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
...	...	...	...	...	...	...

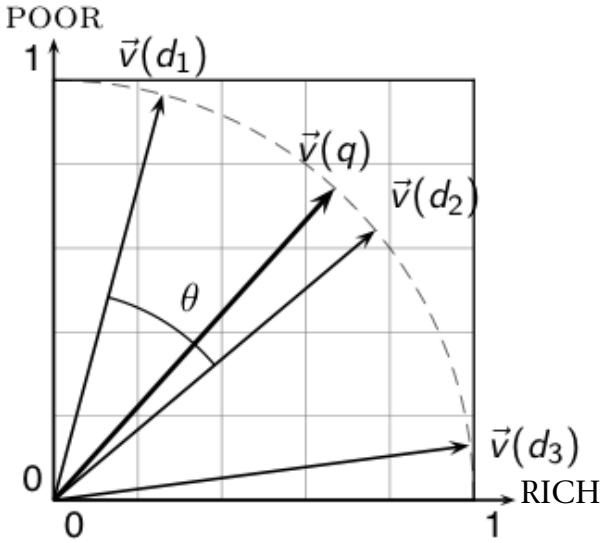
  - Fix a word order in $V = (v_1, v_2, v_3, ..., v_N)$ (in principle, any order will do)
  - Represent *D* as a vector in $\mathbf{R}^N$:

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

- Note: our vector space is HUGE (N ~ 100,000 – 10,000,000)
  - For each word *w*, there is one axis in our vector space!

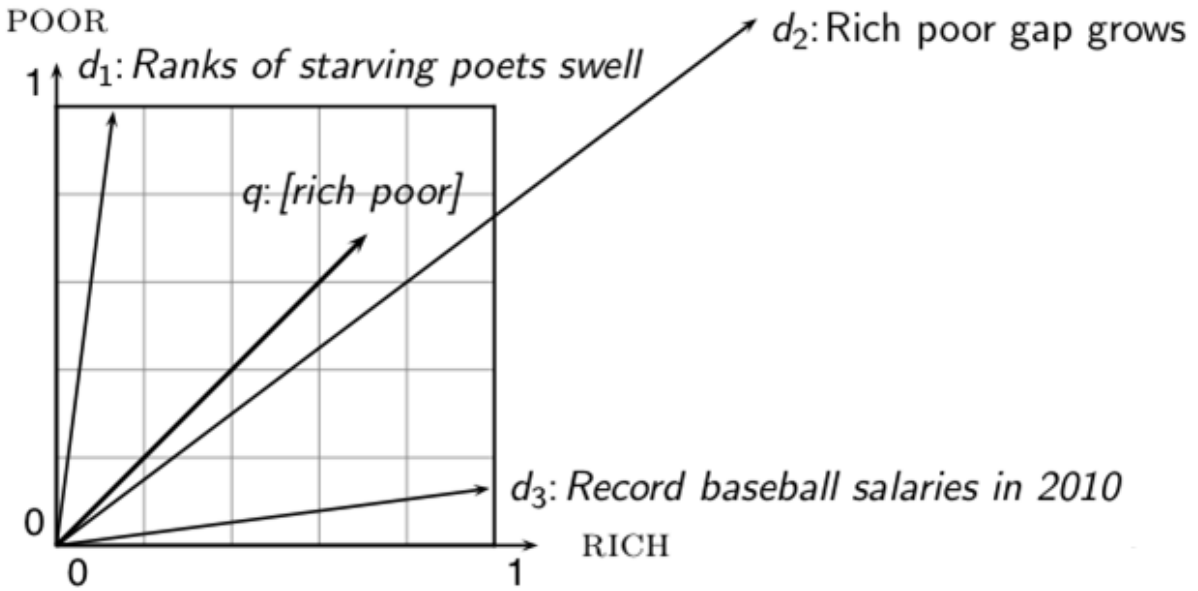
- Define similarity s between documents D_1 and D_2 as

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



- 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)

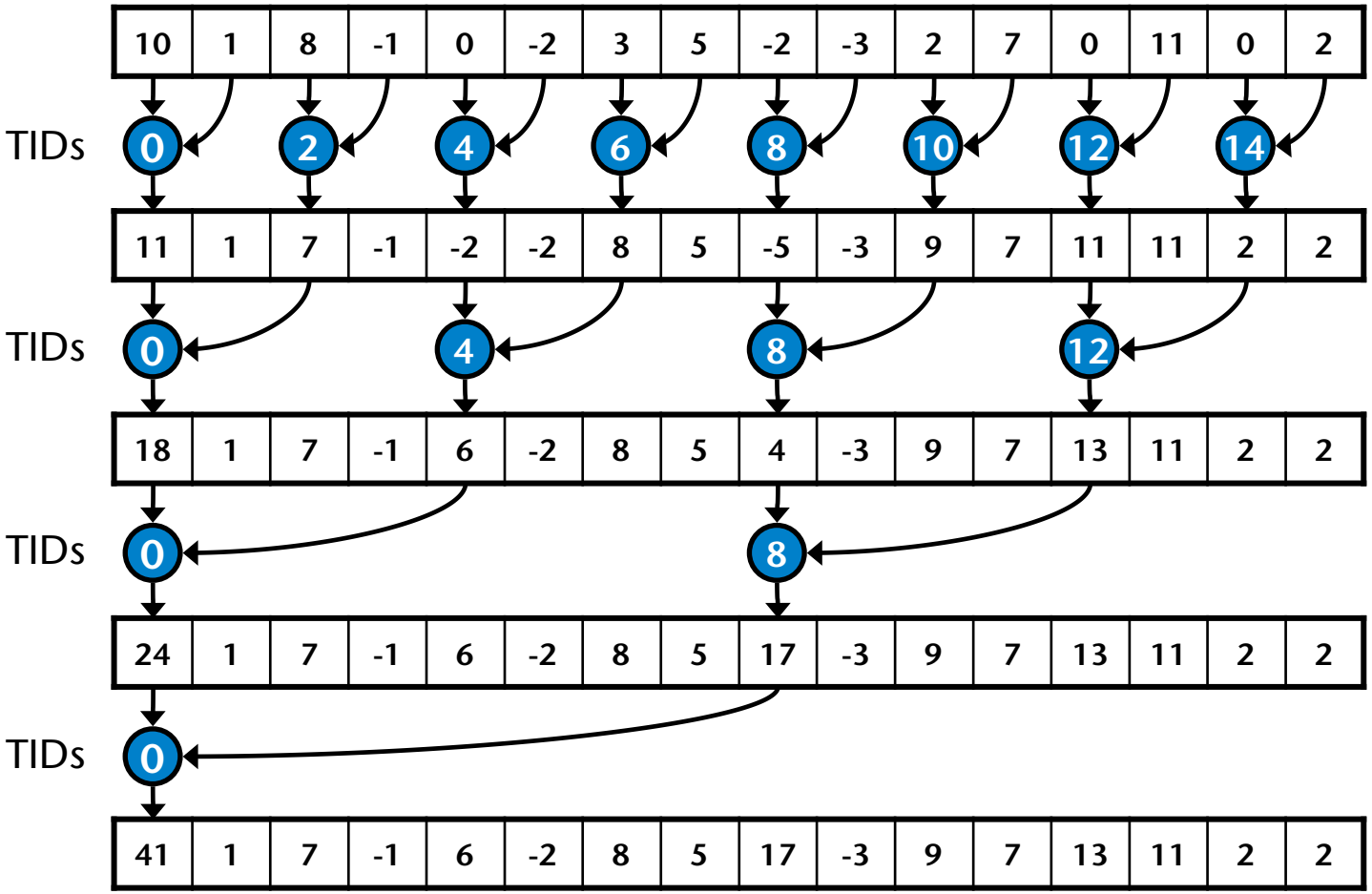
- Why not the Euclidean distance $\|D_1 - D_2\|$?
  - Otherwise: documents D, and D concatenated to itself would be very *dissimilar*!



- 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?

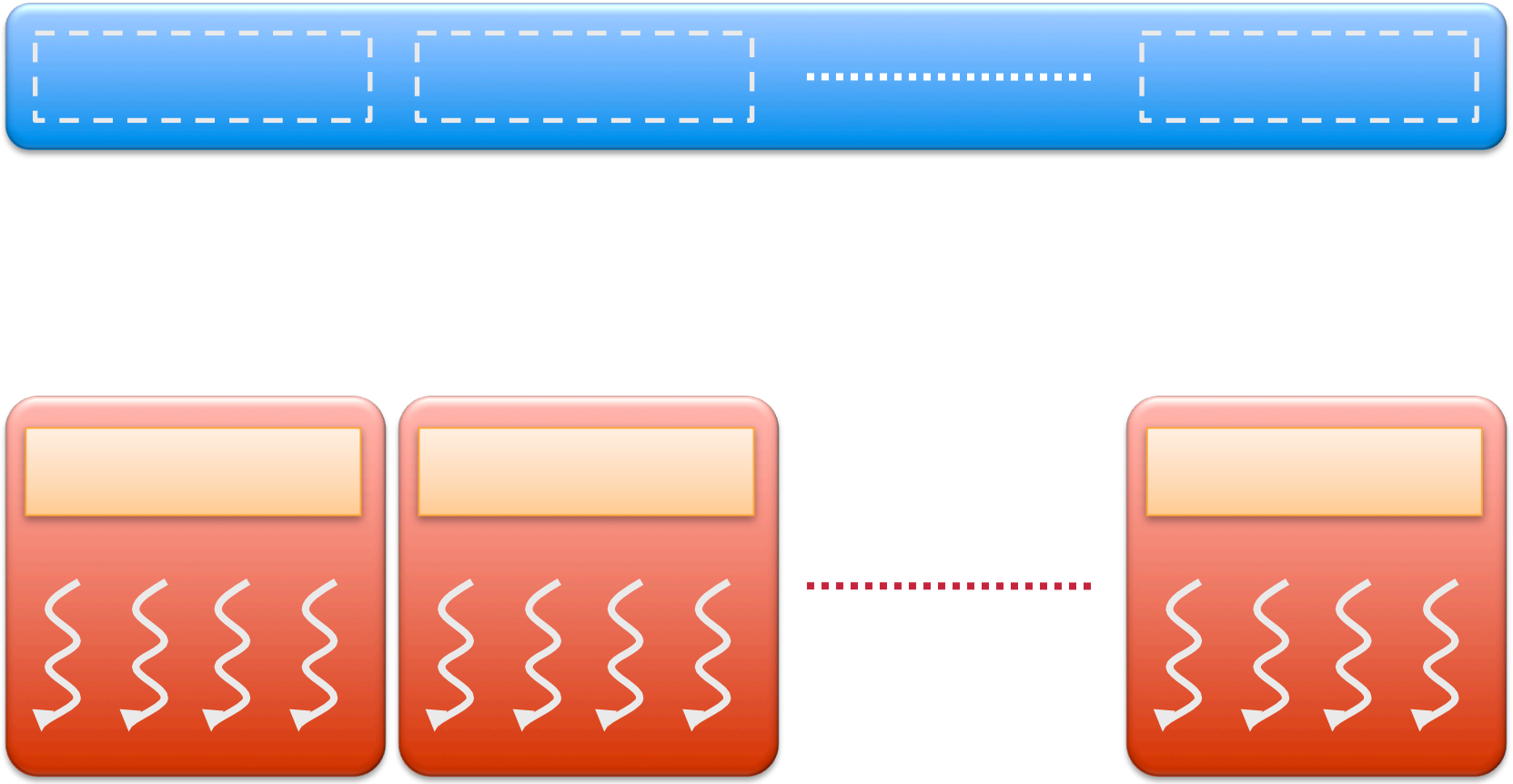


- The kernel for this algorithm:

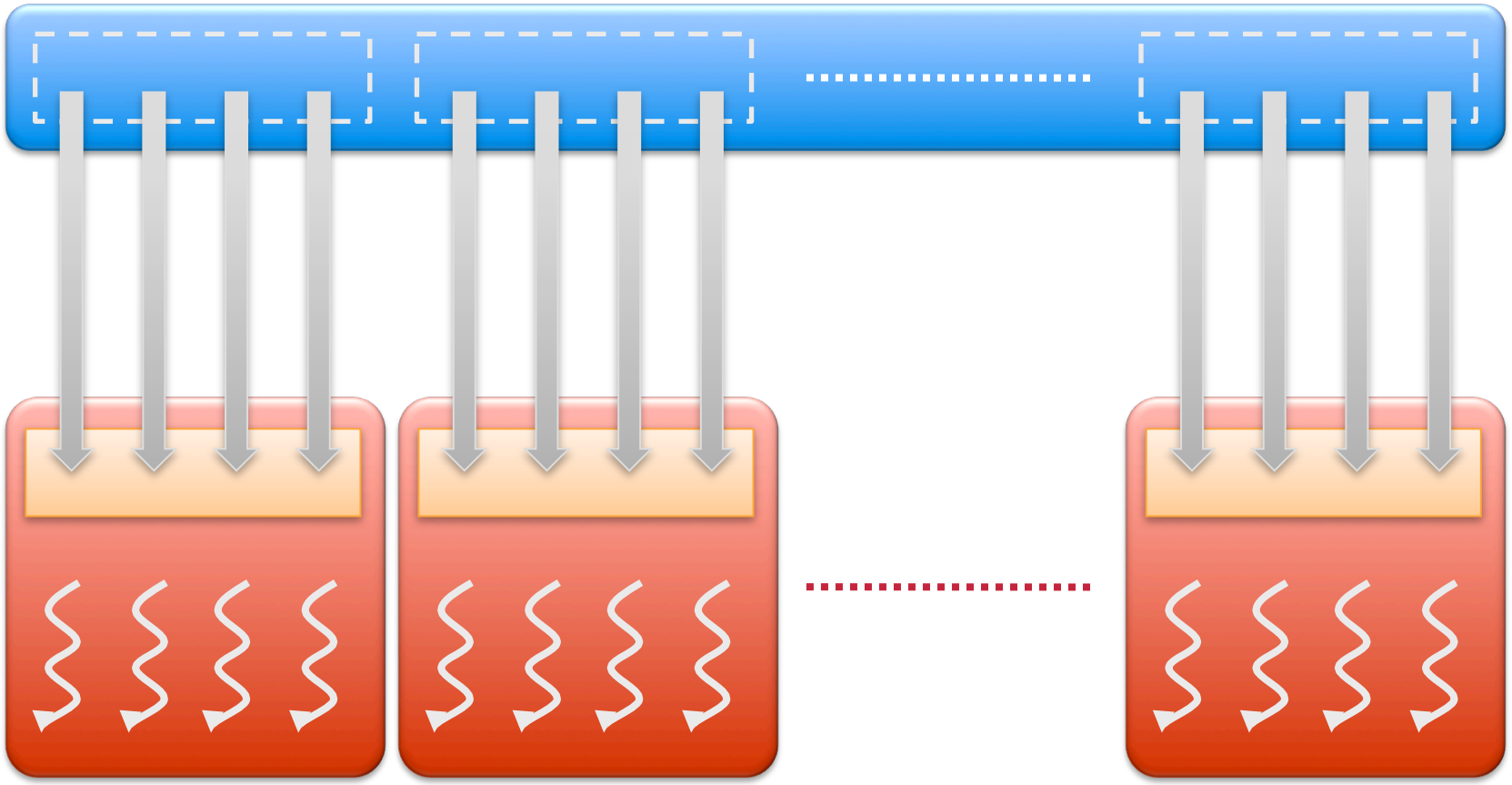
```
// do reduction in shared mem
__syncthreads();
for ( int i = 1; i < blockDim.x; i *= 2 )
{
   if ( threadIdx.x % (2*i) == 0 )
      cache[threadIdx.x] += cache[threadIdx.x + i];
   __syncthreads();
}
```

Problem: highly divergent warps are very inefficient

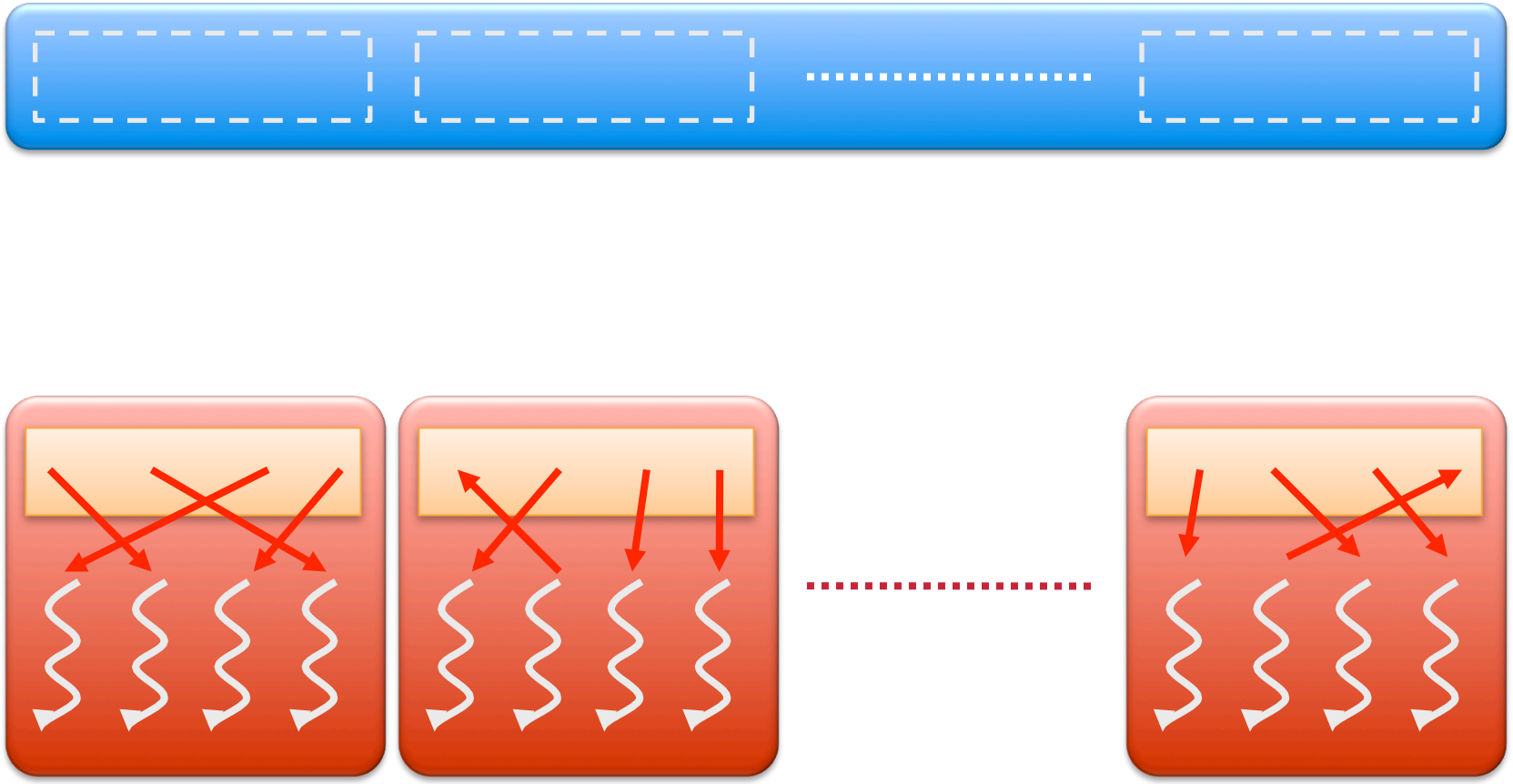
A Common, Massively Parallel Programming Pattern



