CUDA C: race conditions, atomics, locks, mutex, and warps

Will Landau

Iowa State University

October 21, 2013

Outline

Race conditions

Brute force fixes: atomics, locks, and mutex

Warps

Outline

Race conditions

- Let `int *x` point to global memory. `*x++` happens in 3 steps:
  1. Read the value in `*x` into a register.
  2. Add 1 to the value read in step 1.
  3. Write the result back to `*x`.
- If we want parallel threads A and B to both increment `*x`, then we want something like:
  1. Thread A reads the value, 7, from `*x`.
  2. Thread A adds 1 to its value, 7, to make 8.
  3. Thread A writes its value, 8, back to `*x`.
  4. Thread B reads the value, 8, from `*x`.
  5. Thread B adds 1 to its value, 8, to make 9.
  6. Thread B writes the value, 9, back to `*x`.

Race conditions

- But since the threads are parallel, we might actually get:
  1. Thread A reads the value, 7, from `*x`.
  2. Thread B reads the value, 7, from `*x`.
  3. Thread A adds 1 to its value, 7, to make 8.
  4. Thread A writes its value, 8, back to `*x`.
  5. Thread B adds 1 to its value, 7, to make 8.
  6. Thread B writes the value, 8, back to `*x`.

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);
```

CUDA C: race conditions, atomics, locks, mutex, and warps

Will Landau

Race conditions

Brute force fixes: atomics, locks, and mutex

Warps

Example: race_condition.cu

```
  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);

}
```

```
> nvcc race_condition.cu -o race_condition
> ./race_condition
GPU Time elapsed: 0.000148 seconds
a = 88
```

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

Race conditions

- **Race condition**: A computational hazard that arises when the results of the program depend on the timing of uncontrollable events, such as the execution order or threads.
  - Many race conditions are caused by violations of the SIMD paradigm.
  - Atomic operations and locks are brute force ways to fix race conditions.

Outline

Atomics

- **Atomic operation**: an operation that forces otherwise parallel threads into a bottleneck, executing the operation one at a time.
- In `colonel()`, replace

```
*a_d += 1;
```

with an atomic function,

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

to fix the race condition in `race_condition.cu`.

race_condition_fixed.cu

```
#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);
```

race_condition_fixed.cu

```
  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);

}
```

race_condition_fixed.cu

```
> nvcc race_condition_fixed.cu -arch sm_20 -o race_condition_fixed
> ./race_doncition_fixed
GPU Time elapsed: 0.01485 seconds
a = 1000000
```

- We got the right answer this time, and execution was slower because we forced the threads to execute the addition sequentially.
- If you're using builtin atomic functions like `atomicAdd()`, use the `-arch sm_20` flag in compilation.
  - This is to make sure you're using CUDA compute capability (version) 2.0 or above.

CUDA C builtin atomic functions

- With CUDA compute capability 2.0 or above, you can use:
  - `atomicAdd()`
  - `atomicSub()`
  - `atomicMin()`
  - `atomicMax()`
  - `atomicInc()`
  - `atomicDec()`
  - `atomicAdd()`
  - `atomicExch()`
  - `atomicCAS()`
  - `atomicAnd()`
  - `atomicOr()`
  - `atomicXor()`
- For documentation, refer to the CUDA C programming guide.

`atomicCAS(int *address, int compare, int val)`: needed for locks

1. Read the value, `old`, located at `address`.
2. `*address = (old == compare) ? val : old;`
3. Return `old`.

Locks and mutex

- **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.

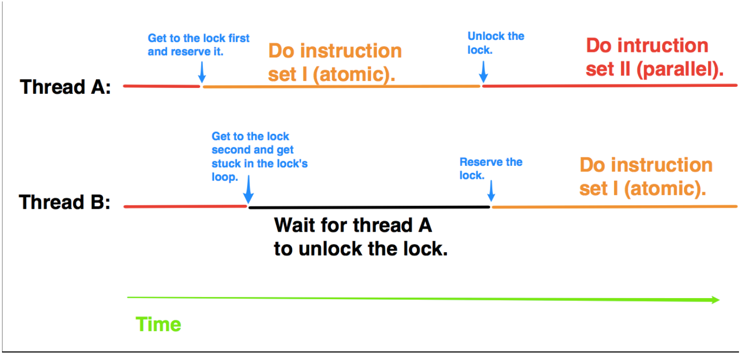
```
__global__ void someKernel(void){
  Lock myLock;

  // some parallel code

  mylock.lock();
  // some sequential code
  mylock.unlock();

  // some parallel code
}
```

The concept


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

Lock.h

```
struct Lock {
  int *mutex;

  Lock(){
    int state = 0;

    cudaMalloc((void**) &mutex, sizeof(int));
    cudaMemcpy(mutex, &state, sizeof(int), cudaMemcpyHostToDevice);
  }

