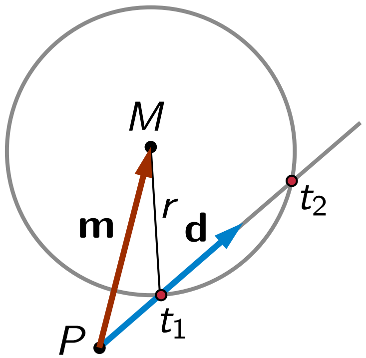
- Access of constant memory on the device (i.e., from a kernel) works just like with any globally declared variable
- Example:

```
__constant__ Sphere c_spheres[MAX_NUM_SPHERES];

__device__
bool intersect( const Ray & ray, int s, Hit * hit )
{
   Vec3 m( c_spheres[s].center - ray.orig );
   float q = m*m - c_spheres[s].radius*c_spheres[s].radius;
   float p = ...
   solve_quadratic( p, q, *t1, *t2 );
   ...
}
```



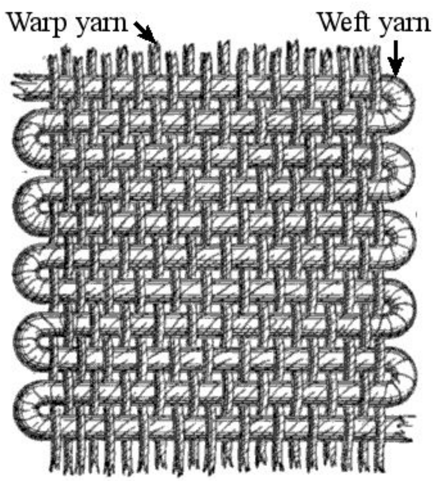
$$(t\cdot\mathbf{d} - \mathbf{m})^2 = r^2 \;\Rightarrow\; t^2 - 2t\cdot\mathbf{md} + \mathbf{m}^2 - r^2 = 0$$

Some Considerations on Constant Memory

- Size of constant memory on the GPU is fairly limited (~48 KB)
  - Check **`cudaDeviceProp`**
- Reads from constant memory *can* be very fast:
  - "Nearby" threads accessing the same constant memory location incur only a single read operation (saves bandwidth by up to factor 16!)
  - Constant memory is cached (i.e., consecutive reads will not incur additional traffic)
- Caveats:
  - If "nearby" threads read from different memory locations → traffic jam!

New Terminology

- "Nearby threads" = all threads within a warp
- Warp := 32 threads next to each other
  - Each block's set of threads is partitioned into *warps*
  - All threads within a warp are executed on a single streaming multiprocessor (SM) in lockstep
- If all threads in a warp read from the same memory location → one read instruction by SM
- If all threads in a warp read from random memory locations → 32 different read instructions by SM, one after another!
- In our raytracing example, everything is fine (if there is no bug ☺)