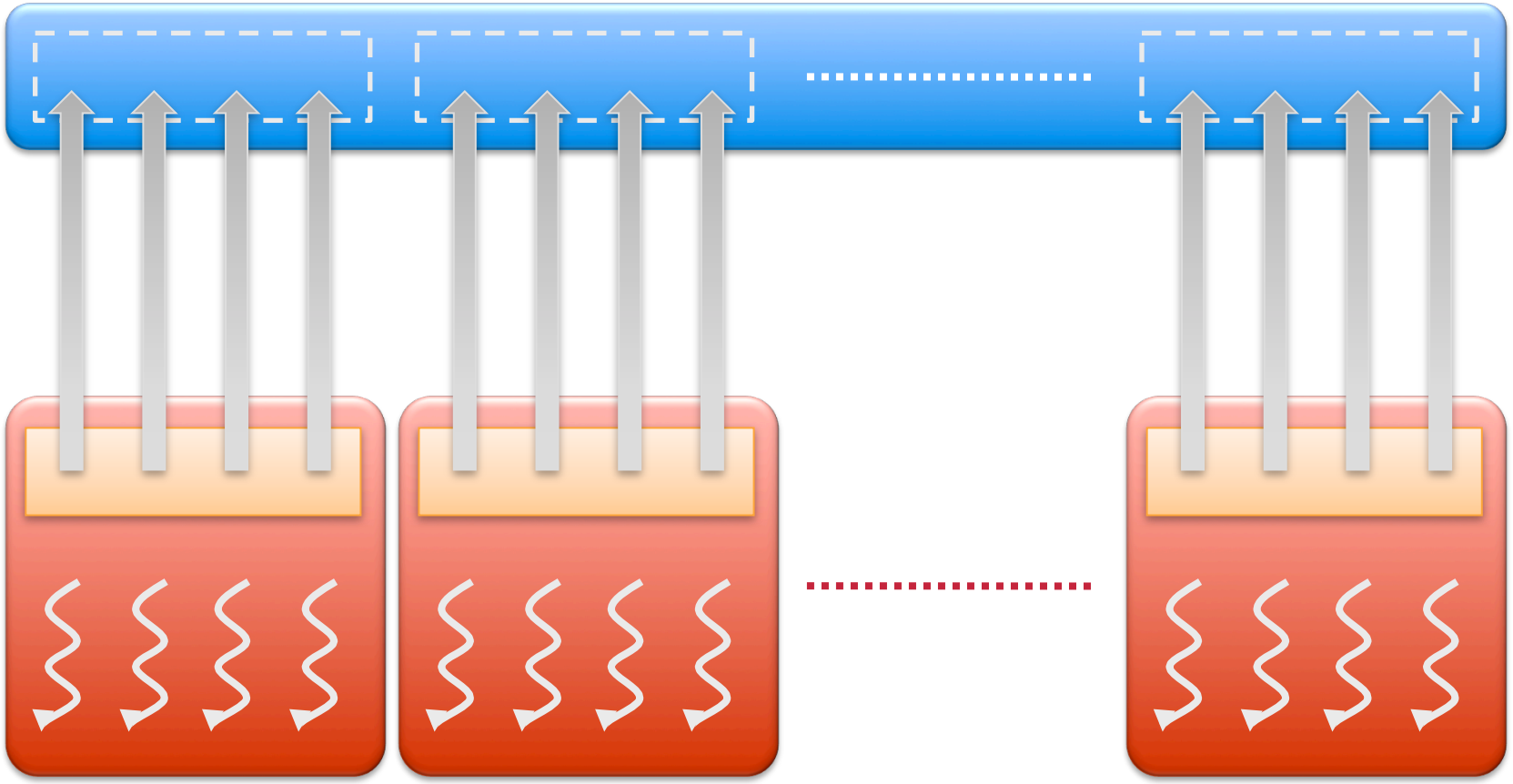
- Partition your domain such that each subset fits into shared memory; handle each data subset with one thread block



- 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

Remarks on Memory (Applies to GPUs and CPUs)

- In our dot product kernel, we could have done everything in global memory, but ...
- Global memory bandwidth is loooow:

Ideal

Reality

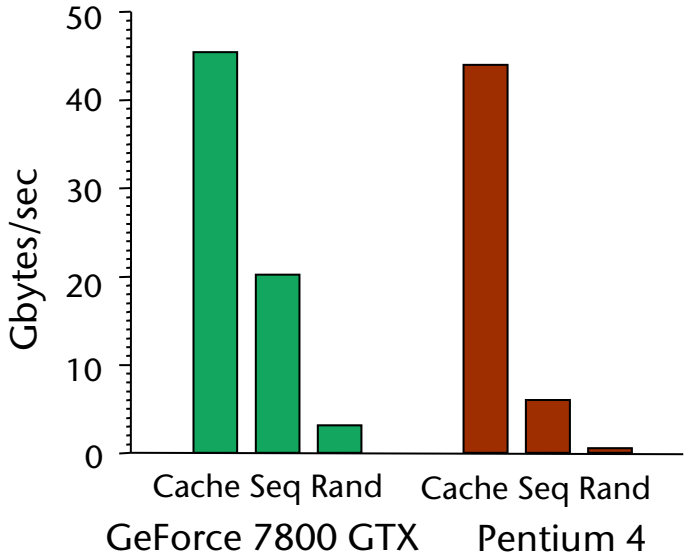
Coalesced Memory Access

- One of the most important optimization techniques for massively parallel algorithm design (on GPUs and — to some degree — CPUs!)

Coalesced memory accesses

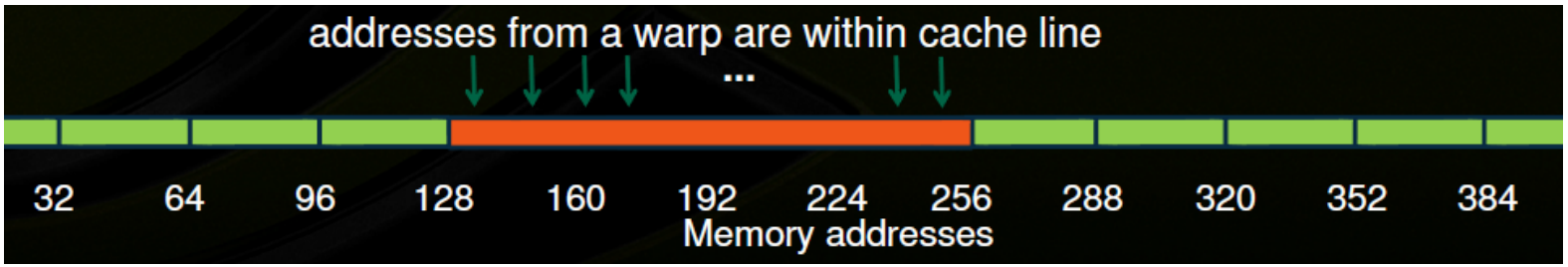
Uncoalesced memory accesses

- When does the GPU win over the CPU?
- Arithmetic intensity of an algorithm := $\frac{\text{number of arithmetic operations}}{\text{amount of transferred bytes}}$
  - Sometimes also called computational intensity
- Unfortunately, many (most?) algorithms have a low arithmetic intensity → they are bandwidth limited
- GPU wins if memory access is "streamed" = coalesced
  - Hence, "stream programming architecture"



How to Achieve Coalesced Access

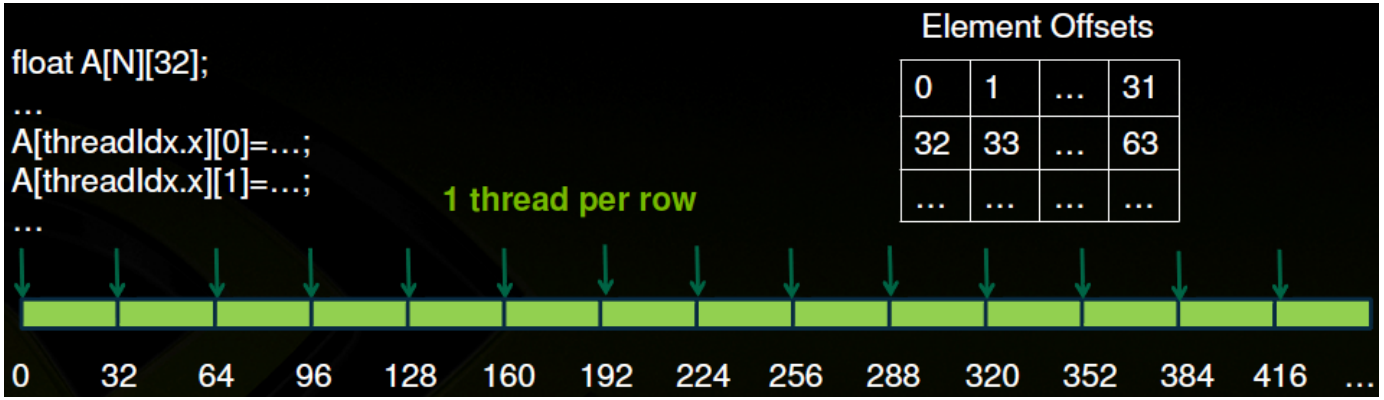
- Addresses from a warp (“thread-vector”) are converted into line requests
  - line sizes: 32B and 128B
  - Goal is to maximally utilize the bytes in these lines



2D Array Access Pattern (row major)

- Consider the following piece in a kernel (e.g., matrix × vector):

```
for ( int j = 0; j < 32; j ++ ) {
   float x = A[treadIdx.x][j];
   ... do something with it ...
```



➢ Uncoalesced access pattern:

- Elements read on 1st SIMT access: 0, 32, 64, ...
- Elements read on 2nd SIMT access: 1, 33, 65, ...
- Also, extra data will be transferred in order to fill the cache line size

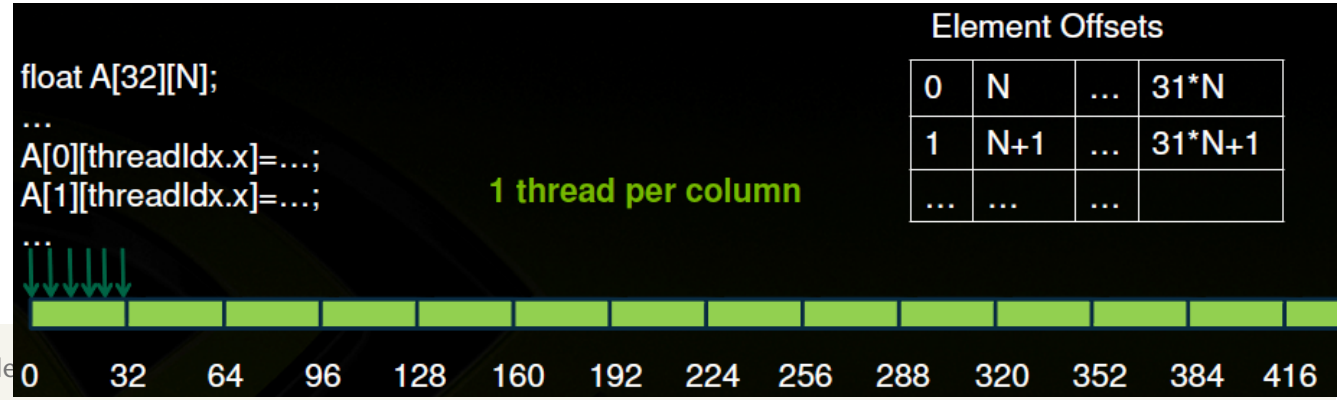
- Generally, most natural access pattern for direct port of a C/C++ code!

Transposed 2D Array Access Pattern

- The "natural" way to store matrices is called row major order
- Column major := store a logical row in a physical column
  - I.e., $A_{00} \rightarrow$ A[0][0] , $A_{01} \rightarrow$ A[1][0] , $A_{02} \rightarrow$ A[2][0] , ...
    $A_{10} \rightarrow$ A[0][1] , $A_{11} \rightarrow$ A[1][1] , $A_{12} \rightarrow$ A[2][1] , ...
    $A_{20} \rightarrow$ A[0][2] , ...
- Transform the code piece (e.g., row×column) to column major:

```
for ( int j = 0; j < 32; j ++ ){
   float x = A[j][treadIdx.x];
   ... do something with it ...
```

- Now, we have coalesced accesses:
  - Elements read on 1st SIMT access: 0, 1, 2, ..., 31
  - Elements read on 2nd SIMT access: 32, 33, ..., 63



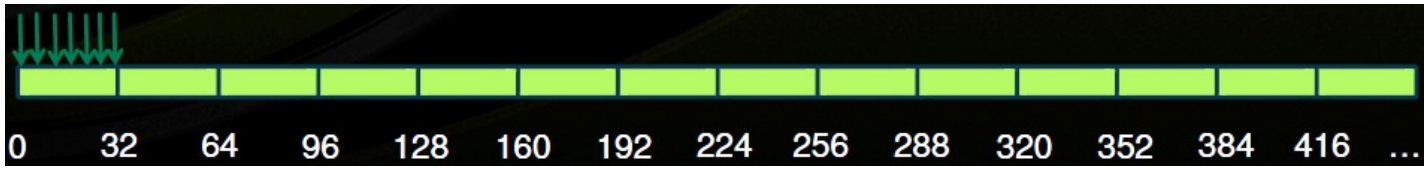
Array of Structure or Structure of Array?

