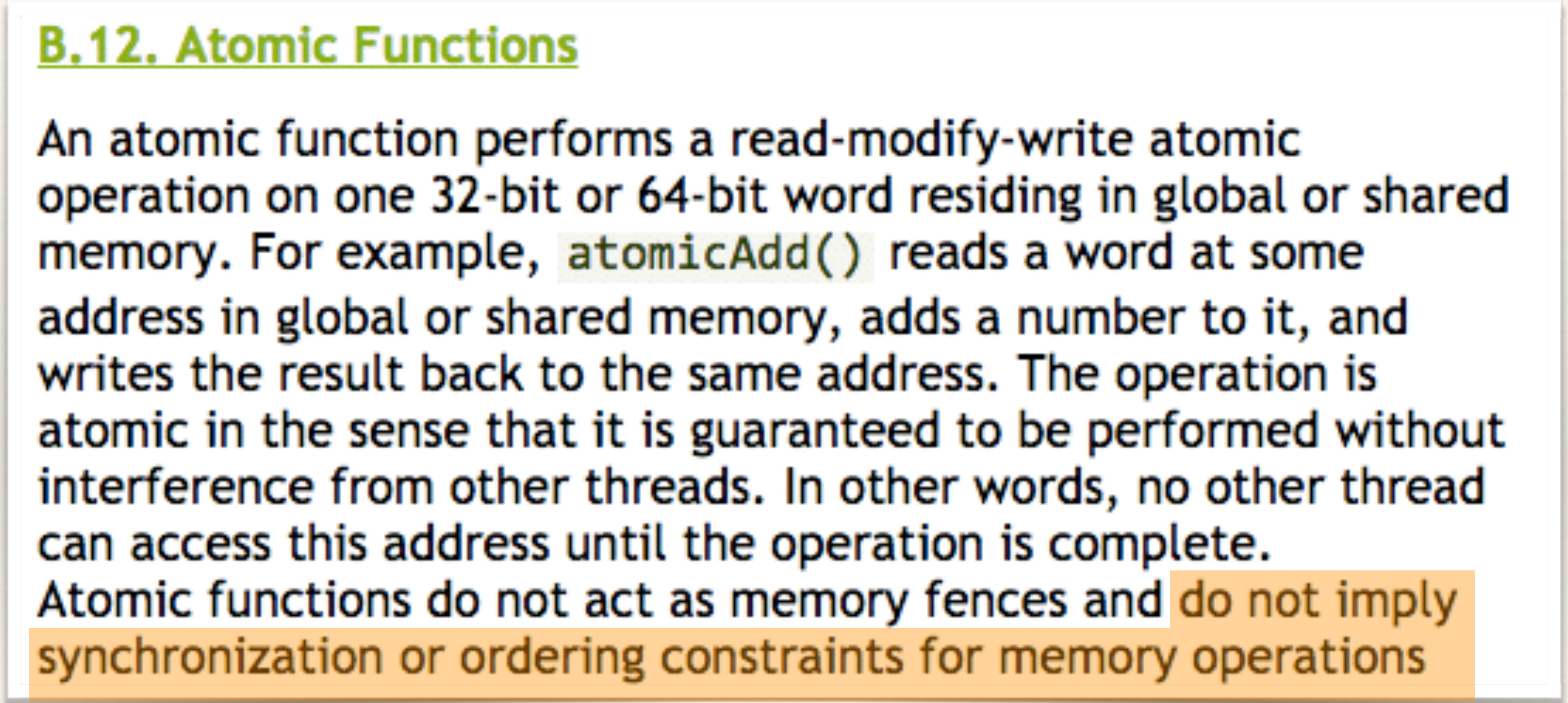
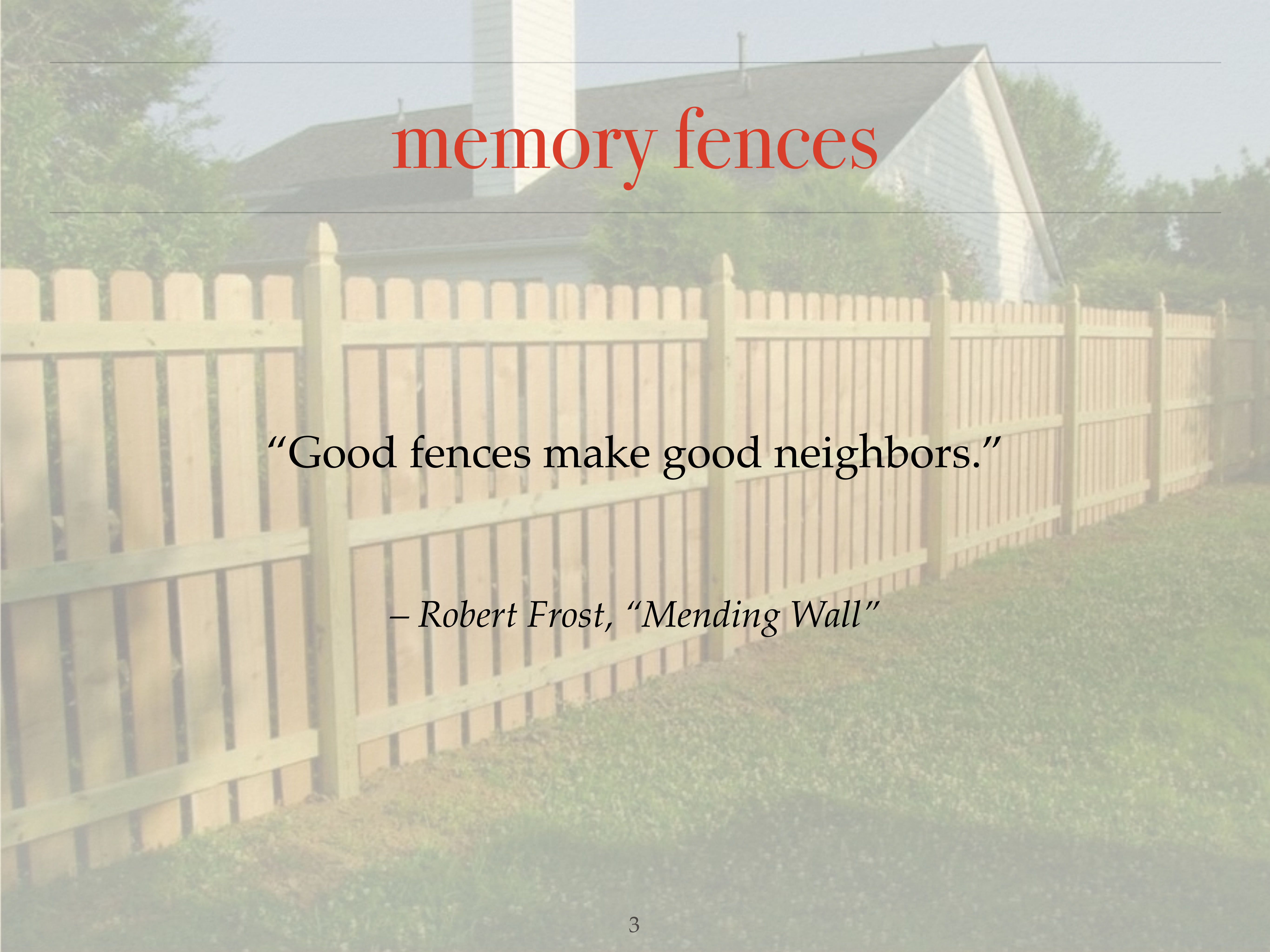
CUDA Synchronization

atomics



B.12. Atomic Functions

An atomic function performs a read-modify-write atomic operation on one 32-bit or 64-bit word residing in global or shared memory. For example, `atomicAdd()` reads a word at some address in global or shared memory, adds a number to it, and writes the result back to the same address. The operation is atomic in the sense that it is guaranteed to be performed without interference from other threads. In other words, no other thread can access this address until the operation is complete.
Atomic functions do not act as memory fences and do not imply synchronization or ordering constraints for memory operations

