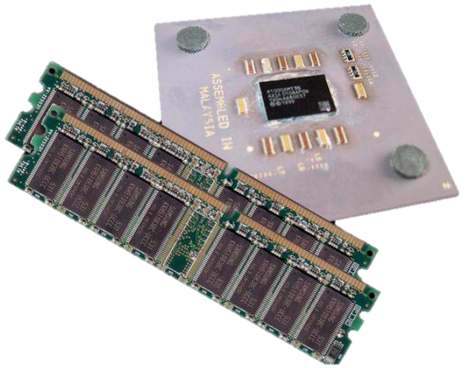
Massively Parallel Algorithms

Introduction to CUDA & Many Fundamental Concepts of Parallel Programming

G. Zachmann
University of Bremen, Germany
cgvr.cs.uni-bremen.de

Hybrid/Heterogeneous Computation/Architecture

- In the future, we'll compute (number-crunching stuff) on both CPU and GPU
- GPU = Graphics Processing Unit
  GPGPU = General Purpose Graphics Processing Unit
- Terminology:
  - Host = CPU and its memory (host memory)
  - Device = GPU and its memory (device memory)



Hello World

- Our first CUDA program:

```
#include <stdio.h>

int main( void )
{
    printf( "Hello World!\n");

    return 0;
}
```

- Compilation:

```
% nvcc -arch=compute_60 -code=sm_70  helloworld.cu –o helloworld
```

- Execution:

```
% ./helloworld
```

- Details (e.g., setting of search paths) will be explained in the lab!

- Now for the real *hello world* program:

```
__global__
void printFromGPU( void )
{
    printf( "world" );
}

int main( void )
{
    printf( "Hello " );
    printFromGPU<<<1,16>>>();          // kernel launch
    cudaDeviceSynchronize();           // important
    printf( "!\n" );
    return 0;
}
```

- Limitations to GPU-side **printf()** apply: see B.16.2 in the *CUDA C Programming Guide* !

New Terminology, New Syntax

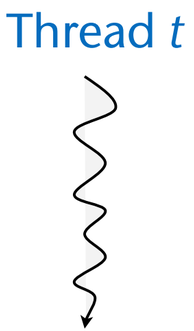
- Kernel := function/program code that is executed on the *device*
  - Syntax for *defining* kernels:

```
__global__ void kernel( parameters )
{
    ... regular C code ...
}
```

    - Note: kernels cannot return a value! → void
    - Kernels can take arguments (using regular C syntax)
  - Syntax for *calling* kernels:

```
kernel<<<b,t>>>( params );
```

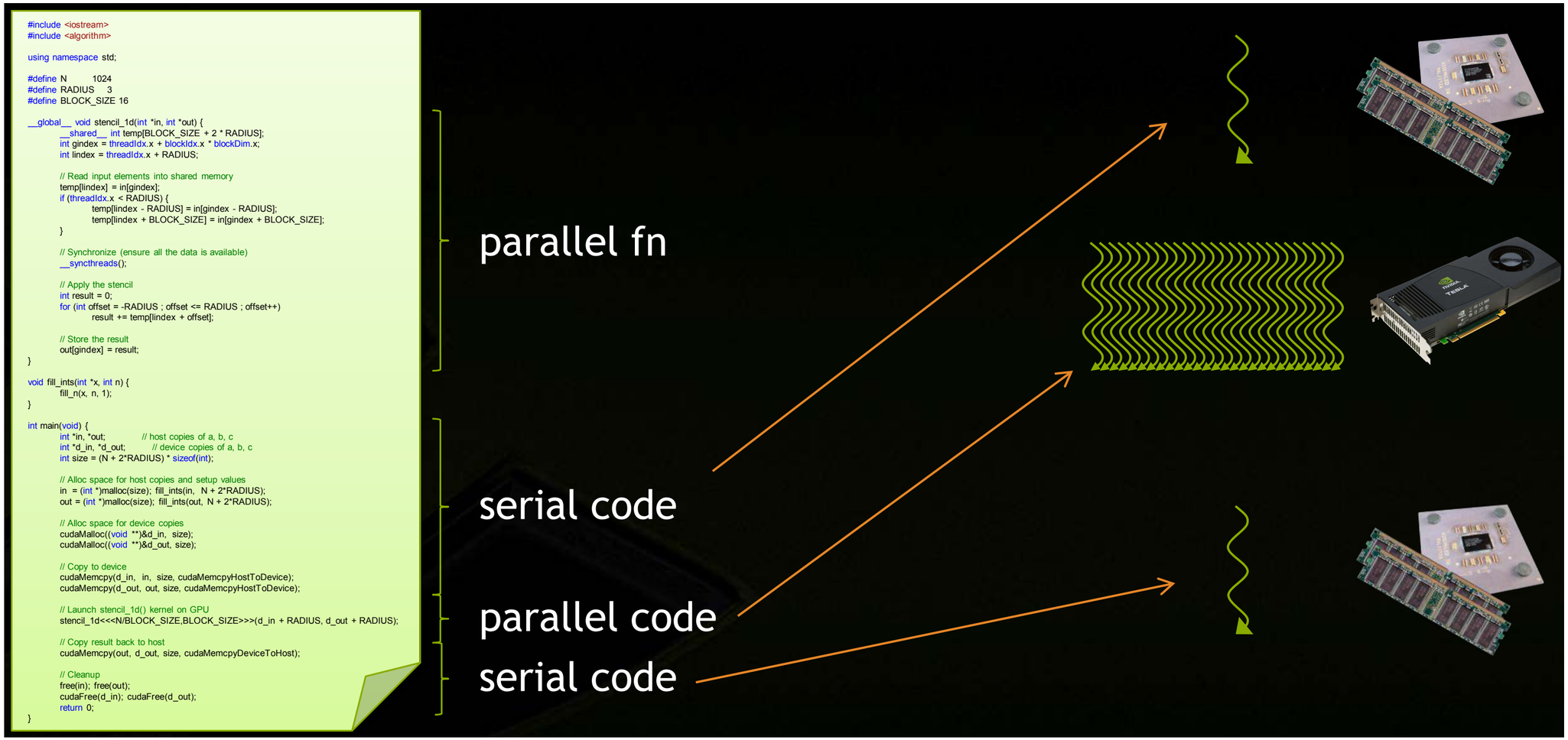
    - Starts b×t many threads in parallel



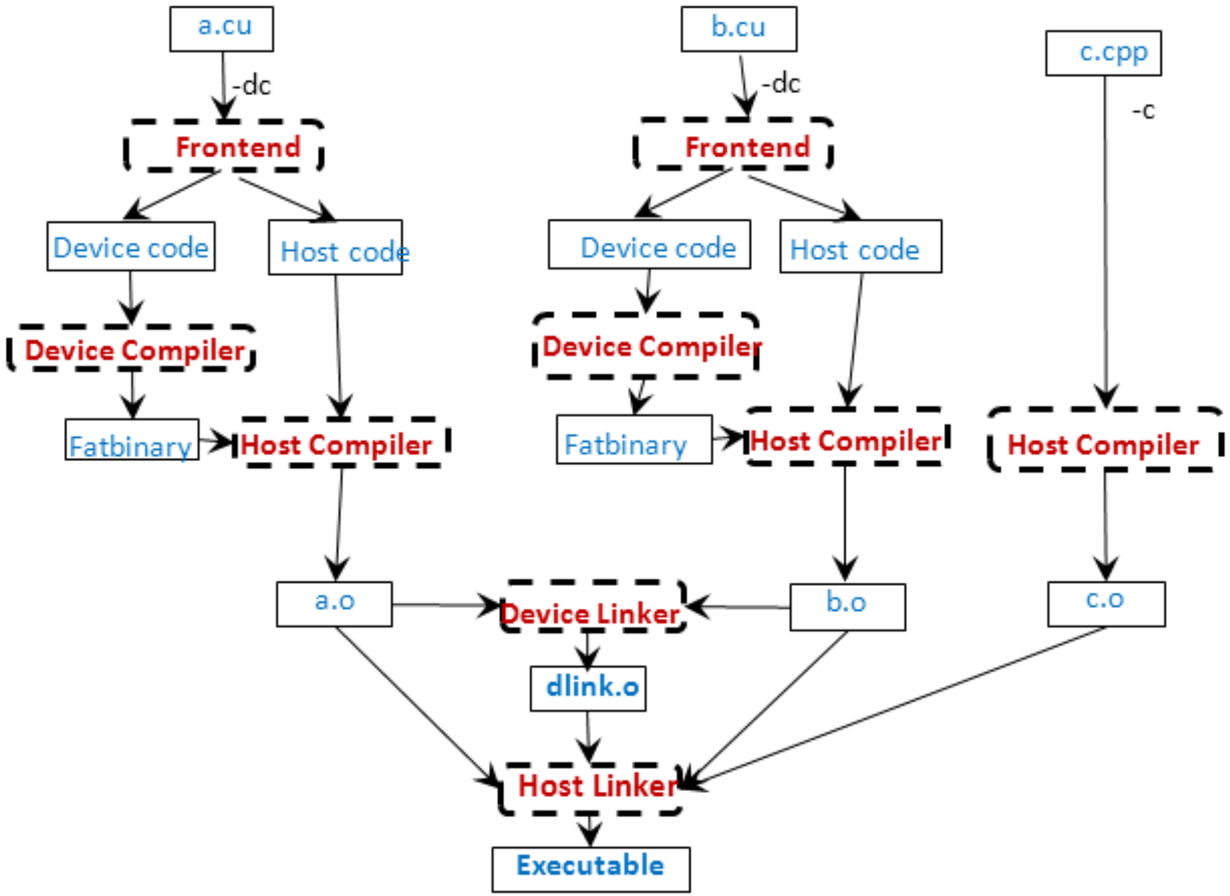
- Thread := one "process" (out of many) executing the **same** kernel
  - Think of multiple copies of the same function (kernel)

Typical Control Flow in Heterogeneous Computing

```
#include <iostream>
#include <algorithm>

using namespace std;

#define N          1024
#define RADIUS     3
#define BLOCK_SIZE 16

__global__ void stencil_1d(int *in, int *out) {
        __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
        int gindex = threadIdx.x + blockIdx.x * blockDim.x;
        int lindex = threadIdx.x + RADIUS;

        // Read input elements into shared memory
        temp[lindex] = in[gindex];
        if (threadIdx.x < RADIUS) {
                temp[lindex - RADIUS] = in[gindex - RADIUS];
                temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
        }

        // Synchronize (ensure all the data is available)
        __syncthreads();

        // Apply the stencil
        int result = 0;
        for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
                result += temp[lindex + offset];

        // Store the result
        out[gindex] = result;
}

void fill_ints(int *x, int n) {
        fill_n(x, n, 1);
}

int main(void) {
        int *in, *out;          // host copies of a, b, c
        int *d_in, *d_out;      // device copies of a, b, c
        int size = (N + 2*RADIUS) * sizeof(int);

        // Alloc space for host copies and setup values
        in  = (int *)malloc(size); fill_ints(in,  N + 2*RADIUS);
        out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

        // Alloc space for device copies
        cudaMalloc((void **)&d_in,  size);
        cudaMalloc((void **)&d_out, size);

        // Copy to device
        cudaMemcpy(d_in,  in,  size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

        // Launch stencil_1d() kernel on GPU
        stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS, d_out + RADIUS);

        // Copy result back to host
        cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

        // Cleanup
        free(in); free(out);
        cudaFree(d_in); cudaFree(d_out);
        return 0;
}
```

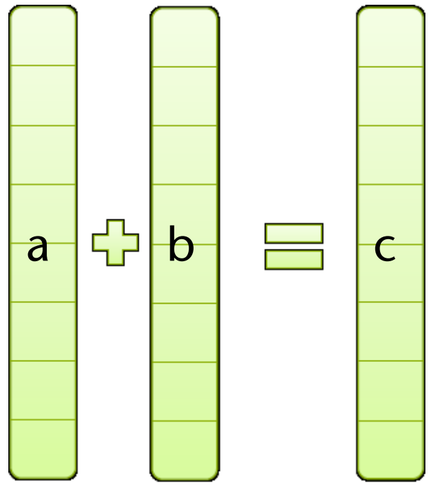


FYI: The compilation process

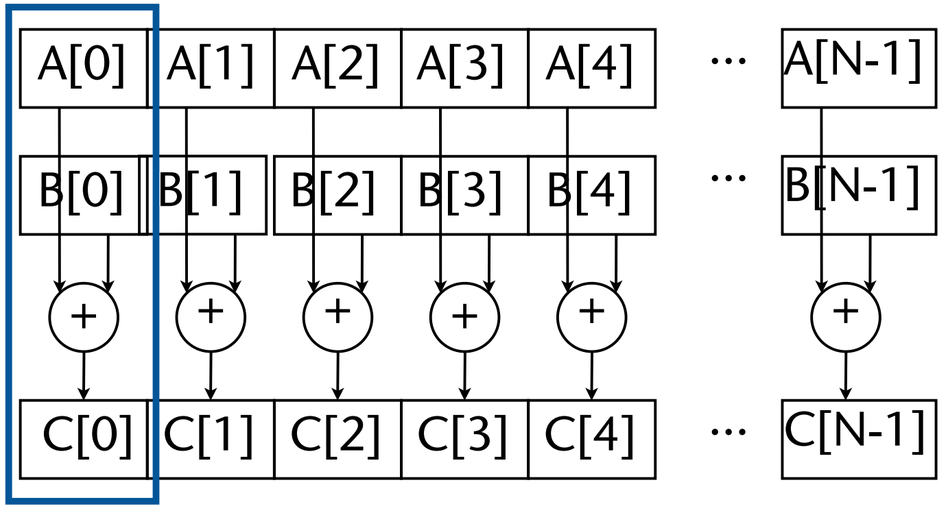


Transferring Data between GPU and CPU

- All data transfer between CPU and GPU must be done by copying ranges of memory (at least for the moment)
- Our next goal:
  fast addition of large vectors



- Idea: *one thread per index*, performing one elementary addition



1. We allocate memory on the host as usual:

```
size_t size = vec_len * sizeof(float);
float * h_a = static_cast<float>( malloc( size ) );
float * h_b = ...  and h_c ...
```

   - Looks familiar? I hoped so ☺ ...

2. Fill vectors **h_a** and **h_b** (see code on the course web page!)

3. Allocate memory on the device:

```
float *d_a, *d_b, *d_c;
cudaMalloc( static_cast<void**>( & d_a), size );
cudaMalloc( static_cast<void**>( & d_b), size );
cudaMalloc( static_cast<void**>( & d_c), size );
```

   - Note the naming convention!

4. Transfer vectors from host to device:

```
cudaMemcpy( d_a, h_a, size, cudaMemcpyHostToDevice );
cudaMemcpy( d_b, h_b, size, cudaMemcpyHostToDevice );
```

5. Write the kernel:

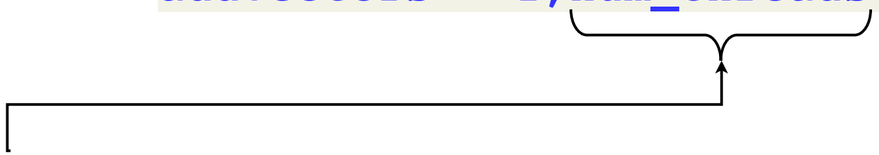
- Launch o*ne thread per element* in the vector

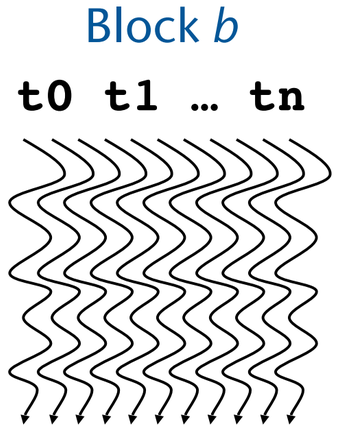
```
__global__
void addVectors( const float *a, const float *b,
                 float *c, unsigned int n       )
{
    unsigned int i = threadIdx.x;
    if ( i < n )
        c[i] = a[i] + b[i];
}
```

- **Yes, this is massively-parallel computation!**

6. And call it:

```
addVectors<<<1,num_threads>>>( d_a, d_b, d_c, vec_len );
```





- This number defines a block of threads
  - All of them run (conceptually) in parallel
  - Sometimes denoted with SIMT (think SIMD)

7. Afterwards, transfer the result back to the host:

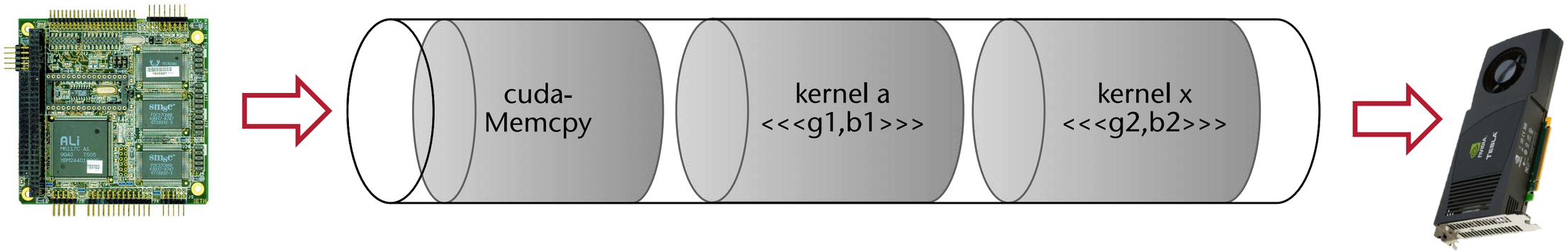
```
cudaMemcpy( h_c, d_c, size, cudaMemcpyDeviceToHost );
```

- See the course web page for the full code *with error checking*

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
  - Successful execution of a kernel launch merely means that the kernel is queued; it may begin executing immediately, or it may execute later when resources become available
- 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 of commands:



- Advantage of asynchronous CUDA calls:
  - CPU can work on other stuff while GPU is working on number crunching
  - Possibility to overlap memcopies and kernel execution (we don't use this special feature in this course)
- You can actually create several pipelines
  - Advantage: make synchronous CUDA calls run in parallel, too! (“overlapping”)
  - Powerful and flexible concept to control parallelism

On Memory Management on the GPU

- The API function:

```
cudaMemcpy( void *dest, void *source,
            unsigned int nbytes,
            enum cudaMemcpyKind direction)
```

  - **cudaMemcpyKind ∈ { cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice }**