- An array of structures (AoS) behaves like row major accesses:

```
struct Point {
   float x; float y; float z;
};
Point PointList[N];
...
PointList[threadIdx.x].x = ...
```

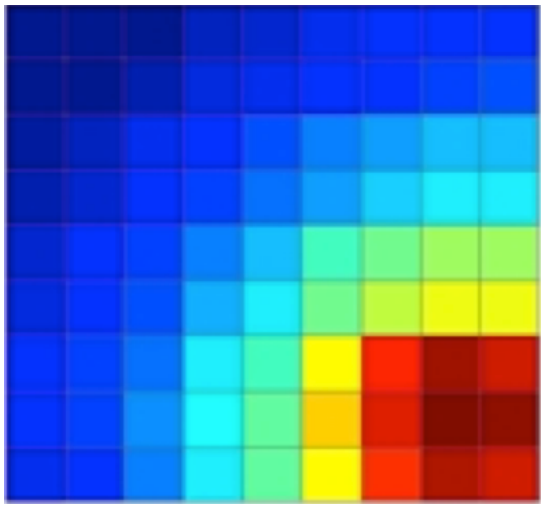
- A structure of arrays (SoA) behaves like column major access:

```
struct PointList {
   float x[N];
   float y[N];
   float z[N];
};
...
PointList.x[threadIdx.x] = ...
```



Simulating Heat Transfer in Solid Bodies

- Assumptions:
  - For sake of illustration, our domain is 2D
  - Discrete domain → 2D grid
    (common approach in simulation)
  - A few designated cells are "heat sources"
    → cells with constant temperature



- Simulation model (simplistic):

$$T_{i,j}^{n+1} = T_{i,j}^{n} + \sum_{(k,l)\in N(i,j)} \mu\left(T_{k,l}^{n} - T_{i,j}^{n}\right)$$

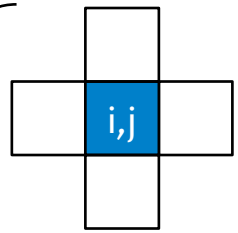
$$\Leftrightarrow \quad T_{i,j}^{n+1} = \left(1 - N\mu\right) T_{i,j}^{n} + \mu \sum_{(k,l)\in N(i,j)} T_{k,l}^{n} \qquad (1)$$

N = number of cells in the neighborhood

- Iterate this (e.g., until convergence to steady-state)

- Do we achieve energy conservation?
- For sake of simplicity, assume

$$N(i,j) = \left\{ \right.$$

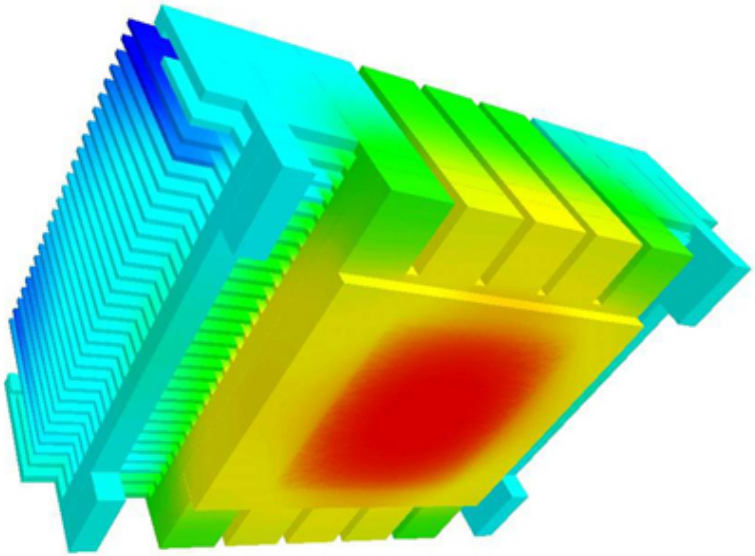


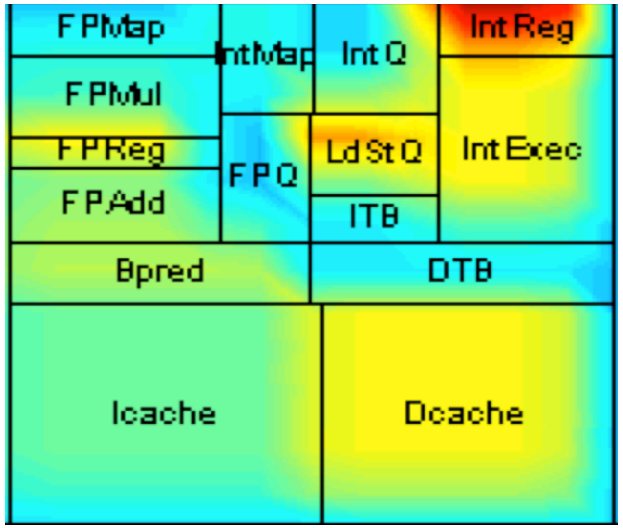
- Energy consumption iff $\sum_{i,j} T_{i,j}^{n+1} \stackrel{!}{=} \sum_{i,j} T_{i,j}^{n}$
- Plugging (1) into (2) yields

$$(1 - N\mu) \sum_{i,j} T_{i,j}^{n} + \mu \underbrace{\sum_{i,j} \sum_{(k,l) \in N(i,j)} T_{k,l}^{n}}_{=0} \stackrel{!}{=} \sum_{i,j} T_{i,j}^{n} \qquad (2)$$

- Therefore, μ is indeed a free material parameter (= "heat flow speed")

- Example: heat simulation of ICs and cooling elements

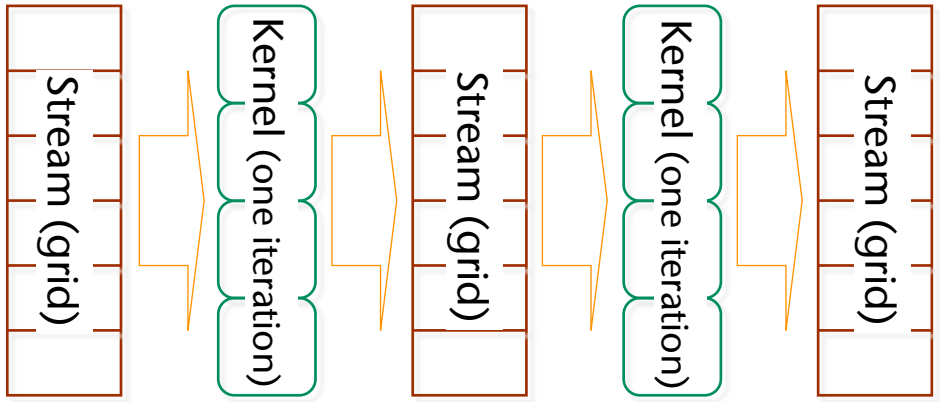




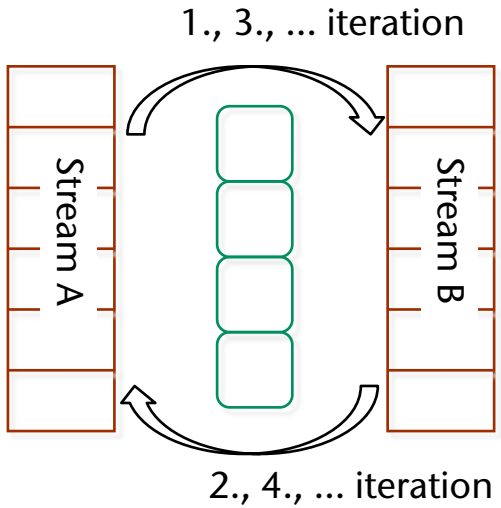
Pattern: Double Buffering

- Observations:
  - Each cell's next state can be computed completely independently

➢ We can arrange our computations like this:



- General parallel programming pattern: double buffering ("ping pong")



Algorithm

- One thread per cell

1. Kernel for re-setting heat sources:

```
if ( cell is heat cell ):
   read temperature from constant "heating stencil"
```

2. Kernel for one transfer step:

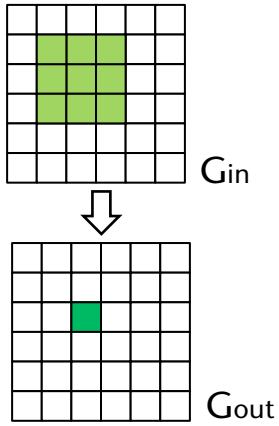
```
Read all neighbor cells  input_grid[tid.x+-1][tid.y+-1]
Accumulate them
Write new temperature in output_grid[tid.x][tid.y]
```

3. Swap pointers to input & output grid (done on host)

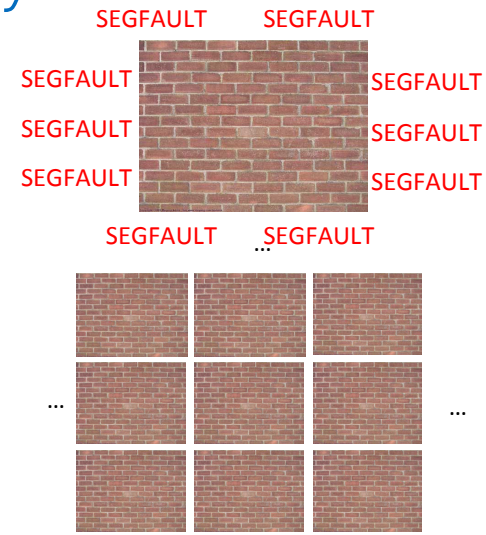
- Challenge: border cells! (very frequent problem in sim. codes)
  - Use if-then-else in above kernel?
  - Use extra kernel that is run only for border cells?
  - Introduce padding around domain? Arrange domain as torus?