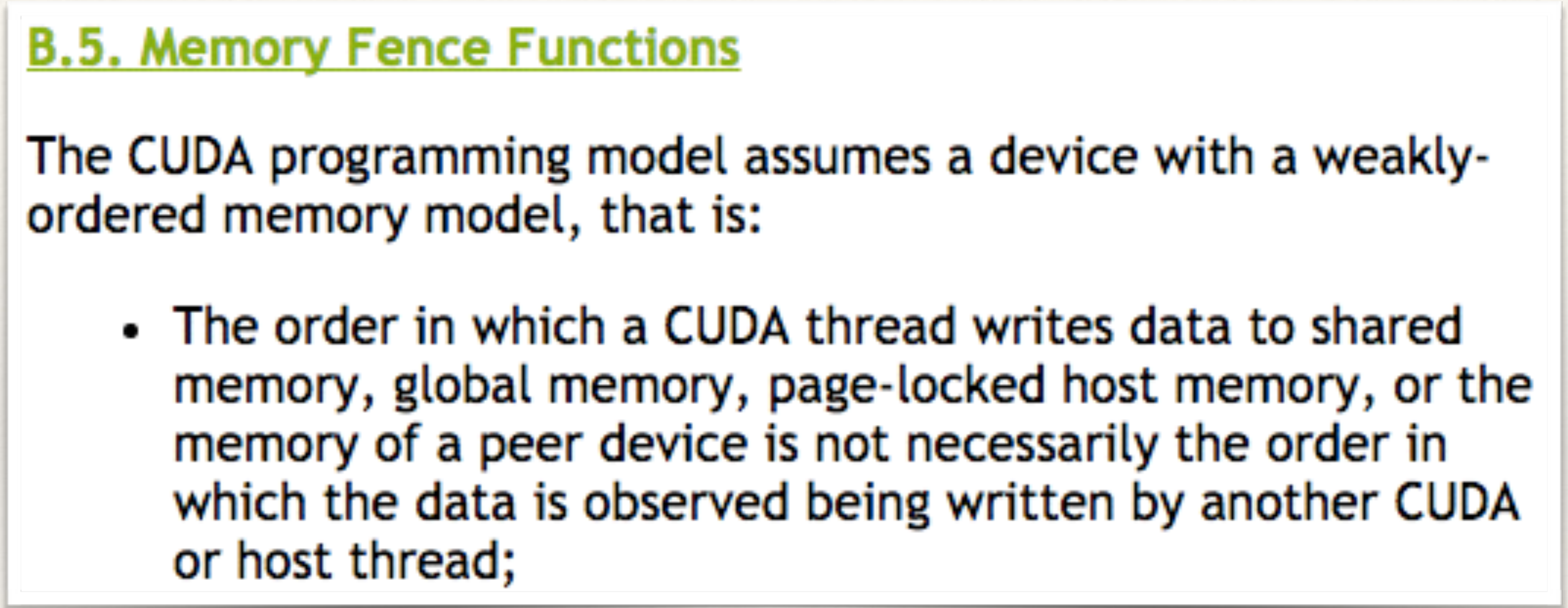


memory fences

“Good fences make good neighbors.”

– Robert Frost, “Mending Wall”

without fences



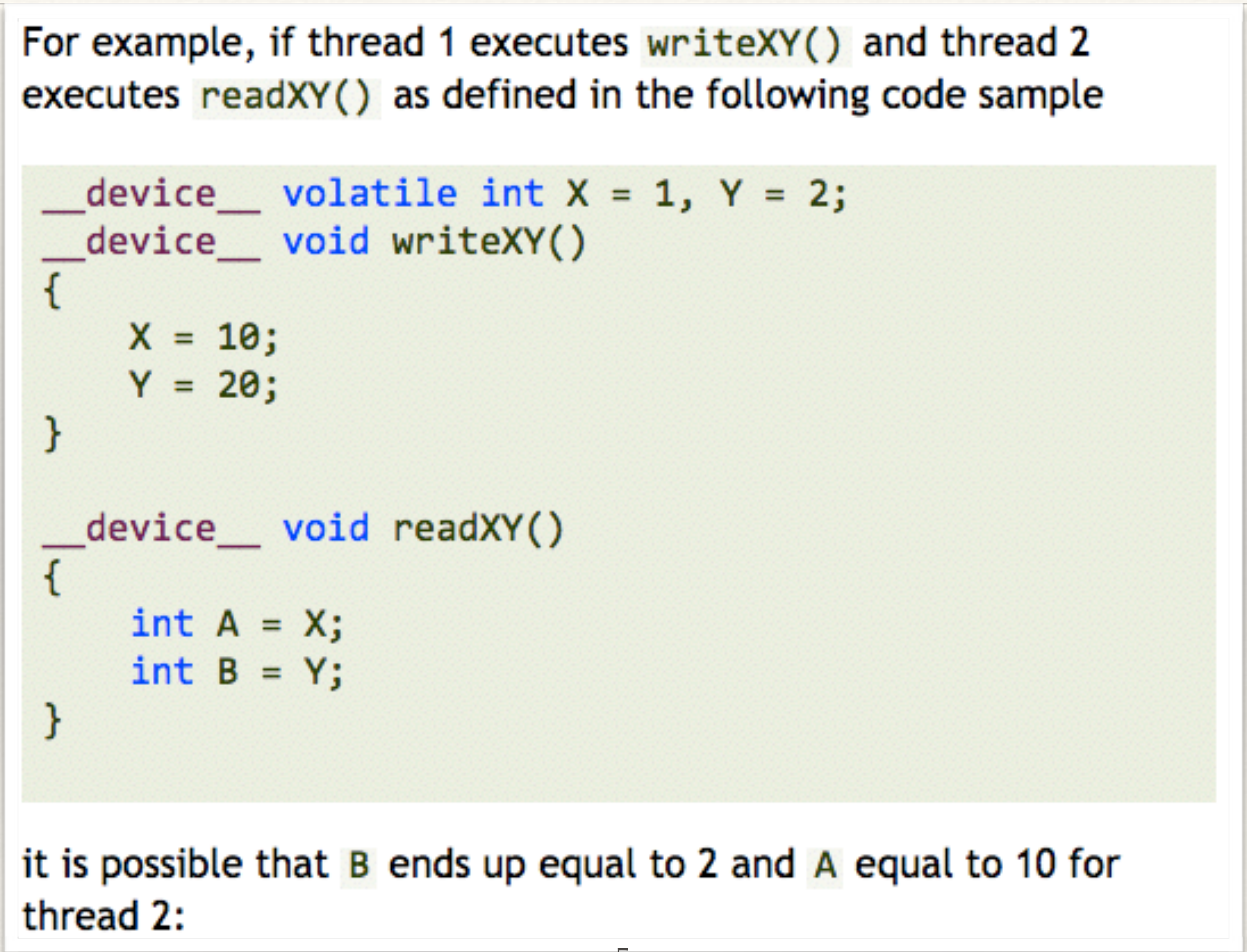
B.5. Memory Fence Functions

The CUDA programming model assumes a device with a weakly-ordered memory model, that is:

- The order in which a CUDA thread writes data to shared memory, global memory, page-locked host memory, or the memory of a peer device is not necessarily the order in which the data is observed being written by another CUDA or host thread;

same (lack of) guarantees for reads

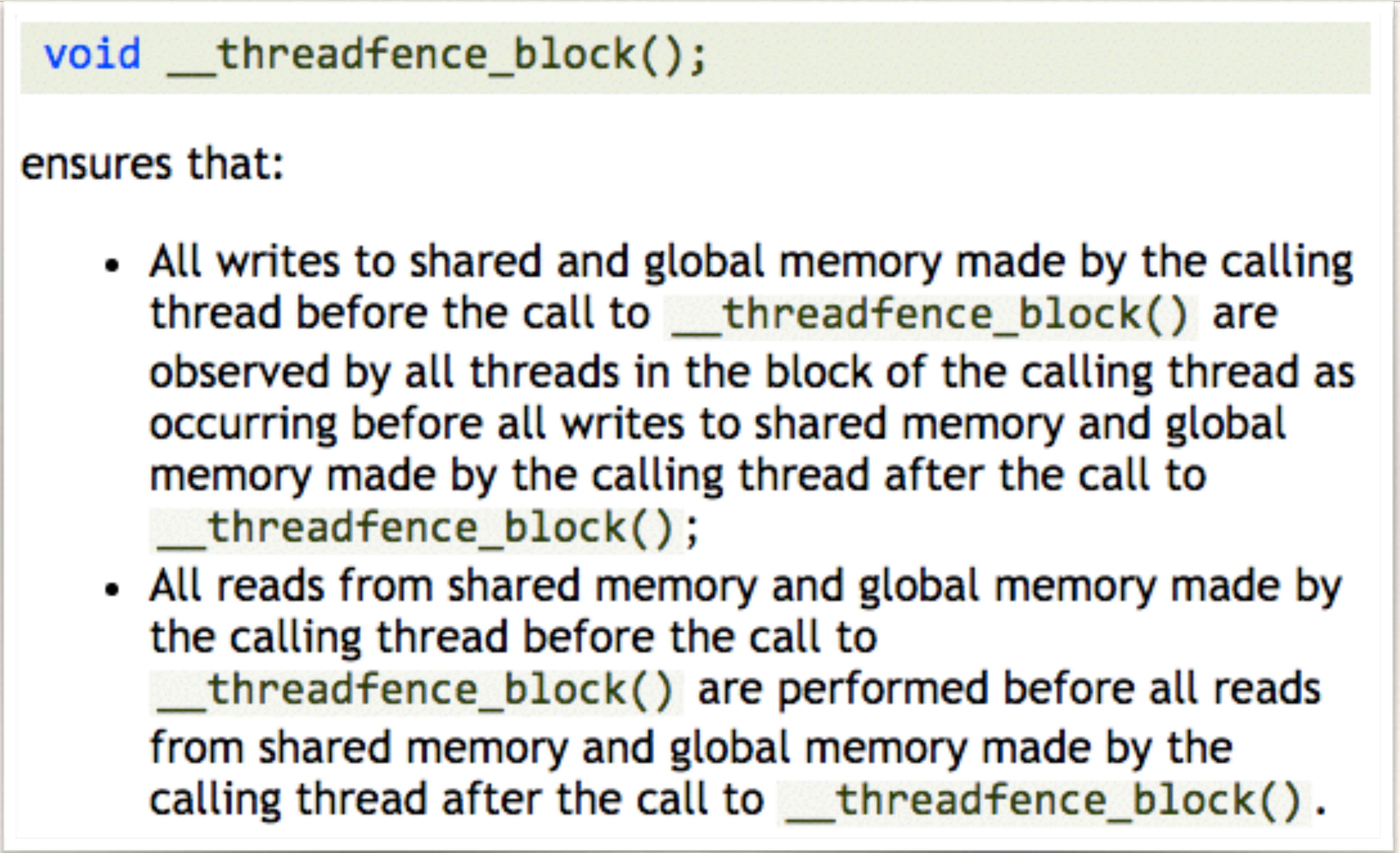
ORLY?