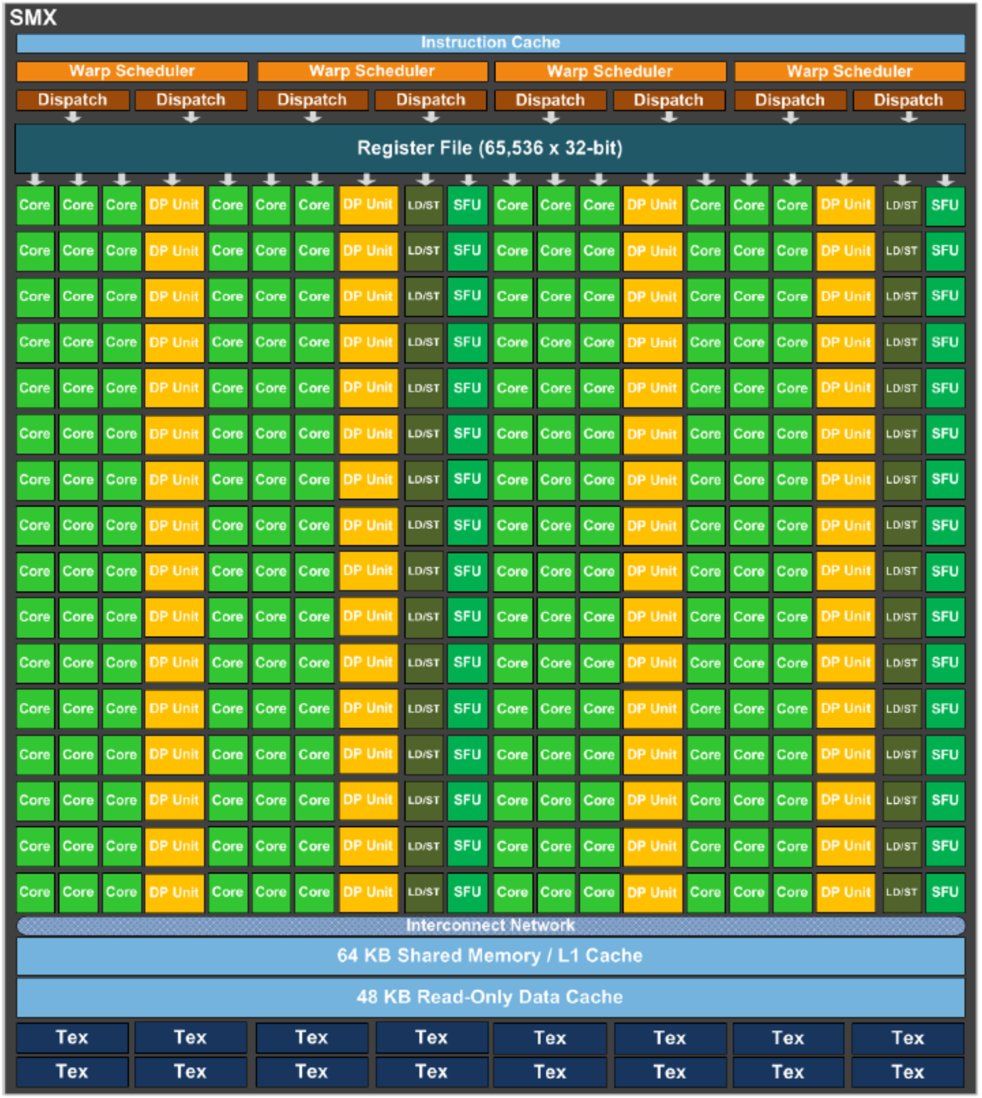


For more details: see "Performance with constant memory" on course web page

Overview of a GPU's Architecture

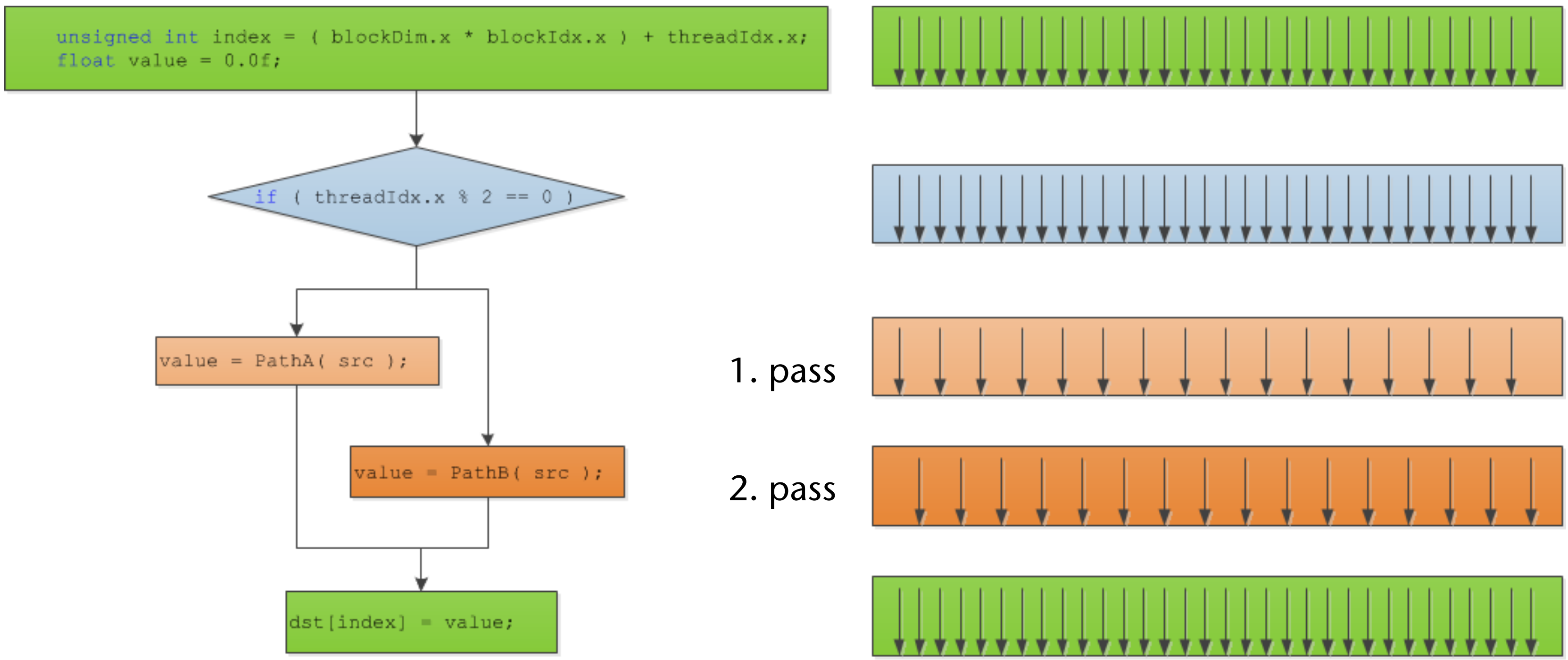
Nvidia's Kepler architecture as of 2012 (192 single-precision cores / 15 SMX)