Texture Memory

- Many computations have the following characteristics:
  - They *iterate* a simple function many times
  - They work on a 2D/3D *grid*
  - We can run *one thread per grid cell*
  - Each thread only needs to look at *neighbor cells*
  - Each iteration transforms an input grid into an output grid

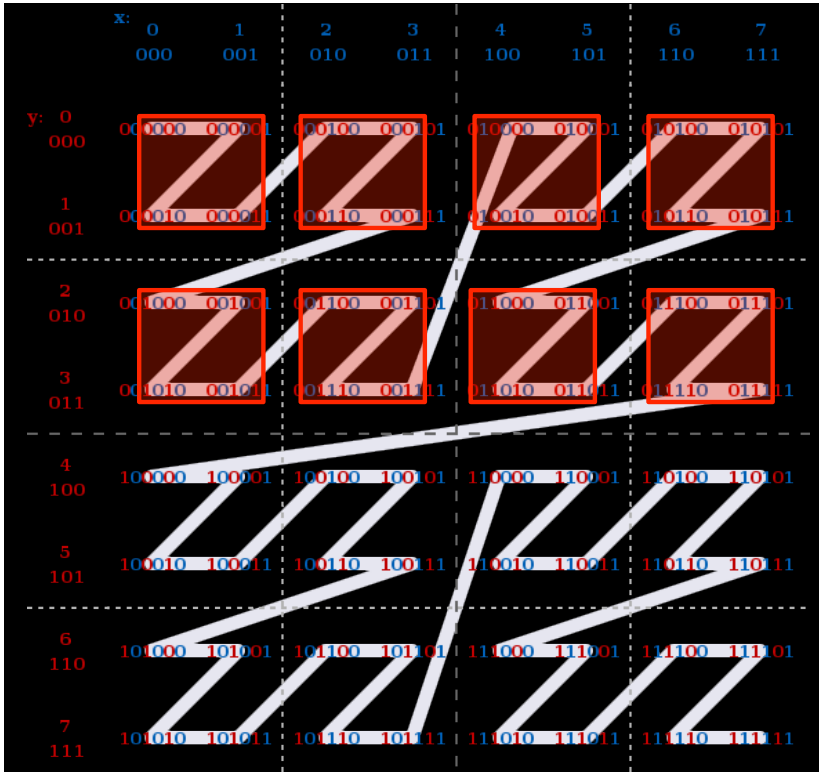


- For this kind of algorithms, there is texture memory:
  - Special cache with optimization for spatial locality
  - Access to neighbor cells is very fast
  - Important: can handle out-of-border accesses automatically by clamping or wrap-around!



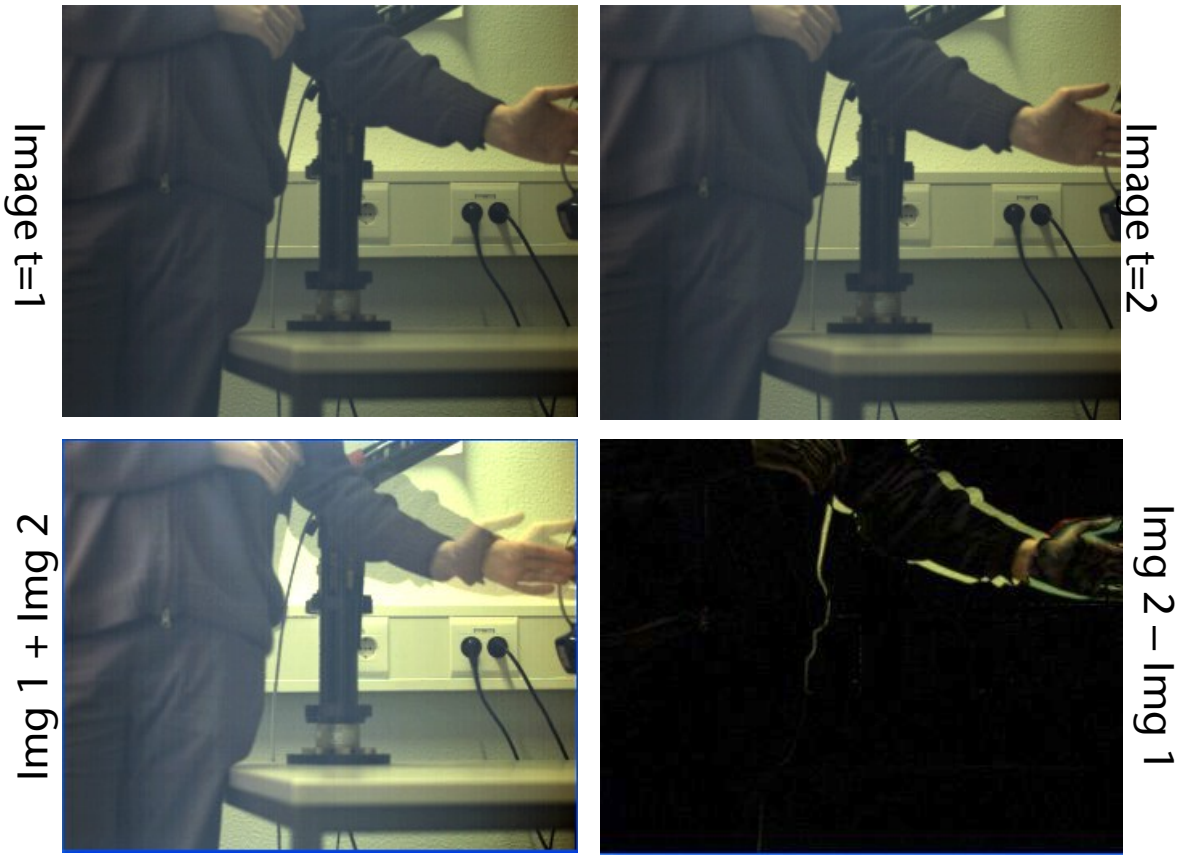
- For the technical details: see "Cuda by Example", Nvidia's "CUDA C Programming Guide",

- The locality-preserving cache is probably achieved by arranging data via a space-filling curve:

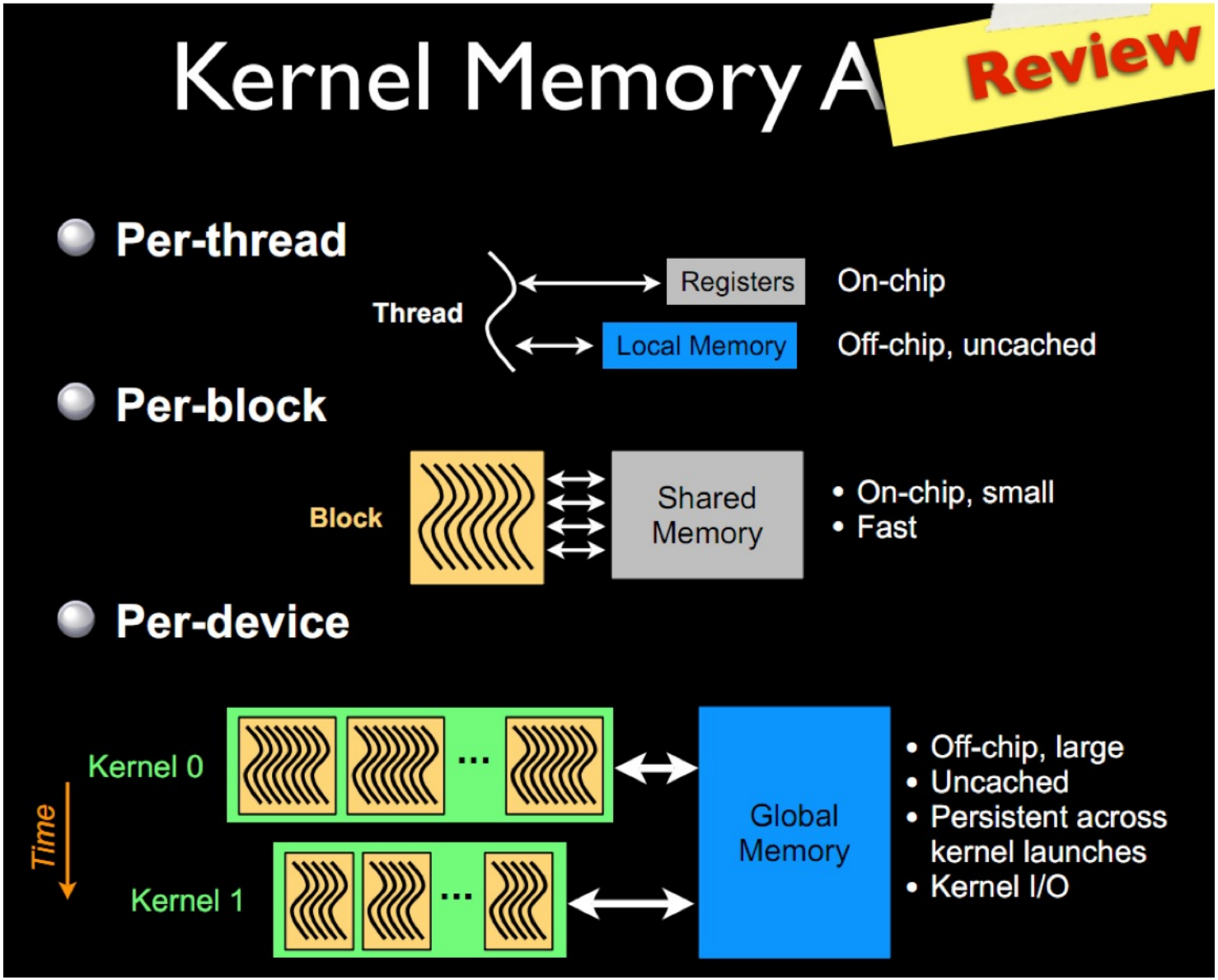


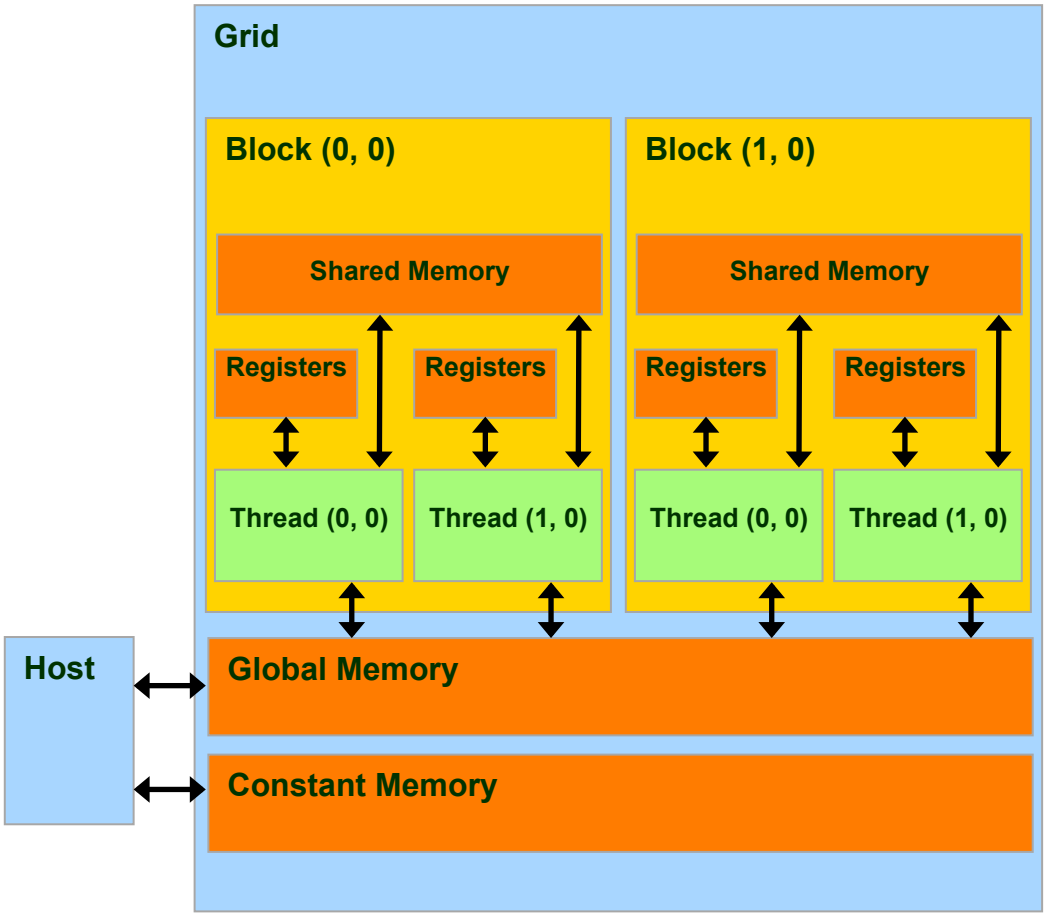
Other Applications of Texture Memory

- Most image processing algorithms exhibit this kind of locality
- Trivial example: image addition / subtraction ⟶ neighboring threads access neighboring pixels



CUDA's Memory Hierarchy




Grid
Block (0, 0)
Block (1, 0)
Shared Memory
Shared Memory
Registers
Registers
Registers
Registers
Thread (0, 0)
Thread (1, 0)
Thread (0, 0)
Thread (1, 0)
Host
Global Memory
Constant Memory

CUDA Variable Type Qualifiers

Variable declaration		Memory	Access	Lifetime
`__device__ __local__`	`int LocalVar;`	local	thread	thread
`__device__ __shared__`	`int SharedVar;`	shared	block	block
`__device__`	`int GlobalVar;`	global	grid	application
`__device__ __constant__`	`int ConstantVar;`	constant	grid	application

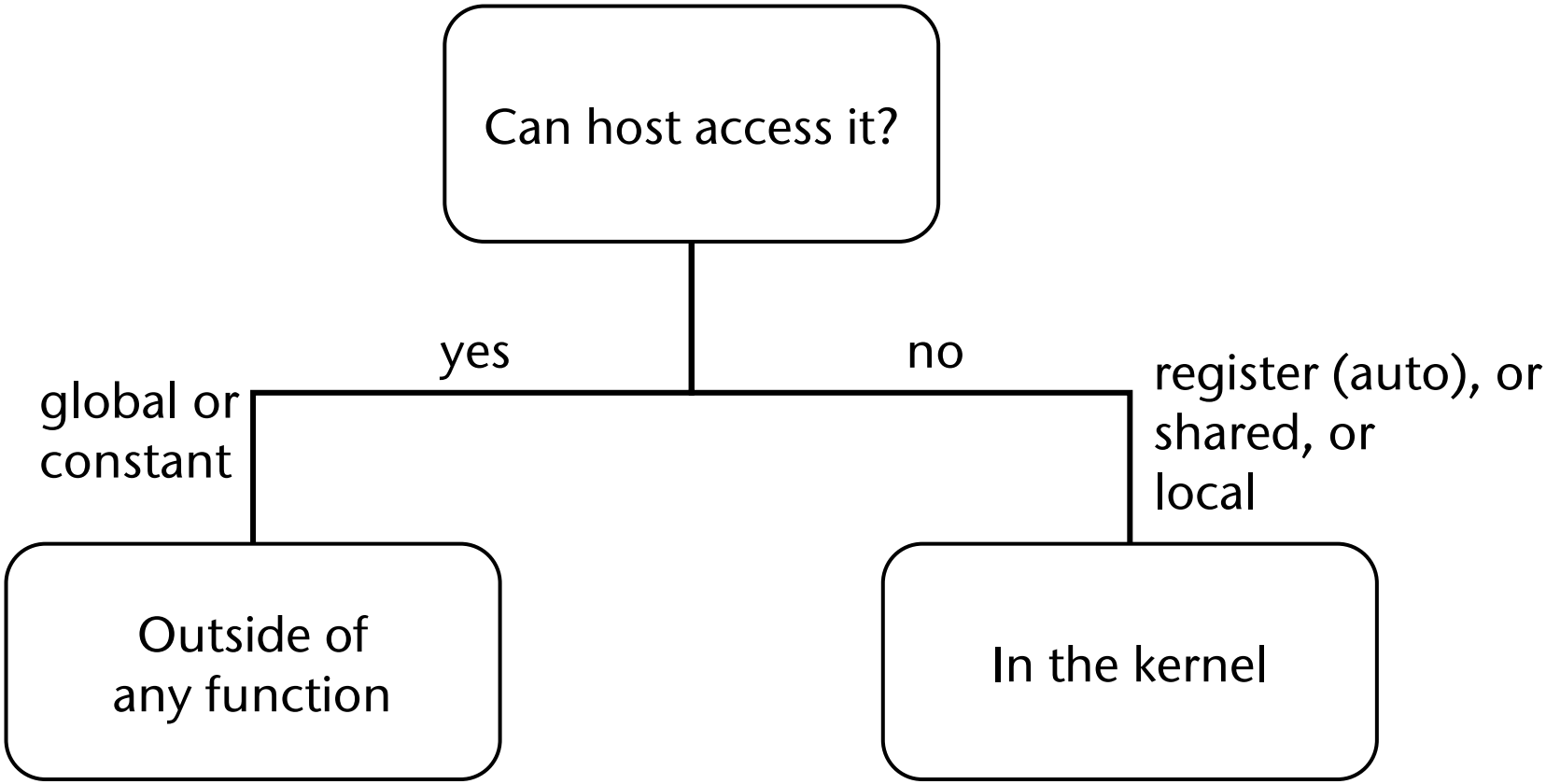
- Remarks:
  - **`__device__`** is optional when used with **`__local__`**, **`__shared__`**, or **`__constant__`**
  - Automatic variables without any qualifier reside in a register
    - Except arrays, which reside in local memory (slow)

CUDA Variable Type Performance

Variable declaration		Memory	Penalty
	`int var;`	register	1x
	`int array_var[10];`	local	100x
`__shared__`	`int shared_var;`	shared	1x
`__device__`	`int global_var;`	global	100x
`__constant__`	`int constant_var;`	constant	1x

- scalar variables reside in fast, on-chip registers
- shared variables reside in fast, on-chip memories
- thread-local arrays & global variables reside in uncached off-chip memory
- constant variables reside in cached off-chip memory

Where to Declare Variables?

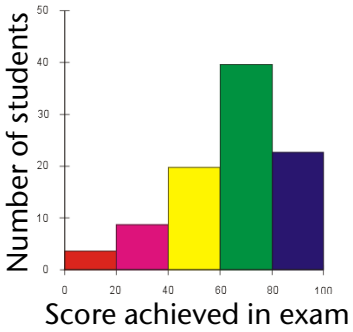


Massively Parallel Histogramm Computation

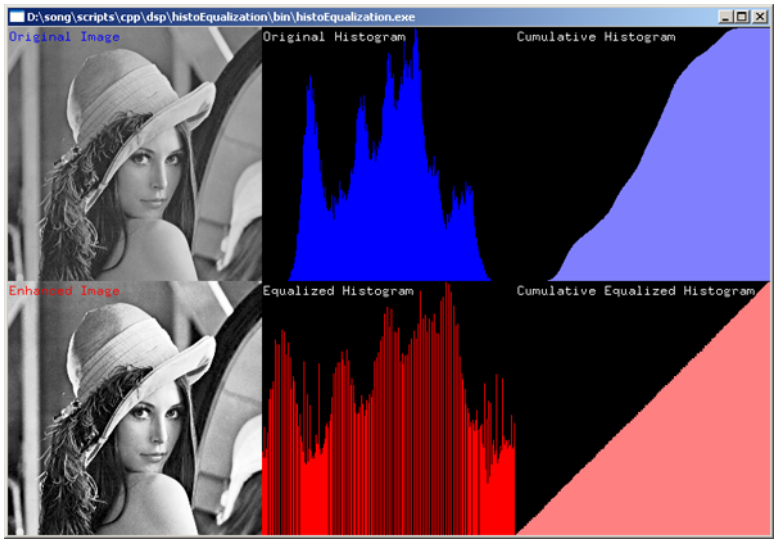
- Definition (for images):

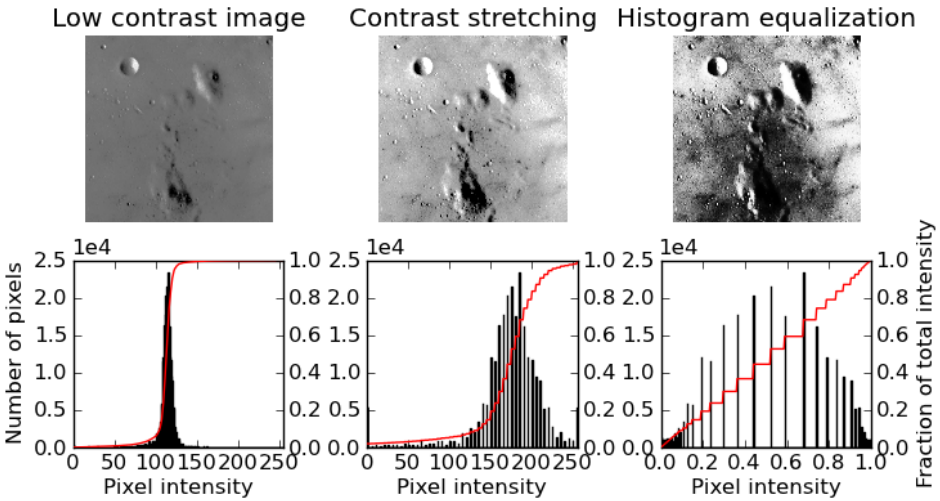
$$h(x) = \#\text{ pixels with level } x$$

$$x \in 0, \ldots, L-1 \qquad L = \#\text{ levels}$$



- Applications: many!
  - Huffman Compression (see Info 2)
  - Image Equalization (see Advanced Computer Graphics)





- The sequential algorithm:

```
unsigned char input[MAX_INP_SIZE];// e.g. image
int input_size;                   // # valid chars in input
unsigned int histogram[256];      // 256 ASCII chars

// clear histogram
for (int i = 0; i < 256; i ++ )
   histogram[i] = 0;
for (int i = 0; i < input_size; i ++ )
   histogram[ input[i] ] ++ ;     // real histogram comput.

// verify histogram
long int total_count = 0;
for (int i = 0; i < 256; i ++ )
   total_count += histogram[i];
if ( total_count != input_size )
   fprintf(stderr, "Error! ..." );
```

- Naïve "massively parallel" algorithm:
  - One thread per bin (e.g., 256)
  - Each thread scans the complete input and counts the number of occurrences of its "own" character
  - At the end, each thread stores its character count in its histogram slot

- Disadvantage: not so massively parallel ...