For example, if thread 1 executes `writeXY()` and thread 2 executes `readXY()` as defined in the following code sample

```
__device__ volatile int X = 1, Y = 2;
__device__ void writeXY()
{
    X = 10;
    Y = 20;
}

__device__ void readXY()
{
    int A = X;
    int B = Y;
}
```

it is possible that `B` ends up equal to 2 and `A` equal to 10 for thread 2:

__threadfence_block

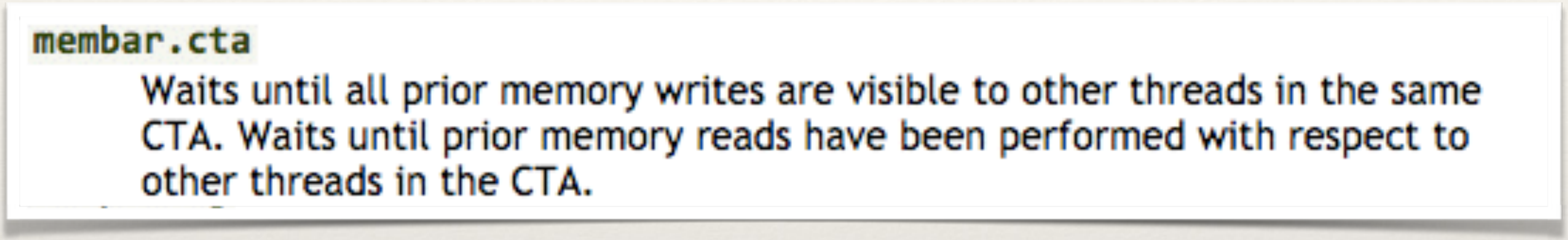


```
void __threadfence_block();
```

ensures that:

- All writes to shared and global memory made by the calling thread before the call to `__threadfence_block()` are observed by all threads in the block of the calling thread as occurring before all writes to shared memory and global memory made by the calling thread after the call to `__threadfence_block()`;
- All reads from shared memory and global memory made by the calling thread before the call to `__threadfence_block()` are performed before all reads from shared memory and global memory made by the calling thread after the call to `__threadfence_block()`.

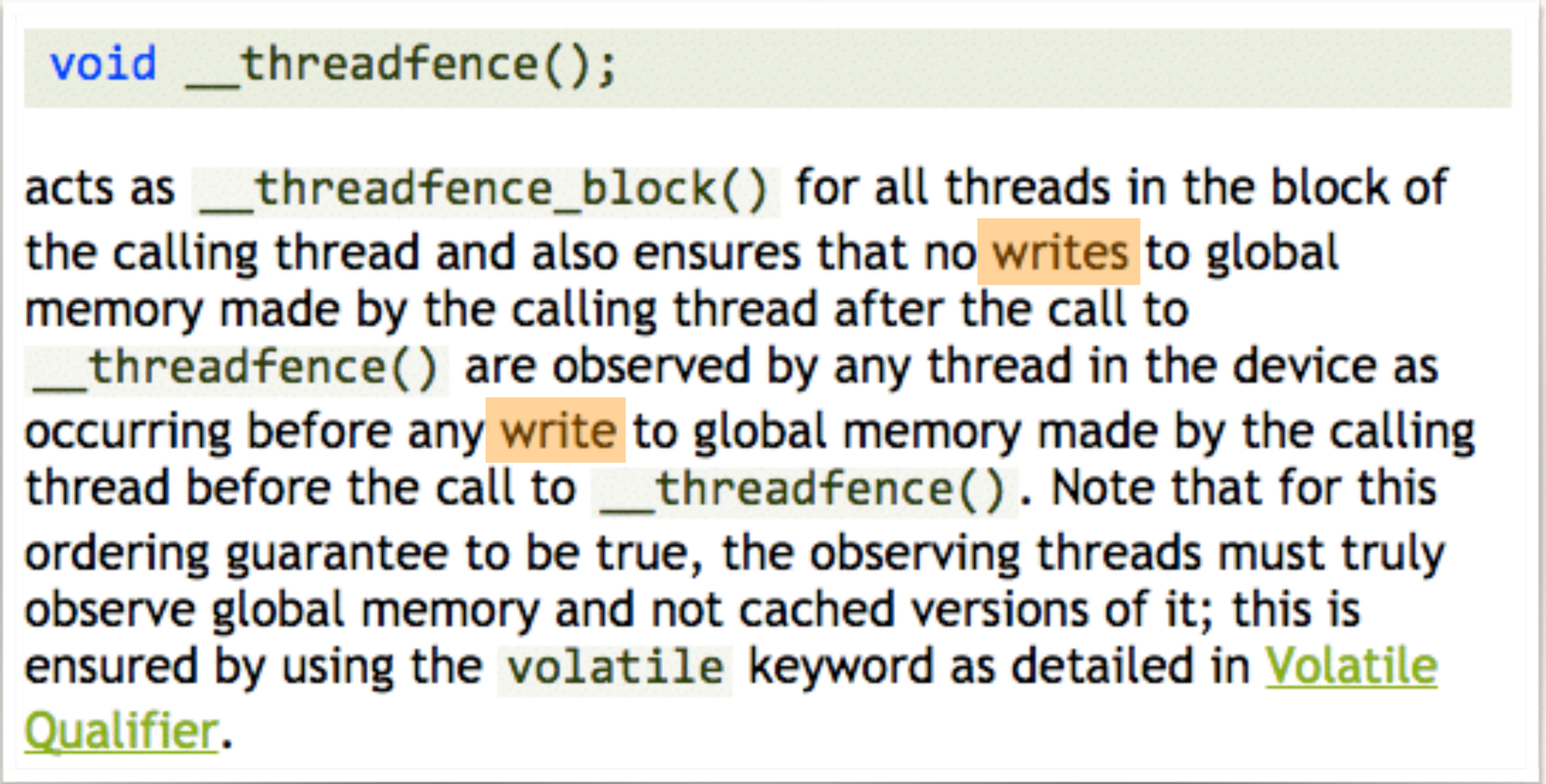
PTX membar.cta



`membar.cta`

Waits until all prior memory writes are visible to other threads in the same CTA. Waits until prior memory reads have been performed with respect to other threads in the CTA.

