Measuring Performance on the GPU

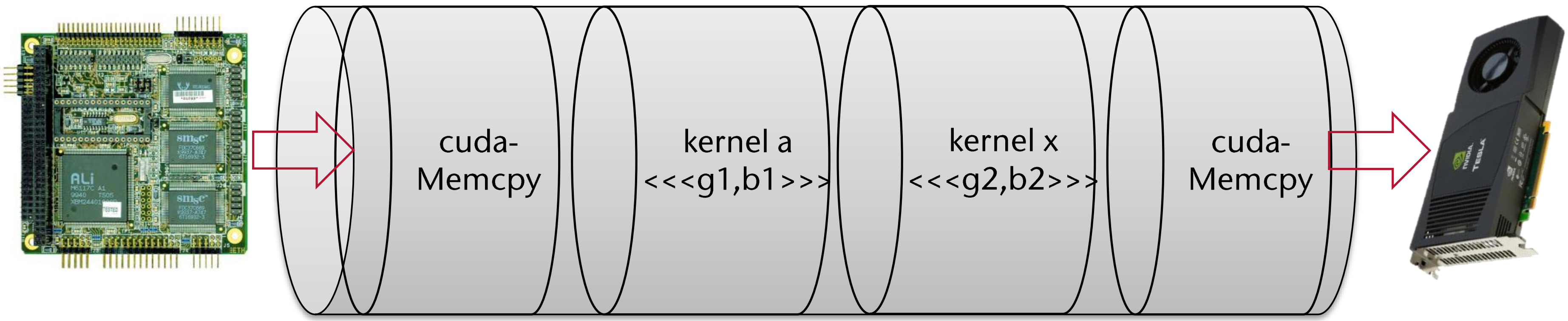
- Advice: experiment with a few different block layouts, e.g., `dim3 threads(16,16)` and `dim3 threads(128,2)` ; then compare performance
- CUDA API for timing: create events

```
// 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:



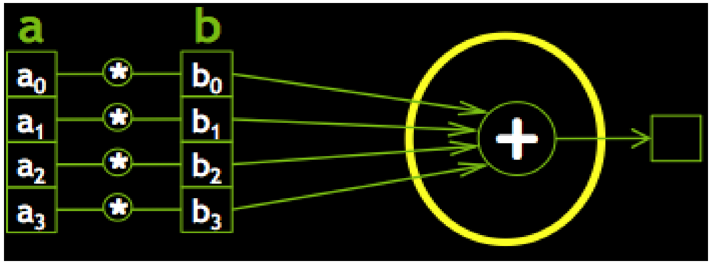
- Advantage of asynchronous CUDA calls:
  - CPU can work on other stuff while GPU is working on number crunching
  - Ability to overlap memcopies and kernel execution (we don't use this special feature in this course)

Why Bother with Blocks?

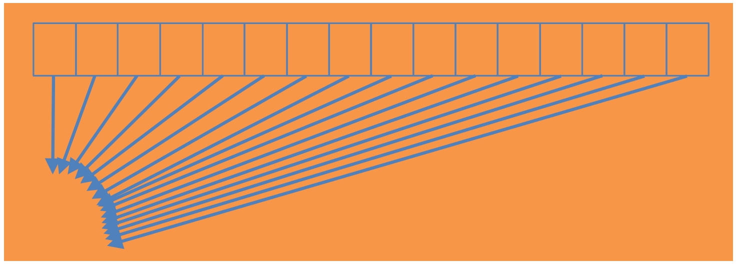
- The concept of *blocks* seems unnecessary:
  - It adds a level of complexity
  - The CUDA compiler could have done the partitioning of a range of threads into a grid of blocks *for us*
- What do we gain?