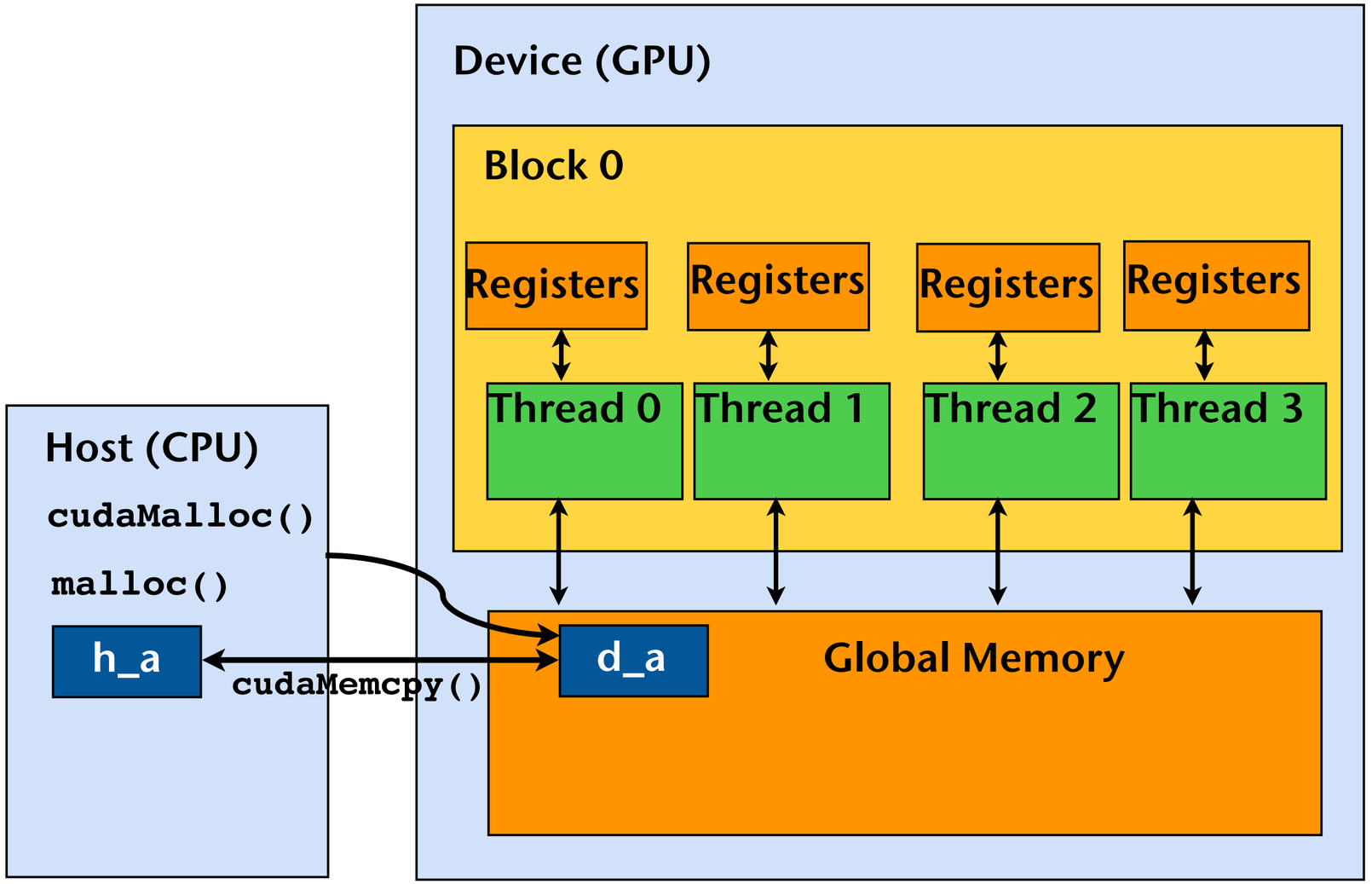
- Mnemonic: like **memcpy()** from Unix/Linux

```
memcpy( void *dst, void *src, unsigned int nbytes )
```

- Blocks CPU until transfer is complete
- CPU thread doesn't start copying until previous CUDA call is complete

Terminology

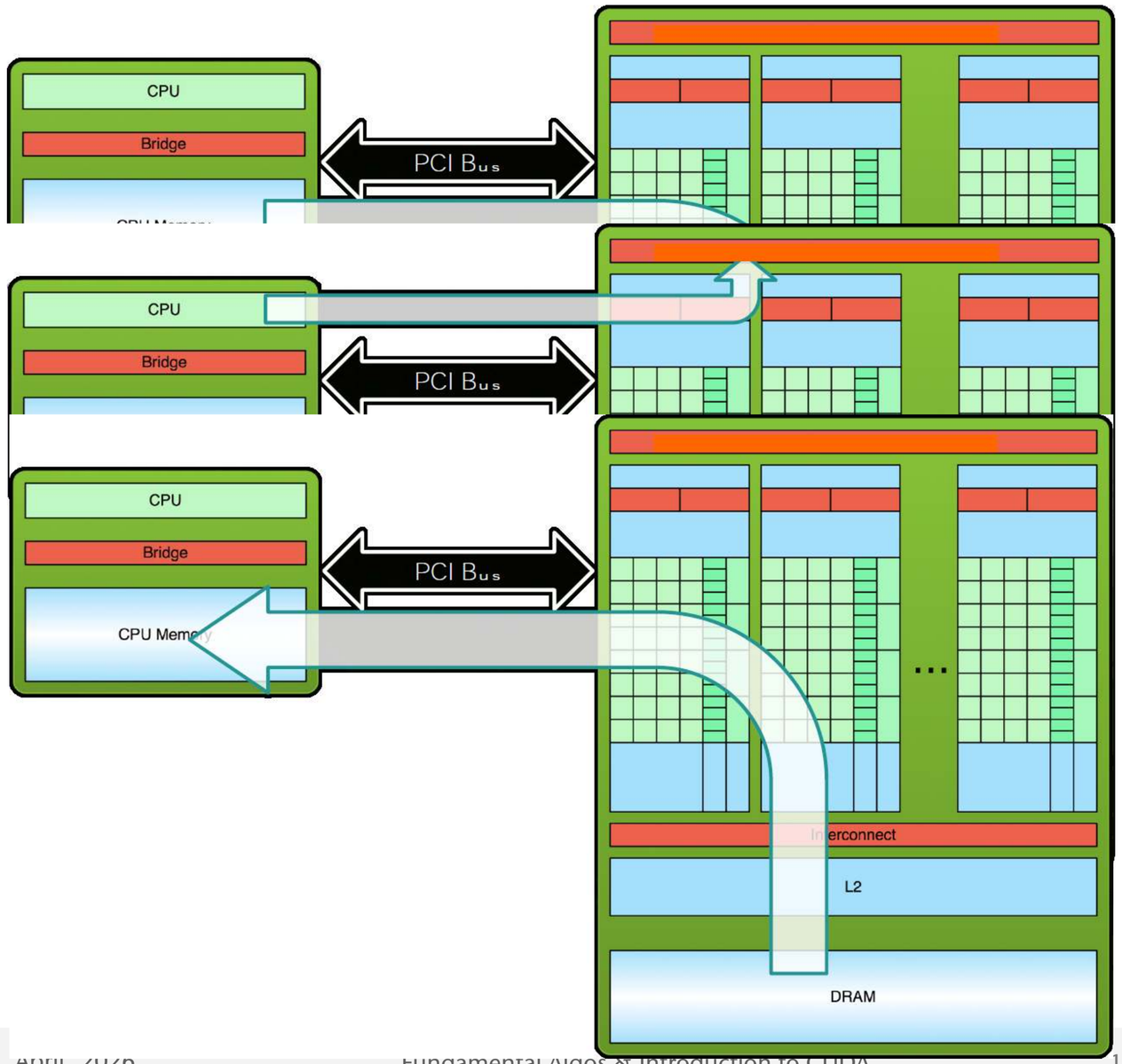
- This memory is called global memory



- The API is extremely simple:
  - **`cudaMalloc(), cudaFree(), cudaMemcpy()`**
  - Modeled after **`malloc(), free(), memcpy()`** from Unix/Linux
- Note: there are two different kinds of pointers!
  - Host memory pointers (obtained from **`malloc()`**)
  - Device memory pointers (obtained from **`cudaMalloc()`**)
  - You can pass each kind of pointers around as much as you like ...
  - But: don't dereference device pointers on the host and vice versa!

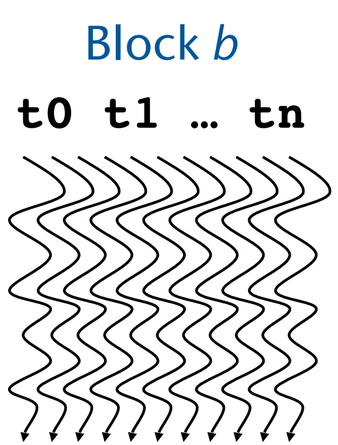
The General Data Flow in Heterogeneous Computing

1. Copy input data from CPU memory to GPU memory
2. Load GPU program(s) and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory



New Concept: Blocks and Grids

- Block of threads = virtualized multiprocessor
  = massively data-parallel task
- Requirements:
  - Each block execution must be independent of others
    - Can run concurrently or sequentially
  - Program is valid for any interleaved execution of blocks
  - Gives scalability
  - Number of threads per block < **`maxThreadsPerBlock`**

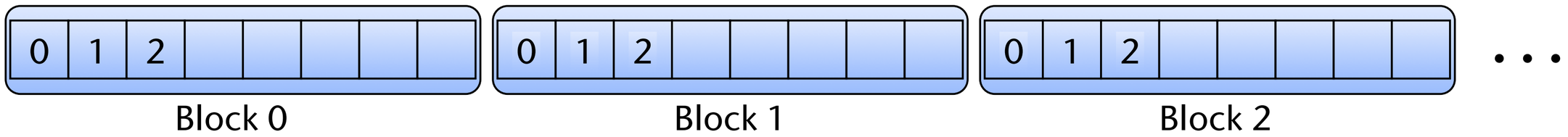


- What if we want to handle vectors larger than **`maxThreadsPerBlock`** ?
- We launch several blocks of our kernel!

```
addVectors<<< 1, num_threads>>>( d_a, d_b, d_c, n );
```

```
addVectors<<< num_blocks, threads_per_block >>>( d_a, d_b, d_c, n );
```

- This gives the following threads layout:

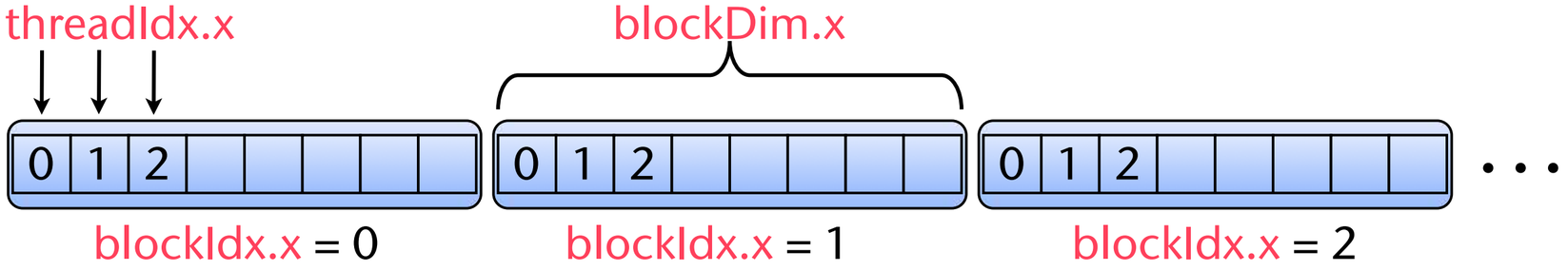


- Grid = set of all blocks

- How can threads index "their" vector element?

```
__global__
void addVectors( const float *a, const float *b,
                 float *c, unsigned int n        )
{
  unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;
  if ( i < n )
    c[i] = a[i] + b[i];
}
```

- The structs **`blockDim`**, **`blockIdx`**, and **`threadIdx`** are predefined in every thread



How Many Threads/Blocks Should We Launch?

- Number of threads per block should be multiple of 32
- The C idiom to calculate the number of blocks:

```
int threads_per_block = 256;            // any k*32 in [1,1024]
int num_blocks = (n + threads_per_block - 1) / threads_per_block;
```

  - where n = total number of threads
- This yields $$\text{num_blocks} = \left\lceil \frac{n}{\text{threads_per_block}} \right\rceil$$

  without any floating-point arithmetic
- Remark: this is the reason for the test **`if ( i < n )`**
- Yes, you should adapt to a programming language's idioms just like with natural languages, too :-)

- There are several limits on **`num_blocks`** and **`threads_per_block`** :
  - **`num_blocks * threads_per_block`** < 65,536 !
  - **`num_blocks < maxGridSize[0]`** !
  - And a few more (see table later)

Thread Layouts for 2D Computational Problems

- Many computational problems have a 2D domain (e.g., in CV)
  - Many others have a 3D domain (e.g., fluids simulation)
- Solution: layout threads in 2D
  - Simplifies index calculations a lot

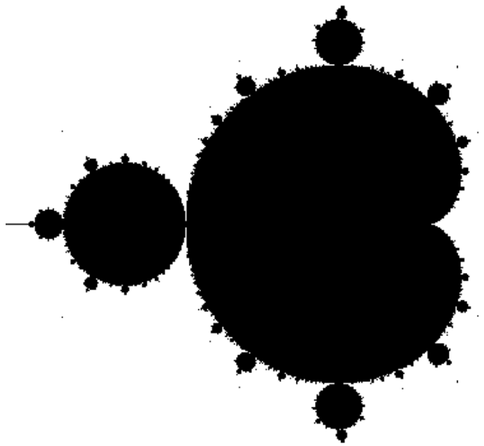
Example: Mandelbrot Set Computation

- Definition:
  - For each $c \in \mathbb{C}$, consider the (infinity) sequence

$$z_{i+1} = z_i^2 + c\ , \quad z_0 = 0$$

  - Define the Mandelbrot set

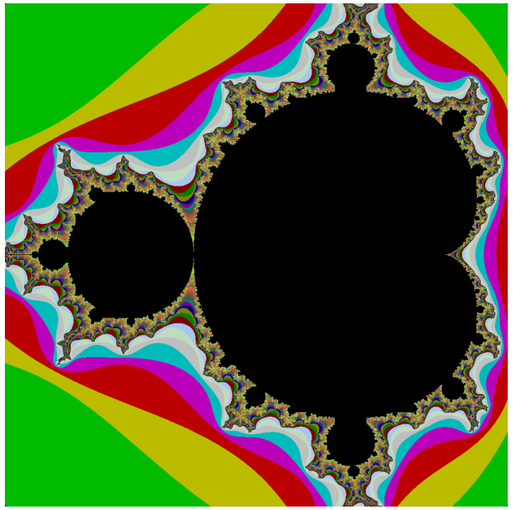
$$\mathbb{M} = \{ c \in \mathbb{C} \mid \text{sequence } (z_i) \text{ remains bounded } \}$$



- Theorem (w/o proof):

$$\exists t : |z_t| > 2 \ \Rightarrow \ c \notin \mathbb{M}$$

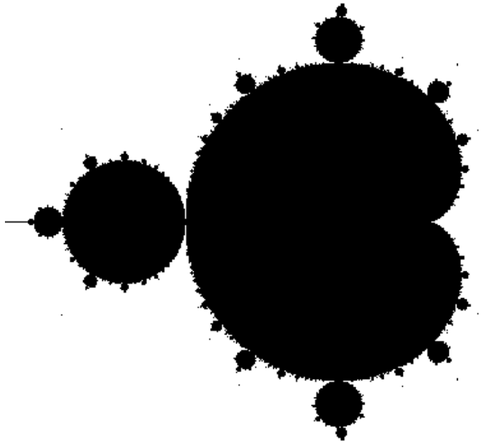
- Visualizing $\mathbb{M}$ nicely with the "escape" algorithm:
  - Color pixel c = (x,y) black, if |z| <2 after "many" iterations
  - Color c depending on the number of iterationsnecessary to reach |zt| > 2

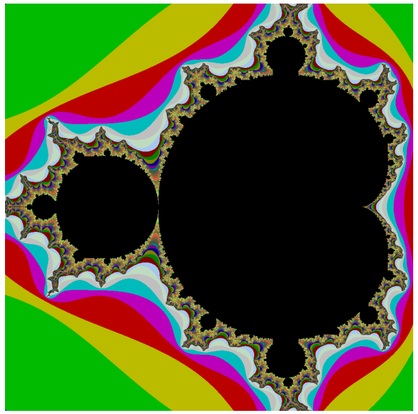


A Few Fun Facts About $\mathbb{M}$ (w/o Proofs)

(with which you can entertain people at a party ☺)

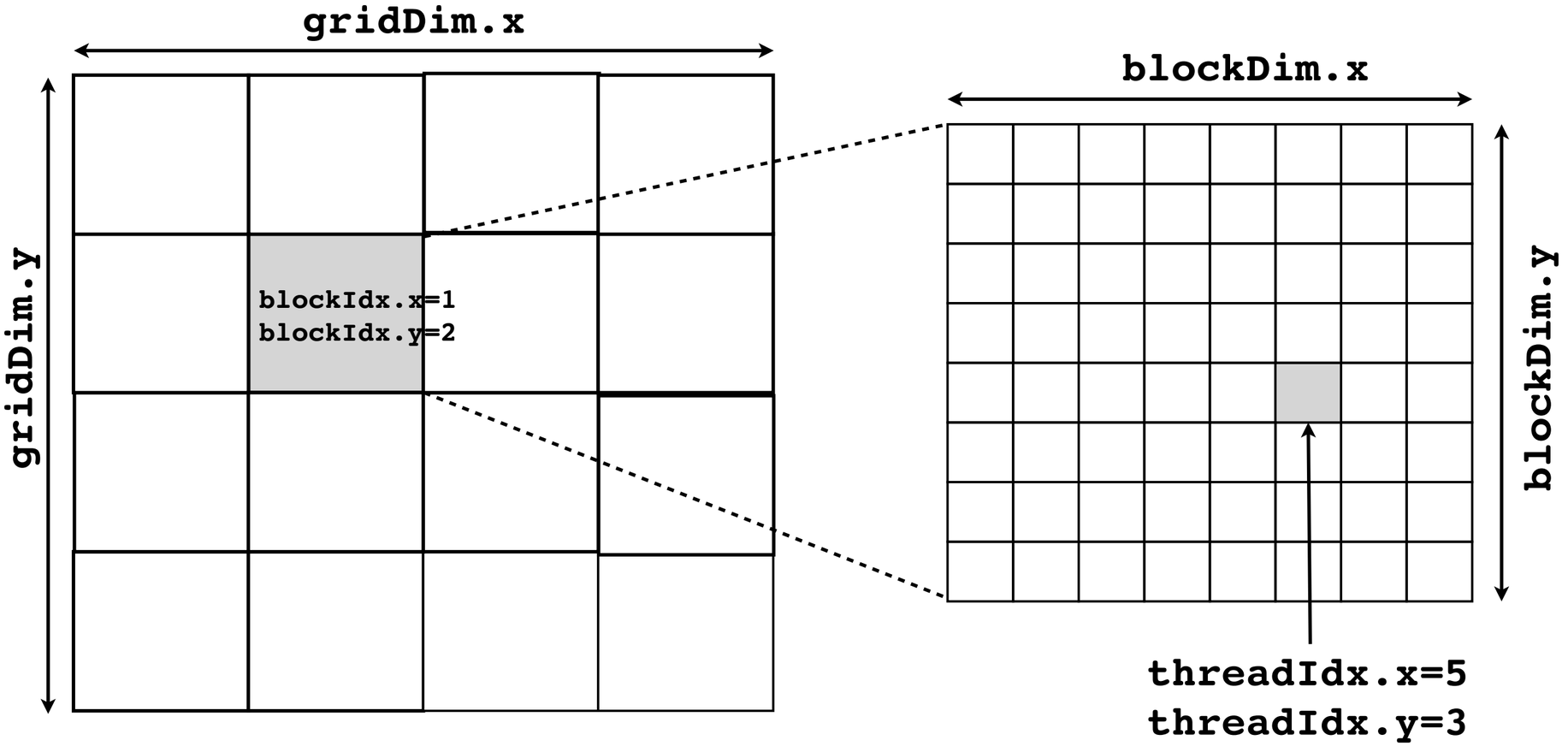
- The length of the border of $\mathbb{M}$ is *infinite*
- $\mathbb{M}$ is *connected* (i.e., all "black" regions are connected with each other)
  - Mandelbrot himself believed $\mathbb{M}$ was disconnected
- For each color, there is exactly one "ribbon" around $\mathbb{M}$, i.e., there is exactly one ribbon of values c, such that $|z_1| > 2$, there is exactly one ribbon of values c, such that $|z_2| > 2$, etc. ...
- Each such "iteration ribbon" goes completely around $\mathbb{M}$ and it is connected (i.e., there are no "gaps" within a ribbon)
- There is an infinite number of "mini Mandelbrot sets", i.e., smaller copies of $\mathbb{M}$