- New approach: one thread per input character
- The setup on the host side:

```
set up device arrays d_input, d_histogram
cudaMemset( d_histogram, 0, 256 * sizeof(int) );
int threadsPerBlock = 256;
int nBlocks = #(multiprocessors on device) * 2;
computeHistogram <<< nBlocks, threadsPerBlock  >>>
                  ( d_input, input_size, d_histogram );
```

- Notes:
  - Letting **`threadsPerBlock`** = 256 makes things much easier in our case here
  - Letting **`nBlocks`** = (number of multiprocessors [SMXs] on the device) * 2 is a good rule of thumb, YMMV
  - On current hardware (Kepler) → ~ 16384 threads

- The kernel on the device side:

```
__global__ void
computeHistogram( unsigned char * input,
                  long int input_size,
                  unsigned int histogram[256] )
{
   int i = threadIdx.x + blockIdx.x * blockDim.x;
   int stride = blockDim.x * gridDim.x;
   while ( i < input_size )
   {
      histogram[ input[i] ] += 1;
      i += stride;
   }
}
```

- Problem: race condition!!

Solution: Atomic Operations

- The kernel with atomic add:

```
__global__ void
computeHistogram( unsigned char * input,
                  long int input_size,
                  unsigned int histogram[256] )
{
   int i = threadIdx.x + blockIdx.x * blockDim.x;
   int stride = blockDim.x * gridDim.x;
   while ( i < input_size )
   {
      atomicAdd( & histogram[input[i]], 1 );
      i += stride;
   }
}
```

- Prototype of atomicAdd():

```
T atomicAdd( T * address, T val )
```

  - where `T` can be **`int`**, **`float`** (and few others)

- Semantic: *while atomicAdd performs its operation on* address, *no other thread can access this memory location*! *(neither read, nor write)*
- Problem: this algorithm is much slower than the sequential one!

  - Lesson: always measure performance against CPU!
- Cause: congestion
  - *Lots* of threads waiting for a *few* memory locations to become available

- Remedy: partial histograms in shared memory

```
computeHistogram( unsigned char * input,
                  long int input_size,
                  unsigned int histogram[256] )
{
   __shared__ unsigned int partial_histo[256];

   partial_histo[ threadIdx.x ] = 0;
   __syncthreads();

   int i = threadIdx.x + blockIdx.x * blockDim.x;
   int stride = blockDim.x * gridDim.x;
   while ( i < input_size ) {
      atomicAdd( & partial_histo[input[i]], 1 );
      i += stride;
   }
   __syncthreads();
   atomicAdd( & histogram[threadIdx.x],
              partial_histo[input[i]], 1 );
}
```

  - Note: now it's obvious why we chose 256 threads/block

More Atomic Operations

- All programming languages / libraries / environments providing for some kind of parallelism/concurrency have one or more of these atomic operations:
  - **`int atomicExch( int* address, int val )`:**
    Read old value at address, store val in address, return old value
  - **`int atomicMin( int* address, int val )`:**
    Read old value at address, compute minimum of old and val, store result in address, return old value
  - **`int atomicAnd( int* address, int val );`**
  - Atomic add
  - And atomic compare-and-swap ...

- The fundamental atomic operation "Compare And Swap":
  - In CUDA: **`int atomicCAS( int* address, int compare, int val )`**
  - Performs this little algorithm atomically:

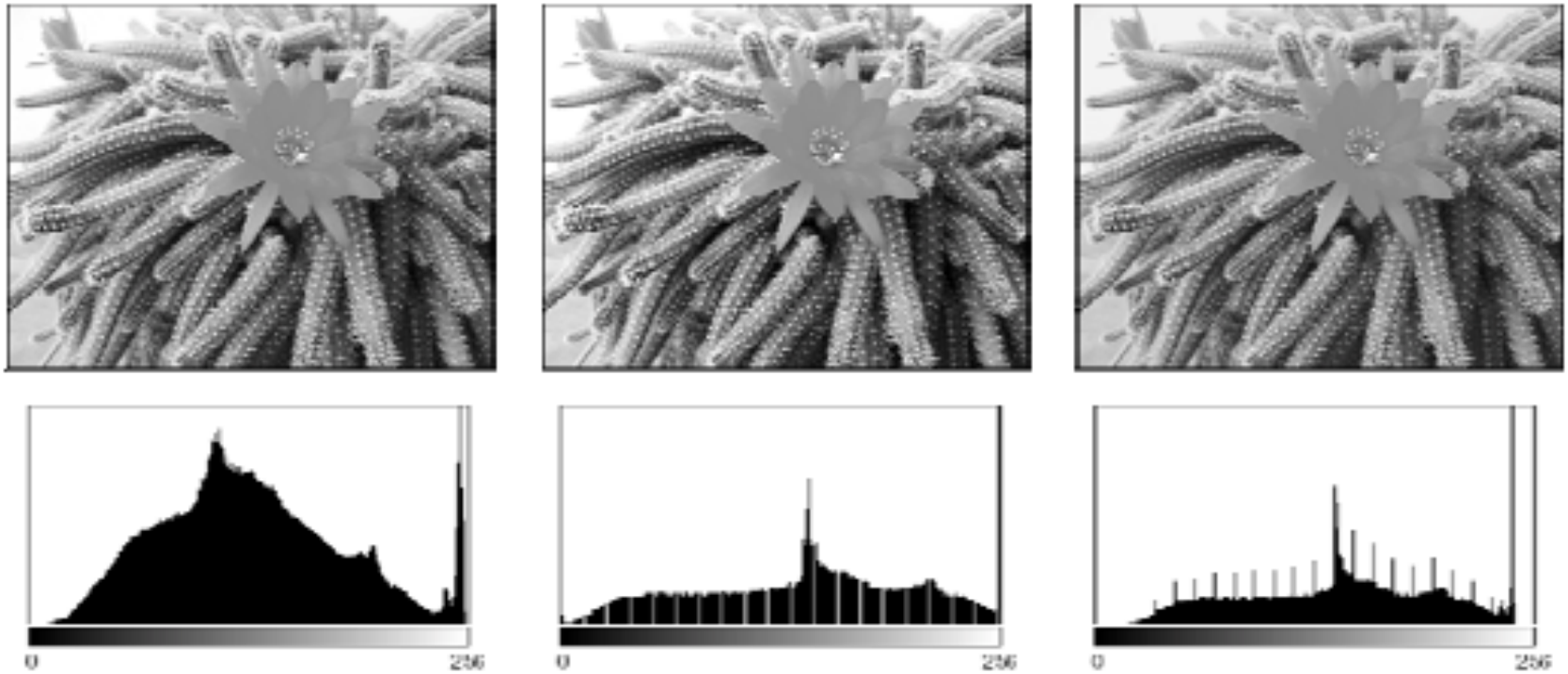
```
atomic_compare_and_swap( address, compare, new_val ):
    old ← value in memory location address
    if compare == old:
        store new_val → memory location address
    return old
```

- Theorem (w/o proof):
  All other atomic operations can be implemented using atomic compare-and-swap.

- Example:

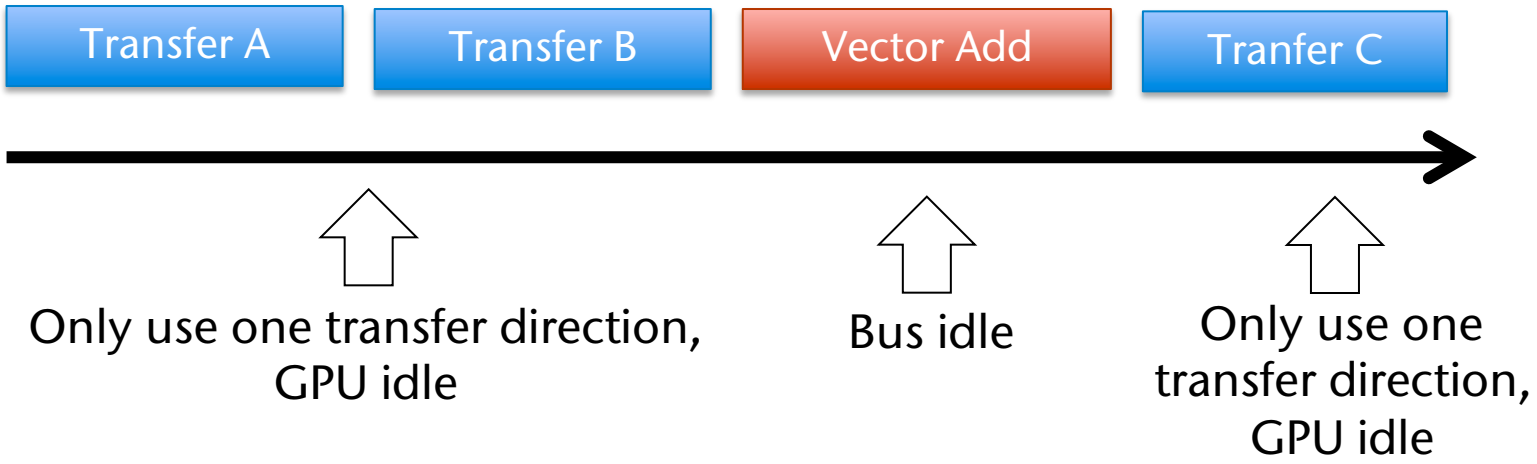
```
atomic_add( address, incr ):
   current_val := value in memory location address
   repeat
      new_val      := current_val + incr
      assumed_val := current_val
      current_val := compare_and_swap( address,
                                       assumed_val,
                                       new_val )
   until assumed_val == current_val
```