One Streaming Multiprocessor

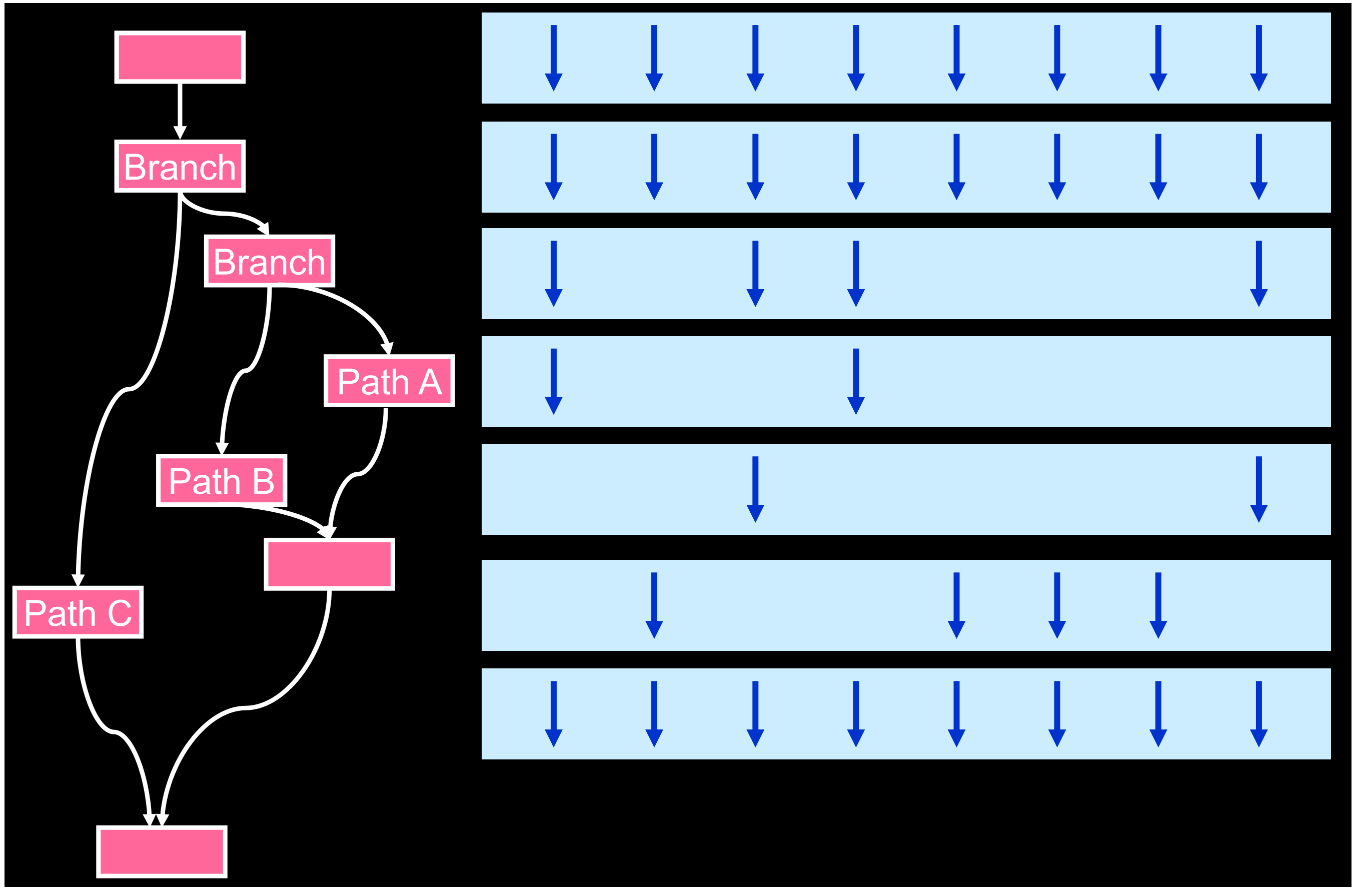


Thread Divergence Revisited

- This execution of threads in *lockstep fashion on one SMX* (think SIMD) is the reason, why thread divergence is so bad
- Thread divergence can occur at each occurrence of **`if-then-else`**, **`while`**, **`for`**, and **`switch`** (all control statements)
- Example:



- The more complex your control flow graph (this is called cyclometric complexity), the more thread divergence can occur!



Consequences for You as an Algorithm Designer / Programmer

- Try to devise algorithms that consist of kernels with very low cyclometric complexity
- Avoid recursion (would probably further increase thread divergence)
  - The other reason is that we would need one stack per thread
  - If your algorithm heavily relies on recursion, then it may not be well suited for massive (data) parallelism!

Measuring Performance on the GPU

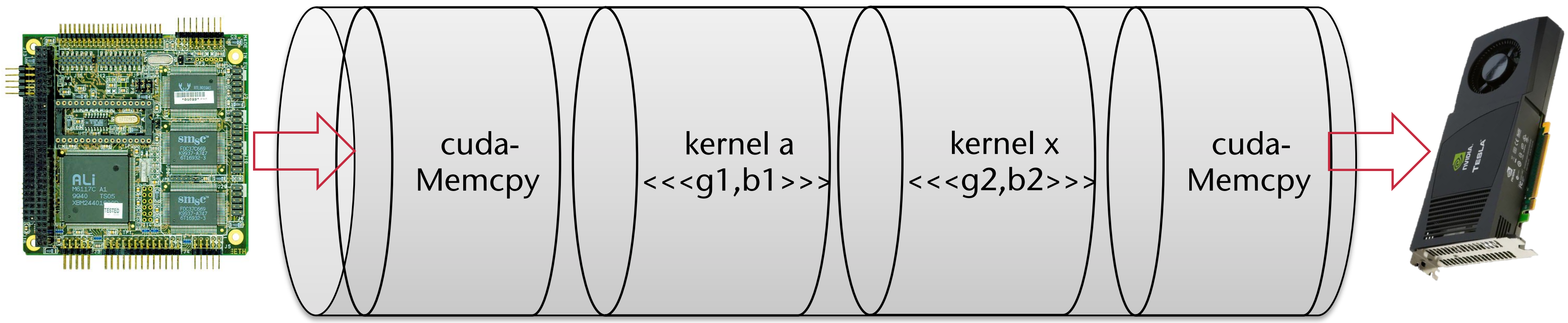
- Advice: experiment with a few different block layouts, e.g., `dim3 threads(16,16)` and `dim3 threads(128,2)` ; then compare performance
- CUDA API for timing: create events

```
// create two "event" structures
cudaEvent_t start, stop;
cudaEventCreate(&start); cudaEventCreate(&stop);
// insert the start event in the queue
cudaEventRecord( start, 0 );
now do something on the GPU, e.g., launch kernel ...

cudaEventRecord( stop, 0 );     // put stop into queue
cudaEventSynchronize( stop );   // wait for 'stop' to finish
float elapsedTime;              // print elapsed time
cudaEventElapsedTime( &elapsedTime, start, stop );
printf("Time to exec kernel = %f ms\n", elapsedTime );
```