  ~Lock(){
    cudaFree(mutex);
  }

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

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

};
```

A closer look at the lock function

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

Example: counting the number of blocks

- Compare the two kernels:

```
__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();
  }
}
```

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 );
```

blockCounter.cu

```
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);
```

blockCounter.cu

```
  //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);
}
```

blockCounter.cu

```
> nvcc blockCounter.cu -arch sm_20 -o blockCounter
> ./blockCounter
blockCounterUnlocked <<< 512, 1024 >>> () counted 47 blocks in
    0.057920 ms.
blockCounter1 <<< 512, 1024 >>> () counted 512 blocks in 0.636064 ms.
```

blockCounter.cu pauses indefinitely with this kernel

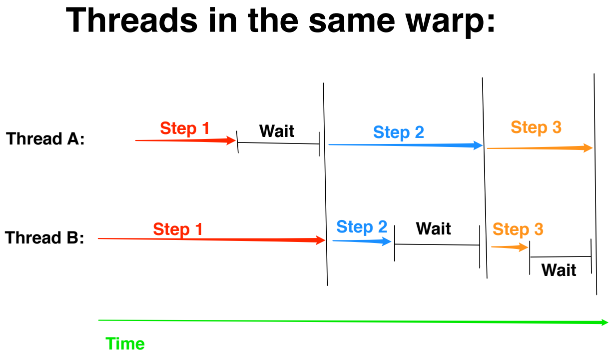
```
__global__ void blockCounter2(Lock lock, int *
    nblocks){
  lock.lock();
  if(threadIdx.x == 0){
    *nblocks = *nblocks + 1;
  }
  lock.unlock();
}
```

- Why? warps.

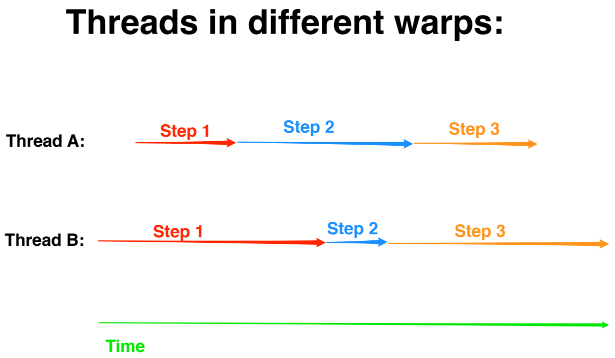
Outline

Warps

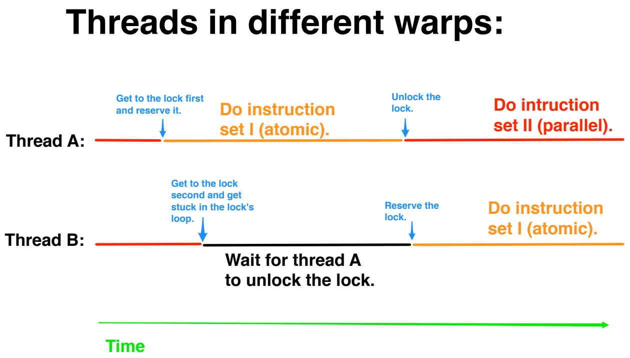
- `Warp`: a group of 32 threads in the same block that execute in lockstep.
  - That is, they synchronize after every step (as if `__syncthreads()` is called as often as possible).
  - All blocks are partitioned into warps.



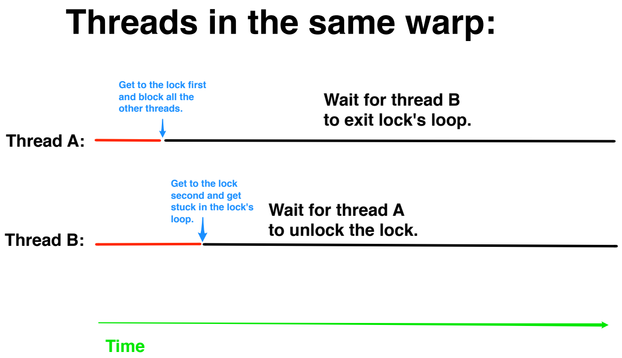
Warps


Threads in different warps:
Thread A:
Step 1
Step 2
Step 3
Thread B:
Step 1
Step 2
Step 3
Time

Warps and locks



Warps and locks



Outline

Race conditions

Brute force fixes: atomics, locks, and mutex

Warps

Resources

- Texts:
  1. J. Sanders and E. Kandrot. CUDA by Example. Addison-Wesley, 2010.
  2. D. Kirk, W.H. Wen-mei, and W. Hwu. *Programming massively parallel processors: a hands-on approach.* Morgan Kaufmann, 2010.
- Code from today:
  - race_condition.cu
  - race_condition_fixed.cu
  - blockCounter.cu
- Dot product with atomic operations:
  - dot_product_atomic_builtin.cu
  - dot_product_atomic_lock.cu

That's all for today.

- Series materials are available at http://will-landau.com/gpu.