Partitioning the Domain into Blocks & Threads

- Embarrassingly parallel: one thread per pixel, each pixel computes their own z-sequence, then sets the color



- Set up threads layout, here a 2D arrangement of blocks & threads:

```
dim3 threads( 16, 16 );
dim3 blocks( img_size/threads.x, img_size/threads.y );
```

  - Here, we assume image size = multiple of 32!
    - Simplifies calculation of number of blocks
    - Also simplifies kernel: we don't need to check whether thread is out of range
    - See example code on web page how to ensure that

- Definition of **`dim3`** (done in CUDA's header files):

```
class dim3
{
   public:
     unsigned int x, y, z;
};
```

- Usage for launching a 2D kernel:

```
dim3 threads( n_tx, n_ty );
dim3 blocks(n_bx, n_by );
kernel<<< blocks, threads>>>(...);
```

- Convenience constructors for dim3:

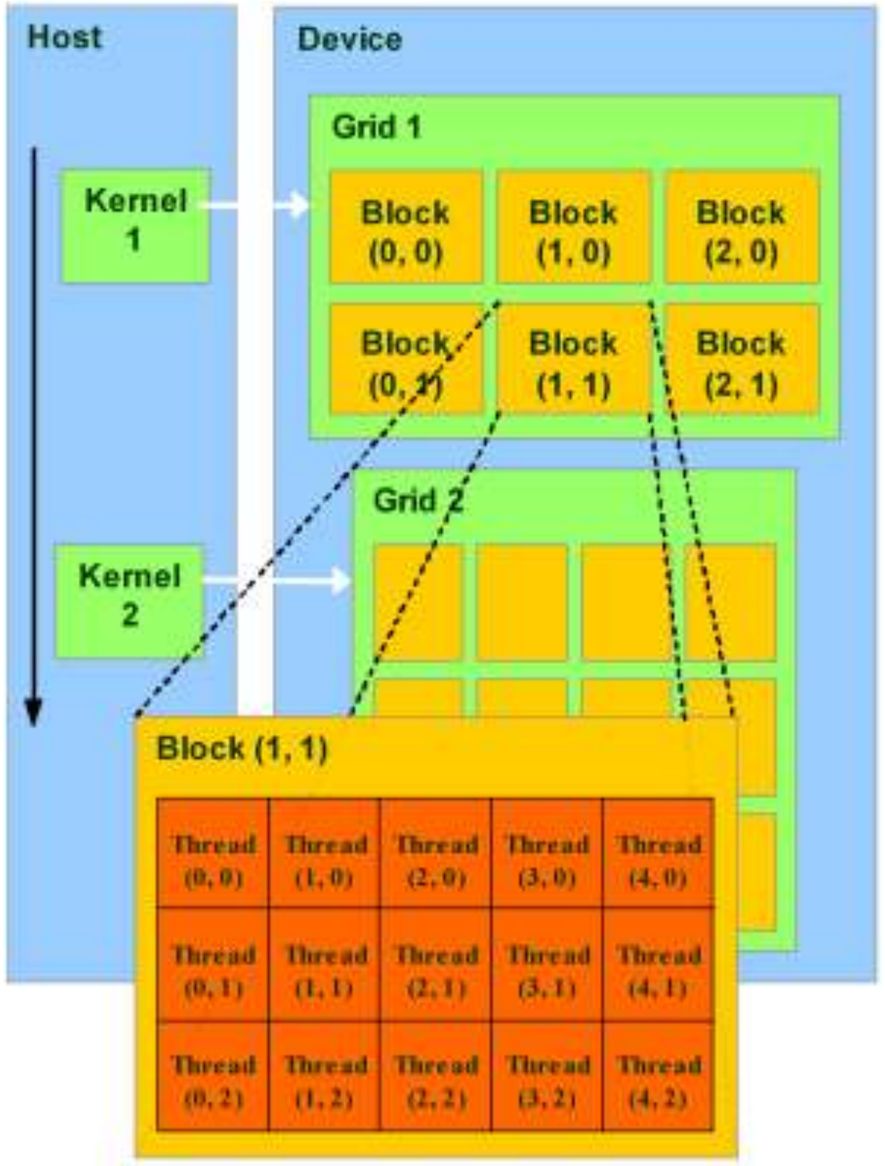
```
dim3 layout(nx);  = dim3 layout(nx,1); =  dim3 layout(nx,1,1);

dim3 layout(nx,ny);  =  dim3 layout(nx,ny,1);
```

- Launching a kernel like this:
  is equivalent to:

```
kernel<<<N,M>>>(...);
```

```
dim3 threads(M,1);
dim3 blocks(N,1);
kernel<<<blocks,threads>>>(...);
```

- In general, the layout of threads can change from kernel to kernel:



Computing the Mandelbrot Set on the GPU

- Usual code for allocating memory, here a bitmap:

```
const unsigned int bitmap_size = img_size * img_size * 4;
h_bitmap = new unsigned char[bitmap_size];
cudaMalloc( (void**) &d_bitmap, bitmap_size );
```

- Launch kernel:

```
mandelImage<<< n_blocks, n_threads >>>( d_bitmap, img_size );
```

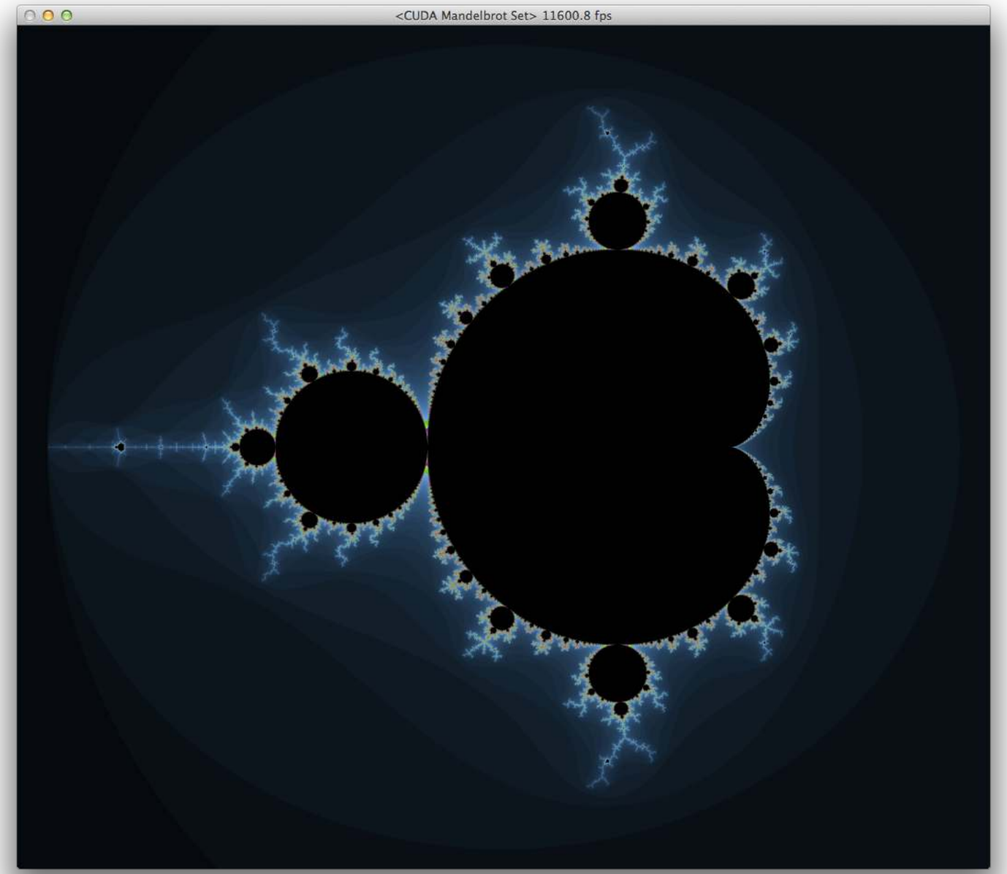
Implementation of the Kernel (simplified)

```
// Compute a Mandelbrot square image around (0,0)
// bitmap:   one pixel = one char4
// img_size: #pixels in x/y
// Warning: bitmap must hold img_size^2 many pixels!

__global__
void mandelImage( char4 * bitmap, const int img_size )
{
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;
  int offset = x + y * (gridDim.x * blockDim.x);

  int isOutsideM = isPointInMandelbrot( x, y, img_size, 1.0 );

  bitmap[offset].x = 255 * isOutsideM;       // red = outside
  bitmap[offset].y = bitmap[offset].z = 0;   // G/B channels
  bitmap[offset].w = 255;                    // alpha (ignore it)
}
```

```
__device__
int isPointInMandelbrot( int x, int y,
                         const int img_size, float scale )
{
  cuComplex c( (float)(x - img_size/2) / (img_size/2),
               (float)(y - img_size/2) / (img_size/2)  );
  c *= scale;
  cuComplex z( 0.0, 0.0 );          // z_i of the sequence

  for ( int i = 0; i < 200; i ++ )
  {
    z = z*z + c;                    // uses overloaded operators
    if ( z.magnitude2() > 4.0f )    // |z|^2 > 2^2 -> outside
      return i;
  }

  return 0;
}
```

```
struct cuComplex                    // define a class for complex numbers
{
   float r, i;                      // real, imaginary part

   __device__                       // constructor
   cuComplex( float a, float b ) : r(a), i(b)  {}

   __device__                       // |z|^2
   float magnitude2( void )
   {
      return r * r + i * i;
   }

   __device__                        // z1 * z2
   cuComplex operator * (const cuComplex & a)
   {
      return cuComplex(r*a.r - i*a.i, i*a.r + r*a.i);
   }
   // for more: see example code on web page
};
```

Demo



Reimplement using imgui,
use CUDA 7 features -
possible on Mac?
⟶ mini project

Keys:

e	= reset
s	= toggle CPU/GPU
mouse	= pan
shift+mouse	= zoom
c	= change colors
d/D	= increase/decrease detail

Demos/Mandelbrot/Mandelbrot

Three different kinds of functions in CUDA

	Executed on:	Only callable from:
`__device__ float deviceFunc();`	device	device
`__global__ void kernelFunc();`	device	host/device
`__host__ float hostFunc();`	host	host

- Remarks:
  - __global__ defines a kernel function
  - Each '__' consists of two underscore characters
  - A kernel function must return void
  - __device__ and __host__ can be used together

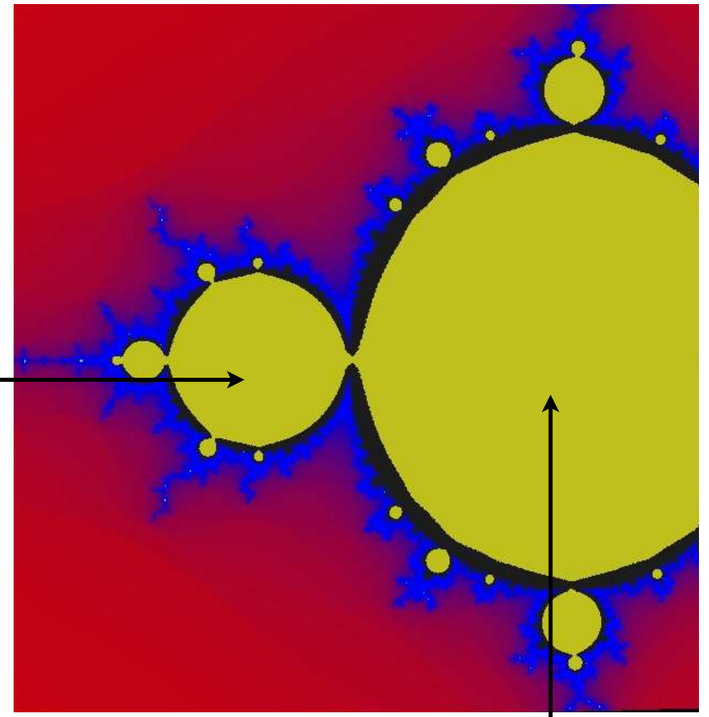
"Dual-Use" Code for Device and Host

- Example for **__device__** and **__host__**: a neat trick to make **cuComplex** usable on both device and host

```
struct cuComplex      // define a class for complex numbers
{
   float r, i;        // real, imaginary part

   __device__  __host__
   cuComplex( float a, float b ) : r(a), i(b)  {}

   __device__ __host__
   float magnitude2( void )
   {
      return r * r + i * i;
   }
   // etc. ...
};
```

Note on a Non-Trivial Mandelbrot Optimization

- An "optimization": utilize following property
  - The sequence of z_i can either converge towards a single (complex) value,
  - or it can end up in a cycle of values,
  - or it can be chaotic.
- Idea:
  - Try to recognize such cycles: if you realize that a thread is caught in a cycle, exit immediately (happens early for c-values in $\mathbb{M}$, but "far" from border)
  - Maintain an array of the *k* most recent elements of the sequence
- Last time I checked: 4x slower than the brute-force version! (YMMV)



Querying the Device for its Capabilities

- How do you know how many threads can be in a block, etc.?
- Query your GPU, like so:

```
int devID;
cudaGetDevice( &devID );          // GPU currently in use
cudaDeviceProp props;
cudaGetDeviceProperties( &props, devID );

unsigned int threads_per_block = props.maxThreadsPerBlock;
```

For Your Reference: the Complete Table of the cudaDeviceProp

DEVICE PROPERTY	DESCRIPTION
`char name[256];`	An ASCII string identifying the device (e.g., `"GeForce GTX 280"`)
`size_t totalGlobalMem`	The amount of global memory on the device in bytes
`size_t sharedMemPerBlock`	The maximum amount of shared memory a single block may use in bytes
`int regsPerBlock`	The number of 32-bit registers available per block
`int warpSize`	The number of threads in a warp
`size_t memPitch`	The maximum pitch allowed for memory copies in bytes
`int maxThreadsPerBlock`	The maximum number of threads that a block may contain
`int maxThreadsDim[3]`	The maximum number of threads allowed along each dimension of a block
`int maxGridSize[3]`	The number of blocks allowed along each dimension of a grid
`size_t totalConstMem`	The amount of available constant memory

DEVICE PROPERTY	DESCRIPTION
`int major`	The major revision of the device's compute capability
`int minor`	The minor revision of the device's compute capability
`size_t textureAlignment`	The device's requirement for texture alignment
`int deviceOverlap`	A boolean value representing whether the device can simultaneously perform a `cudaMemcpy()` and kernel execution
`int multiProcessorCount`	The number of multiprocessors on the device
`int kernelExecTimeoutEnabled`	A boolean value representing whether there is a runtime limit for kernels executed on this device
`int integrated`	A boolean value representing whether the device is an integrated GPU (i.e., part of the chipset and not a discrete GPU)
`int canMapHostMemory`	A boolean value representing whether the device can map host memory into the CUDA device address space
`int computeMode`	A value representing the device's computing mode: default, exclusive, or prohibited
`int maxTexture1D`	The maximum size supported for 1D textures

DEVICE PROPERTY	DESCRIPTION
`int maxTexture2D[2]`	The maximum dimensions supported for 2D textures
`int maxTexture3D[3]`	The maximum dimensions supported for 3D textures
`int maxTexture2DArray[3]`	The maximum dimensions supported for 2D texture arrays
`int concurrentKernels`	A boolean value representing whether the device supports executing multiple kernels within the same context simultaneously

Problem Partitioning: Example Adding Huge Vectors

- Vectors of size 100,000,000 are not uncommon in high-performance computing (HPC) ...
- Problem: your input, e.g. the vectors, can be larger than the maximally allowed size along one dimension
  - I.e., what if

```
vec_len > maxThreadsDim[0] * maxGridSize[0]?
```

- Abstract solution: choose a grid layout *different* from the problem domain

- The thread layout:

```
dim3 threads(16,16);                              // = 256 threads per block
int  n_threads_pb = threads.x * threads.y;
int  n_blocks = (vec_len + n_threads_pb - 1) / n_threads_pb;
int  n_blocks_sqrt = (int)( ceilf( sqrtf( n_blocks ) ) );
dim3 blocks( n_blocks_sqrt, n_blocks_sqrt );
```

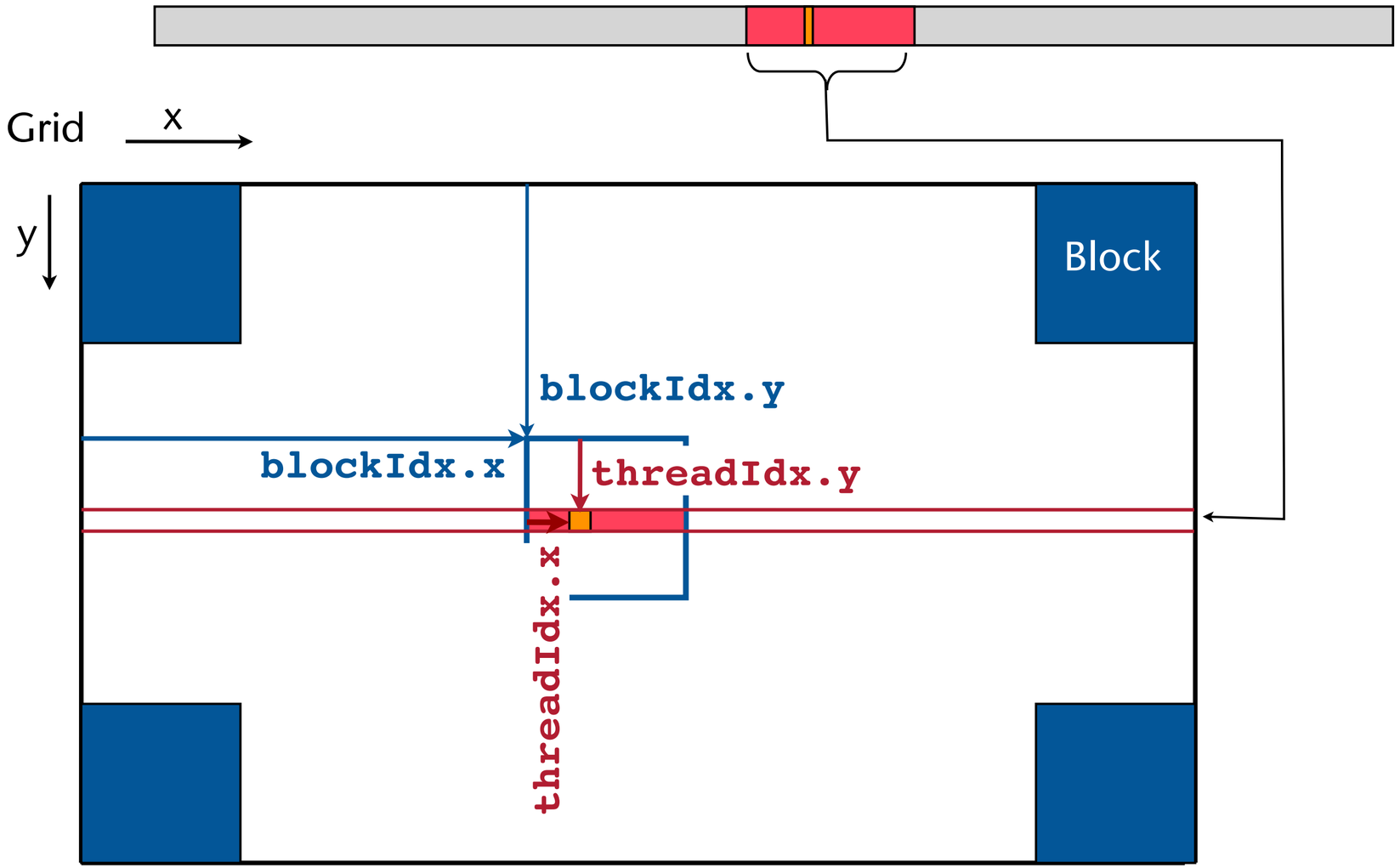
- Kernel launch:

```
addVectors<<< blocks, threads >>>( d_a, d_b, d_c, n );
```

