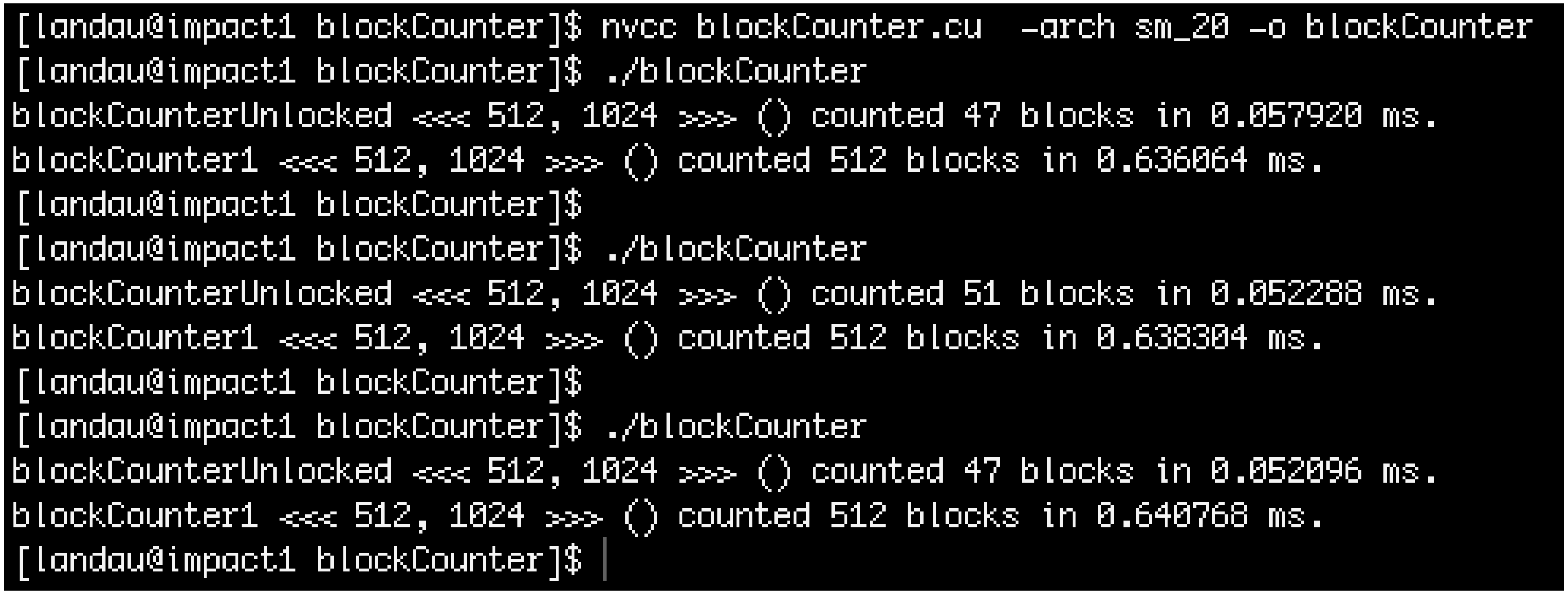
blockCounter.cu

```
#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 STALL OUT

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

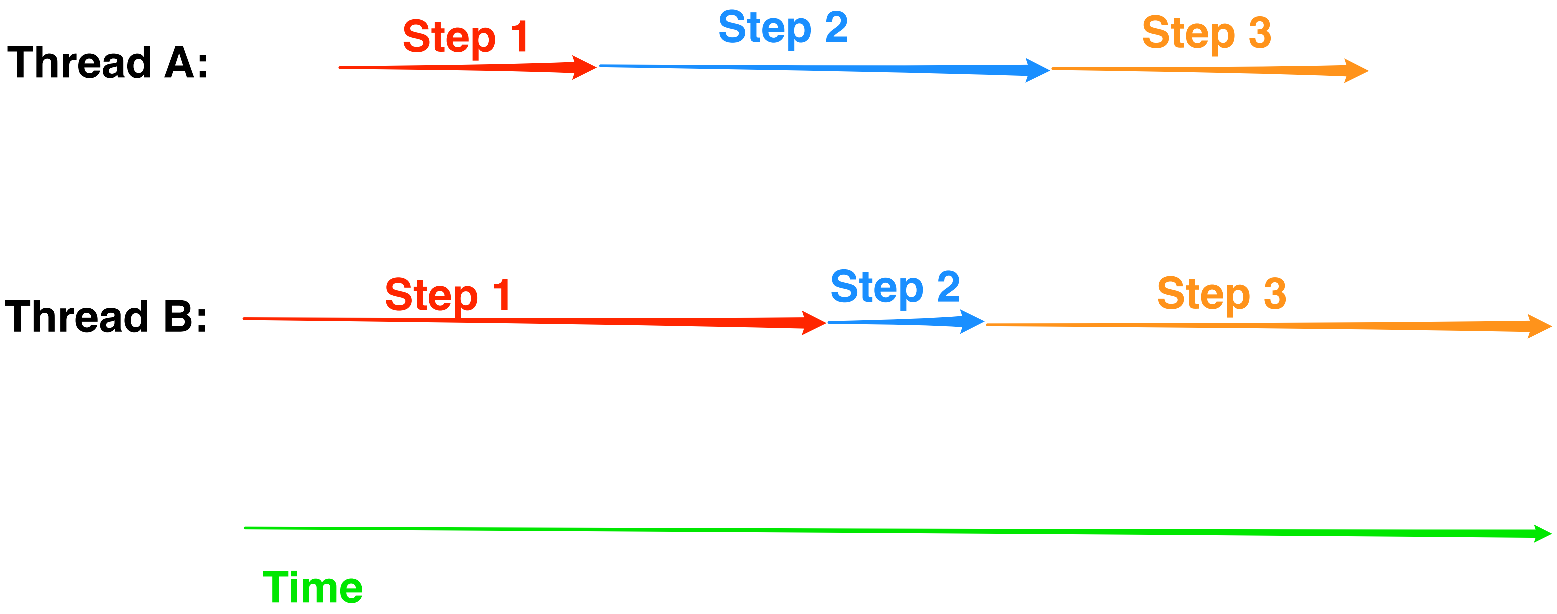