- Unlike parallel blocks, *threads within a block* have mechanisms to communicate & synchronize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

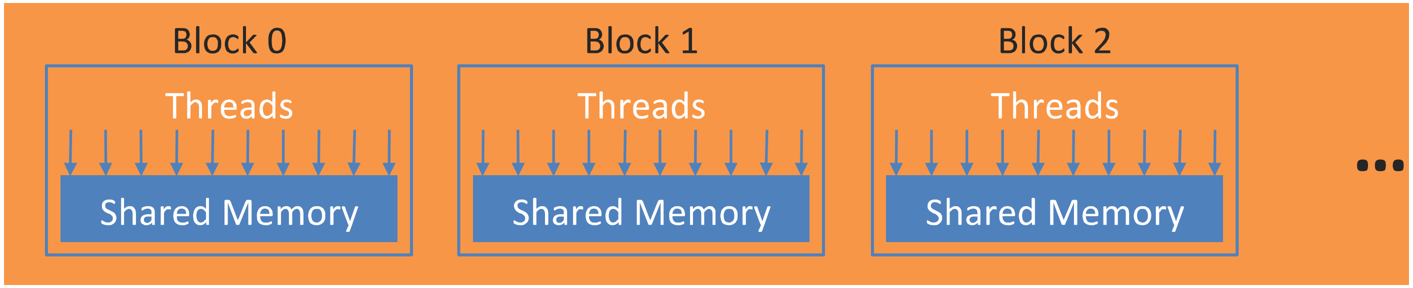


- Naïve (pseudo-parallel) algorithm:
  - Compute vector **z** with $z_i = x_i y_i$ in parallel
  - Transfer vector z back to CPU, and do summation sequentially
- Another (somewhat) naïve solution:
  - Compute vector **z** in parallel
  - Do summation of all z_i in thread 0

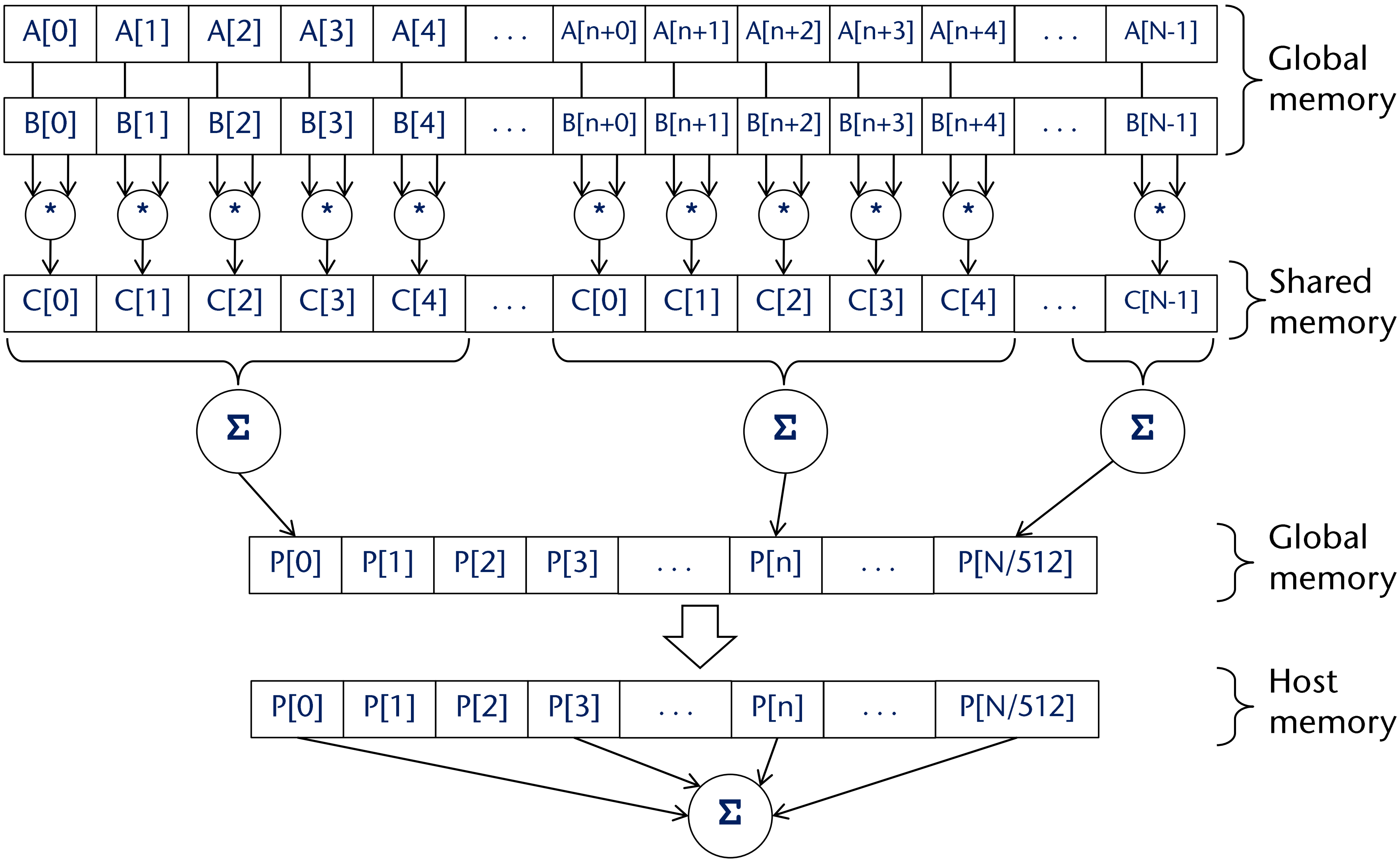


Cooperating Threads / Shared Memory

- 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__`

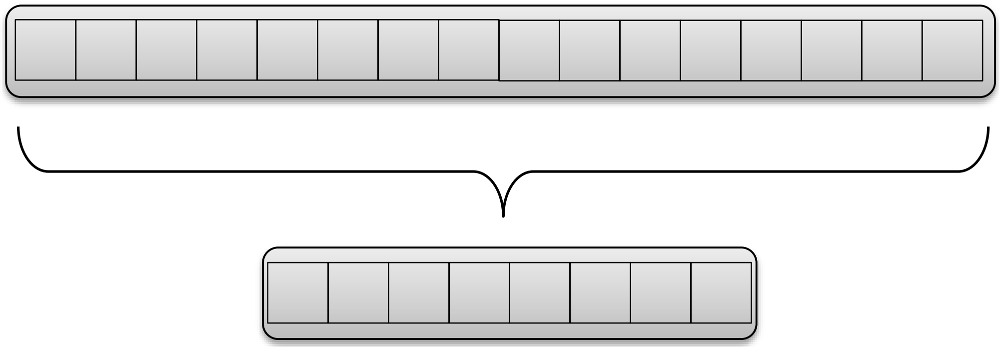


Overview of the Efficient Dot Product

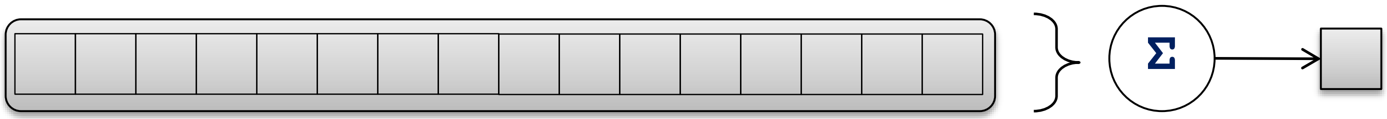


Terminology

- The term "reduction" always means that the output stream/vector of a kernel is smaller than the input

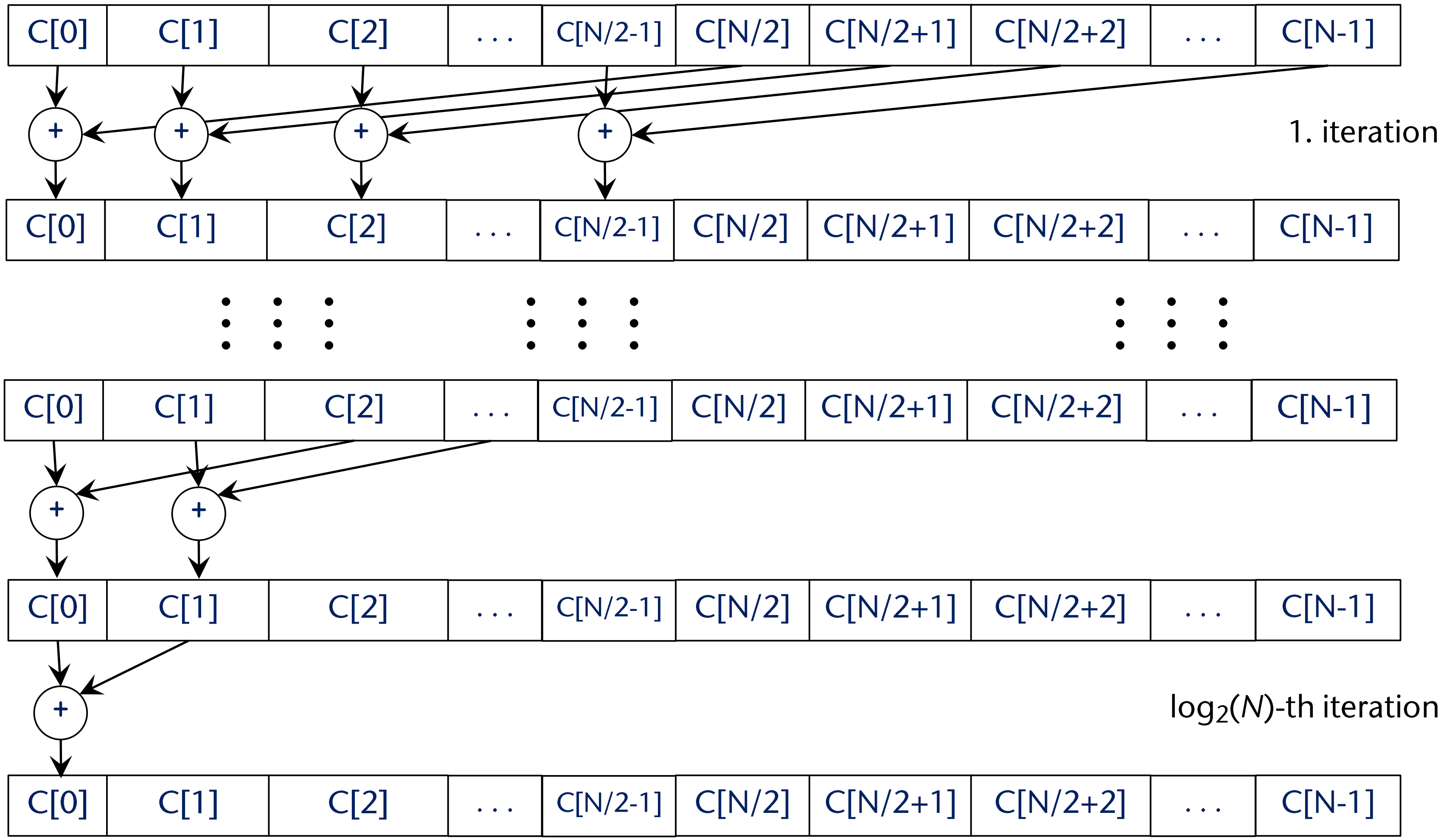


- Examples:
  - Dot product; takes 2 vectors, outputs 1 scalar = summation reduction