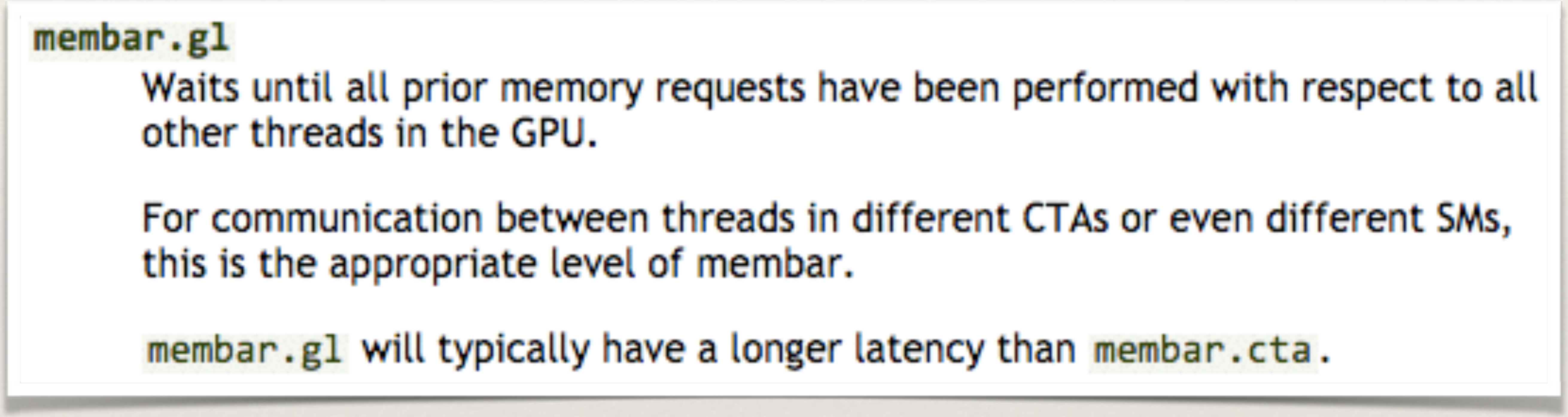
__threadfence



```
void __threadfence();
```

acts as `__threadfence_block()` for all threads in the block of the calling thread and also ensures that no writes to global memory made by the calling thread after the call to `__threadfence()` are observed by any thread in the device as occurring before any write to global memory made by the calling thread before the call to `__threadfence()`. Note that for this ordering guarantee to be true, the observing threads must truly observe global memory and not cached versions of it; this is ensured by using the `volatile` keyword as detailed in Volatile Qualifier.

PTX membar.gl



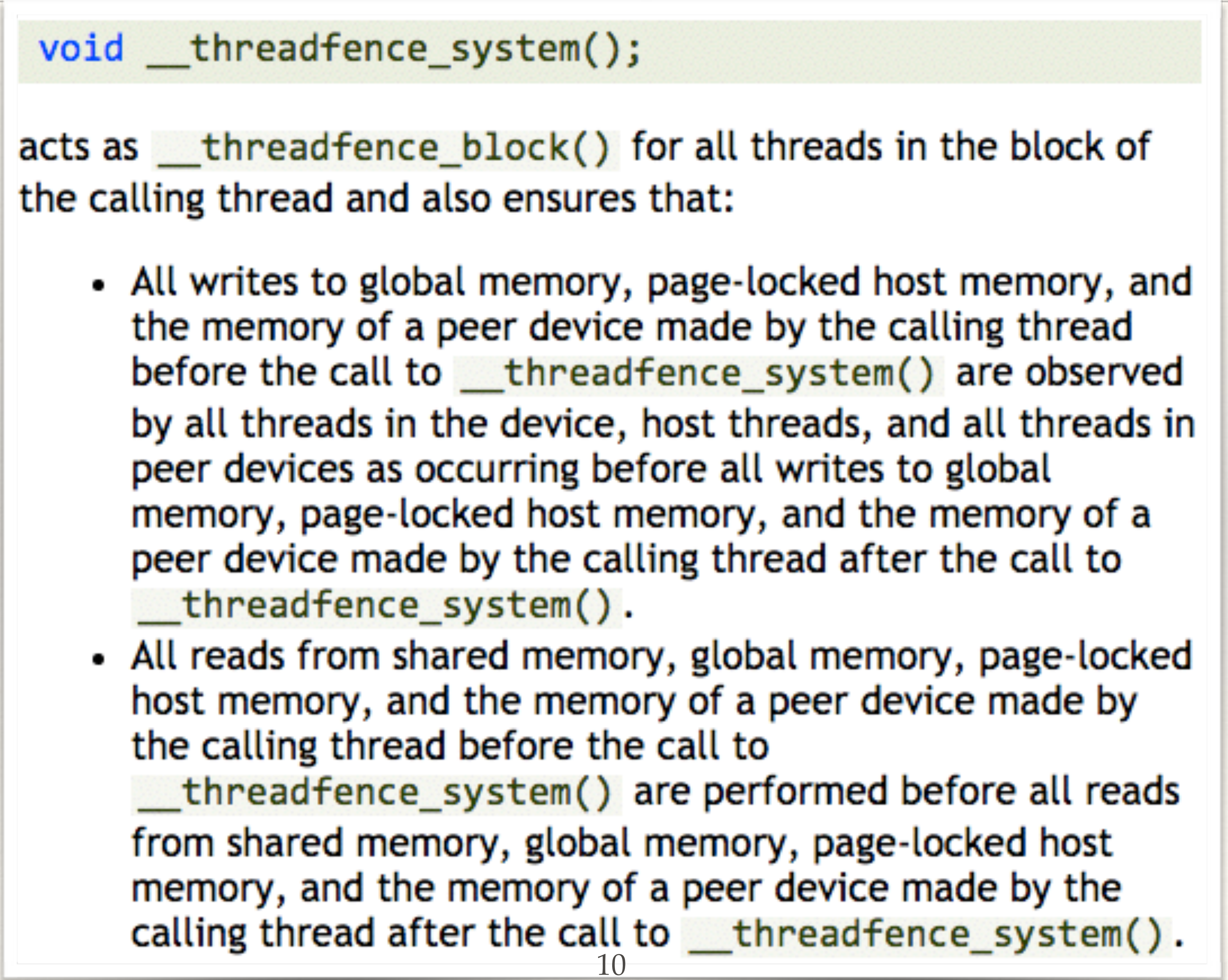
`membar.gl`

Waits until all prior memory requests have been performed with respect to all other threads in the GPU.

For communication between threads in different CTAs or even different SMs, this is the appropriate level of membar.

`membar.gl` will typically have a longer latency than `membar.cta`.

__threadfence_system

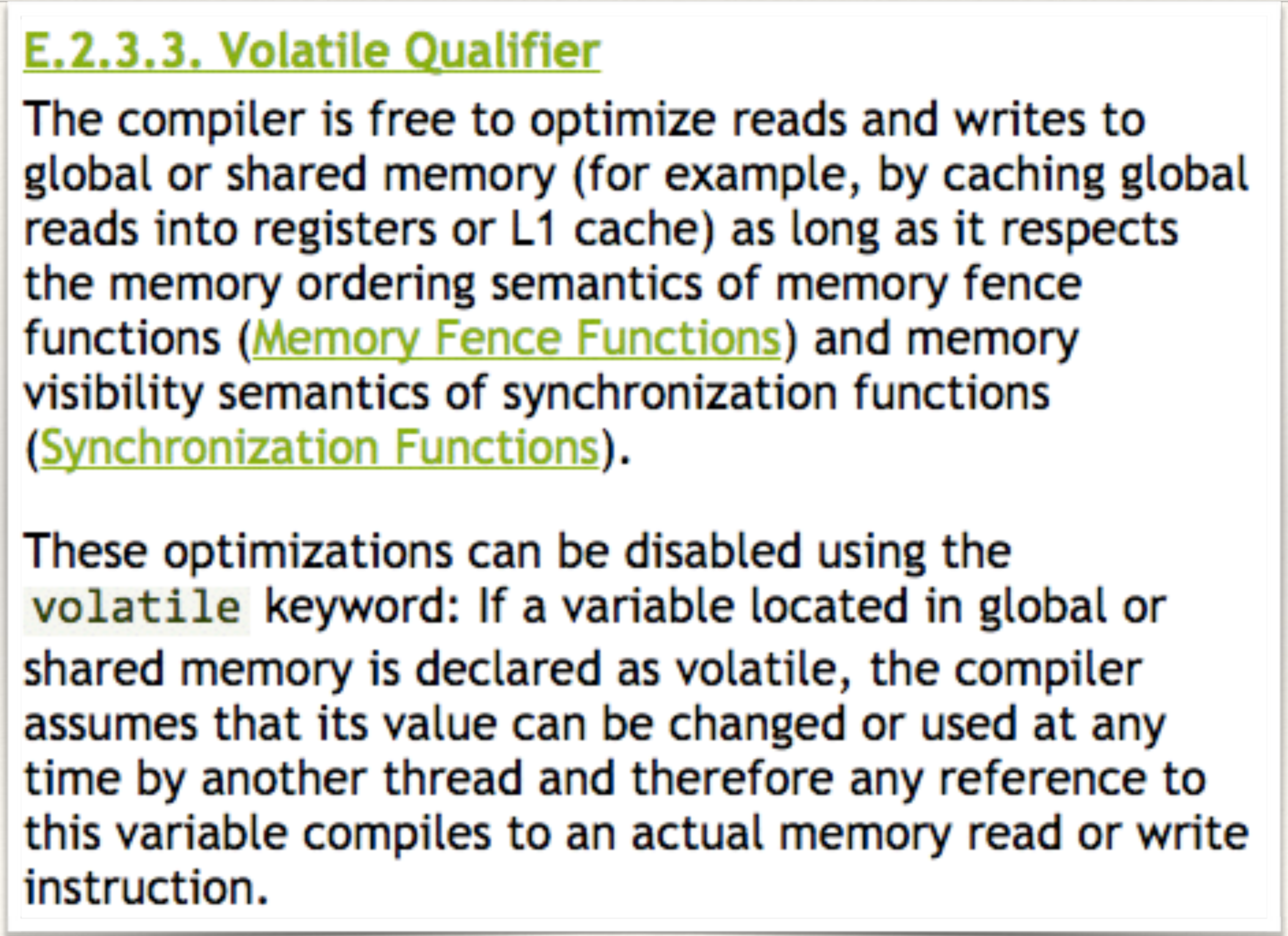


```
void __threadfence_system();
```

acts as `__threadfence_block()` for all threads in the block of the calling thread and also ensures that:

- All writes to global memory, page-locked host memory, and the memory of a peer device made by the calling thread before the call to `__threadfence_system()` are observed by all threads in the device, host threads, and all threads in peer devices as occurring before all writes to global memory, page-locked host memory, and the memory of a peer device made by the calling thread after the call to `__threadfence_system()`.
- All reads from shared memory, global memory, page-locked host memory, and the memory of a peer device made by the calling thread before the call to `__threadfence_system()` are performed before all reads from shared memory, global memory, page-locked host memory, and the memory of a peer device made by the calling thread after the call to `__threadfence_system()`.

volatile

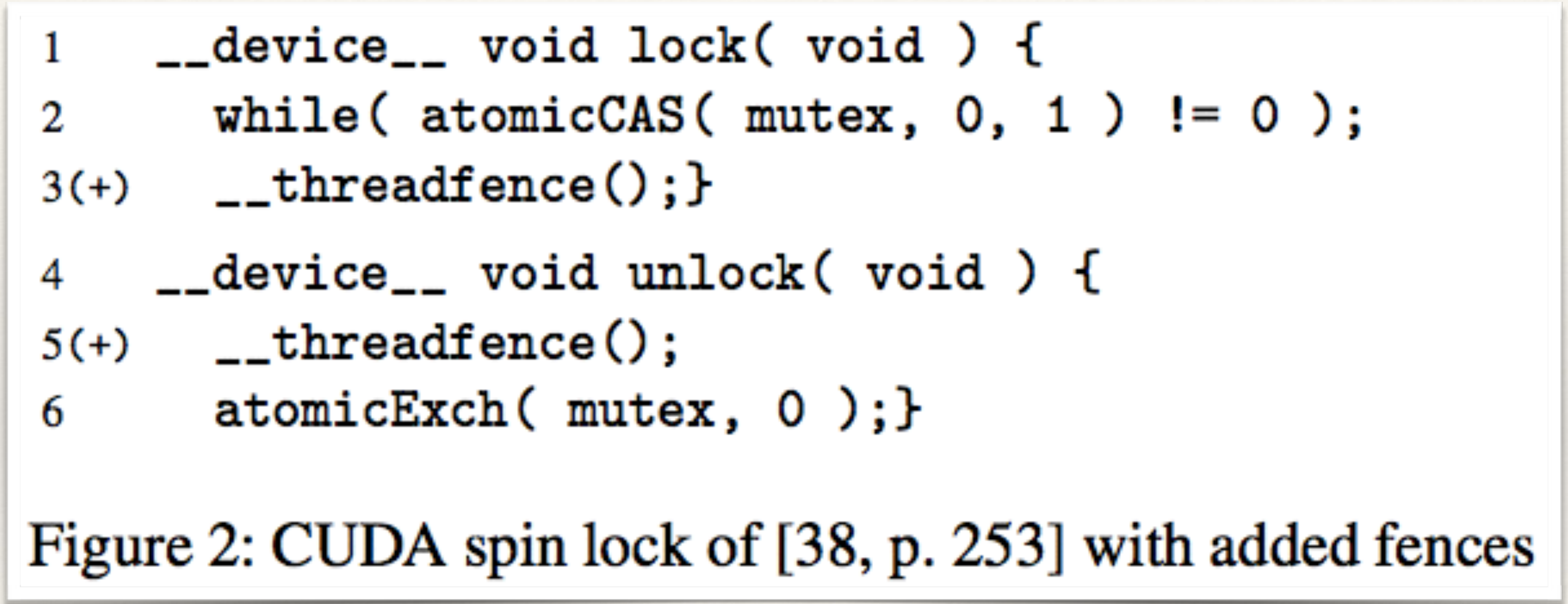


E.2.3.3. Volatile Qualifier

The compiler is free to optimize reads and writes to global or shared memory (for example, by caching global reads into registers or L1 cache) as long as it respects the memory ordering semantics of memory fence functions (Memory Fence Functions) and memory visibility semantics of synchronization functions (Synchronization Functions).

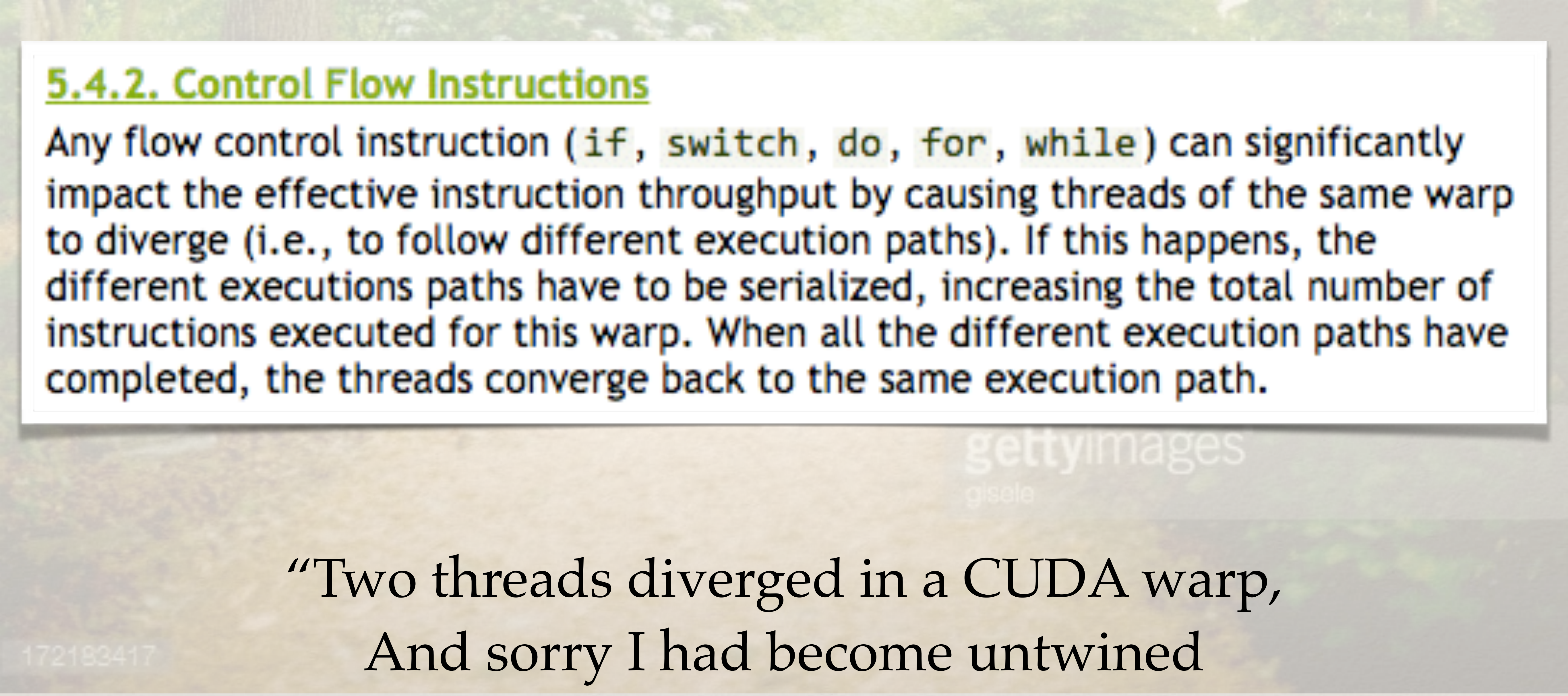
These optimizations can be disabled using the `volatile` keyword: If a variable located in global or shared memory is declared as volatile, the compiler assumes that its value can be changed or used at any time by another thread and therefore any reference to this variable compiles to an actual memory read or write instruction.