- Index computation in the kernel:

```
unsigned int tid_x = blockDim.x * blockIdx.x + threadIdx.x;
unsigned int tid_y = blockDim.y * blockIdx.y + threadIdx.y;
unsigned int i = tid_y * (blockDim.x * gridDim.x) + tid_x;
```

Visualization of the index computation

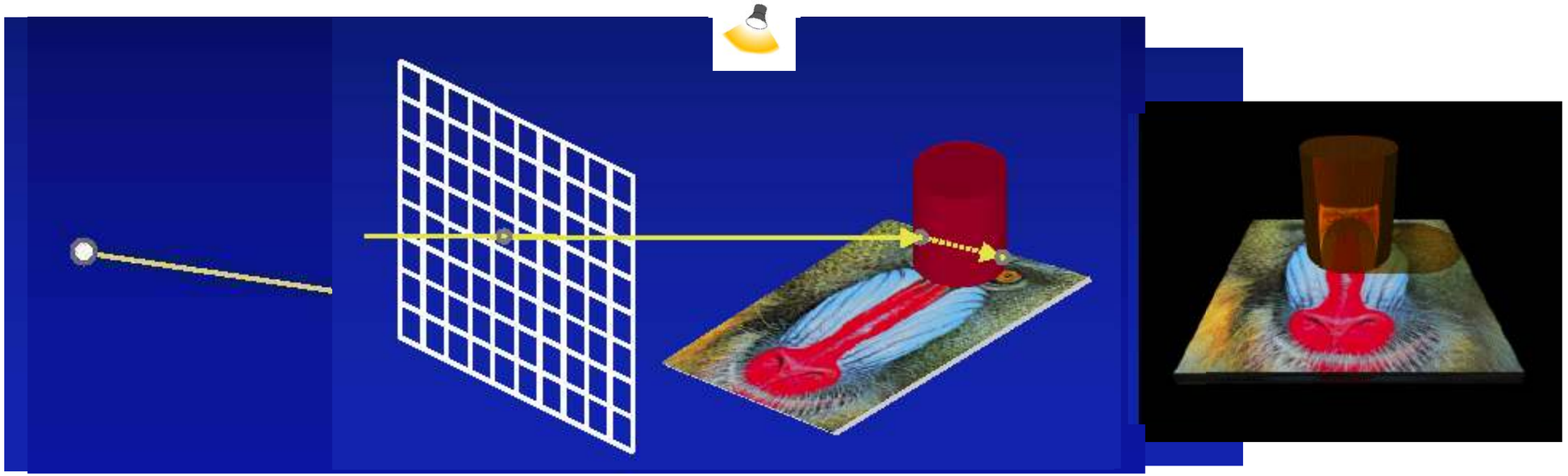


Constant Memory

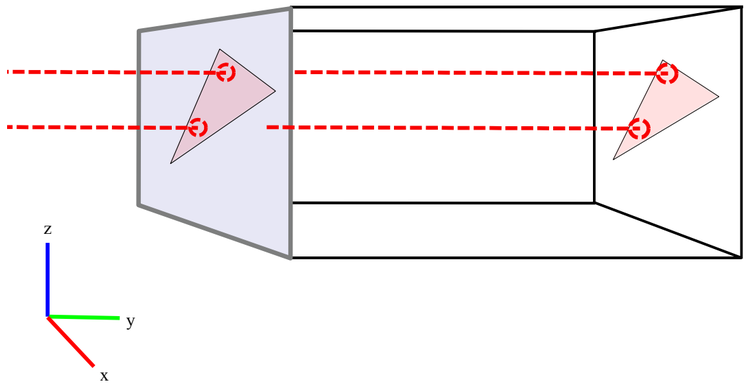
- Why is it so important to declare constant variables/instances in C/C++ as **`const`** ?
- It allows the compiler to ...
  - optimize your program a lot
  - do more type-checking
- Something similar exists in CUDA → constant memory

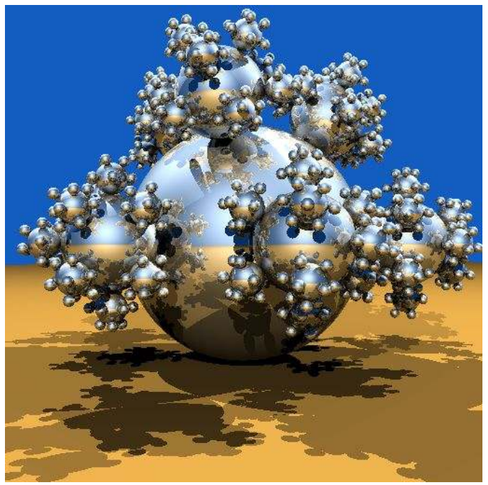
Example: a Simple Raytracer

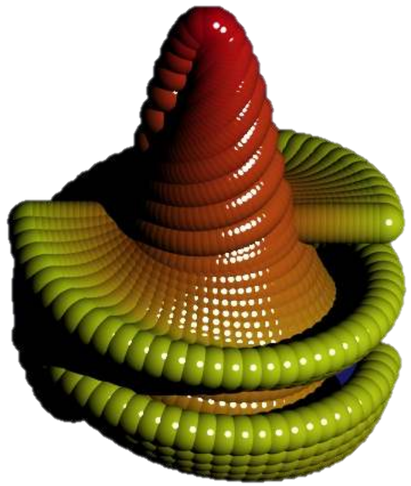
- The ray-tracing principle:
- Shoot rays from camera through every pixel into scene (primary rays)
- If the rays hits more than one object, then consider only the first hit
- From there, shoot rays to all light sources (shadow feelers)
- If a shadow feeler hits another obj → point is in shadow w.r.t. that light source.
  Otherwise, evaluate a lighting model (e.g., Phong [see "Computer graphics"])
- If the hit object is glossy, then shoot reflected rays into scene (secondary rays) → recursion
- If the hit object is transparent, then shoot refracted ray → more recursion

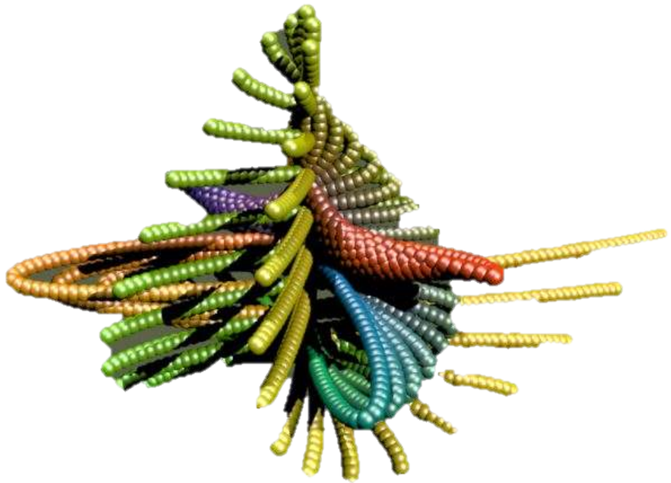


- Simplifications (for now):
  - Only primary rays and shadow rays
  - Camera is at infinity → primary rays are orthogonal to image plane
  - Only spheres: so easy, every raytracer has them ☺









The data structures

```
struct Sphere
{
   Vec3 center;           // center of sphere
   float radius;
   Color r, g, b;         // color of sphere

   __device__
   bool intersect( const Ray & ray, Hit * hit )
   {
      ...
   }
};
```

The mechanics on the host side

```
int main( void )
{
   // create host/device bitmaps (see Mandelbrot ex.)
   ...
   Sphere * h_spheres = new Sphere[n_spheres];
   // generate spheres, or read from file

   // transfer spheres to device (later)

   // generate image by launching kernel
   // assumption: img_size = multiple of block-size!
   dim3 threads(16,16);
   dim3 blocks( img_size/treads.x, img_size/treads.y );
   raytrace<<<blocks,threads>>>( d_bitmap );

   // display image, clean up, and exit
};
```

Declaration & Data Transfer

- Since our scene of spheres is constant during raytracing, we declare its memory as constant, too:

```
const int MAX_NUM_SPHERES 1000;
__constant__ Sphere c_spheres[MAX_NUM_SPHERES];
```

- Transfer now works by a different kind of cudaMemcpy:

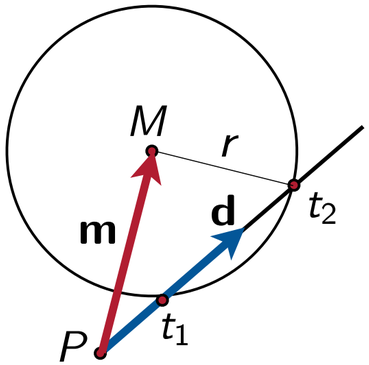
```
int main( void )
{
   ...
   // transfer spheres to device
   cudaMemcpyToSymbol( c_spheres, h_spheres,
                       n_spheres * sizeof(Sphere) );
   ...
};
```

The mechanics on the device side

```
__global__
void raytrace( unsigned char * bitmap )   {
   // map thread id to pixel position
   int x = blockIdx.x * blockDim.x + threadIdx.x;
   int y = blockIdx.y * blockDim.y + threadIdx.y;
   int offset = x + y * (gridDim.x * blockDim.x);

   Ray ray( x, y, camera );                    // generate primary ray

   // check intersection with scene, take closest one
   min_dist = FLT_MAX;                         // "= infinity"
   int hit_sphere = MAX_INT;                   // closest sphere hit, if any
   Hit hit;
   for ( int i = 0; i < n_spheres; i ++ ) {
      if ( intersect(ray, c_spheres[i], & hit) ) {
         if ( hit.dist < min_dist ) {
            min_dist = hit.dist;              // found a closer hit
            hit_sphere = i;                   // remember sphere; hit info
         }                                    // is already filled
      }
   }
   // compute color at hit point (if any) and set in bitmap
}
```

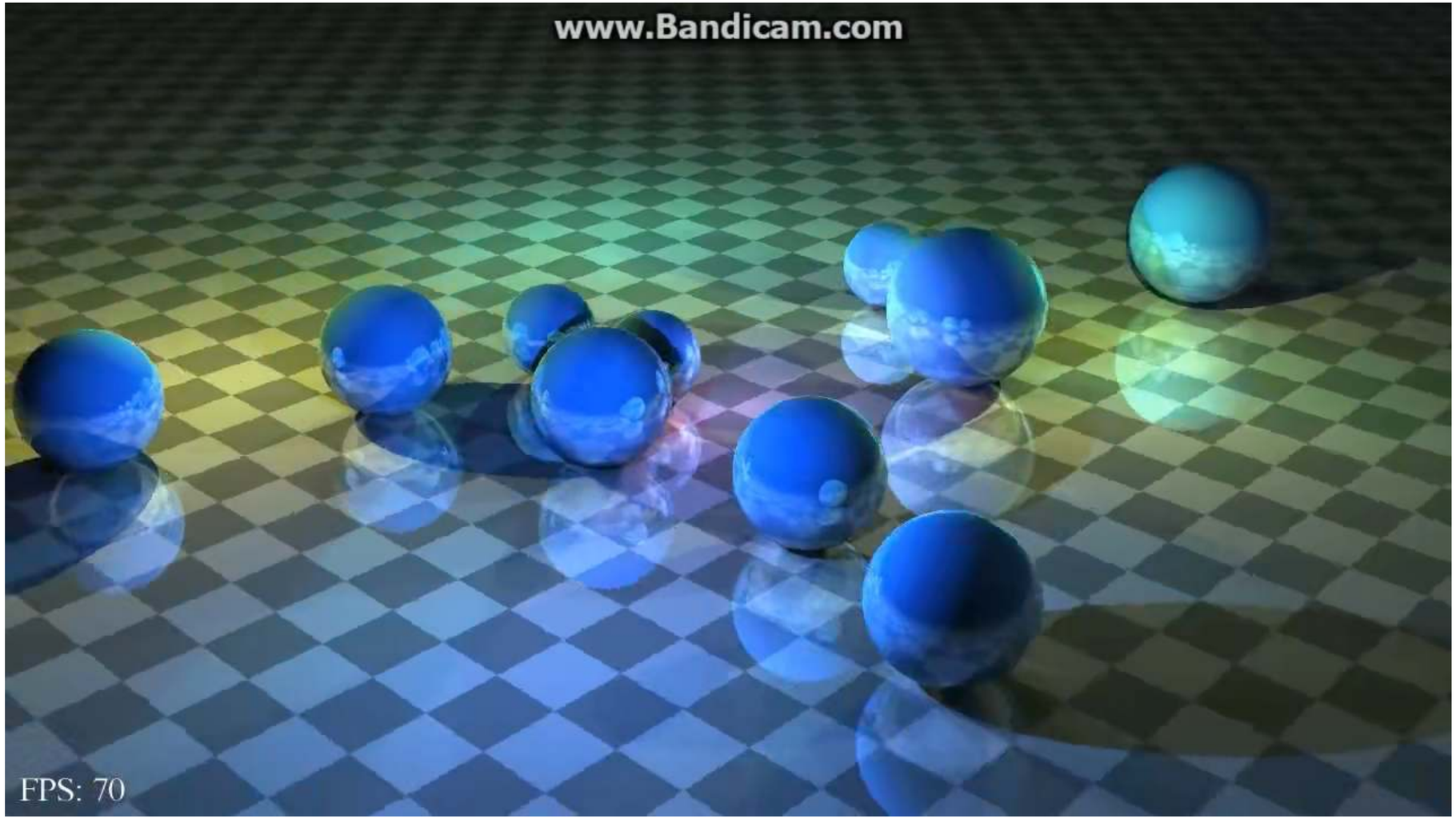
- Access of constant memory on the device (i.e., from a kernel) works just like with any globally declared variable
- Example: intersection routine, implementing

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




```
__constant__ Sphere c_spheres[MAX_NUM_SPHERES];

__device__
bool intersect( const Ray & ray, const Sphere & s, Hit * hit )
{
   Vec3 m( s.center – ray.orig );
   float r = s.radius;
   float q = m*m – r*r;
   float p = ...
   solve_quadratic( p, q, *t1, *t2 );
   ...
}
```

Video: Comparison of Ray-Tracing on GPU v. CPU



GPU = GTX 660 Ti CPU = i5-3570, 3.4 GHz, 4 cores, 4 threads

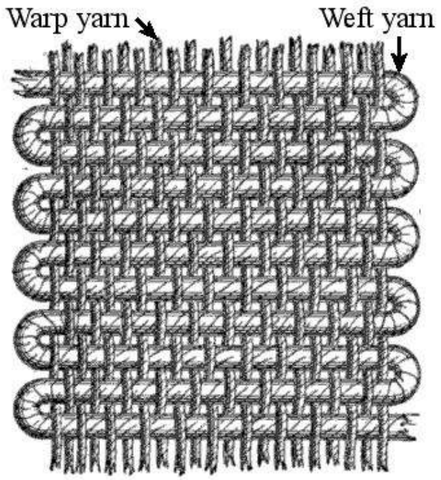
Some Considerations on Constant Memory

- 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

One Streaming Multiprocessor

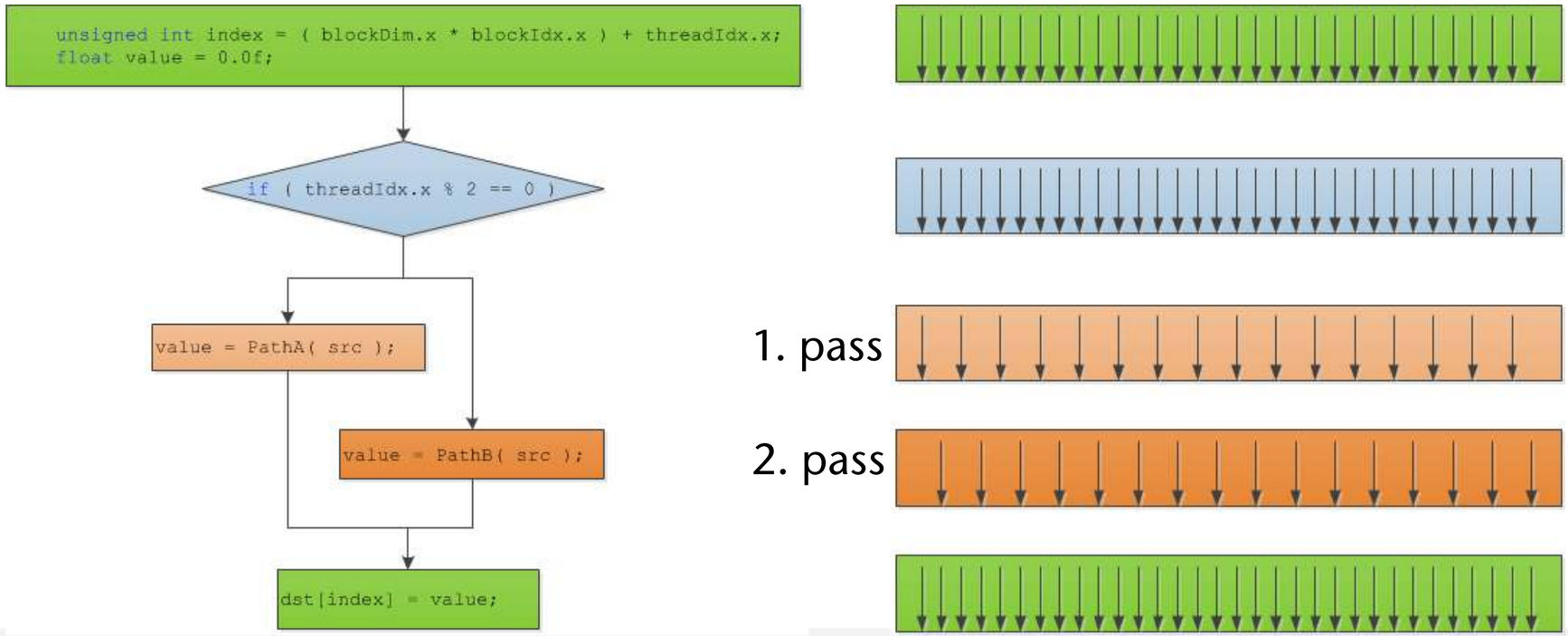
Tensor cores are missing here (more about those later)