  - Min/max of the elements of a vector = min/max reduction



Efficiently Computing the Summation Reduction

- A (common) massively-parallel programming pattern:



The complete kernel for the dot product

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

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

    // Here, for easy reduction,
    // blockDim.x must be a power of 2!

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

        stride /= 2;
    }

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

This code contains a bug!

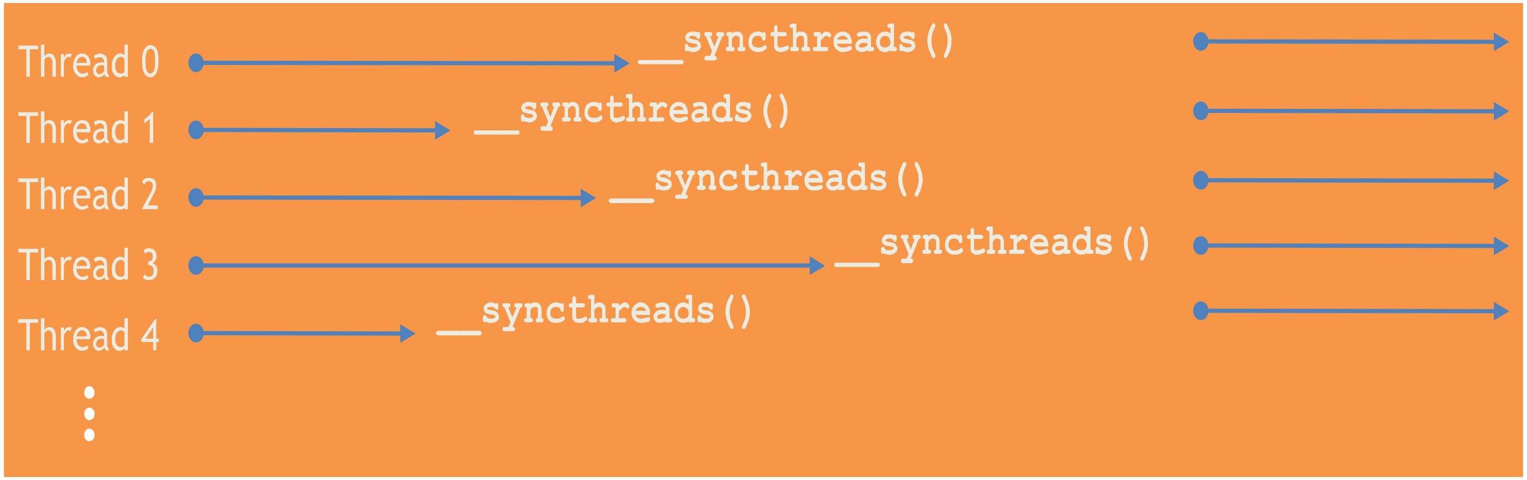
And that bug is probably hard to find!

The complete kernel for the dot product

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

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

   // Here, for easy reduction,
   // blockDim.x must be a power of 2!