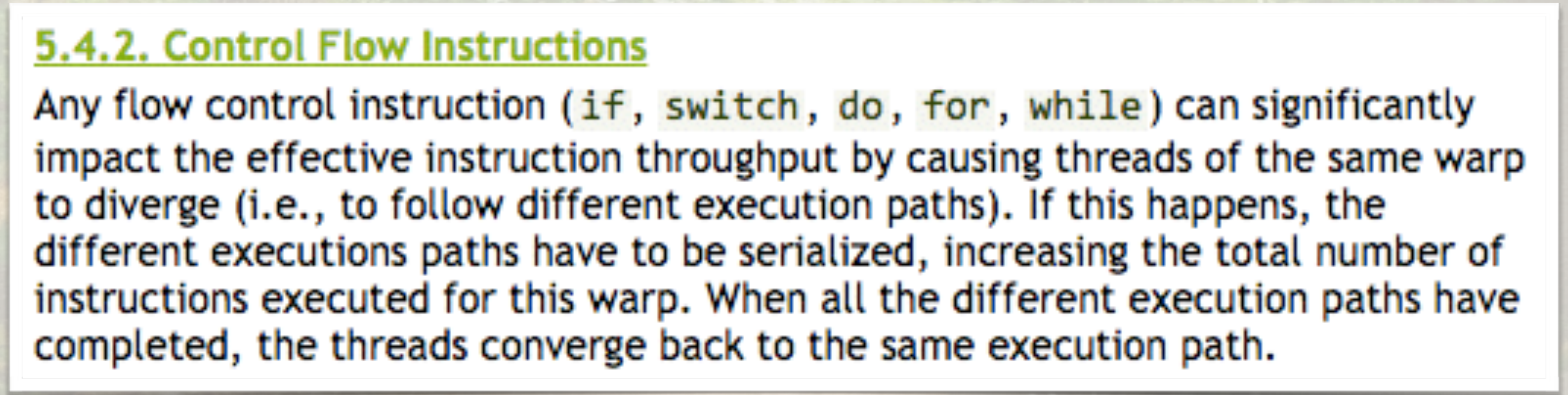
CUDA spinlock?



```
1      __device__ void lock( void ) {
2        while( atomicCAS( mutex, 0, 1 ) != 0 );
3(+)     __threadfence();}
4      __device__ void unlock( void ) {
5(+)     __threadfence();
6        atomicExch( mutex, 0 );}
```

Figure 2: CUDA spin lock of [38, p. 253] with added fences





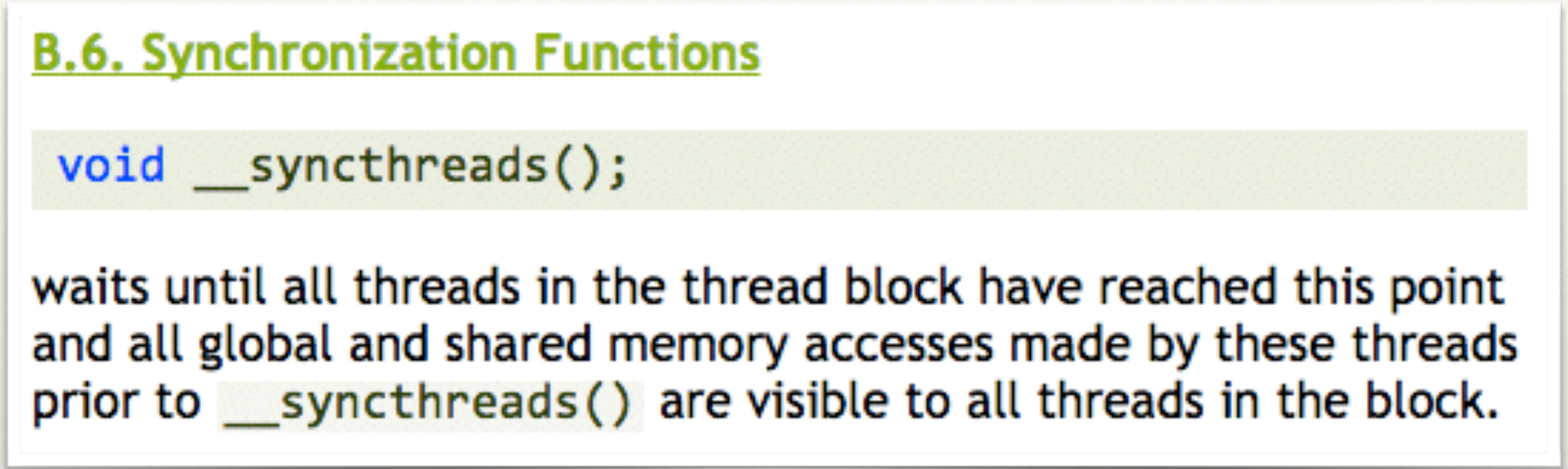
5.4.2. Control Flow Instructions

Any flow control instruction (`if`, `switch`, `do`, `for`, `while`) can significantly impact the effective instruction throughput by causing threads of the same warp to diverge (i.e., to follow different execution paths). If this happens, the different executions paths have to be serialized, increasing the total number of instructions executed for this warp. When all the different execution paths have completed, the threads converge back to the same execution path.

"Two threads diverged in a CUDA warp,
And sorry I had become untwined
from my PC by a branch so sharp,
I had to ask Nvidia Corp:
the order of paths was undefined."

– *"Robert Frost", The Thread Not Taken*

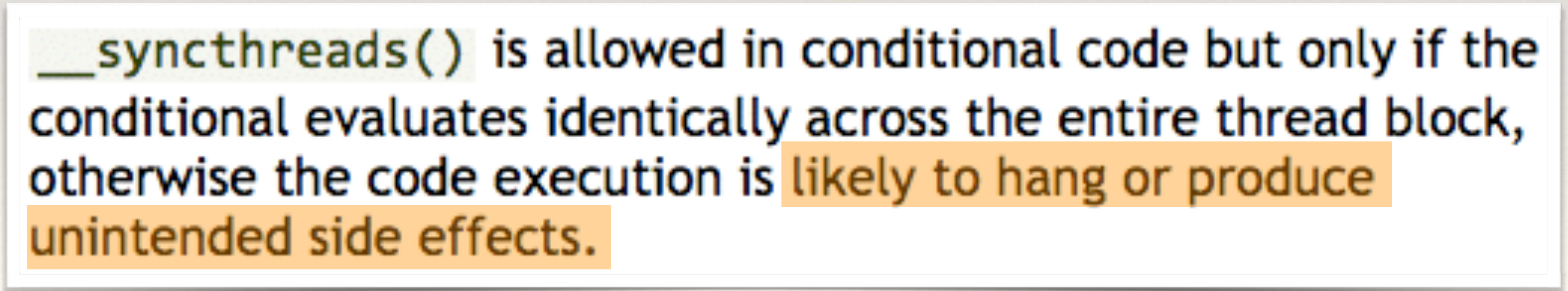
__syncthreads()