On CPU/GPU Synchronization

- All kernel launches are asynchronous:
  - Control returns to CPU immediately
  - Kernel starts executing once all previous CUDA calls have completed
  - You can even launch another kernel without waiting for the first to finish
    - They will still be executed one after another
- Memcopies are synchronous:
  - Control returns to CPU once the copy is complete
  - Copy starts once all previous CUDA calls have completed
- `cudaDeviceSynchronize()`:
  - Blocks until all previous CUDA calls are complete

- Think of GPU & CPU as connected through a pipeline:



- Advantage of asynchronous CUDA calls:
  - CPU can work on other stuff while GPU is working on number crunching
  - Ability to overlap memcopies and kernel execution (we don't use this special feature in this course)

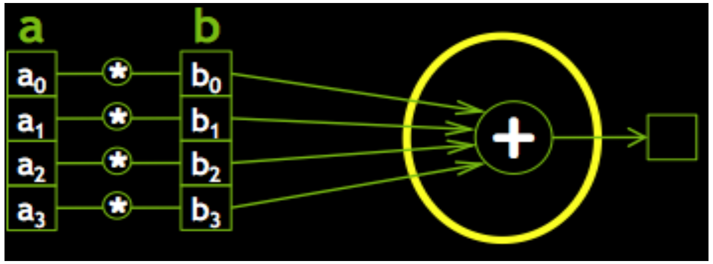
Why Bother with Blocks?

- The concept of *blocks* seems unnecessary:
  - It adds a level of complexity
  - The CUDA compiler could have done the partitioning of a range of threads into a grid of blocks *for us*
- What do we gain?
- Unlike parallel blocks, *threads within a block* have mechanisms to communicate & synchronize very quickly

Computing the Dot Product

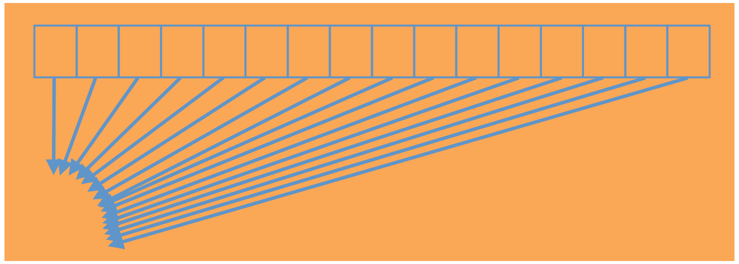
- Next goal: compute
$$d = \mathbf{x} \cdot \mathbf{y} = \sum_{i=0}^{N} x_i y_i$$
for large vectors

- We know how to do ($x_i y_i$) on the GPU, but how do we do the summation?



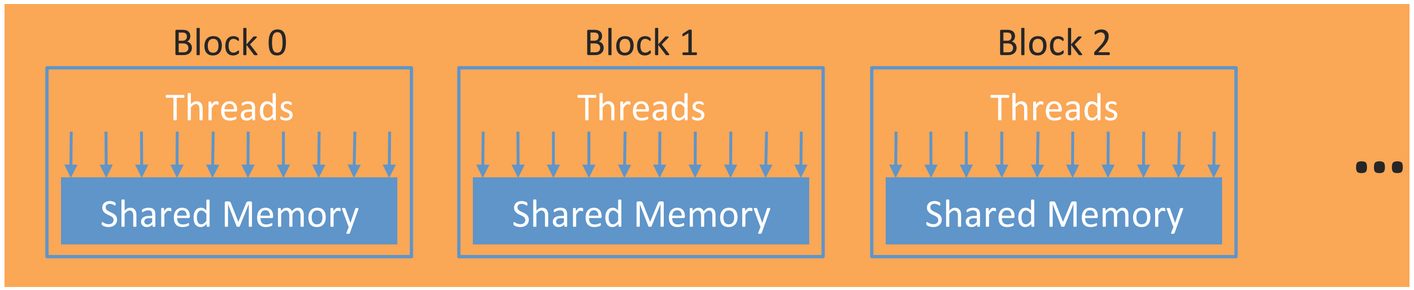
- Naïve (pseudo-parallel) algorithm:
  - Compute vector **z** with $z_i = x_i y_i$ in parallel
  - Transfer vector z back to CPU, and do summation sequentially

- Another (somewhat) naïve solution:
  - Compute vector **z** in parallel
  - Do summation of all z_i in thread 0