  __syncthreads();
   int stride = blockDim.x/2;
   while ( stride != 0 ) {
      if ( threadIdx.x < stride )
         cache[threadIdx.x] += cache[threadIdx.x + stride];
      __syncthreads();
      stride /= 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*



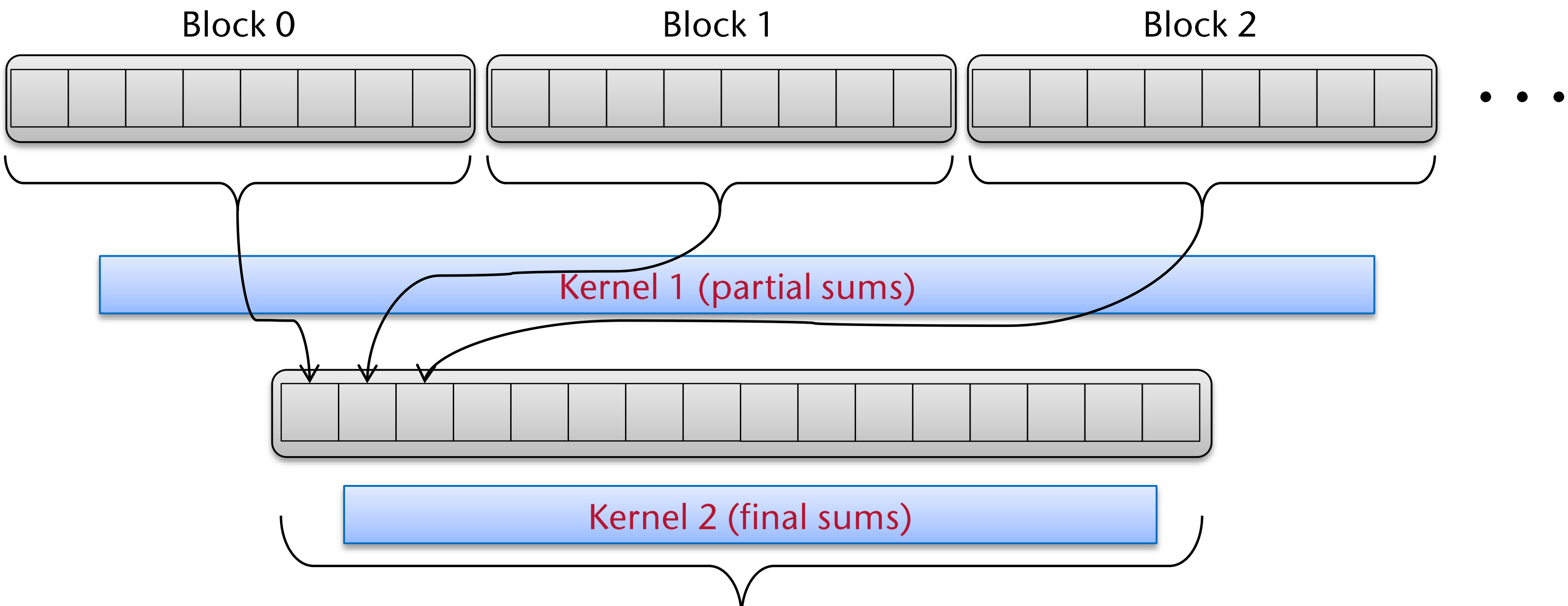
- Warning: threads are only synchronized *within a block*!

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 for achieving barrier right before 2nd reduction:
  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 N
{
    // just like before ...

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

This code contains a bug!

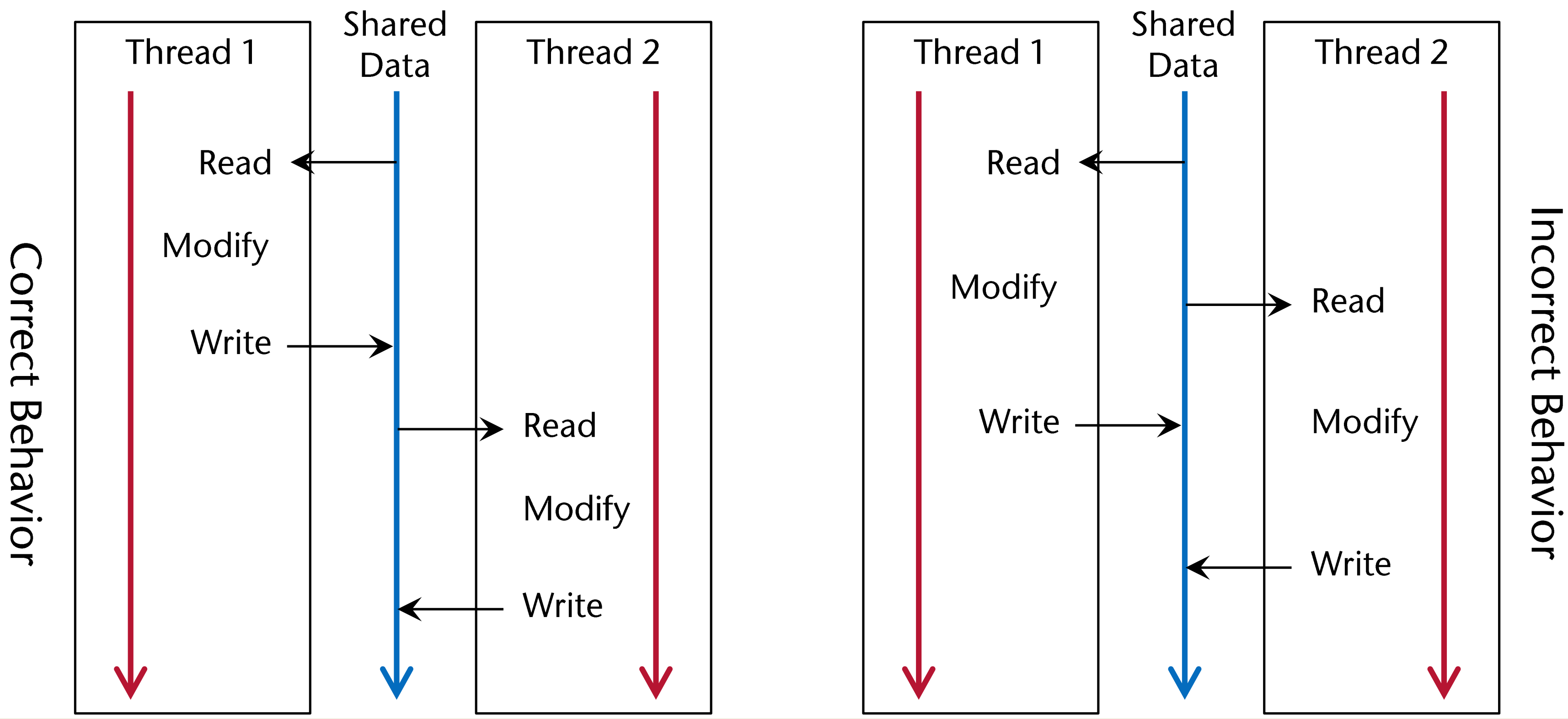
It makes your GPU hang ...!

  - 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 in *different* warps / SMs

Thread 0

```
cache[0] = x[0];
y[0] = cache[0] * cache[1];

cache[0] = x[2];
y[2] = cache[0] * cache[1];

cache[0] = x[4];
y[4] = cache[0] * cache[1];

...
```

Thread 1

Read-after-write hazard!

```
cache[1] = x[1];
y[1] = cache[0] * cache[1];

cache[1] = x[3];
y[3] = 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() ——————————————
...