Development of New Architectures is (Still) Pretty Dynamic

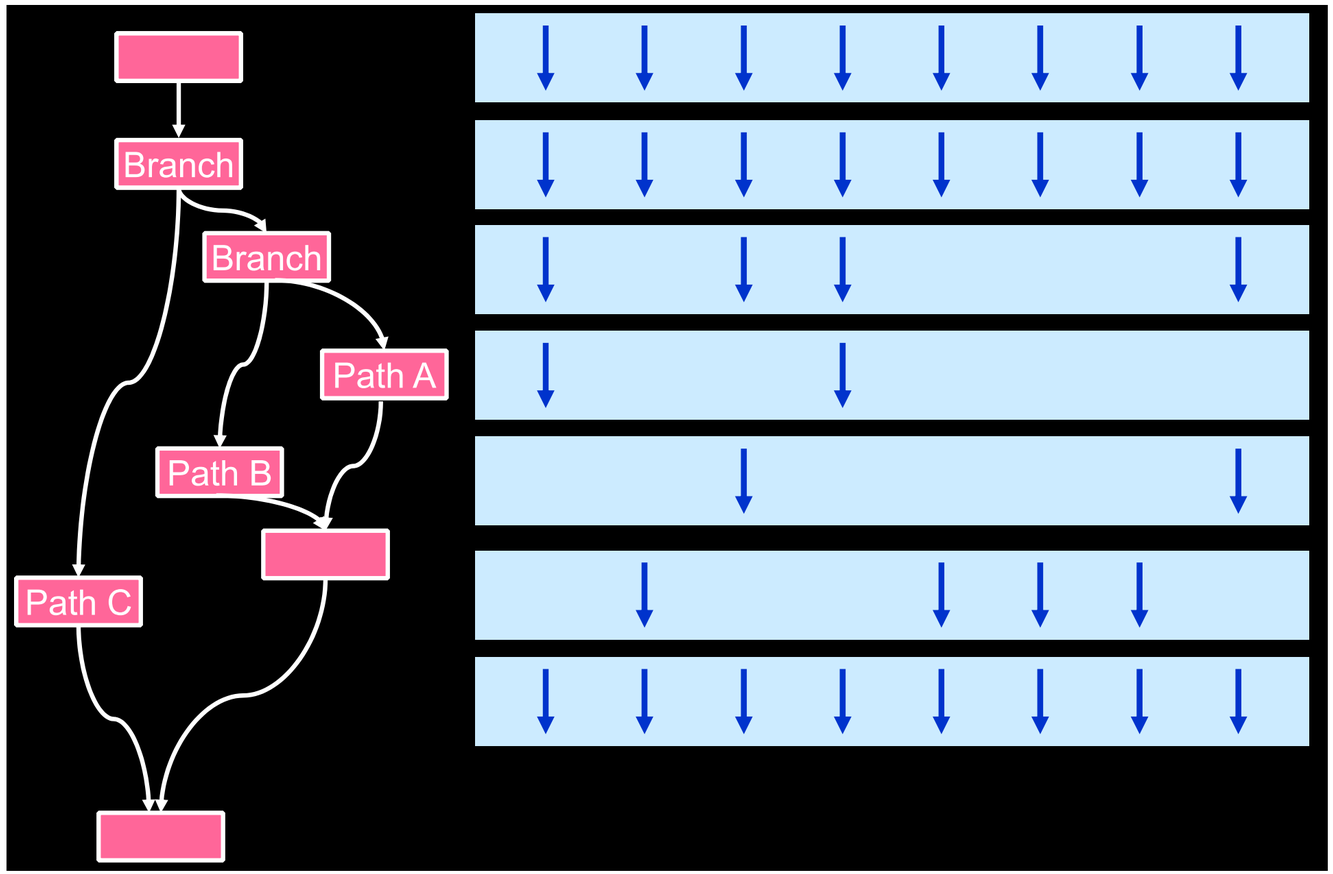
Architecture	Exemplary GPU Model	Compute Capability	Important Features
Tesla	GeForce 8800 GTX	1.0 – 1.3	Basic
Fermi	GeForce GTX 480	2.0 – 2.1	Ballots, 32-bit floating point atomics, 3D grids
Kepler	GeForce GTX 780	3.0 – 3.7	Shuffle, unified memory, dynamic parallelism
Maxwell	GeForce GTX 980	5.0 – 5.3	Half-precision floating point operations
Pascal	GeForce GTX 1080	6.0 – 6.2	64-bit floating point atomics
Volta	TITAN V	7.0 – 7.2	Tensor cores
Turing	GeForce RTX 2080	7.5	More concurrency, RTX cores (not compute)
Ampere	GeForce RTX 3090	8.0 – 8.6	L2 Cache Residency Management
Lovelace	?	9.0 – ?	?

Thread Divergence Revisited

- This execution of threads in lockstep fashion on one SM (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`** (branching flow control statements)
- Example:



- The more complex your control flow graph (this is called cyclomatic 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 cyclomatic complexity
- Avoid recursion
  - The SM needs to maintain one stack per warp
  - Recursion gets tricky in combination with inactive threads
  - If your algorithm heavily relies on recursion, then it may not be well suited for massive (data) parallelism!

Measuring Performance on the GPU

- General advice: experiment with a few different block layouts, e.g.,

  `dim3 threads(16,16)` up to `dim3 threads(128,2)`

  then compare performance
- CUDA API for timing:

```
// create two "event" structures
cudaEvent_t start, stop;
cudaEventCreate(&start); cudaEventCreate(&stop);
// insert the start event in the command 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;              // compute elapsed time
cudaEventElapsedTime( &elapsedTime, start, stop );
printf("Time to exec kernel = %f ms\n", elapsedTime );
```

Remarks on Memory (Applies to GPUs and CPUs)

- In our vector addition kernel, we stored everything in what is called global memory, but ...
- Bandwidth to global memory is sloooow

Ideal

Reality

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?
- All threads within the same block share very fast memory
- Unlike parallel blocks, threads within a block have mechanisms to communicate & synchronize very quickly

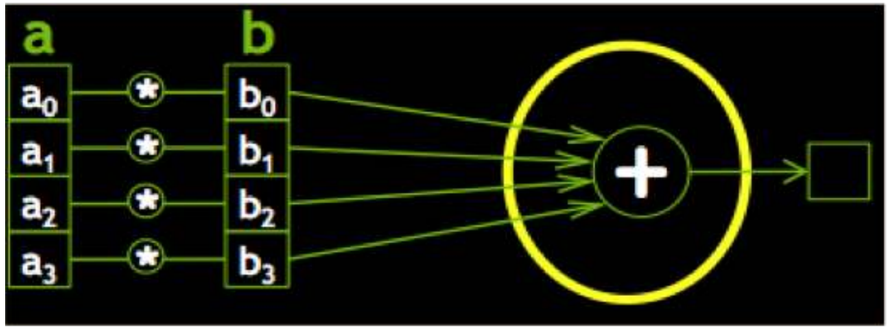
Computing the Dot Product

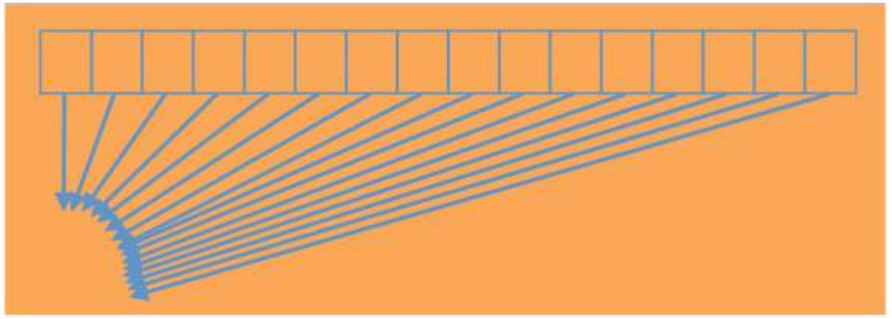
- Next goal: compute

$$d = \mathbf{x} \cdot \mathbf{y} = \sum_{i=0}^{N} x_i y_i$$

  for large vectors

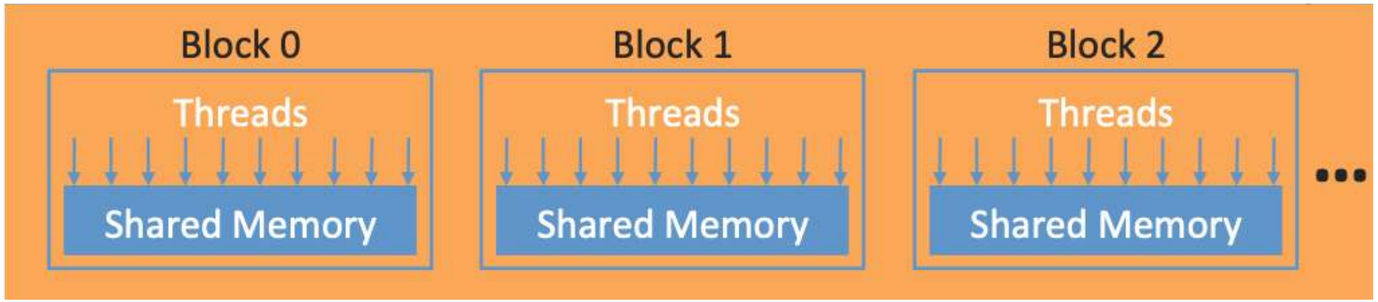
- We know how to do ($x_i y_i$) on the GPU, but how do we do the summation?
- Naïve (pseudo-parallel) algorithm:
  - Compute vector **z** with $z_i = x_i y_i$ in parallel
  - Transfer vector **z** back to CPU, and do summation sequentially
- Another (somewhat) naïve solution:
  - Compute vector **z**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



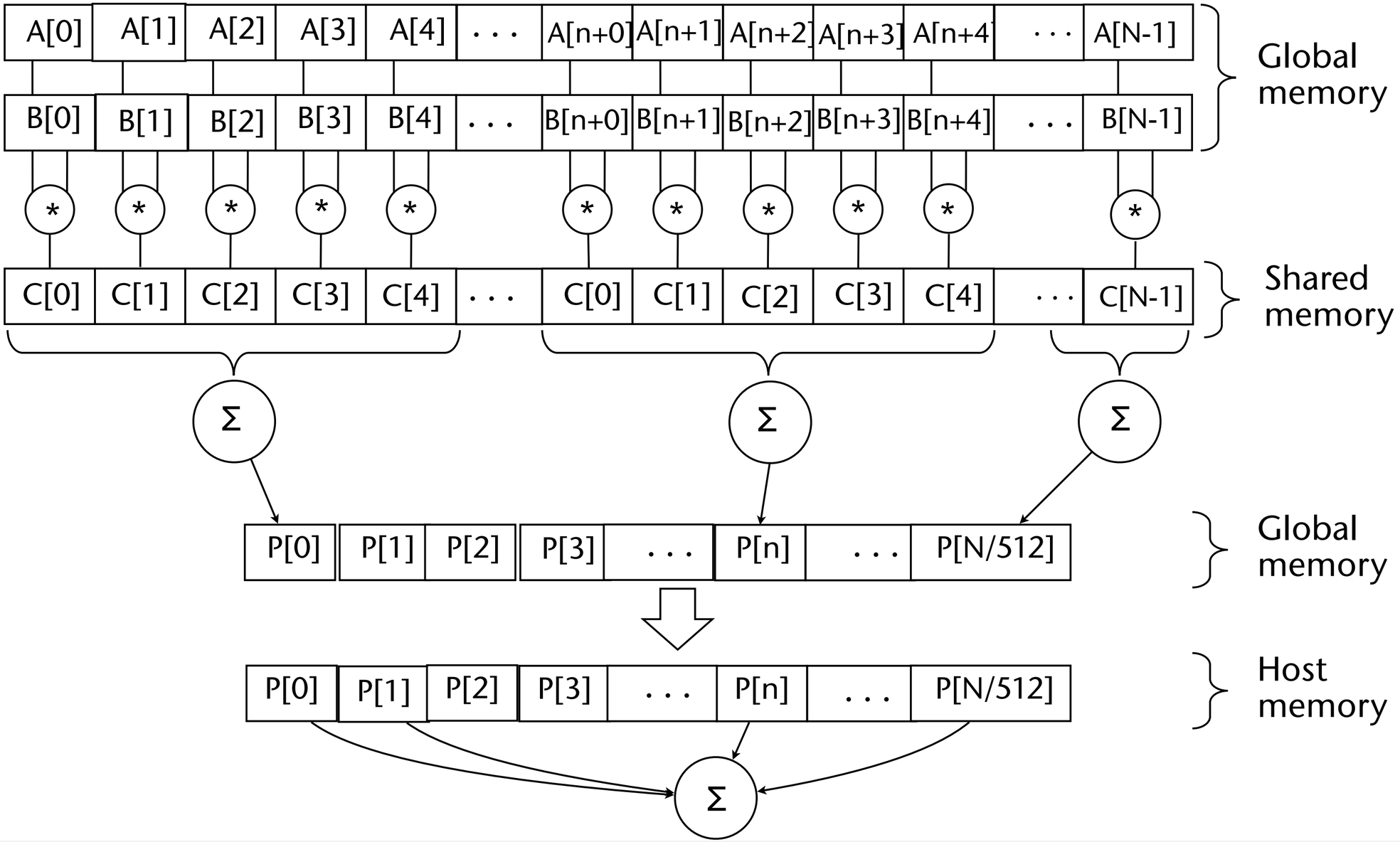


Cooperating Threads / Shared Memory

- A block of threads can have some amount of shared memory
- *All threads within a block* have the same "view" of this 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__`**

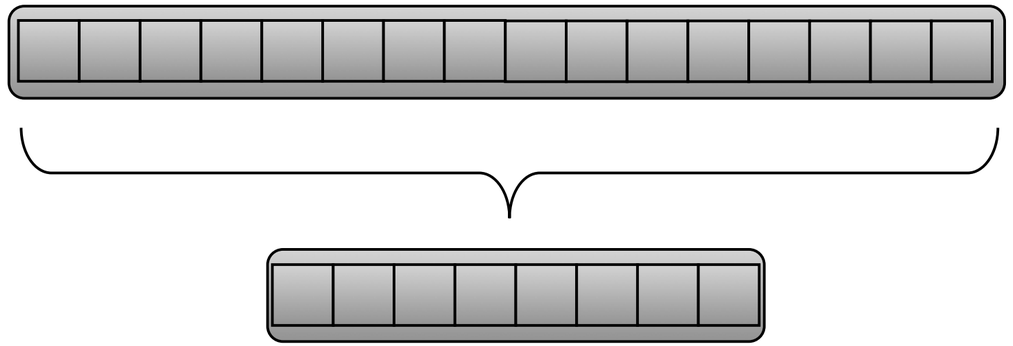


Overview of the Efficient Dot Product

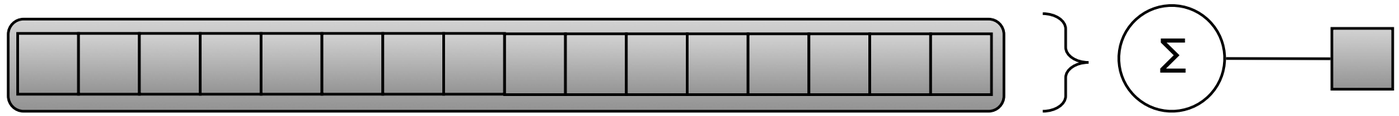


Terminology

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

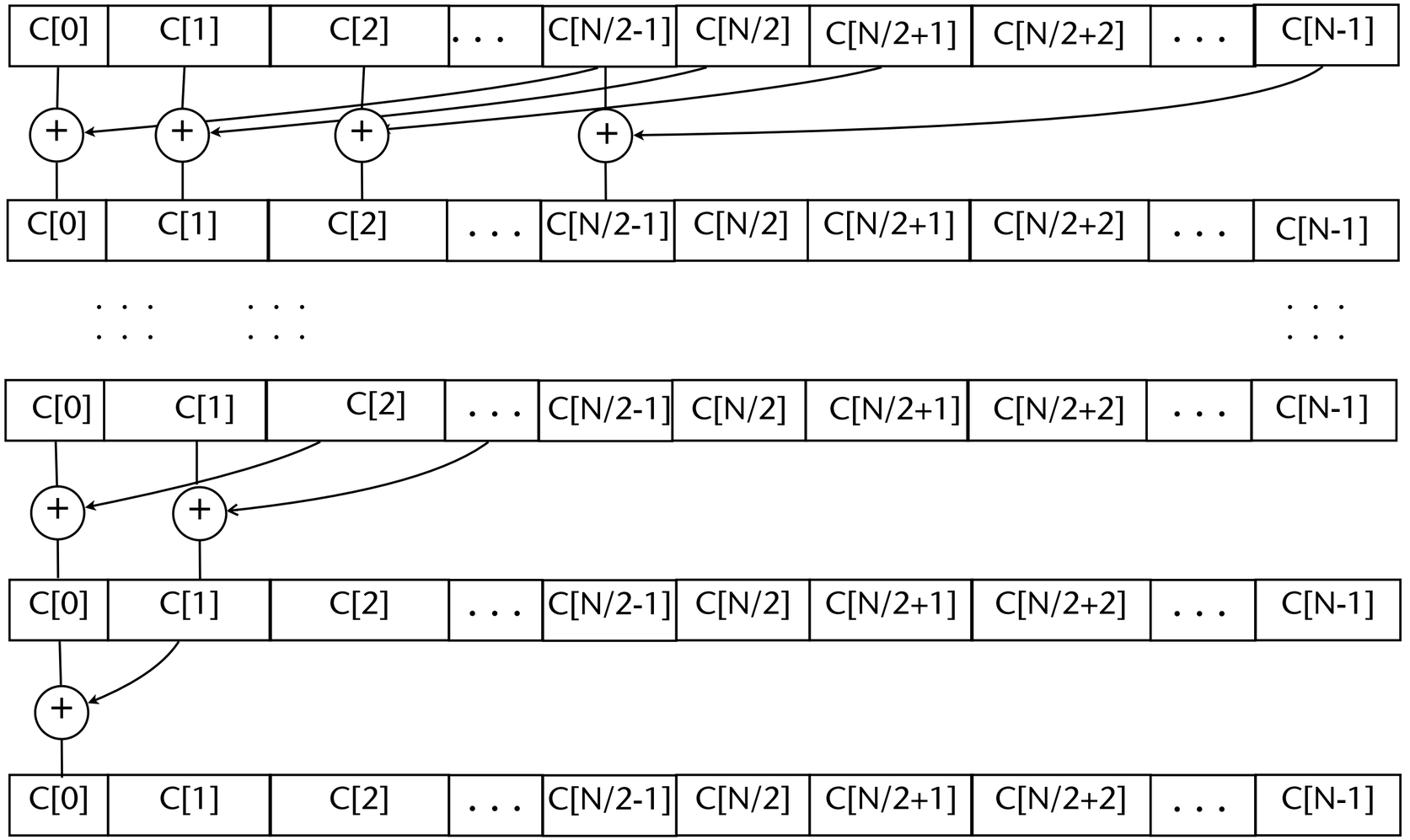


- Examples:
  - Dot product; takes 2 vectors, outputs 1 scalar = summation reduction
  - Min/max of the elements of a vector = min/max reduction