WHY? BECAUSE OF WARPS!

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

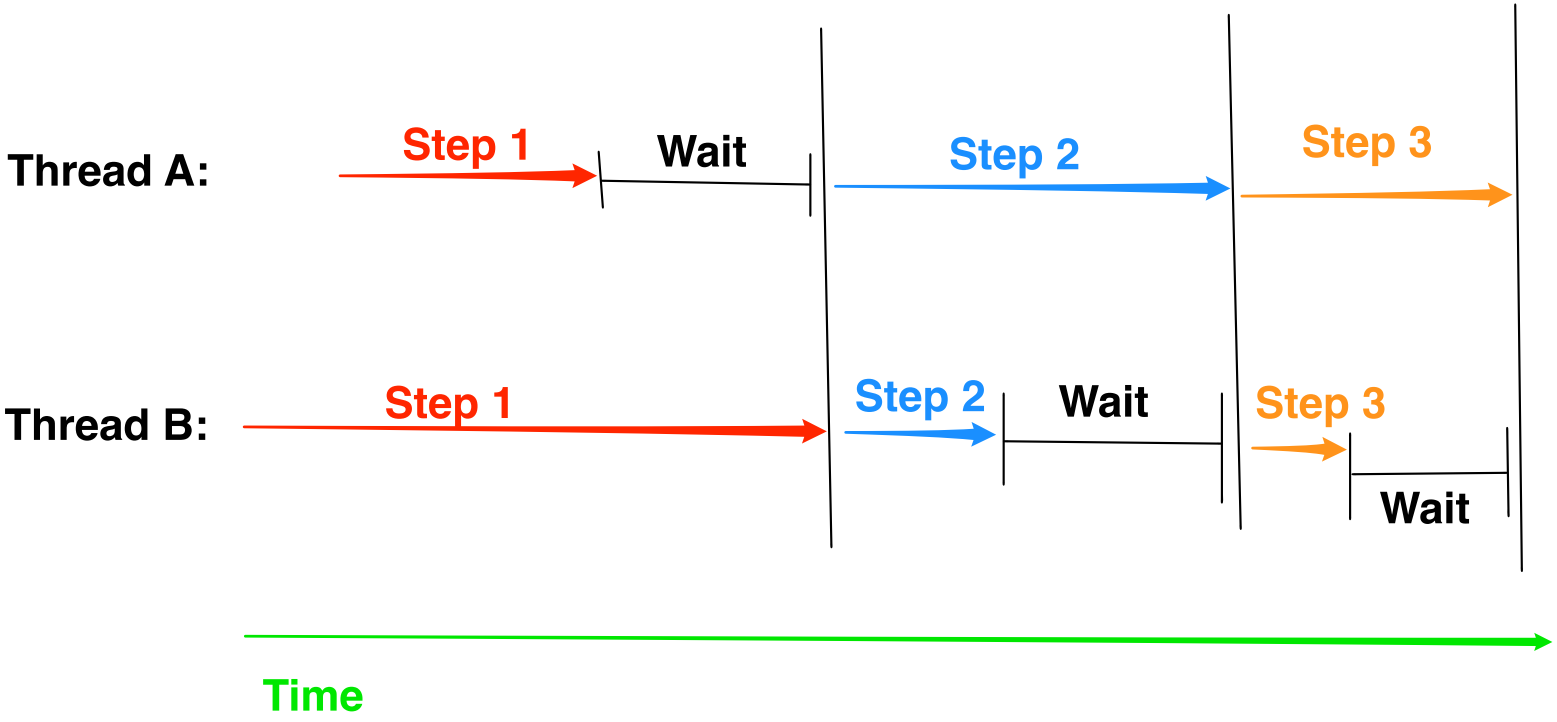
Warp: a group of 32 threads that execute together in lockstep: that is, all threads in the warp synchronize after every single step.