```

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/

Application of Dot Product: Document Similarity

- Task: compute "similarity" of documents (think Google)
  - One of the fundamental tasks in information retrieval (IR)
- Example: search engine / database of scientific papers needs to suggest similar papers for a given one
- Assumption: all documents are over a given, fixed vocabulary V consisting of N words (e.g., all English words)
  - Consequence: set of words, V, occurring in the docs is known & fixed
- Assumption: don't consider word order → *bag of words* model
  - Consequence: "*John is quicker than Mary*" = "*Mary is quicker than John*"

- Representation of a document D:
  - For each word $w \in V$: determine $f(w)$ = frequency of word w in D
  - Example:

	Anthony & Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
ANTHONY	157	73	0	0	0	1
BRUTUS	4	157	0	2	0	0
CAESAR	232	227	0	2	1	0
CALPURNIA	0	10	0	0	0	0
CLEOPATRA	57	0	0	0	0	0
MERCY	2	0	3	8	5	8
WORSER	2	0	1	1	1	5
...	...	...	...	...	...	...

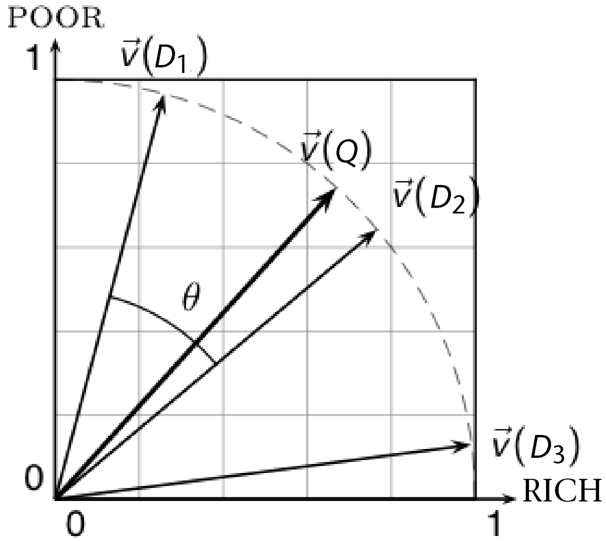
  - Fix a word order in $V = (v_1, v_2, v_3, \ldots, v_N)$ (in principle, any order will do)
  - Represent D as a vector in $\mathbf{R}^N$:

$$D = \left(f(v_1), f(v_2), f(v_3), \ldots, f(v_N) \right)$$

- Note: our vector space is HUGE ($N \sim 100{,}000 - 10{,}000{,}000$)