B.6. Synchronization Functions

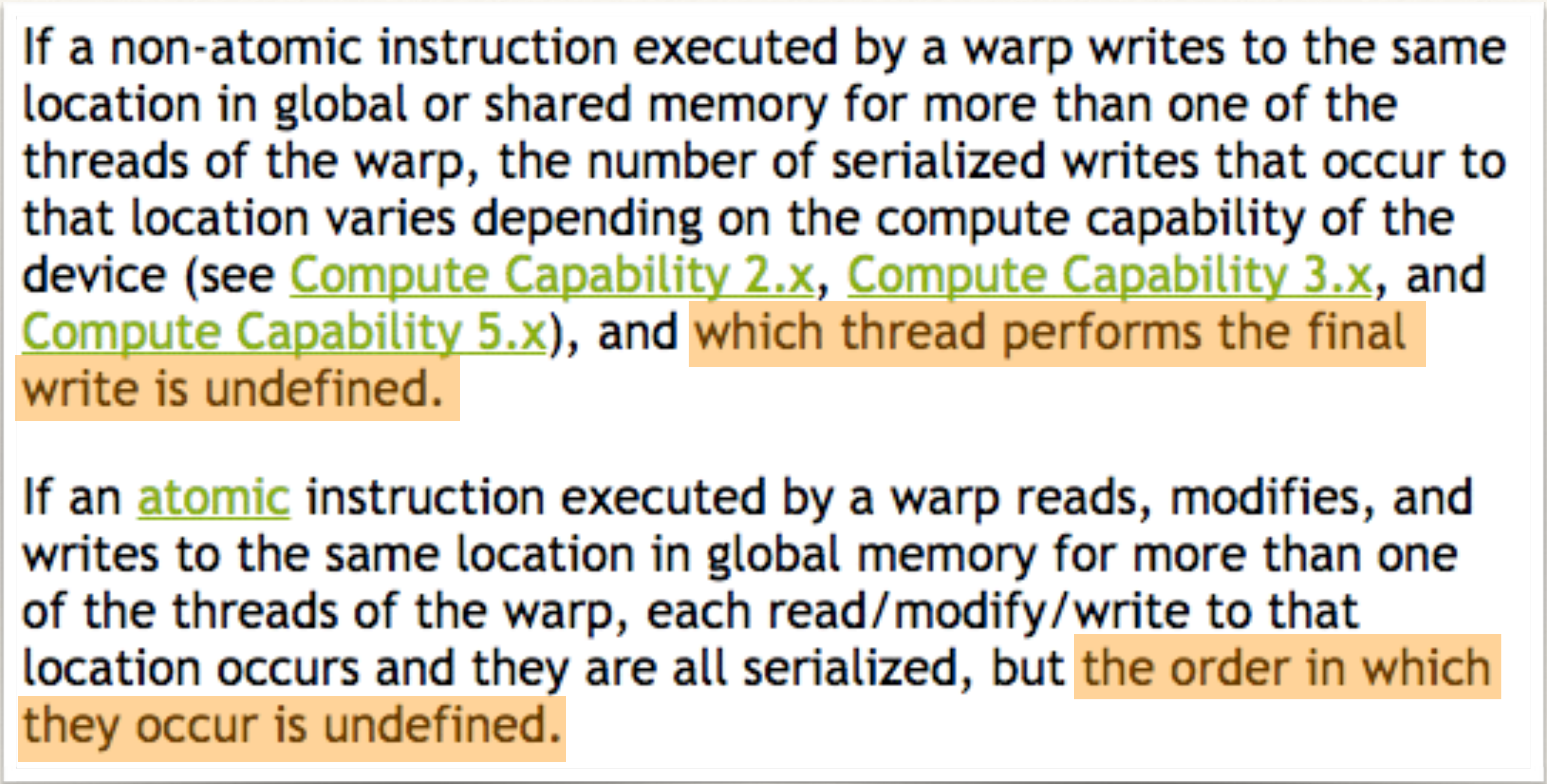
```
void __syncthreads();
```

waits until all threads in the thread block have reached this point and all global and shared memory accesses made by these threads prior to `__syncthreads()` are visible to all threads in the block.



`__syncthreads()` is allowed in conditional code but only if the conditional evaluates identically across the entire thread block, otherwise the code execution is likely to hang or produce unintended side effects.

intra-warp synchronization



If a non-atomic instruction executed by a warp writes to the same location in global or shared memory for more than one of the threads of the warp, the number of serialized writes that occur to that location varies depending on the compute capability of the device (see Compute Capability 2.x, Compute Capability 3.x, and Compute Capability 5.x), and which thread performs the final write is undefined.

If an atomic instruction executed by a warp reads, modifies, and writes to the same location in global memory for more than one of the threads of the warp, each read/modify/write to that location occurs and they are all serialized, but the order in which they occur is undefined.

<spinlock PTX demo>

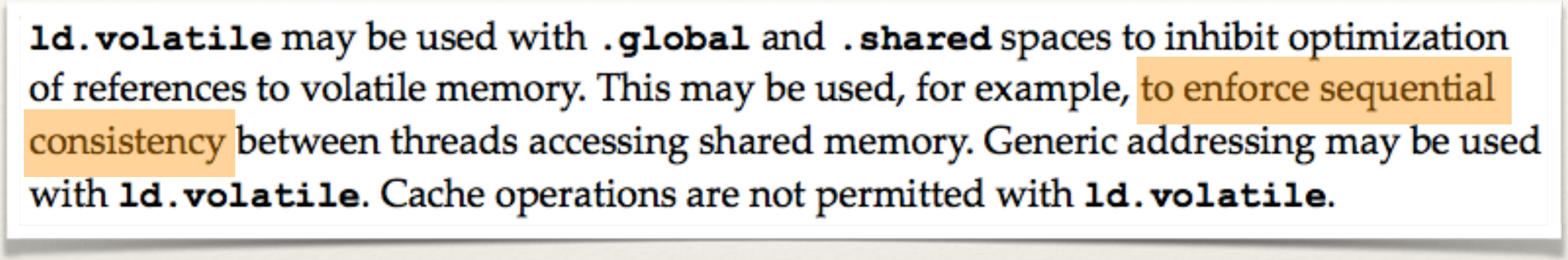
PTX

- virtual ISA for Nvidia GPUs
- RISC-like ISA, load-store, 3-operand
- destination register is on the **left**

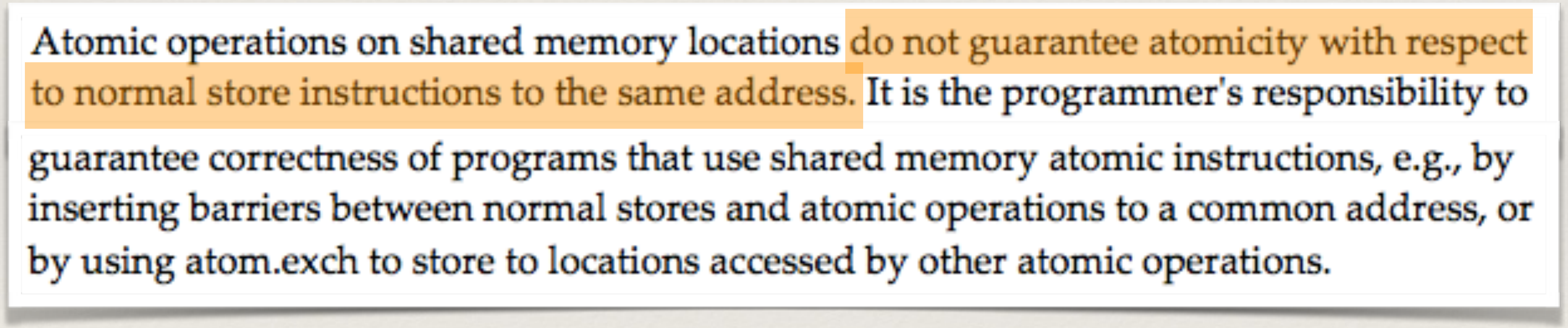
PTX load/store caching

qualifier	meaning	load	store
.ca	cache at all levels	default	
.wb	write back caching		default
.cg	cache at L2 (global cache)	yes	yes
.cs	streaming (mark as LRU)	yes	yes
.lu	last use (read & invalidate)	yes	

PTX tidbits



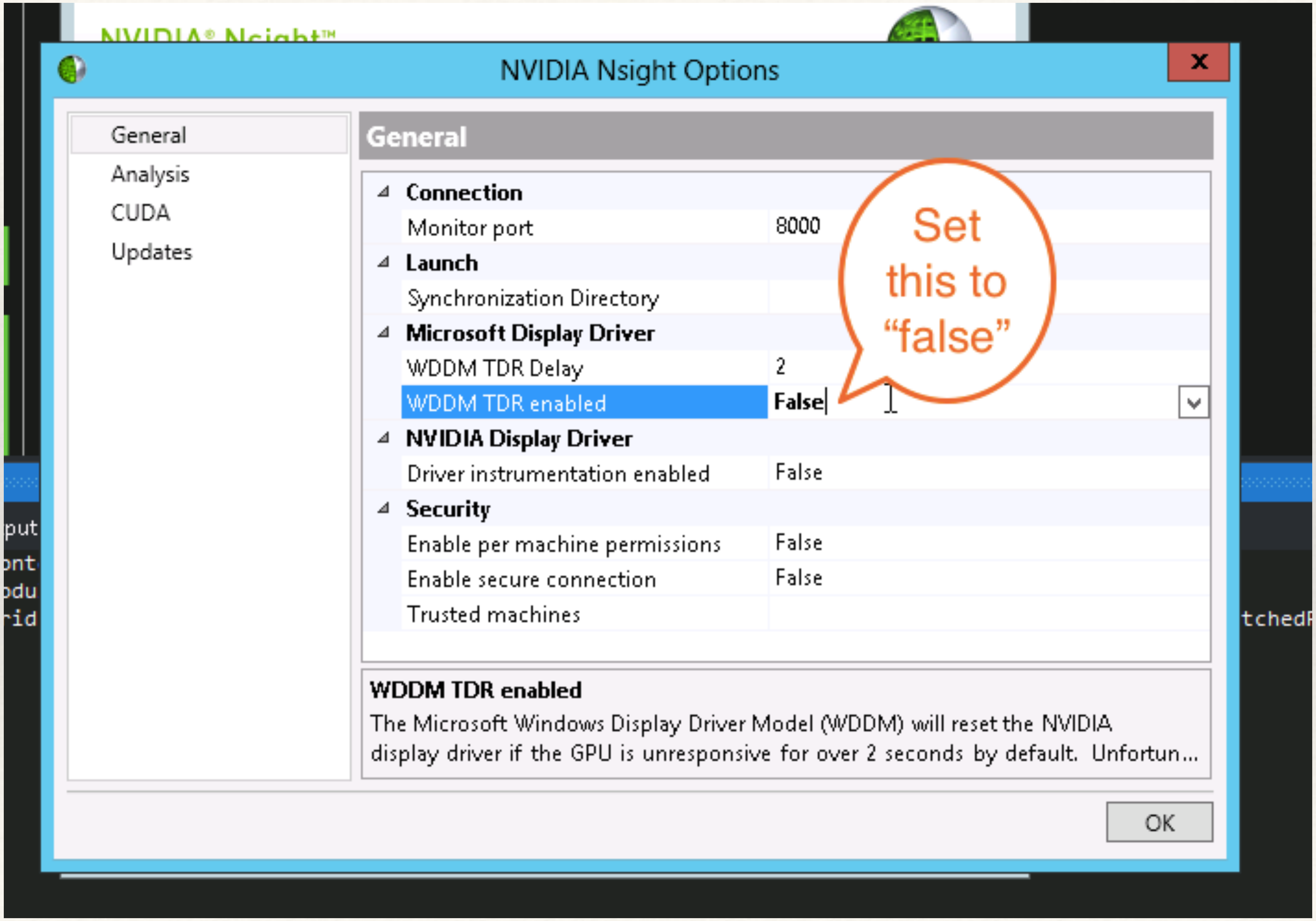
ld.volatile may be used with **.global** and **.shared** spaces to inhibit optimization of references to volatile memory. This may be used, for example, to enforce sequential consistency between threads accessing shared memory. Generic addressing may be used with **ld.volatile**. Cache operations are not permitted with **ld.volatile**.