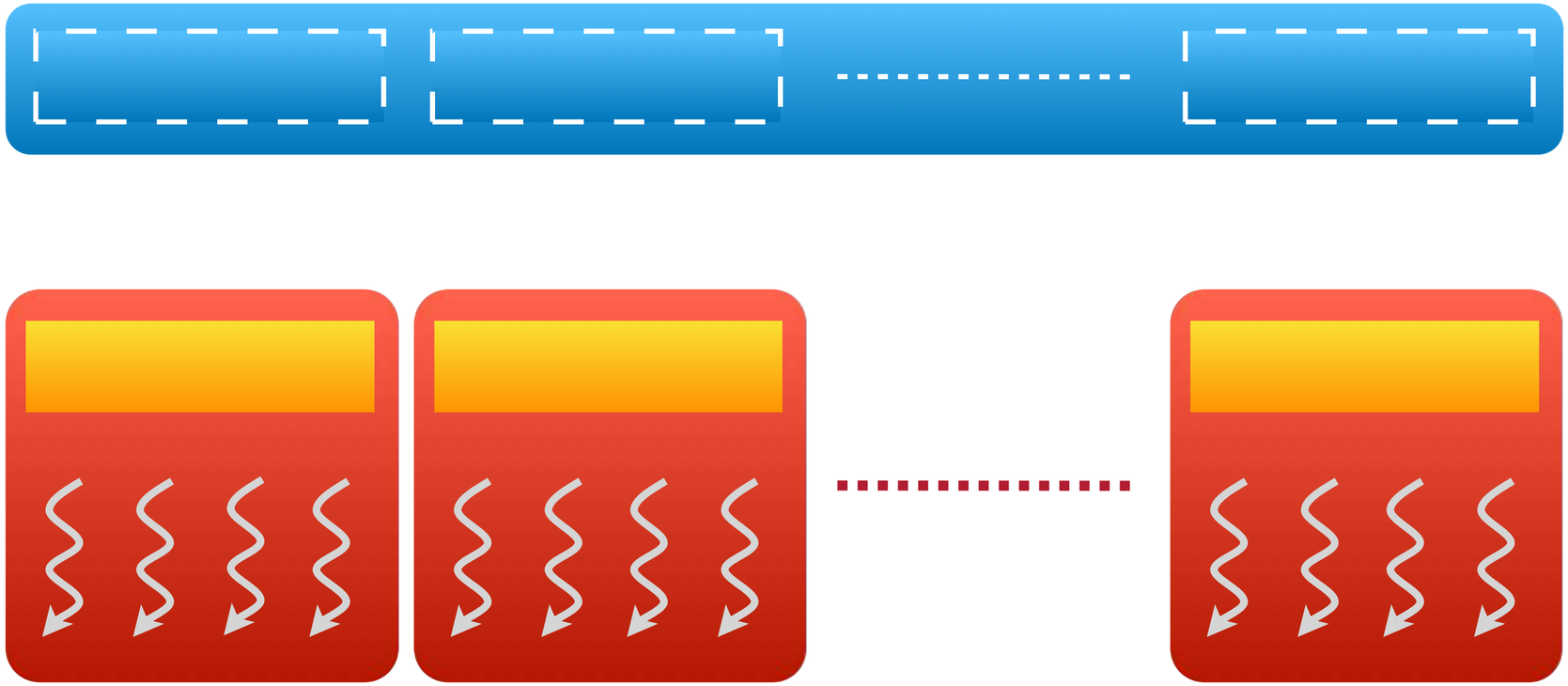


Efficiently Computing the Summation Reduction

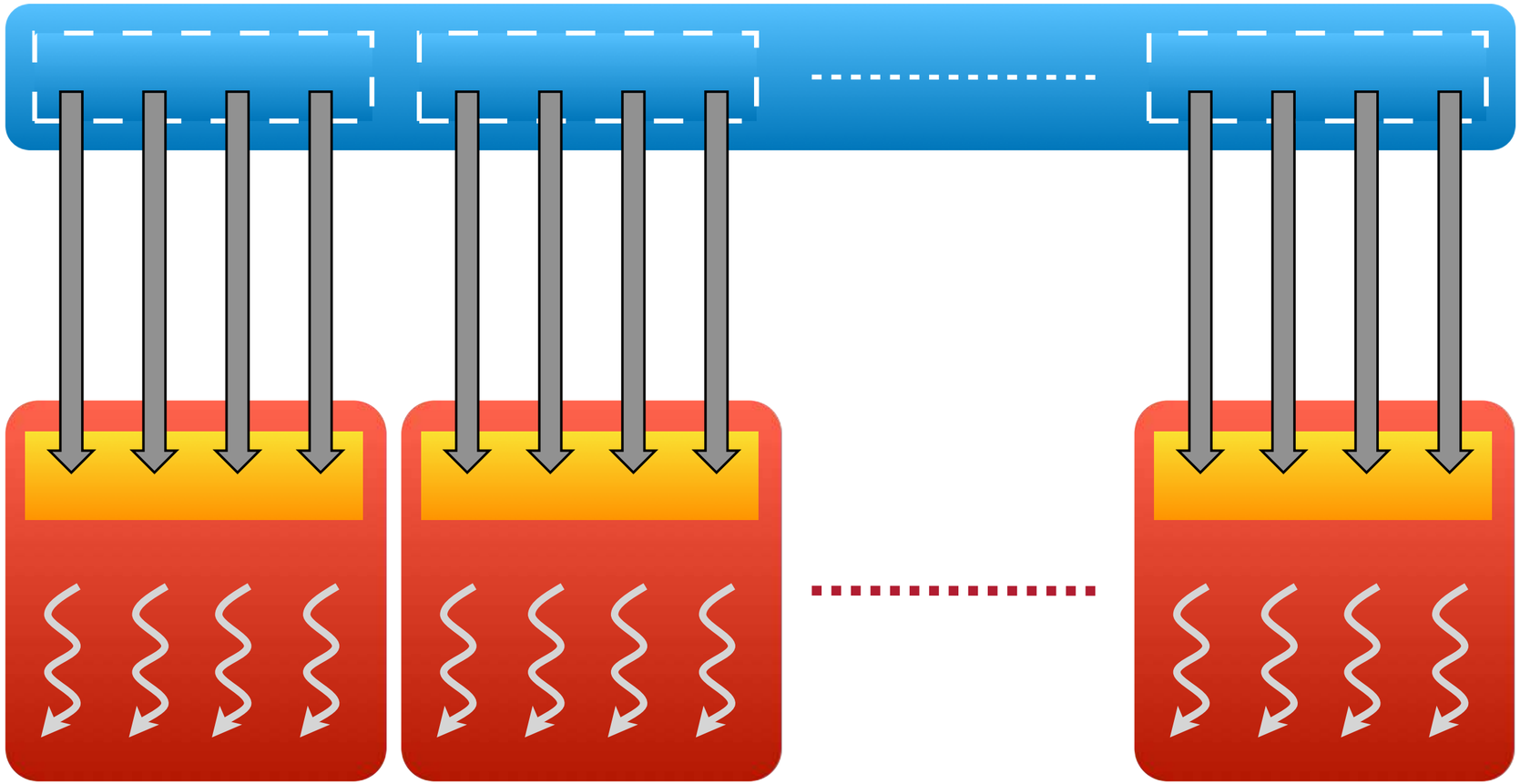
- A (common) massively-parallel programming pattern:



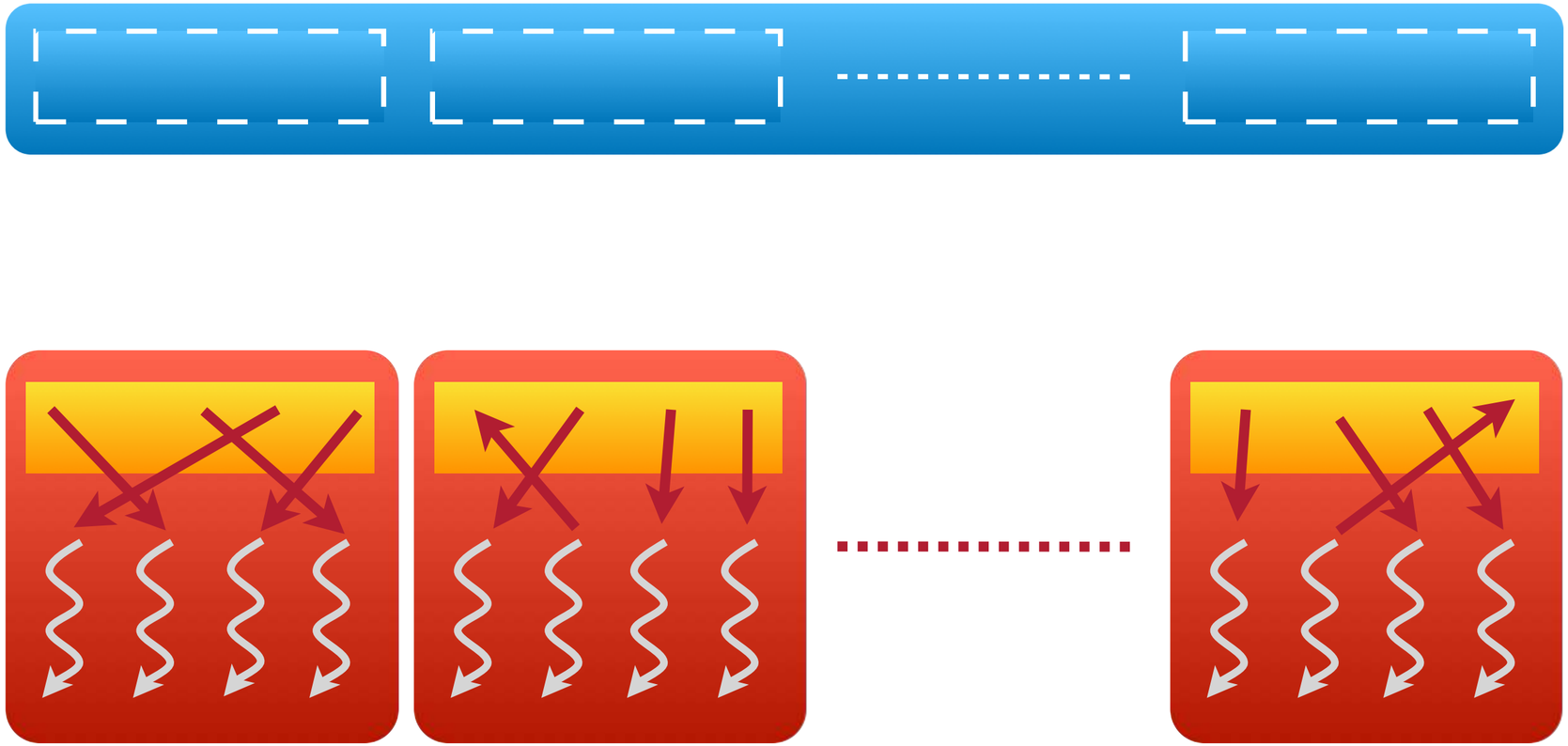
A Common, Massively Parallel Programming Pattern



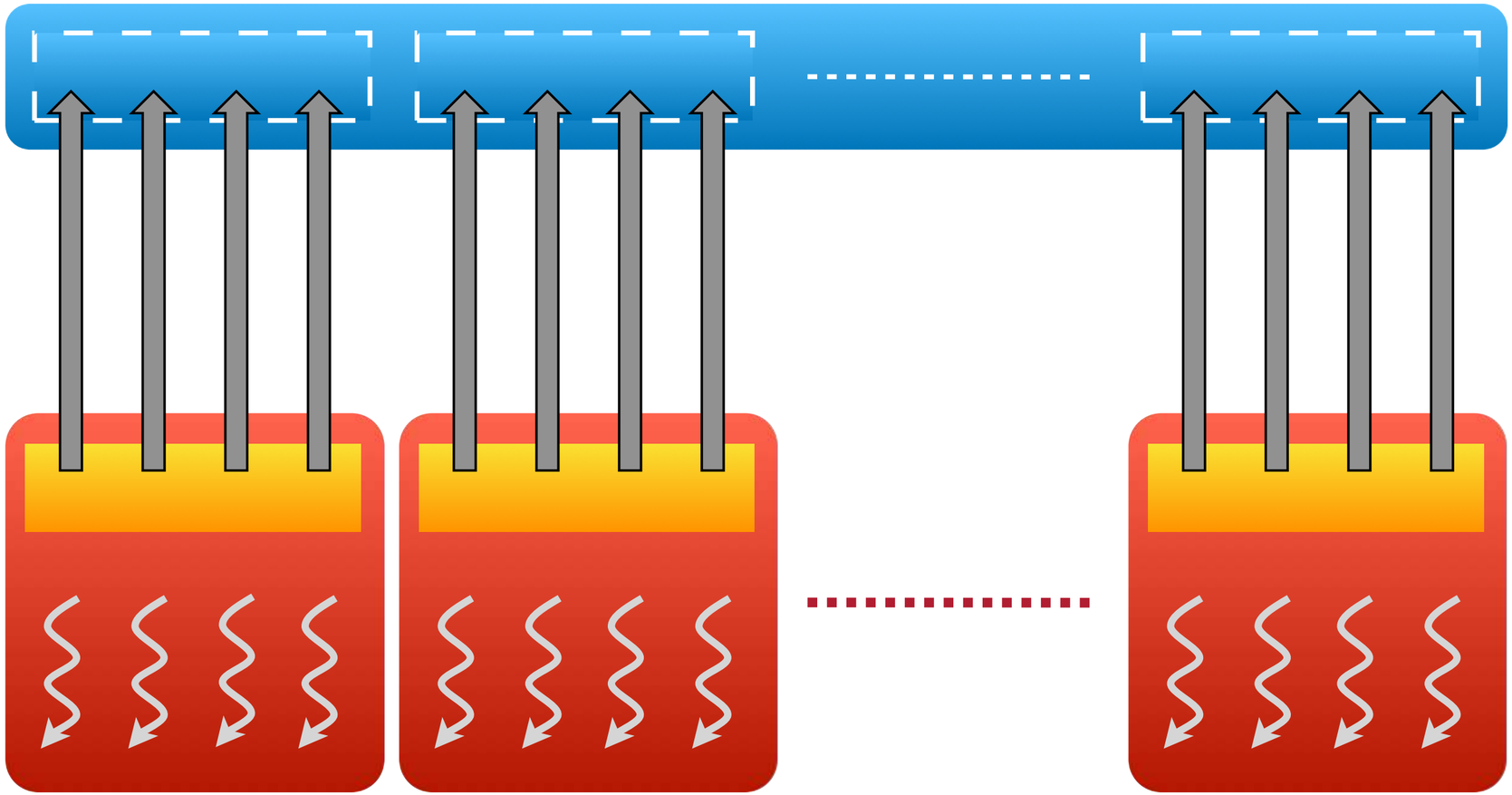
- Partition your domain such that each subset fits into shared memory; handle each data subset with one thread block



- Load the subset from global memory to shared memory; exploit memory-level parallelism by loading one piece per thread



Perform the computation on the subset in shared memory



- Copy the result from shared memory back to global memory

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];
}
```

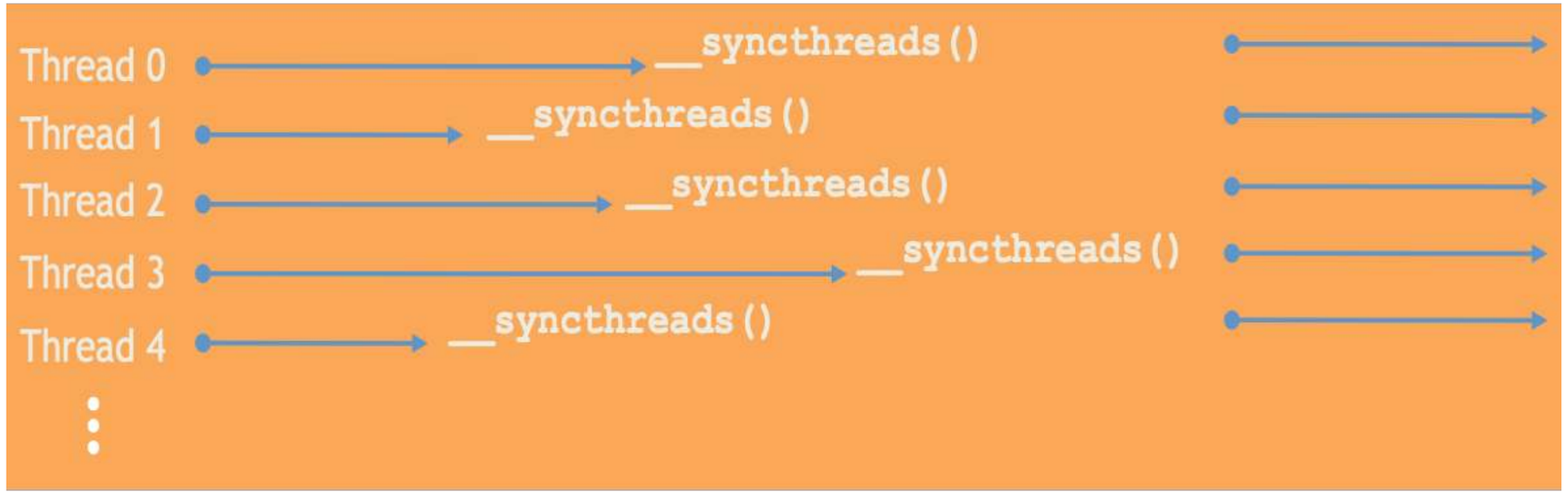
This code contains a bug!

And that bug is probably hard to find!

New Concept: Barrier Synchronization