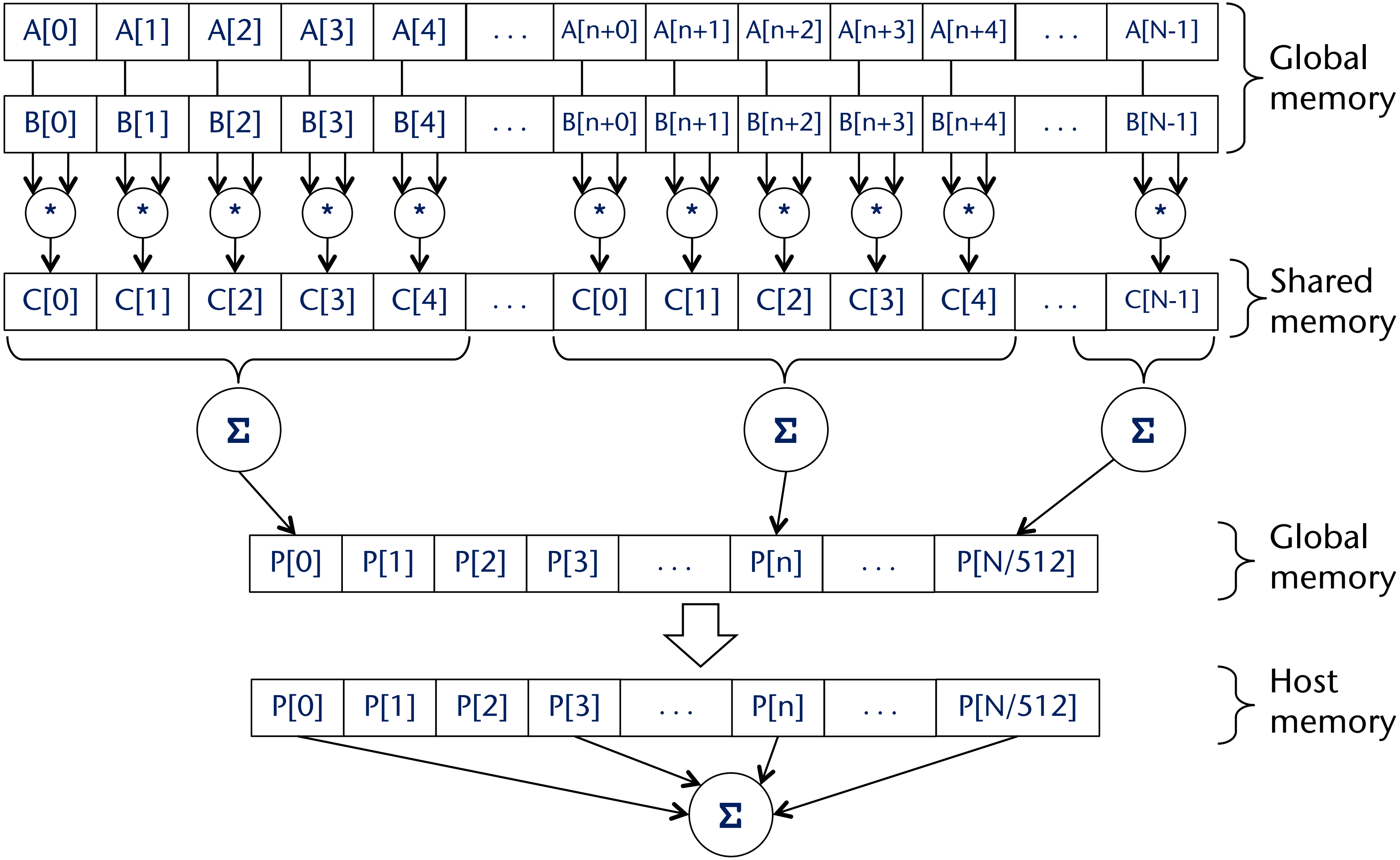


Cooperating Threads / Shared Memory

- Shared Memory:
  - A block of threads can have some amount of shared memory
  - All threads within a block have the same "view" of this
    - Just like with global memory
  - BUT, access to shared memory is much faster!
    - Kind of a user-managed cache
  - Not visible/accessible to other blocks
  - Every block has their own copy
    - So allocate only enough for one block
  - Declared with qualifier `__shared__`

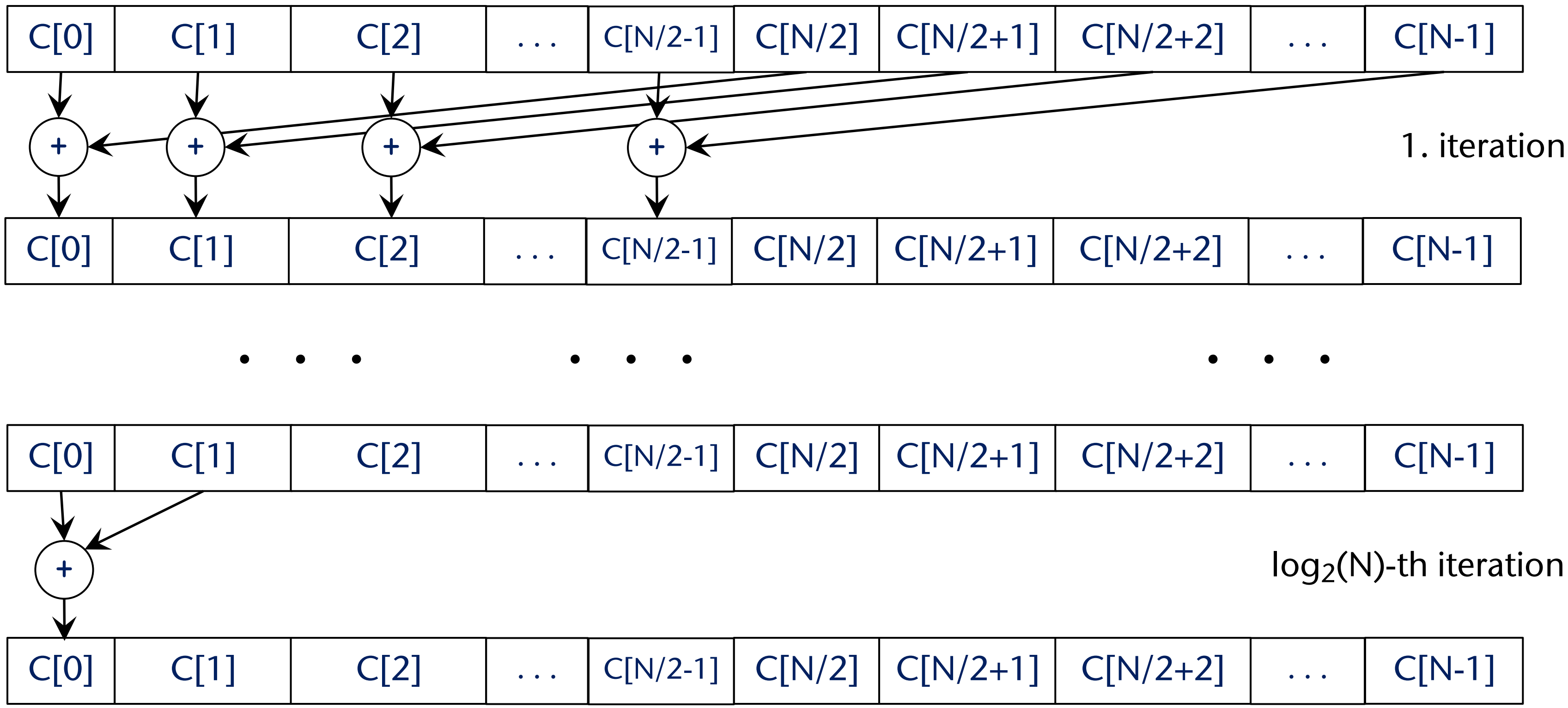


Overview of the Efficient Dot Product



Efficiently Computing the Summation Reduction

- Terminology: computing a smaller output vector (stream) from one/more larger input vectors is called reduction
  - Here summation reduction
- The pattern here:



The complete kernel for the dot product

```
__global__
void dotprod( float *a, float *b, float *p, int N )
    __shared__  float cache[threadsPerBlock];
    int tid = threadIdx.x + blockIdx.x * blockDim.x;

    if ( tid < N )
        cache[threadIdx.x] = a[tid] * b[tid];

    // for reductions, threadsPerBlock must be a pow

    int i = blockDim.x/2;
    while ( i != 0 ) {
        if ( threadIdx.x < i )
            cache[threadIdx.x] += cache[threadIdx.x + i];

        i /= 2;
    }

    // last thread copies partial sum to global memory
    if ( threadIdx.x == 0 )
        p[blockIdx.x] = cache[0];
}
```

This code contains a bug!

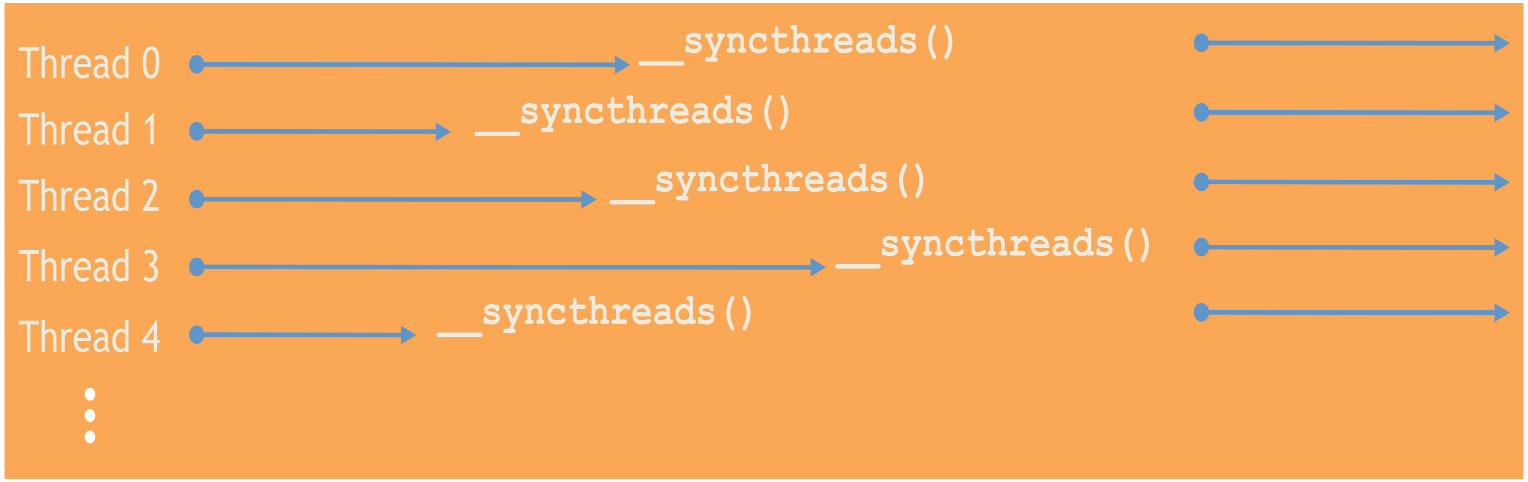
And that bug is probably hard to find!

The complete kernel for the dot product

```
__global__
void dotprod( float *a, float *b, float *p, int N ) {
   __shared__ float cache[threadsPerBlock];
   int tid = threadIdx.x + blockIdx.x * blockDim.x;

   if ( tid < N )
        cache[threadIdx.x] = a[tid] * b[tid];

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

   // last thread copies partial sum to global memory
   if ( threadIdx.x == 0 )
      p[blockIdx.x] = cache[0];
}
```

New Concept: Barrier Synchronization

- The command implements what is called a barrier synchronization (or just "barrier"):
  All threads wait at this point in the execution of their program, until all other threads have arrived at this *same point*