Image Restoration Using Histograms

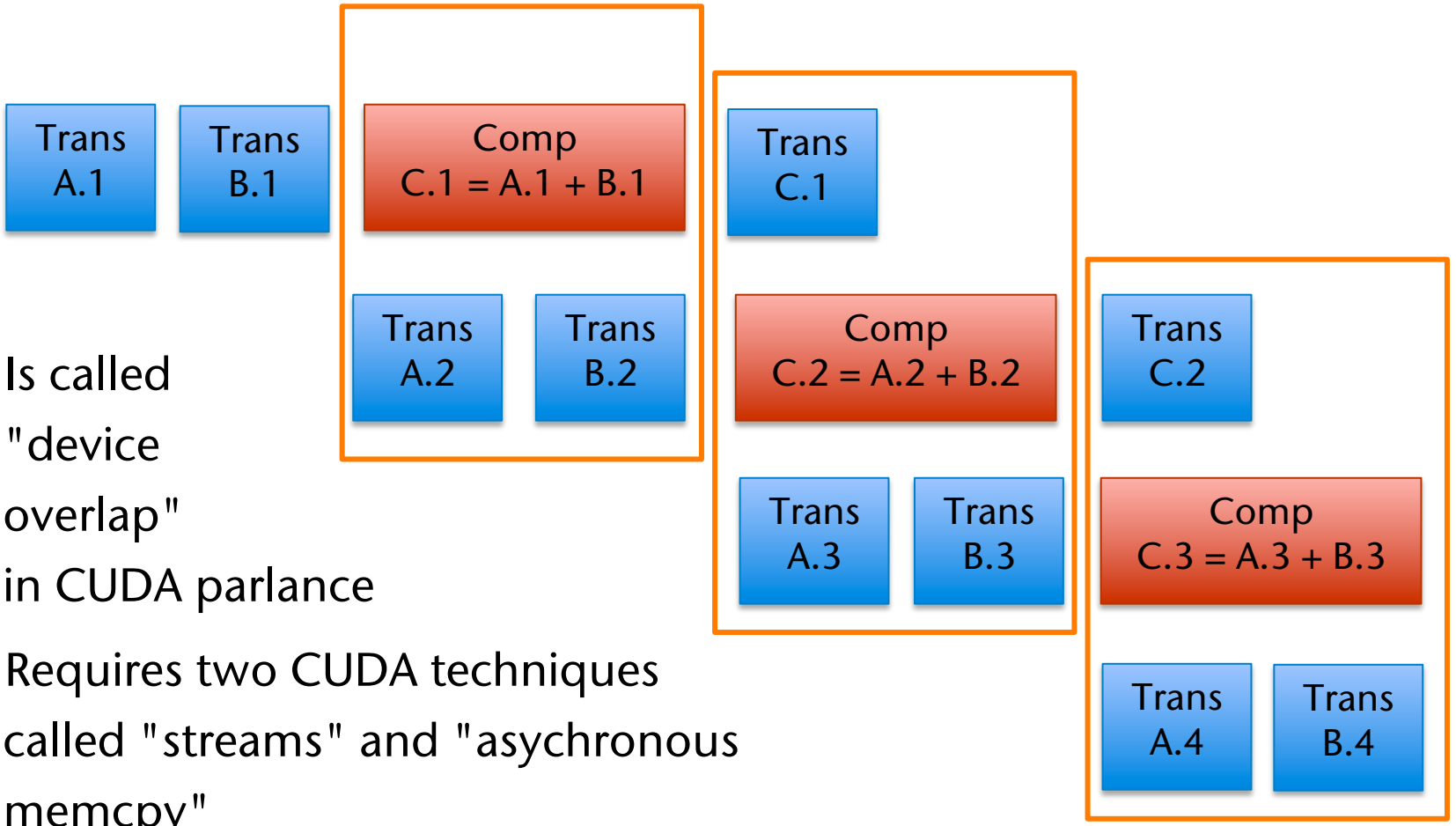


Advanced GPU & Bus Utilization

- Problem with performance, if lots of transfer between GPU↔CPU:



- Solution: pipelining (the "other" *parallelism paradigm*)



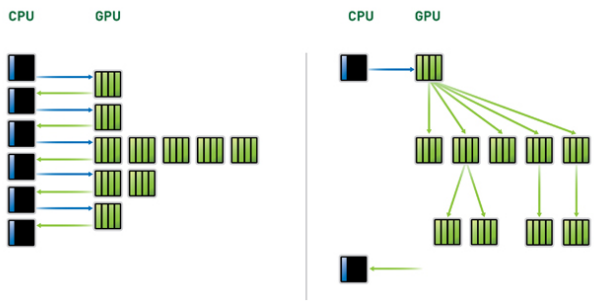
- Is called "device overlap" in CUDA parlance
- Requires two CUDA techniques called "streams" and "asychronous memcpy"

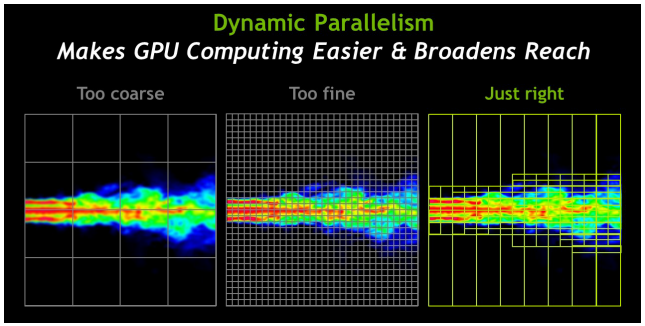
For More Information on CUDA ...

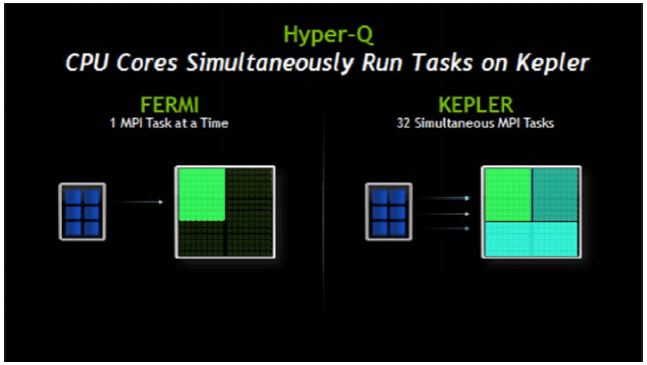
- *CUDA C Programming Guide* (zur Programmiersprache)
- *CUDA C Best Practices Guide* (zur Performance-Steigerung)
- */Developer/NVIDIA/CUDA-5.0/doc/html/index.html*
  (zum Runtime API)

Concepts we Have Not Covered Here

- Dynamic parallelism (threads can launch new threads)
  - Good for irregular data parallelism (e.g., tree traversal, multi-grids)
- Running several tasks at the same time on a GPU (via MPI; they call it "Hyper-Q")



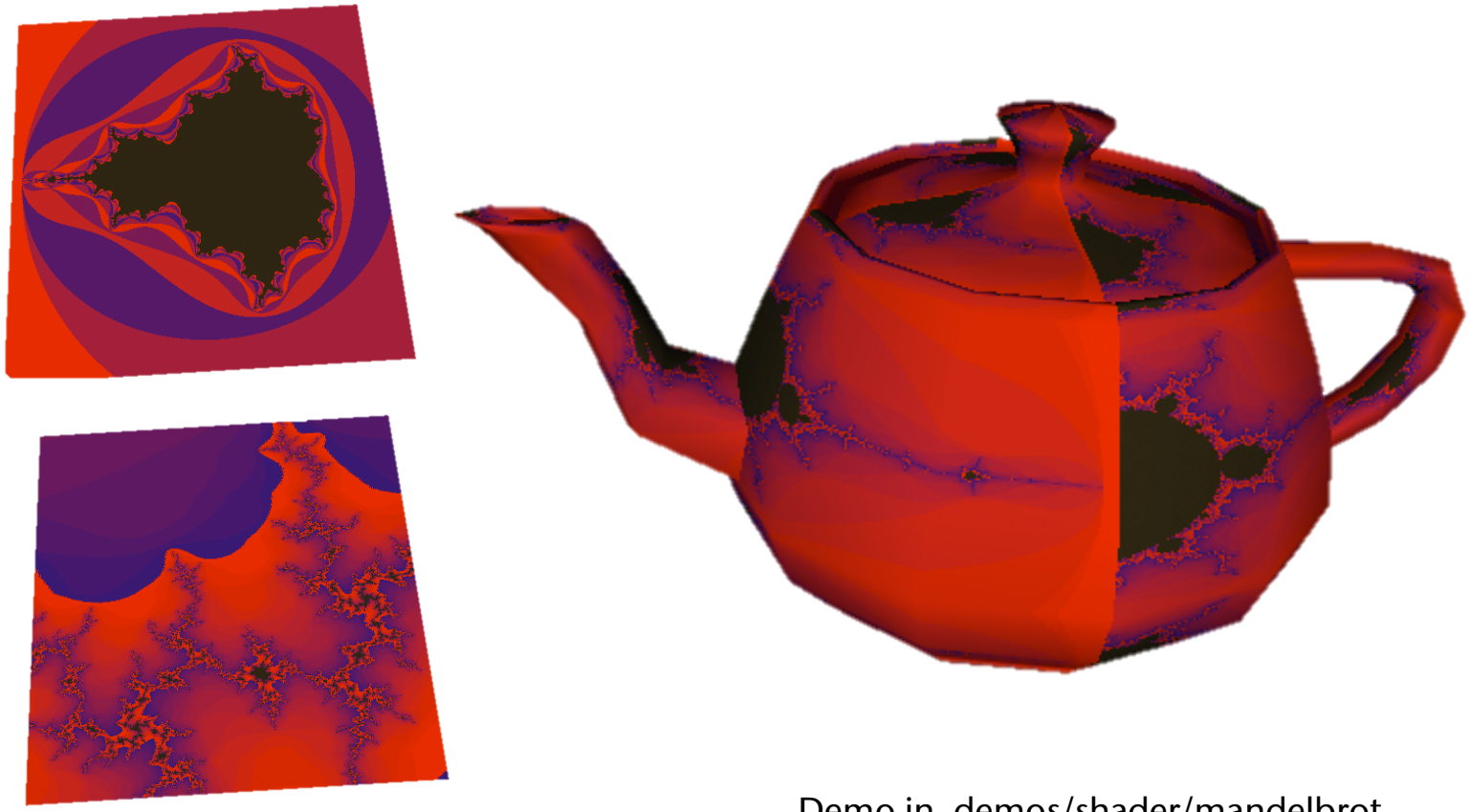




- See:
  - "Introduction to CUDA 5.0" on the course web page
  - "CUDA C Programming Guide" at docs.nvidia.com/cuda/index.html

- Graphics Interoperability:
  - Transfer images directly from CUDA memory to OpenGL's framebuffer
- Dynamic shared memory
- Asynchronous memory copies between host ↔ device
- Dynamic memory allocation in the kernel
  - Can have serious performance issues
- Pinned CPU memory (
- CUDA Streams
- Multi-GPU programming, GPU-to-GPU memory transfer
- Zero-copy data transfer
- Libraries: CUBLAS, Thrust, ...
- Voting functions (__all(), __any())

- With Graphics Interoperability, you can render results from CUDA directly in a 3D scene, e.g. by using them as textures



Demo in demos/shader/mandelbrot