  - For each word w, there is one axis in our vector space!

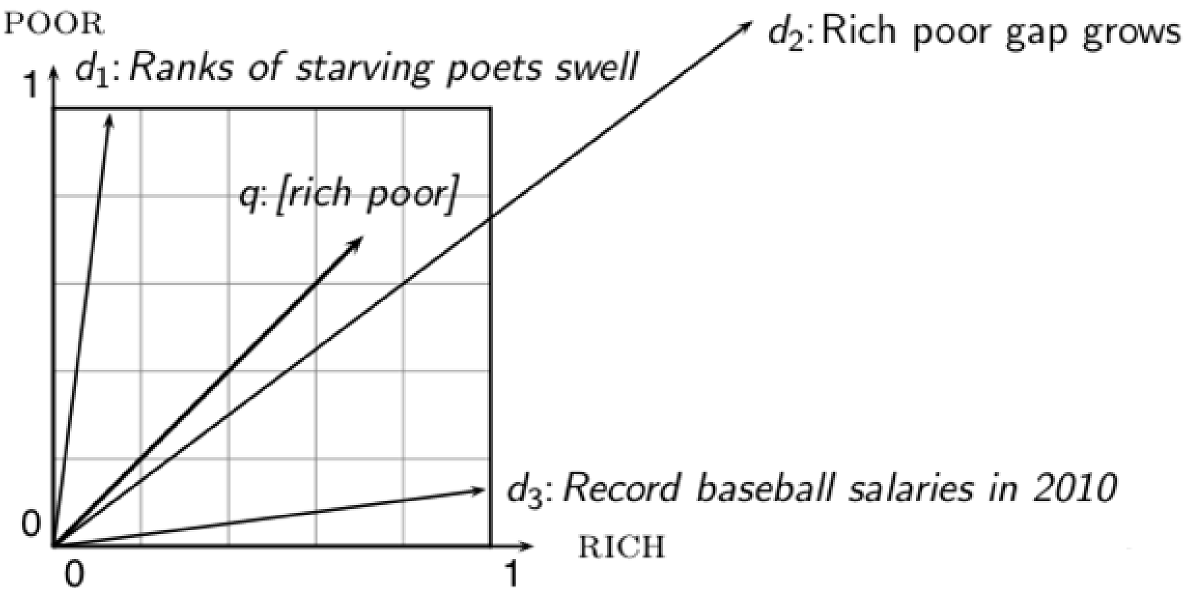
- Define similarity s between documents D_1 and D_2 as

$$s(D_1, D_2) = \frac{D_1 \cdot D_2}{\|D_1\| \cdot \|D_2\|} = \cos(D_1, D_2)$$



- This similarity measure is called "vector space model"
  - One of the most frequently used similarity measures in IR
- Note: our definition is a slightly simplified version of the commonly used one (we omitted the tf-idf weighting)

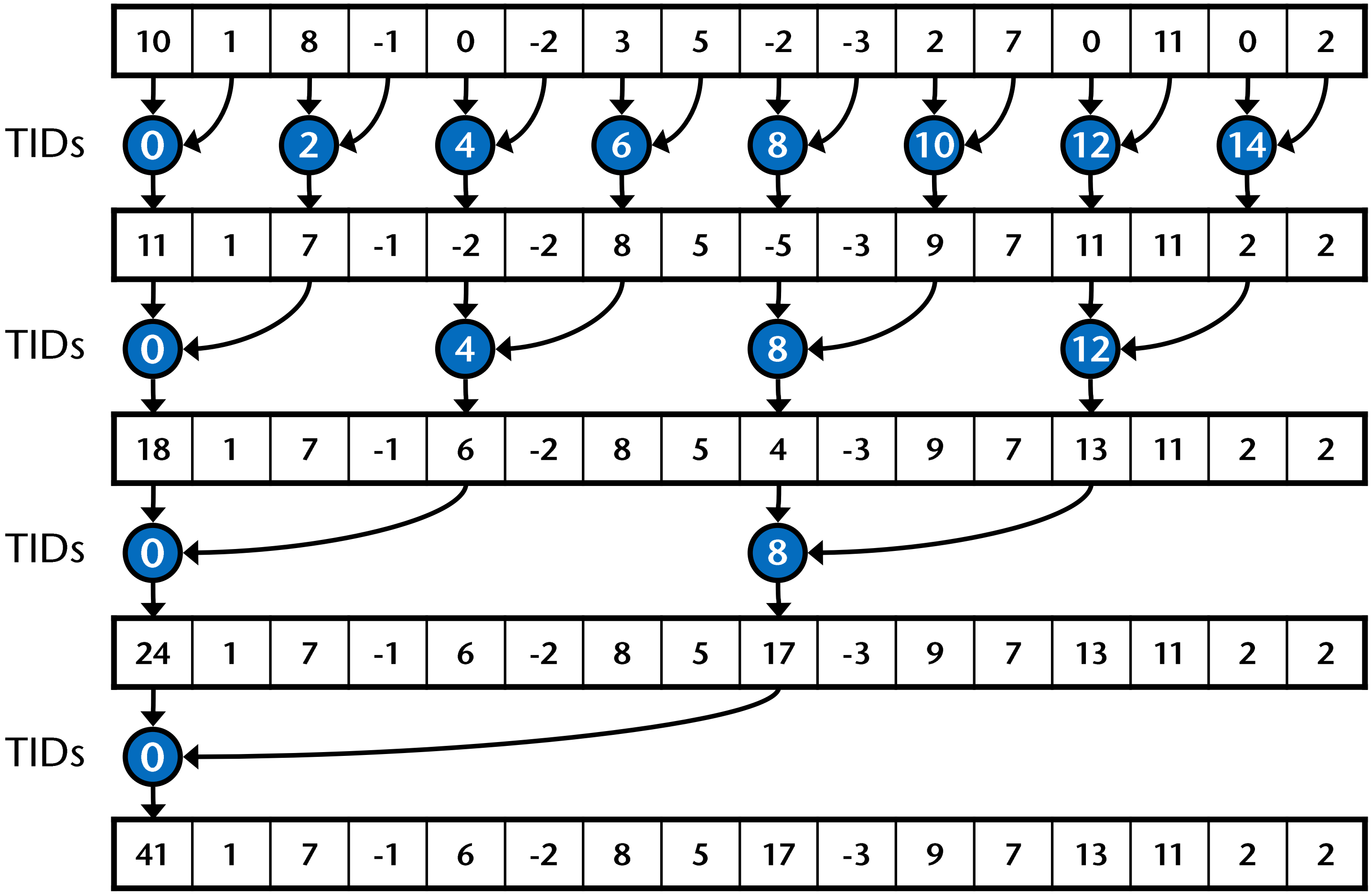
- Why not the Euclidean distance $\|D_1 - D_2\|$?
  - Otherwise: documents *D*, and *D* concatenated to itself would be very *dissimilar*!



- Why do we need the normalization by $\frac{1}{\|D_1\| \cdot \|D_2\|}$?
  - Same reason ...

Parallel Reduction Revisited

- Why didn't we do the reduction this way?



- The kernel for this algorithm:

```
// do reduction in shared mem
__syncthreads();
for ( int i = 1; i < blockDim.x; i *= 2 )
{
    if ( threadIdx.x % (2*i) == 0 )
        cache[threadIdx.x] += cache[threadIdx.x + i];
    __syncthreads();
}
```

Problem: highly divergent warps are very inefficient

- Further problem: memory access is not contiguous ☹
  - The GPU likes contiguous memory access ☐

A Real Optimization for Reduction

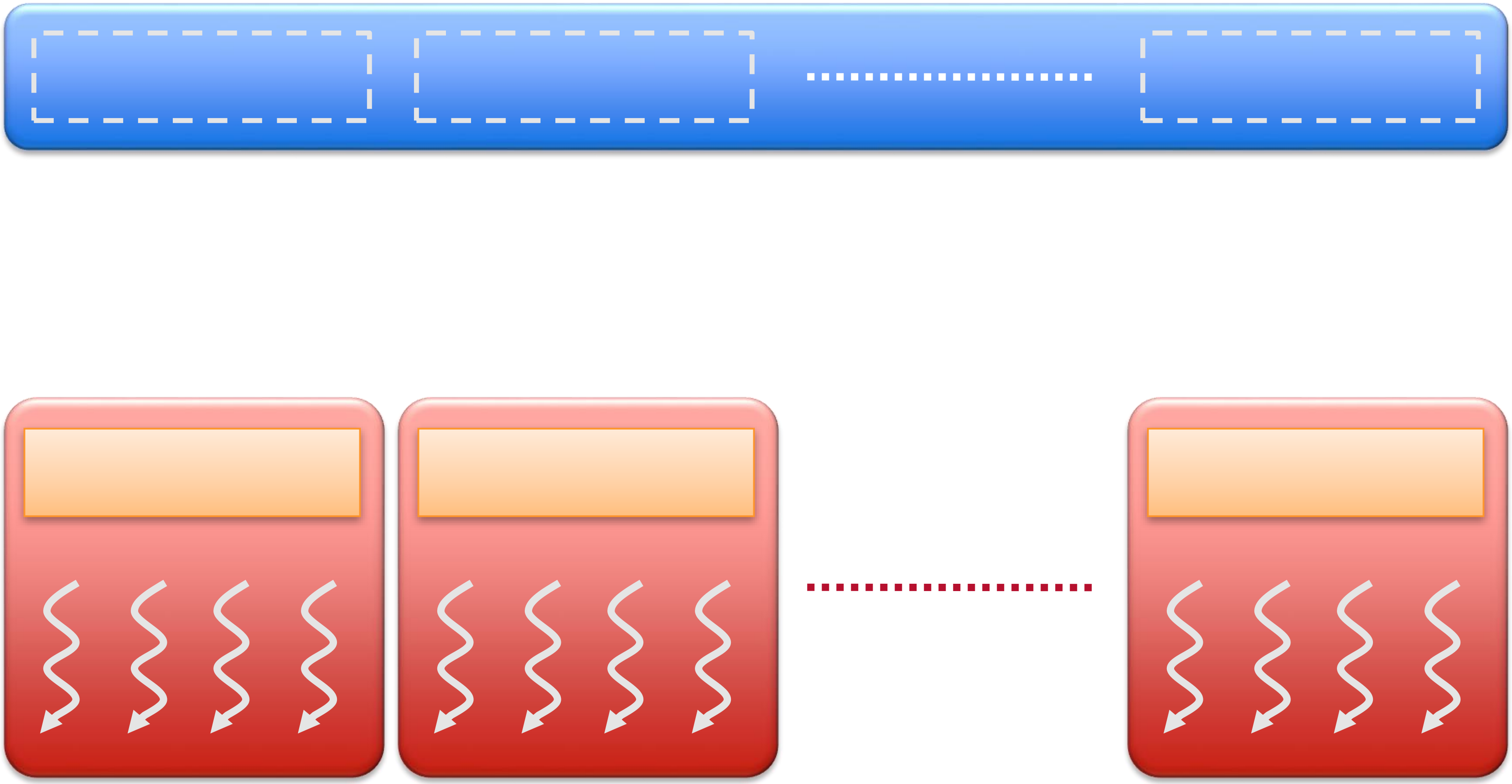