Atomic operations on shared memory locations do not guarantee atomicity with respect to normal store instructions to the same address. It is the programmer's responsibility to guarantee correctness of programs that use shared memory atomic instructions, e.g., by inserting barriers between normal stores and atomic operations to a common address, or by using atom.exch to store to locations accessed by other atomic operations.

Homework 2

CUDA Kernel Timeout == good



General

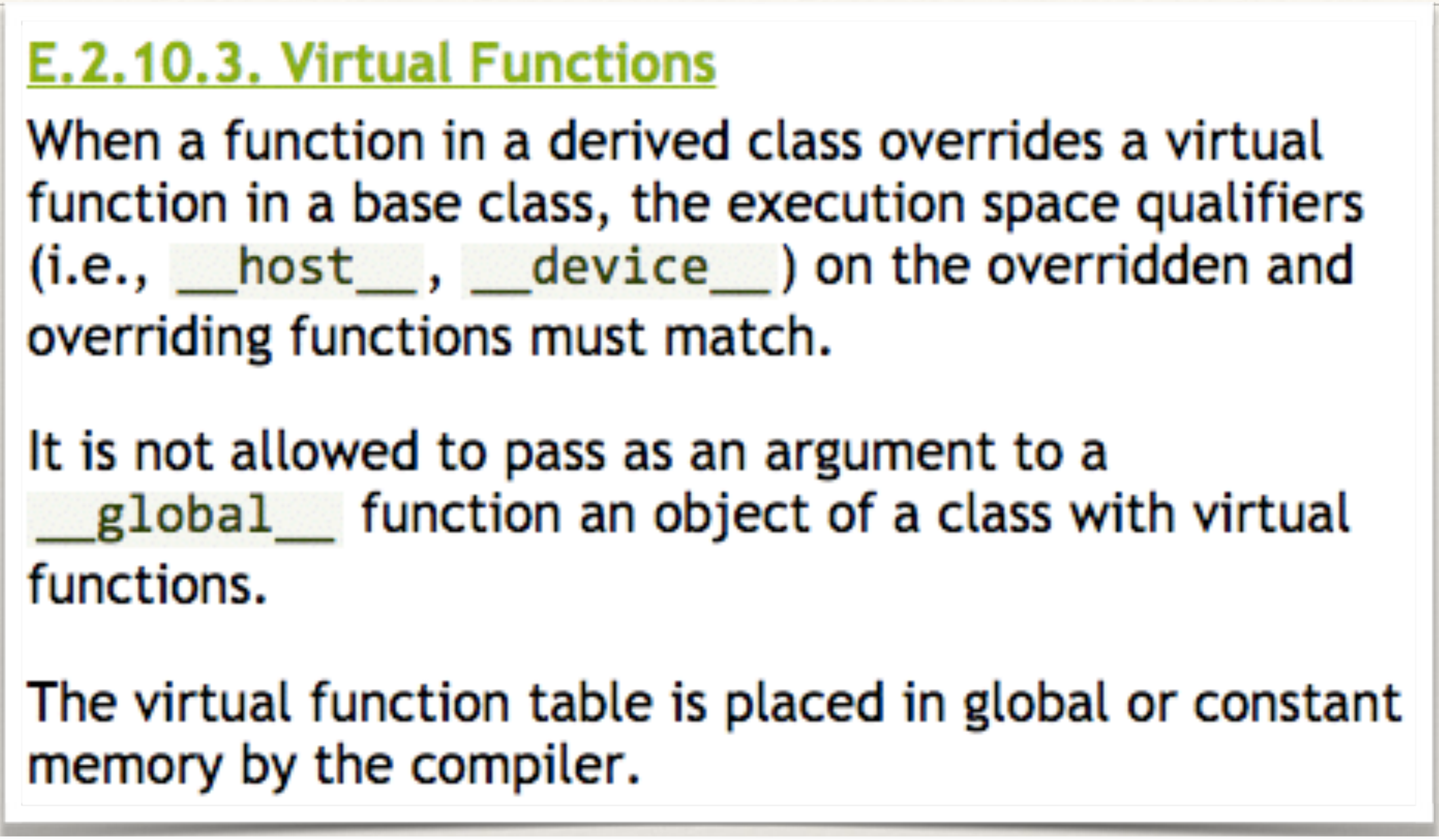
Connection	
Monitor port	8000
Launch	
Synchronization Directory	
Microsoft Display Driver	
WDDM TDR Delay	2
WDDM TDR enabled	False
NVIDIA Display Driver	
Driver instrumentation enabled	False
Security	
Enable per machine permissions	False
Enable secure connection	False
Trusted machines	

WDDM TDR enabled
The Microsoft Windows Display Driver Model (WDDM) will reset the NVIDIA display driver if the GPU is unresponsive for over 2 seconds by default. Unfortun...

__managed__

```
__device__ __managed__ int d_counter = 0;

void main() {
  d_counter = 10;

  myKernel<<<8,16>>>();

  cudaStatus = cudaDeviceSynchronize();
  checkCudaErrors(cudaStatus);

  printf("%d", d_counter);
}
```

C++ virtual functions

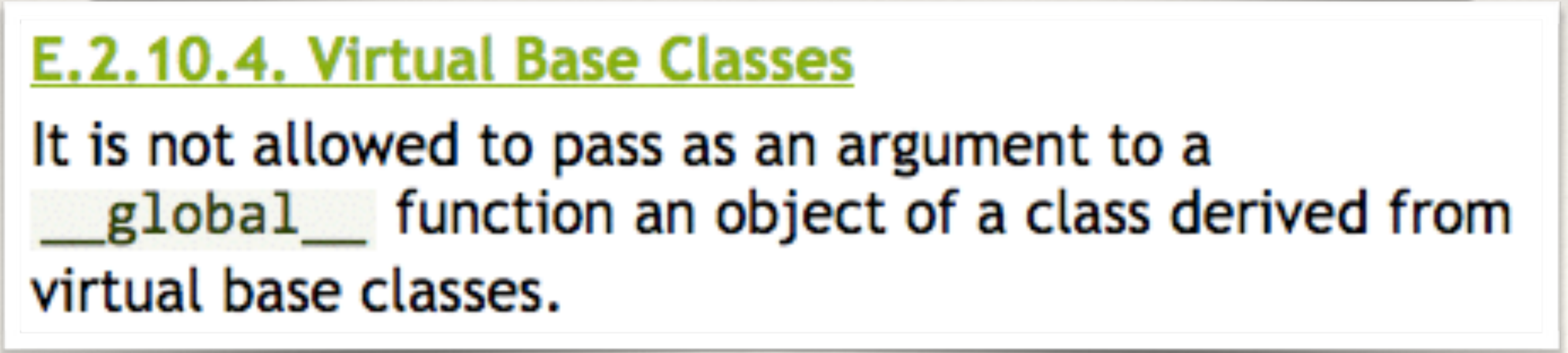


E.2.10.3. Virtual Functions

When a function in a derived class overrides a virtual function in a base class, the execution space qualifiers (i.e., `__host__`, `__device__`) on the overridden and overriding functions must match.

It is not allowed to pass as an argument to a `__global__` function an object of a class with virtual functions.

The virtual function table is placed in global or constant memory by the compiler.



E.2.10.4. Virtual Base Classes

It is not allowed to pass as an argument to a `__global__` function an object of a class derived from virtual base classes.