```
__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 sake of simplicity, 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];
}
```

- The command implements what is called a barrier synchronization (or just "barrier"):



All threads of the block wait at this point in the execution of their program, until all other threads of the same block have arrived at this *same* point → we say "barrier is fulfilled"

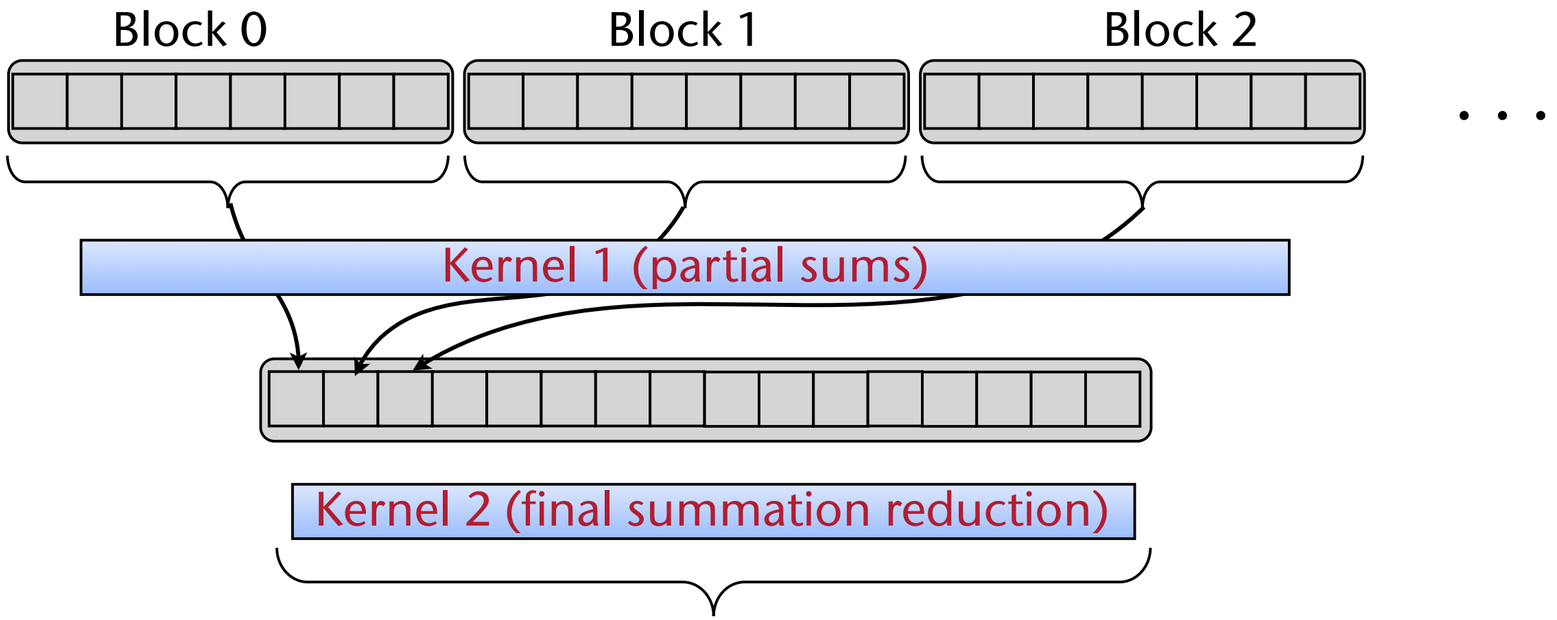
- Warning: threads will only be synchronized *within a block*!

The Complete Dot Product Program

```
// allocate host & device arrays h_a, d_a, etc.
// h_p, 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 Entirely on the GPU

- Sometimes, 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:
  - Idea: only wait for threads that were actually writing to memory ...
- Bug: the barrier will never be fulfilled!

```
__global__
void dotprod( float *a, float *b, float *c, int N )
{
   setup code 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!

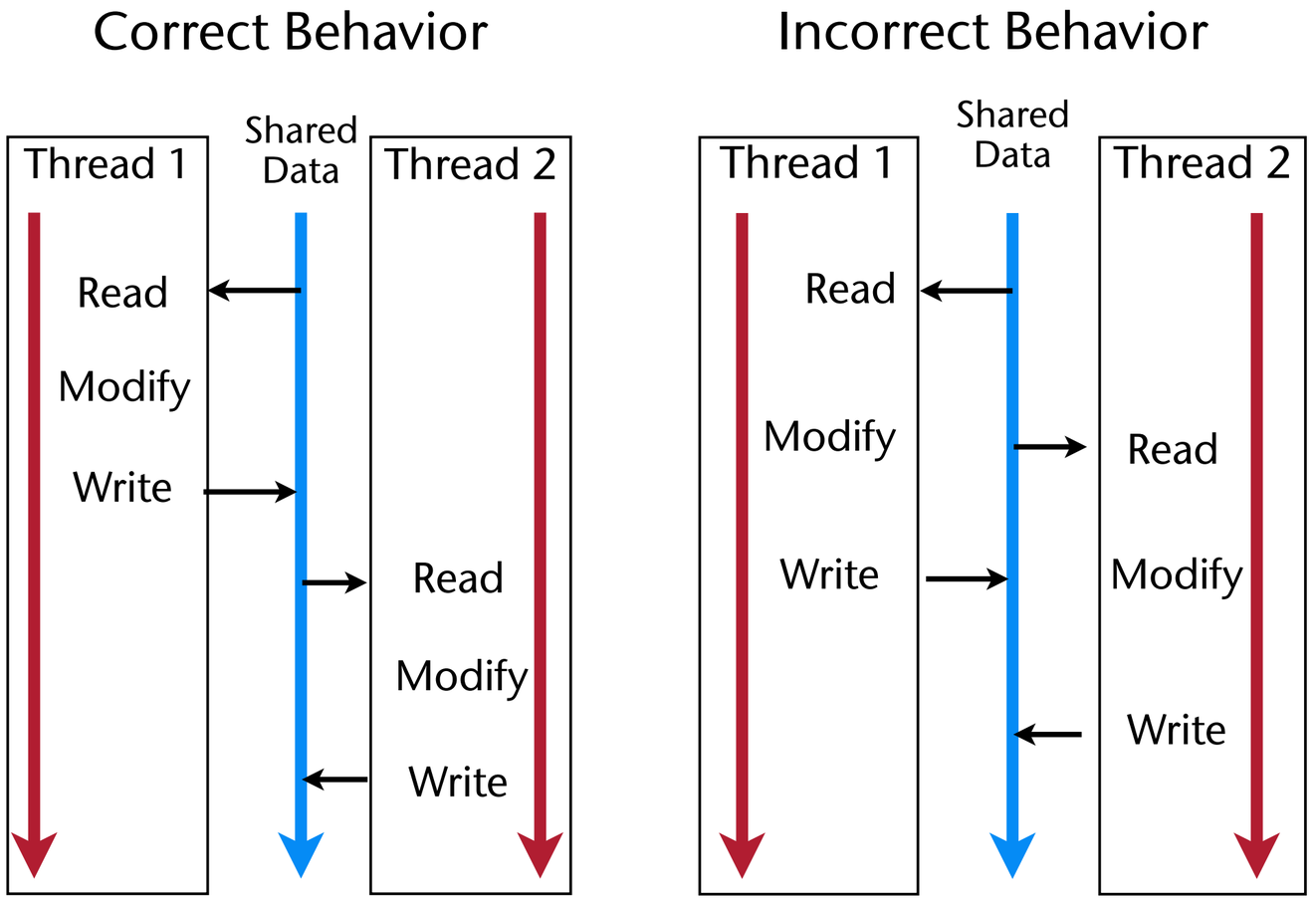
It makes your GPU hang ...!

- Another example, same problem:

```
__global__
void kernel( float radius, .. )
{
   if ( radius > 0.001 )
   {
      area = PI * radius * radius;
      __syncthreads();
   }
   else
   {
      area = 0.0;
   }
   ...
```

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)



Types of 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

    $y = (x_0*x_1, x_0*x_1, x_2*x_3, x_2*x_3, x_4*x_5, x_4*x_5, \ldots)$

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

```
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 *within* a warp, i.e., in lockstep:
  - Everything is fine

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

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

RAW hazard

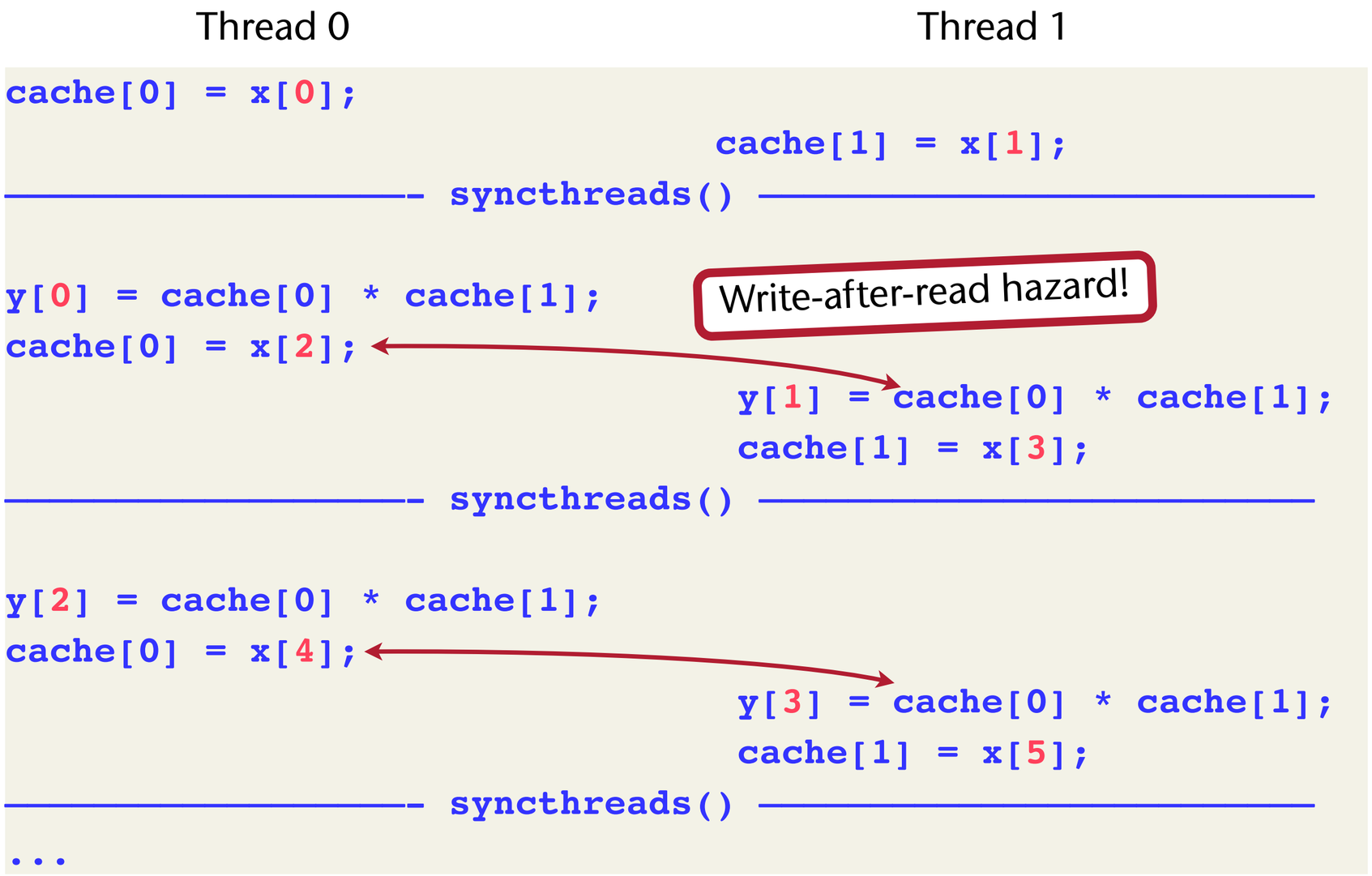
Thread 0	Thread 1
`cache[0] = x[0];`	
`y[0] = cache[0] * cache[1];`	
	`cache[1] = x[1];`
	`y[1] = cache[0] * cache[1];`
`cache[0] = x[2];`	
`y[2] = cache[0] * cache[1];`	
	`cache[1] = x[3];`
	`y[3] = cache[0] * cache[1];`
`cache[0] = x[4];`	
`y[4] = cache[0] * cache[1];`	
	`cache[1] = x[5];`
	`y[5] =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

Read *must* occur *after* write - if read happens *before* write, then result is *wrong*.

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];
   }
}
```

WAR hazard


Thread 0
Thread 1
cache[0] = x[0];
cache[1] = x[1];
__syncthreads()
y[0] = cache[0] * cache[1];
cache[0] = x[2];
Write-after-read hazard!
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()
...

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();
   }
}
```

Just for sake of illustration - this is not a good CUDA implementation!

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 the 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 is asked 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 an order for all words in
  $V = (v_1, v_2, v_3, ..., v_N)$
  (in principle, any order will do)
- Represent D as a vector in $\mathbf{R}^N$:

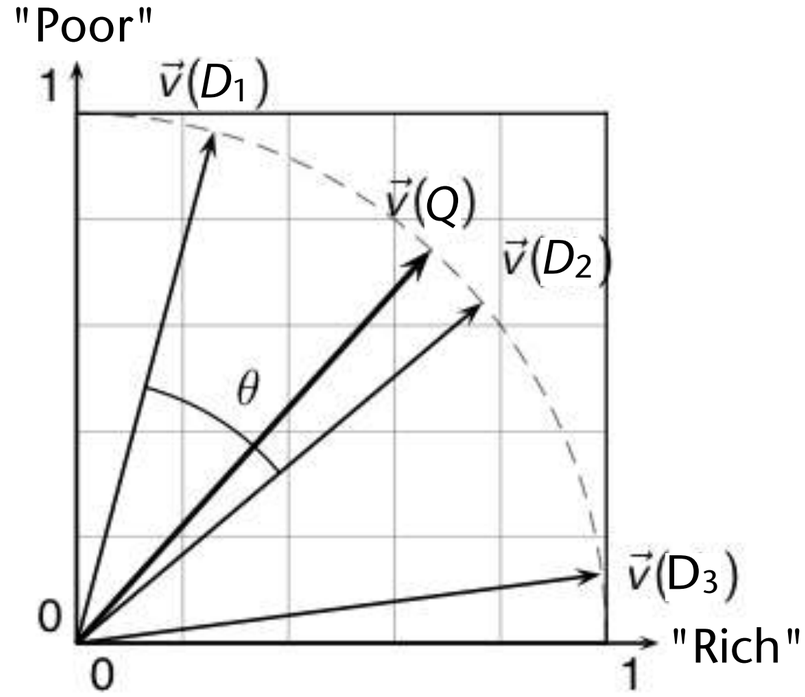
$$D = \left(f(v_1), f(v_2), f(v_3), \ldots, f(v_N) \right)$$

- Note: our vector space is HUGE (N ~ 10,000 – 1,000,000)
  - For each word w, there is one axis in our vector space!

Document Similarity

- Define similarity s between documents D_1 and D_2 as

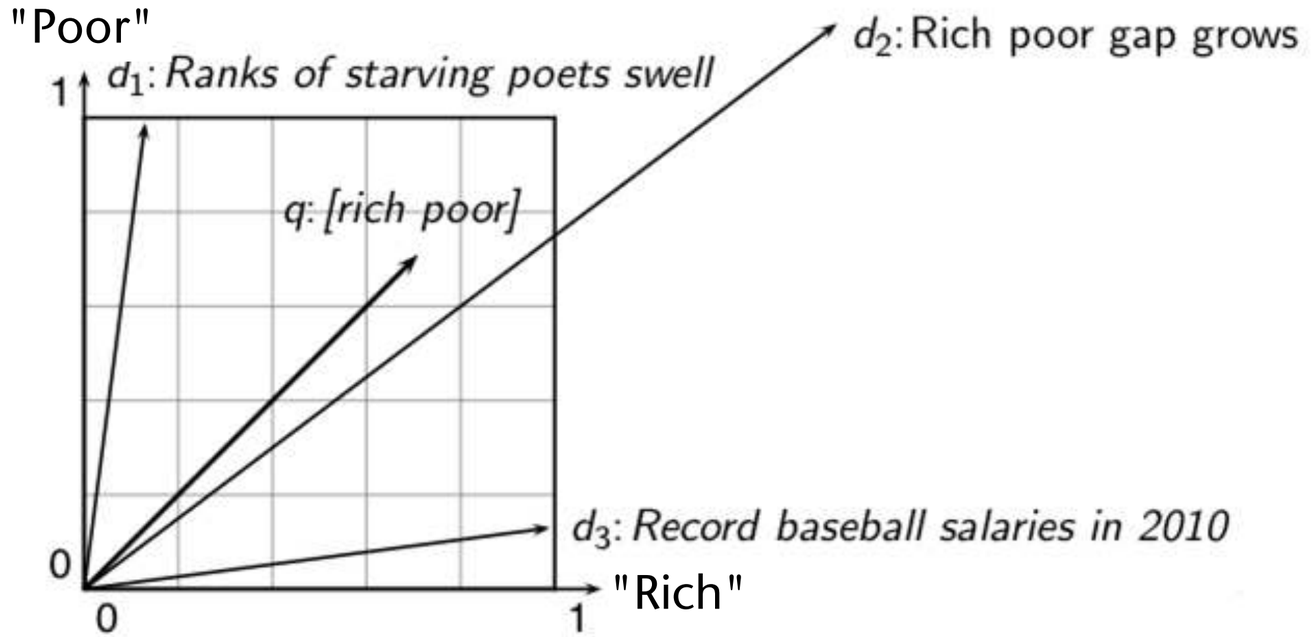
$$s(D_1, D_2) = \frac{D_1 \cdot D_2}{\|D_1\| \cdot \|D_2\|} = \cos(D_1, D_2)$$



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

Notes on the Similarity Measure

- Why not the Euclidean distance $\|D_1 - D_2\|$?
  - Otherwise: documents *D* and *DD* (i.e., *D* concatenated with itself) would be very *dissimilar*!
- Why do we need the normalization by $\frac{1}{\|D_1\| \cdot \|D_2\|}$?
  - Same reason ...

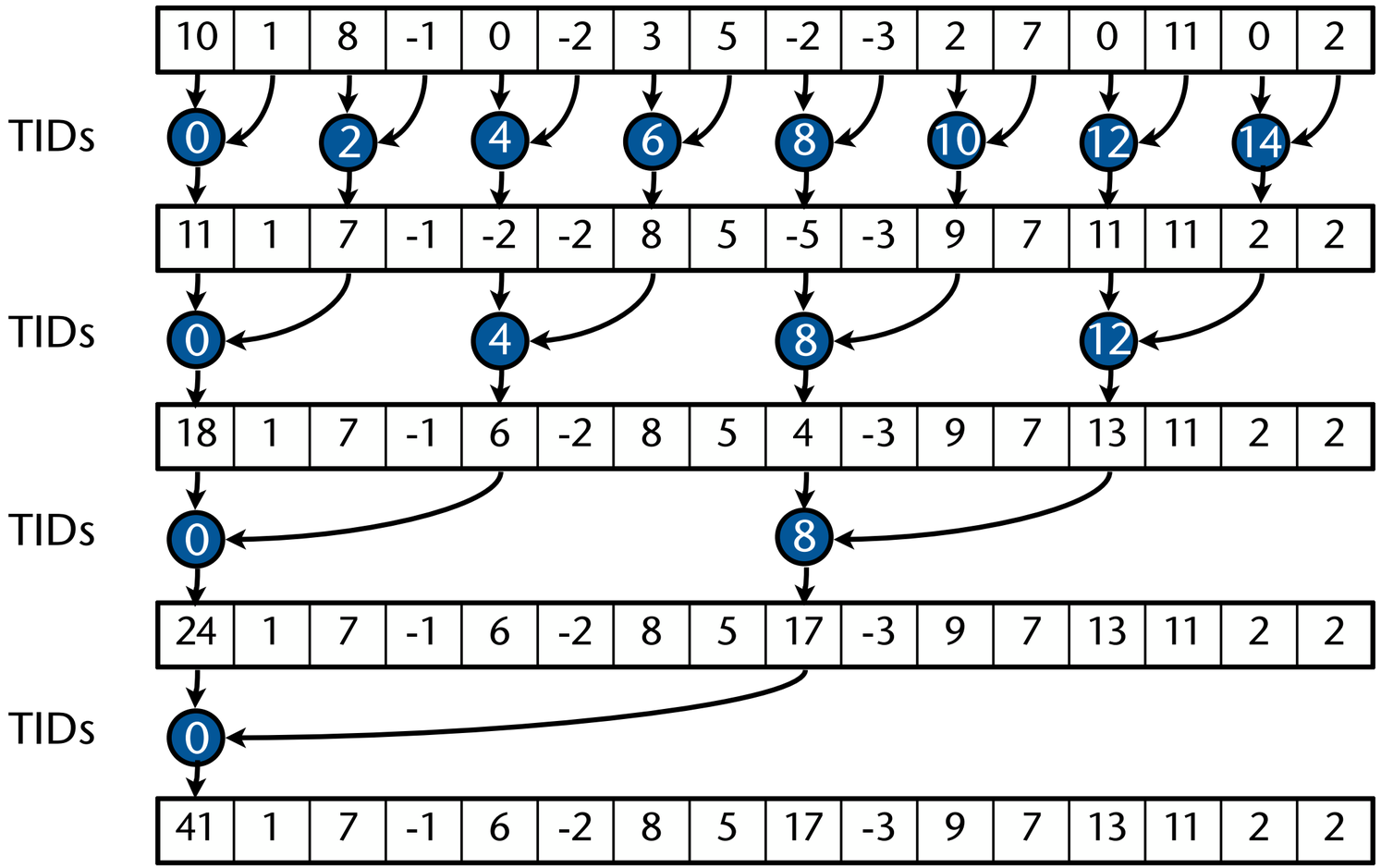


Information Retrieval at Its Best

- Espresso? But I ordered a cappuccino!
- Don't worry, the cosine distance between them is so small that they are almost the same thing.