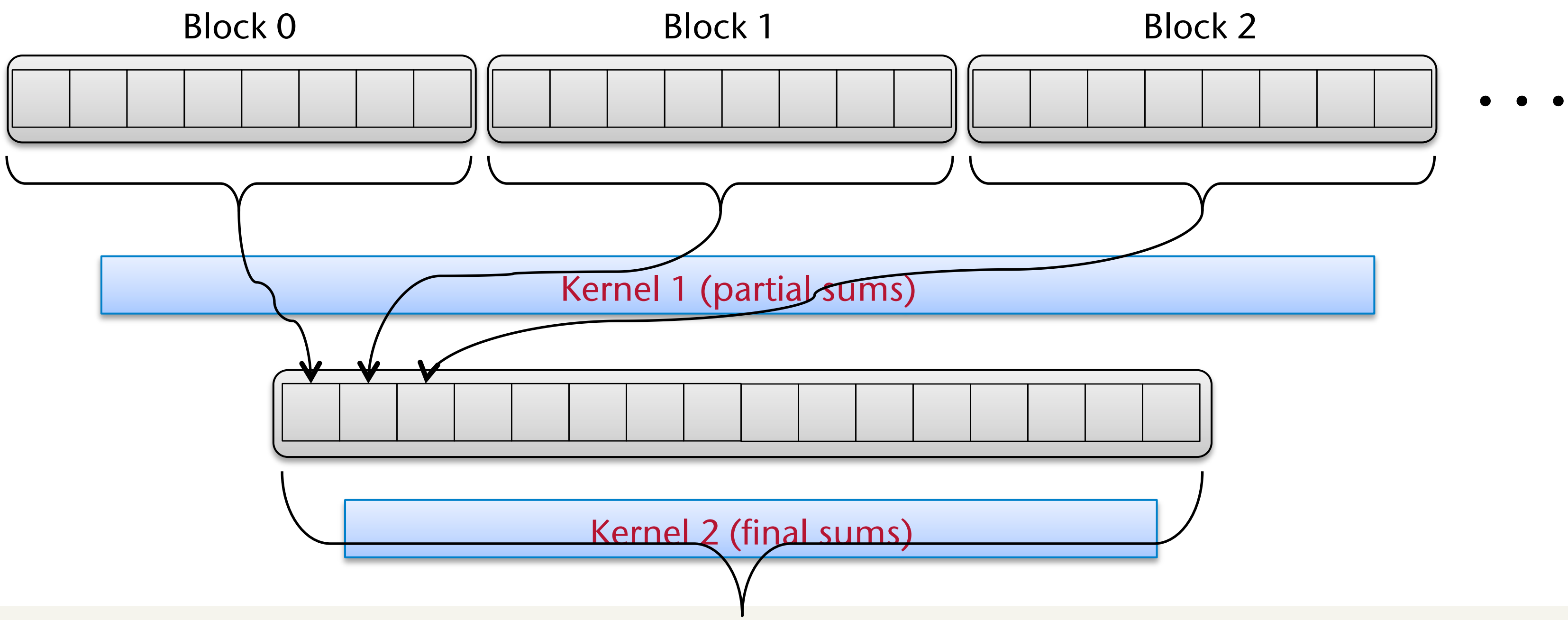
- Warning: threads are only synchronized within a block!

The Complete Dot Product Program

```
// allocate host & device arrays h_a, d_a, etc.
// h_c, d_p = arrays holding partial sums

dotprod<<< nBlocks, nThreadsPerBlock >>>( d_a, d_b, d_p, N );

transfer d_p -> h_p

float prod = 0.0;
for ( int i = 0; i < nBlocks, i ++ )
   prod += h_p[i];
```

How to Compute the Dot-Product Completely on the GPU

- You might want to compute the dot-product complete on the GPU
  - Because you need the result on the GPU anyway
- Idea:
  1. Compute partial sums with one kernel
  2. With another kernel, compute final sum of partial sums
- Gives us automatically a sync/barrier between first/second kernel



A Caveat About Barrier Synchronization

- You might consider optimizing the kernel like so:

```
__global__
void dotprod( float *a, float *b, float *c, int
{
    // just like before ...

    // incorrectly optimized reduction
    __syncthreads();
    int i = blockDim.x/2;
    while ( i != 0 ) {
        if ( threadIdx.x < i )
        {
            cache[threadIdx.x] += cache[threadIdx.x + i];
            __syncthreads();
        }
        i /= 2;
    }
    // rest as before ...
```

This code contains a bug!

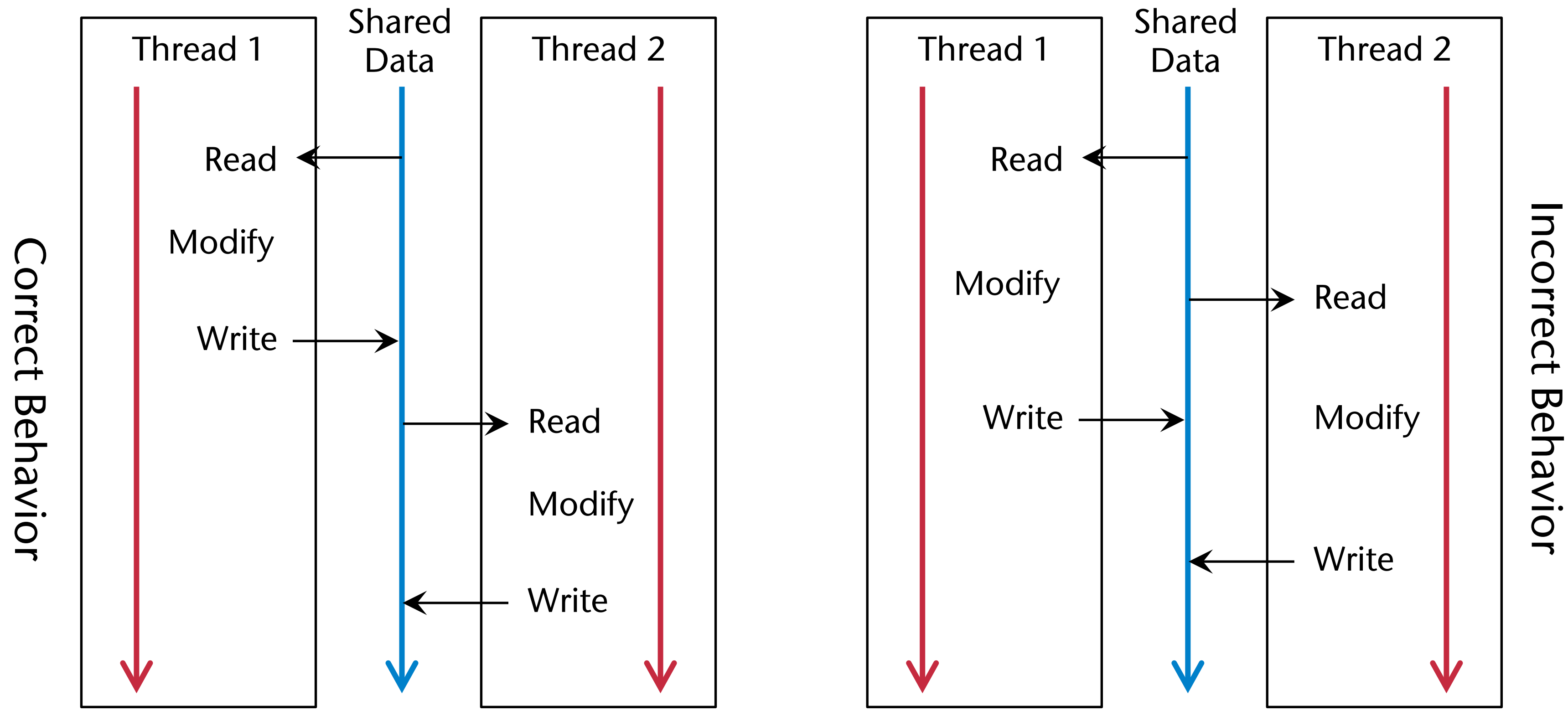
It makes your GPU hang ...!