- Reduction usually does not do a lot of computations
  - Called low arithmetic intensity (more on that later)
- Try to maximize bandwidth by reducing the instruction overhead
  - Here: try to get rid of any instruction that is not load/store/arithmetic
  - I.e., get rid of address arithmetic and loop instructions
- Observation:
  - As reduction proceeds, # active threads decreases
  - When stride <= 32, only one warp of threads is left
- Remember: instructions within warp are SIMD (lock-stepped)
- Consequence:
  - No __syncthreads() necessary
  - No `if (threadIdx.x < stride)` necessary, because of lock-stepped threads within the warp (i.e., `if` doesn't save work anyway)

- Optimization: unroll last 6 iterations (=log(32))

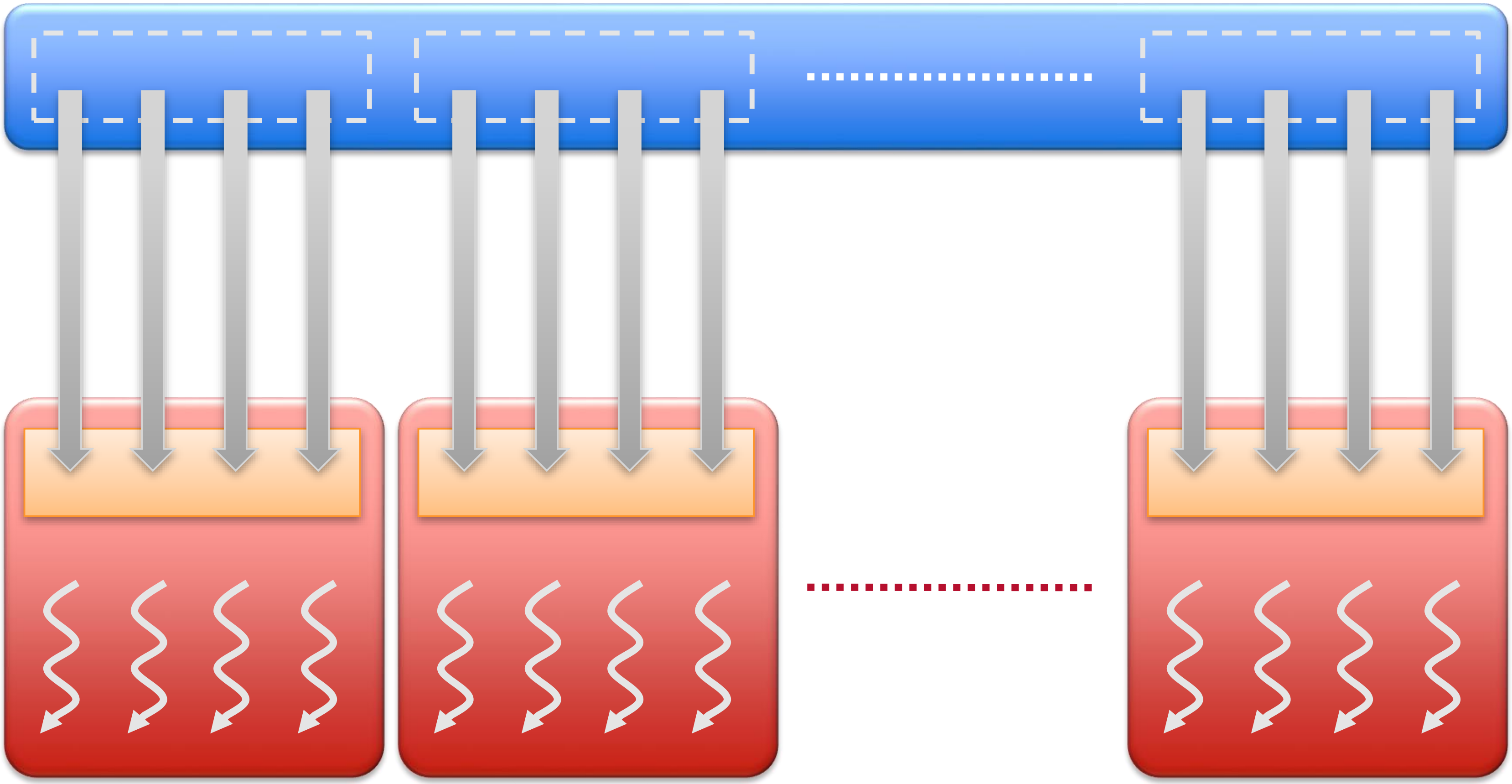
```
int stride = blockDim.x/2;
while ( stride > 32 ) {
   if ( threadIdx.x < stride )
      cache[threadIdx.x] += cache[threadIdx.x + stride];
   __syncthreads();
   stride /= 2;
}

if ( threadIdx.x < 32 )
{
   sdata[tid] += sdata[tid + 32];
   sdata[tid] += sdata[tid + 16];
   sdata[tid] += sdata[tid +  8];
   sdata[tid] += sdata[tid +  4];
   sdata[tid] += sdata[tid +  2];
   sdata[tid] += sdata[tid +  1];
}
```

- Note: This saves useless work in all warps, not just the last one
- Gives almost factor 2 speedup over previous version!