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

Threads in different warps:



Threads in the same warp:



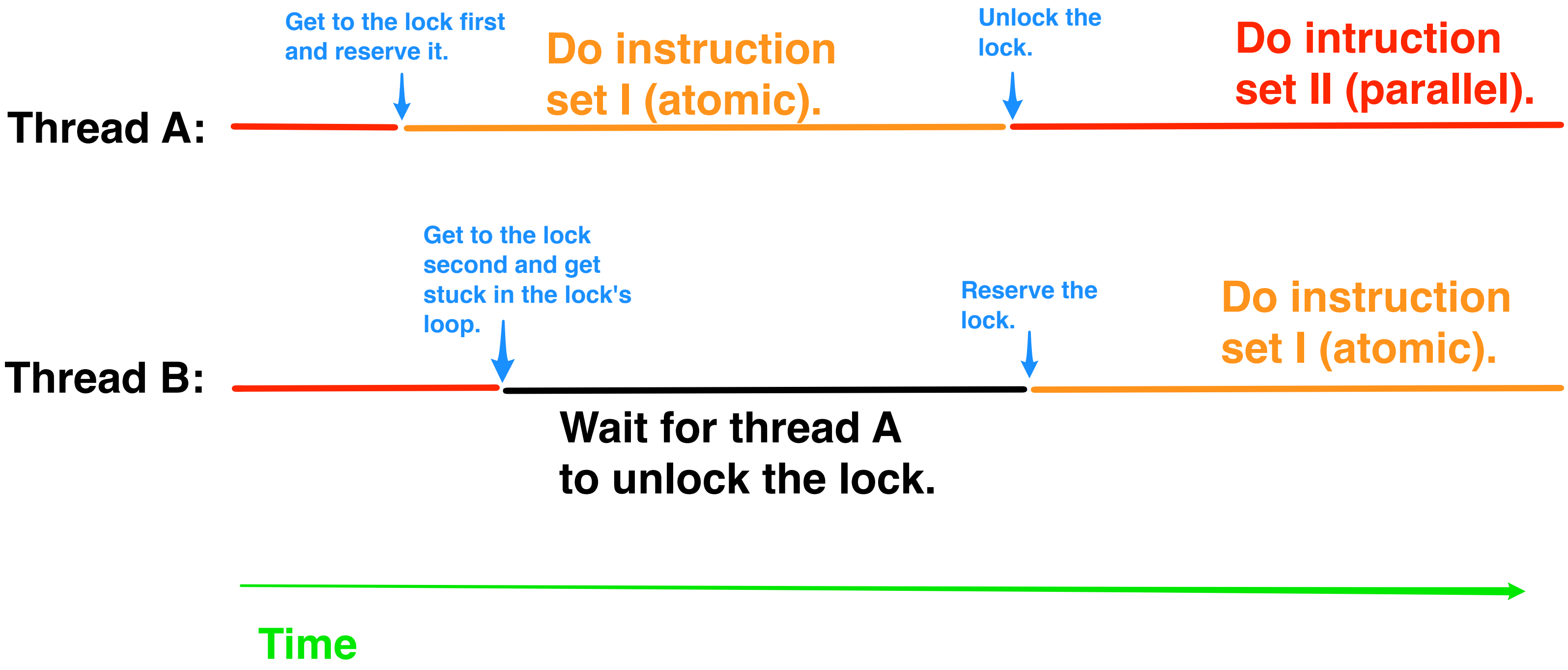
Now, let's look at the lock function again:

```
__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:



Thread A:

Get to the lock first and reserve it.

Do instruction set I (atomic).

Unlock the lock.

Do intruction set II (parallel).

Thread B:

Get to the lock second and get stuck in the lock's loop.

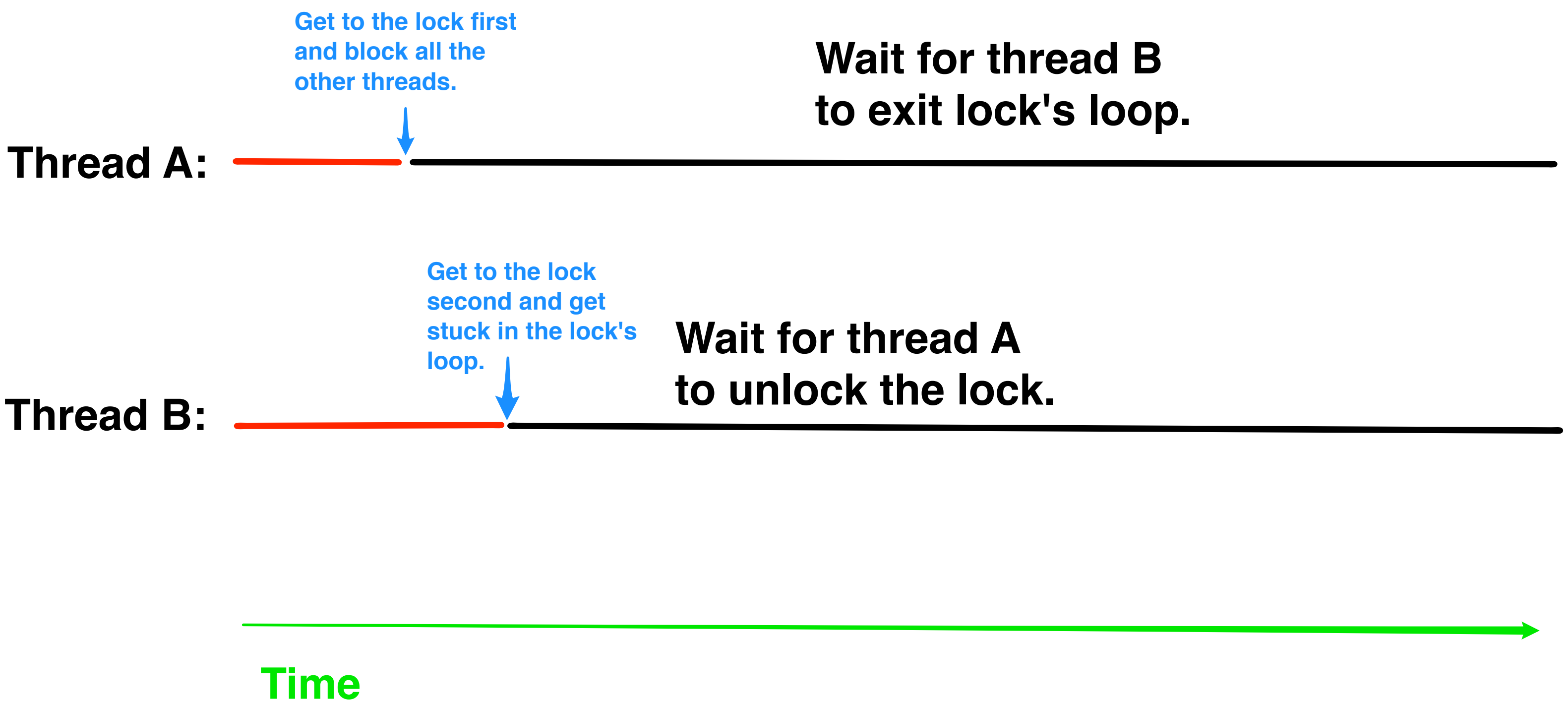
Wait for thread A to unlock the lock.

Reserve the lock.

Do instruction set I (atomic).

Time

Threads in the same warp:



BEWARE THE CATCH-22!

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.