  - Idea: only wait for threads that were actually writing to memory ...

- Bug: the barrier will never be fulfilled!

New Concepts & Terminology

- A race condition occurs when overall program behavior depends upon relative timing of two (or more) event sequences
- Frequent case: two processes (threads) read-modify-write the same memory location (variable)



Race Conditions

- Race conditions come in three different kinds of hazards:
  - *Read-after-write hazard* (RAW): true data dependency, most common type
  - *Write-after-read hazard* (WAR): anti-dependency (basically the same as RAW)
  - *Write-after-write hazard* (WAW): output dependency
- Consider this (somewhat contrived) example:
  - Given input vector *x*, compute output vector

$$y = (x_0{*}x_1, x_0{*}x_1, x_2{*}x_3, x_2{*}x_3, x_4{*}x_5, x_4{*}x_5, \dots)$$

  - Approach: two threads, one for odd/even numbered elements

```
kernel( const float * x, float * y, int N )  {
   __shared__ cache[2];
   for ( int i = 0; i < N/2; i ++ )   {
      cache[threadIdx.x] = x[ 2*i + threadIdx.x];
      y[2*i + threadIdx.x] = cache[0] * cache[1];
   }
}
```

- Execution in a warp, i.e., in lockstep:

Thread 0	Thread 1
`cache[0] = x[0];` `y[0] = cache[0] * cache[1];`	`cache[1] = x[1];` `y[1] = cache[0] * cache[1];`
`cache[0] = x[2];` `y[2] = cache[0] * cache[1];`	`cache[1] = x[3];` `y[3] = cache[0] * cache[1];`

`cache[0] = x[4];`
`y[4] = cache[0] * cache[1];`
`...`
	`cache[1] = x[5];`
	`y[5] = cache[0] * cache[1];`

  - Everything is fine

- In the following, we consider execution on different warps / SMs

Thread 0 | Thread 1

Read-after-write hazard!

```
cache[0] = x[0];
y[0] = cache[0] * cache[1];
                                   cache[1] = x[1];
                                   y[1] = cache[0] * cache[1];
cache[0] = x[2];
y[2] = cache[0] * cache[1];
                                   cache[1] = x[3];
                                   y[3] = cache[0] * cache[1];
cache[0] = x[4];
y[4] = cache[0] * cache[1];
                                   cache[1] = x[5];
                                   y[5] = cache[0] * cache[1];
...
```

- Remedy:

```
kernel( const float * x, float * y, int N )
{
   __shared__ cache[2];
   for ( int i = 0; i < N/2; i ++ )
   {
      cache[threadIdx.x] = x[ 2*i + threadIdx.x];
      __syncthreads();
      y[2*i + threadIdx.x] = cache[0] * cache[1];
   }
}
```

Thread 0 | Thread 1

```
cache[0] = x[0];
                                        cache[1] = x[1];
---------------------- syncthreads() ----------------------
y[0] = cache[0] * cache[1];
cache[0] = x[2];
                                        y[1] = cache[0] * cache[1];
                                        cache[1] = x[3];
---------------------- syncthreads() ----------------------
y[2] = cache[0] * cache[1];
cache[0] = x[4];
                                        y[3] = cache[0] * cache[1];
                                        cache[1] = x[5];
---------------------- syncthreads() ----------------------
...
```

(Re-)Write-after-read hazard!

- Final remedy:

```
kernel( const float * x, float * y, int N )
{
   __shared__ cache[2];
   for ( int i = 0; i < N/2; i ++ )
   {
      cache[threadIdx.x] = x[ 2*i + threadIdx.x];
      __syncthreads();
      y[2*i + threadIdx.x] = cache[0] * cache[1];
      __syncthreads();
   }
}
```

- Note: you'd never design the algorithm this way!

Digression: Race Conditions are an Entrance Door for Hackers

- Race conditions occur in all environments and programming languages (that provide some kind of parallelism)
- CVE-2009-2863:
  - Race condition in the Firewall Authentication Proxy feature in Cisco IOS 12.0 through 12.4 allows remote attackers to bypass authentication, or bypass the consent web page, via a crafted request.
- CVE-2013-1279:
  - Race condition in the kernel in Microsoft [...] Windows Server 2008 SP2, R2, and R2 SP1, Windows 7 Gold and SP1, Windows 8, Windows Server 2012, and Windows RT allows local users to gain privileges via a crafted application that leverages incorrect handling of objects in memory, aka "Kernel Race Condition Vulnerability".
- Many more: search for "race condition" on http://cvedetails.com/