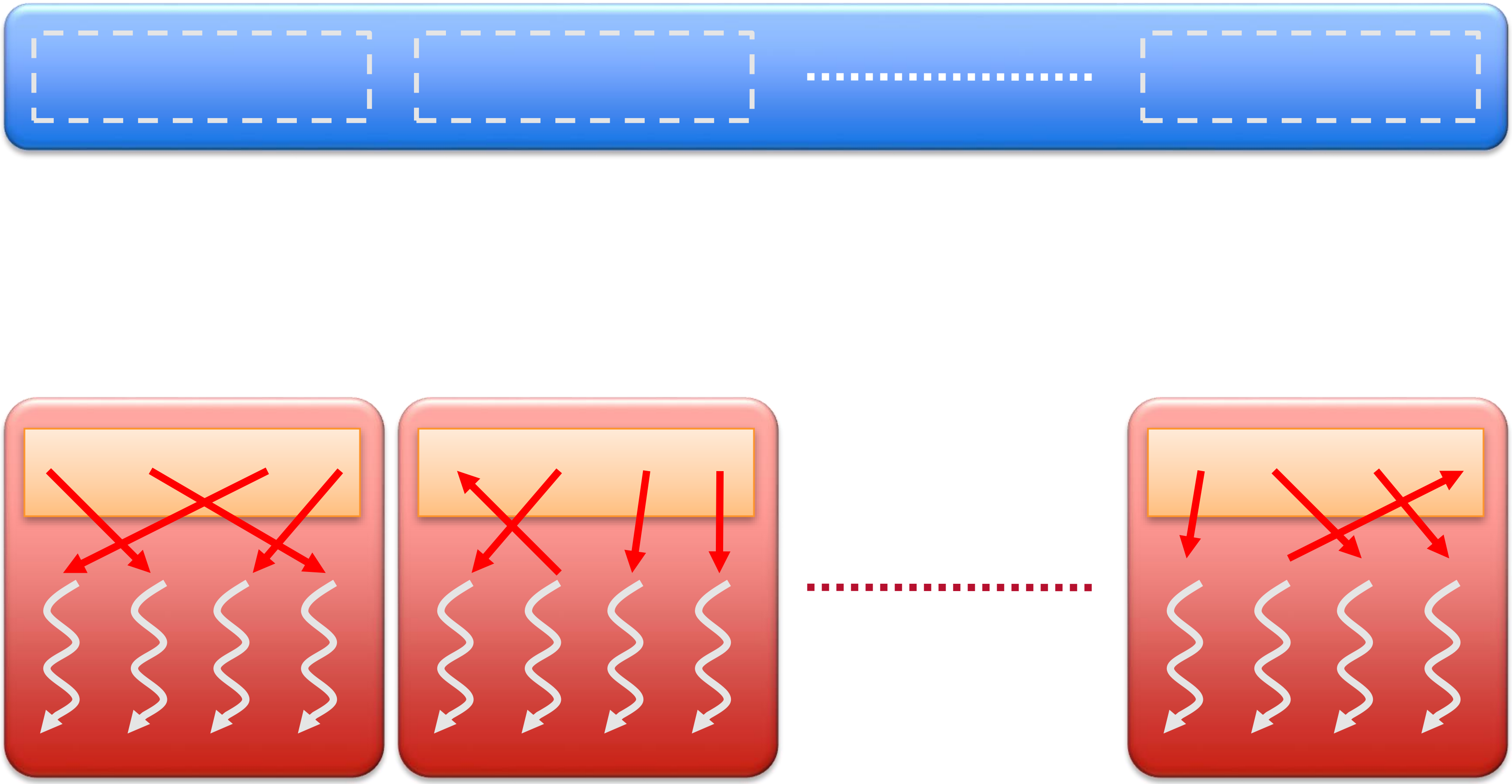
A Common, Massively Parallel Programming Pattern



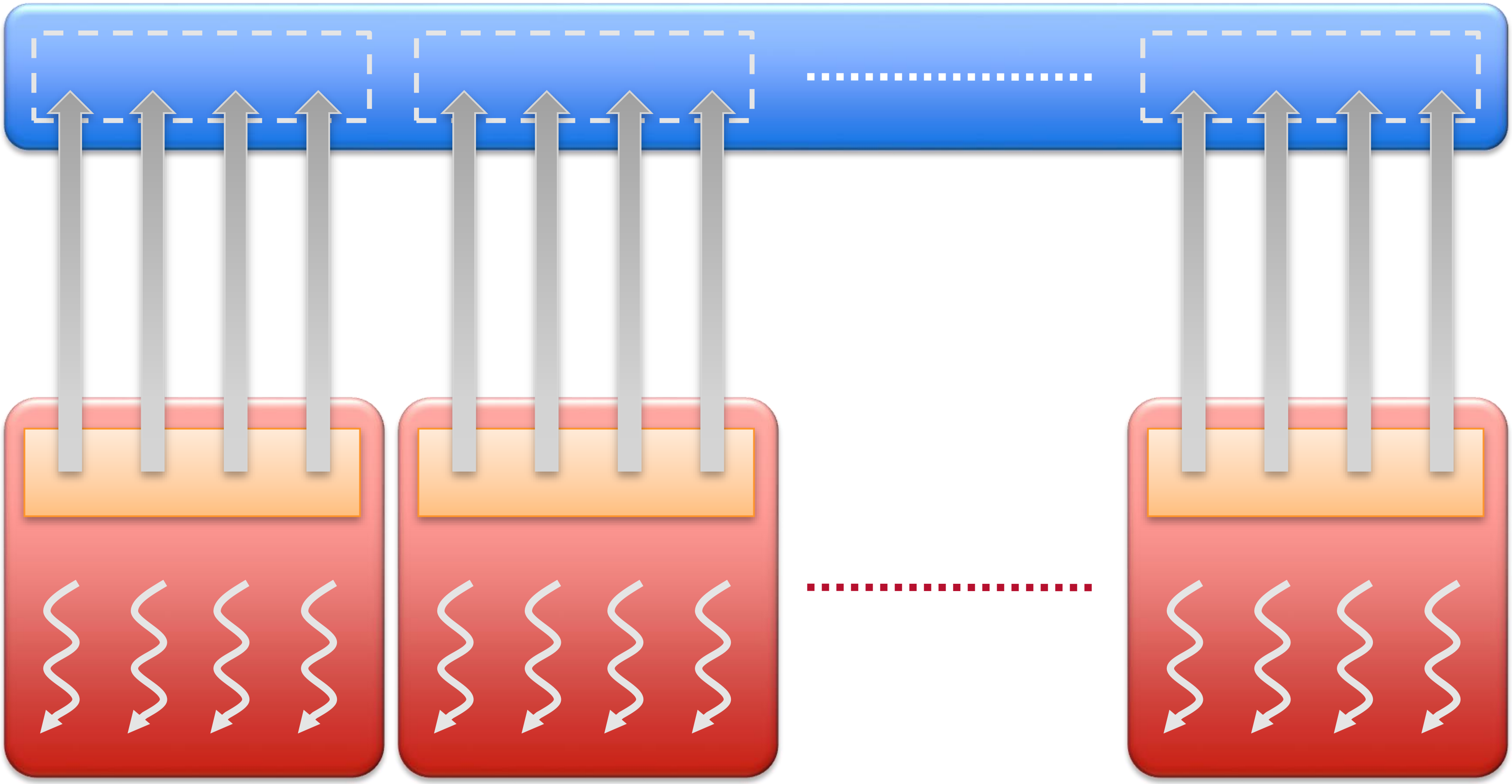
- Partition your domain such that each subset fits into shared memory; handle each data subset with one thread block



- Load the subset from global memory to shared memory; exploit memory-level parallelism by loading one piece per thread; don't forget to synchronize all threads before continuing!



- Perform the computation on the subset in shared memory



- Copy the result from shared memory back to global memory