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

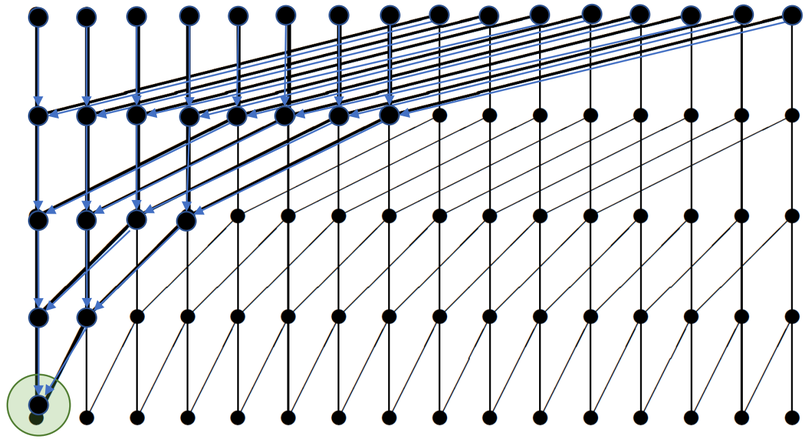
- Further problem: memory access is not coalsced ☹
  - The GPU likes contiguous memory access - more on that in the next chapter

A Real Optimization for Reduction

- Reduction usually does *not* do a lot of computations, mostly memory transfer
- 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 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)

Warp-Level Optimization by Way of Example of Reduction

- Remember this pattern:



- Optimization: *unroll* last 6 iterations (= log(32)+1)
- Gives almost factor 2 speedup over previous version!

```
__global__
void dotprod( ... )
{
   .. beginning as before ..

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

   if ( tid < 32 )
      warpReduce( cache, tid );

   .. do the rest as before ..
```

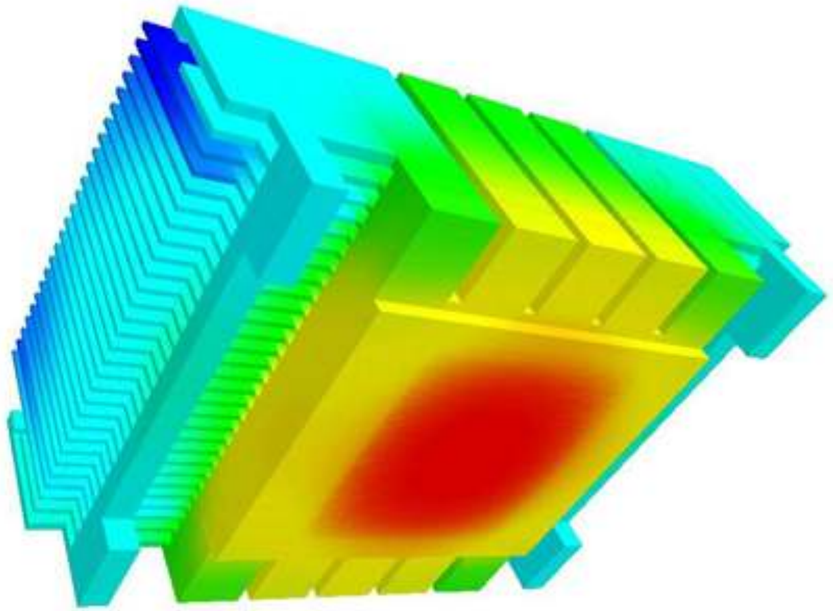
- The unrolled iterations:

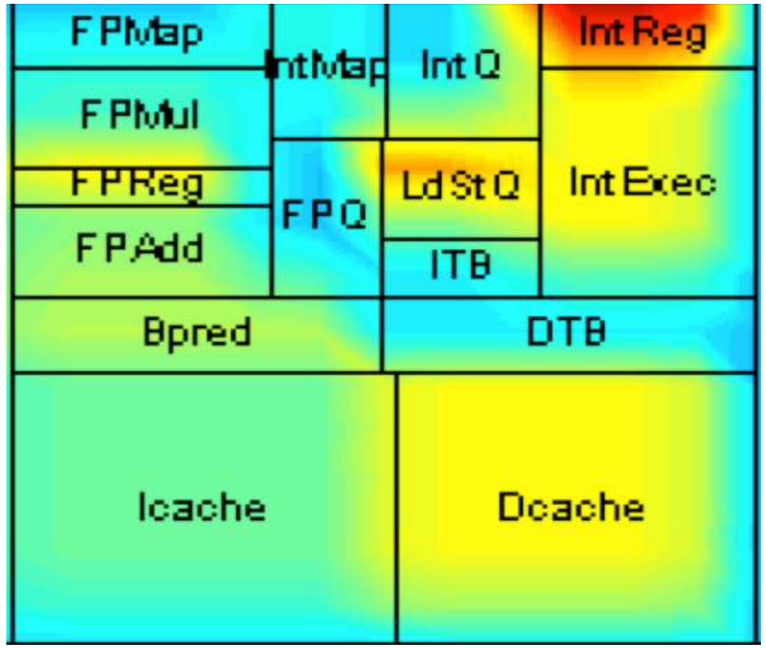
```
__device__
void warpReduce( volatile float *sdata, int tid )
{
   sdata[tid] += sdata[tid + 32];
   sdata[tid] += sdata[tid + 16];
   sdata[tid] += sdata[tid +  8];
   sdata[tid] += sdata[tid +  4];
   sdata[tid] += sdata[tid +  2];
   sdata[tid] += sdata[tid +  1];
}
```

- Bug: has to do with modern compiler optimizations
  - Compiler can decide to keep data in registers (fastest memory of them all) and write to memory *only later*, *not* at the point where write-back is requested by the program
  - **`volatile`** forces *compiler* to actually *always* read & write from/to memory (the compiler doesn't know about the other threads!)

Simulating Heat Transfer in Solid Bodies

- Example: heat simulation of ICs and cooling elements





A Simplistic Model

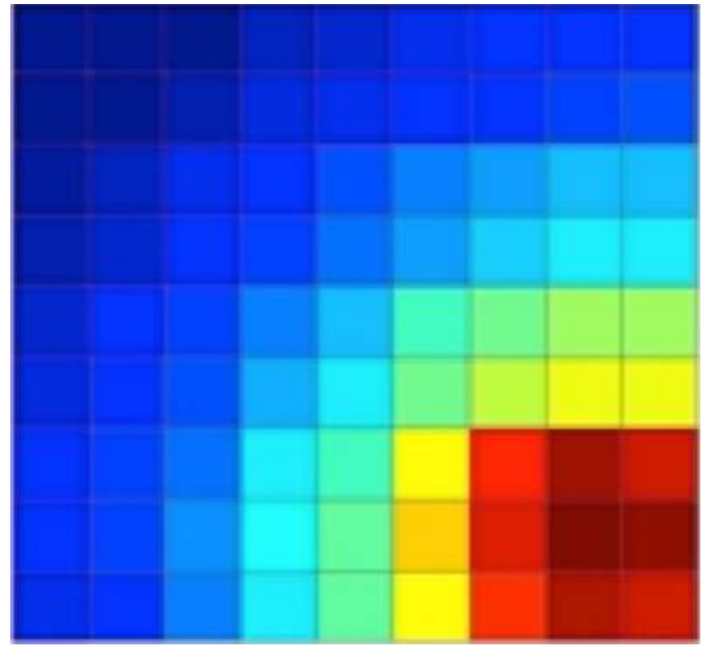
- Assumptions:
  - For sake of illustration, our domain is 2D
  - Discretize domain → 2D grid (common approach in simulation)
  - A few designated cells are "heat sources"
    → cells with constant temperature, i.e., re-set in every iteration
- Simulation model (simplistic):

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

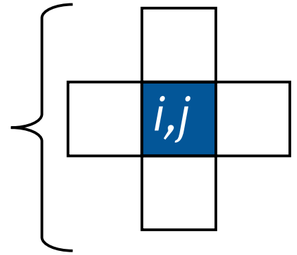
$$T_{i,j}^{n+1} = (1 - N\mu)\,T_{i,j}^{n} + \mu \sum_{(k,l)\in N(i,j)} T_{k,l}^{n} \qquad (1)$$

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

$N(i,j) =$ 

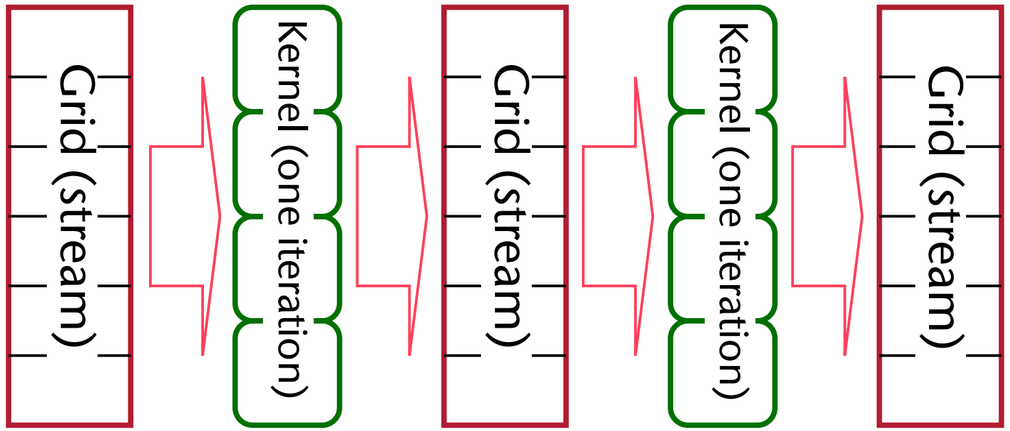
- Energy conservation is maintained, iff $\sum_{i,j} T_{i,j}^{n+1} \stackrel{!}{=} \sum_{i,j} T_{i,j}^{n}$ (2)
- Plugging (1) into (2) yields

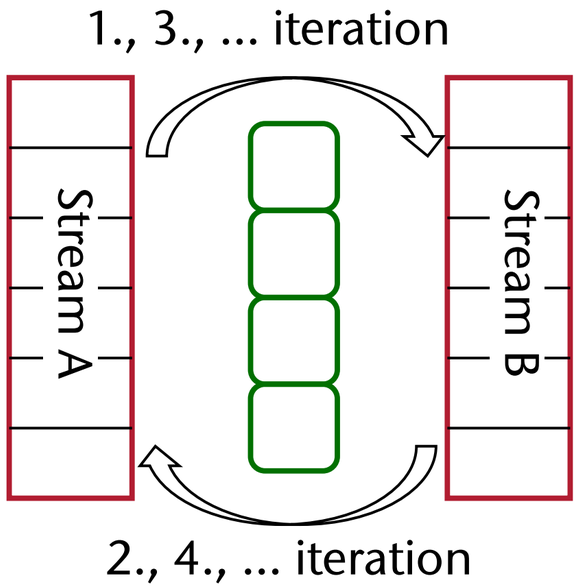
$$\underbrace{(1 - N\mu) \sum_{i,j} T_{i,j}^{n} + \mu \sum_{i,j} \sum_{(k,l) \in N(i,j)} T_{k,l}^{n}}_{= 0} \stackrel{!}{=} \sum_{i,j} T_{i,j}^{n}$$

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

Mass.-Par. Algorithm Design Pattern: Double Buffering

- Observations:
  - Each cell's next state can be computed completely independently
- Instead of allocating new memory for the output of each iteration, we can re-use memory:





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

Algorithm

- One thread per cell
- Kernel for resetting heat sources:

```
if ( cell is in list of heat cells ):
   read temperature from constant "heating stencil"
```

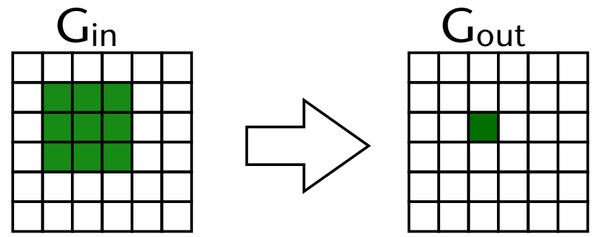
- Kernel for one transfer step:

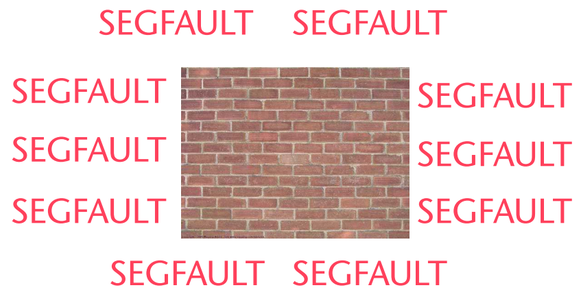
```
Read all neighbor cells:  input_grid[tid.x+-1][tid.y+-1]
Accumulate and compute new temperature for "own" cell
Write new temperature in output_grid[tid.x][tid.y]
```

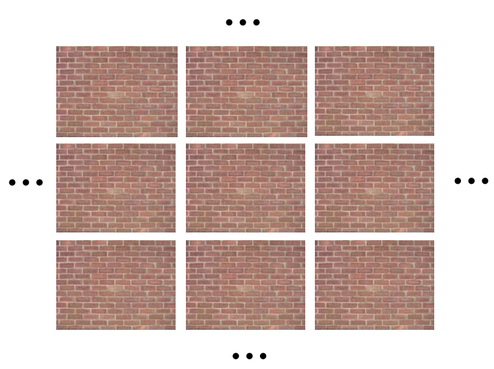
- Swap pointers to input & output grid (done on host) and repeat
- Challenge: *border cells*! (very frequent problem in sim. codes)
  - Use if-then-else in above kernel?
  - Use extra kernel that is run only for border cells?
  - Introduce padding around domain? Arrange domain as torus?

Texture Memory

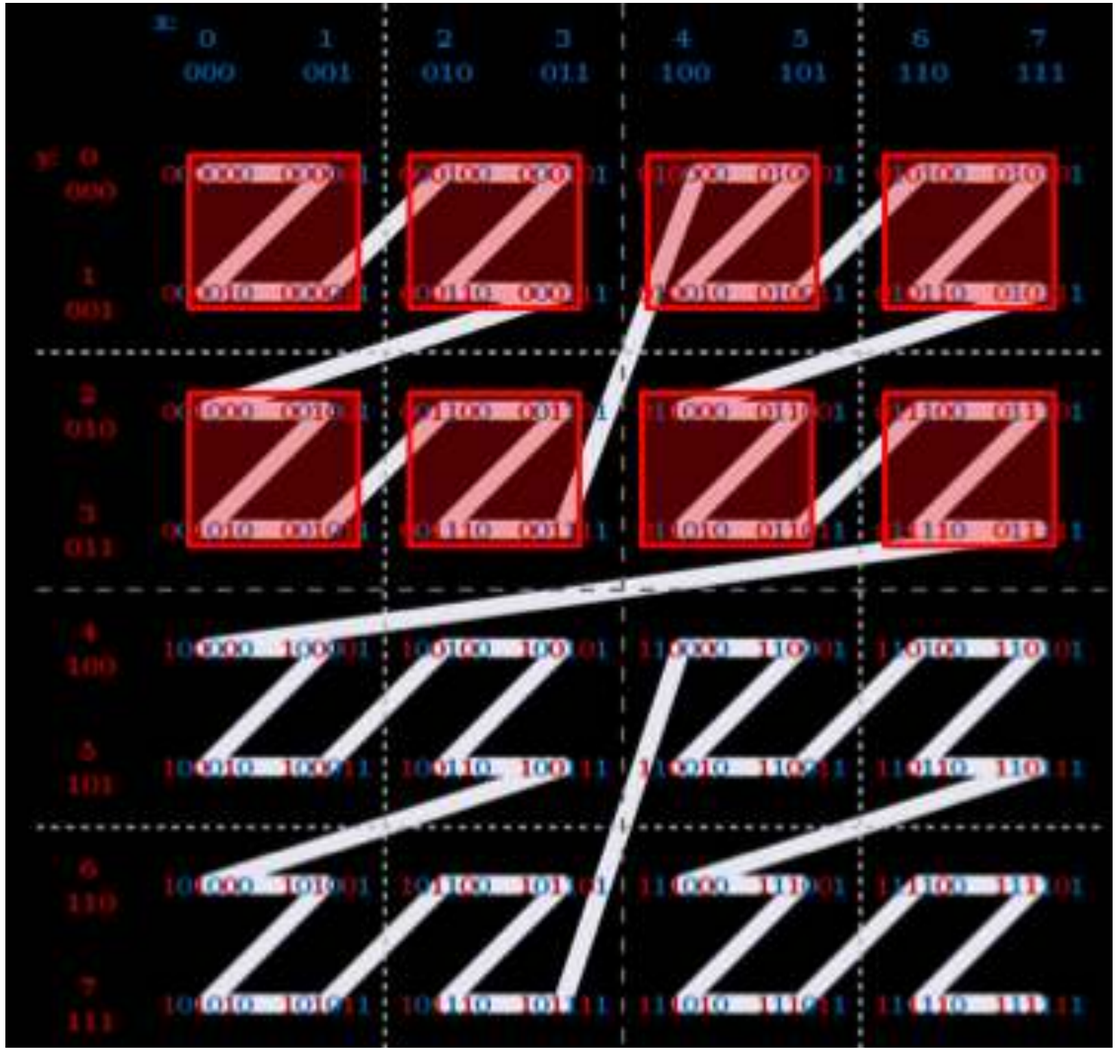
- Many computations have the following characteristics:
  - They work on a 2D/3D *grid*, and each iteration transforms an input grid into an output grid
  - We can run one *thread per grid cell*
  - Each thread only needs to look at *neighbor cells*
- For this kind of algorithms, there is texture memory:
  - 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" or Nvidia's "CUDA C Programming Guide"







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



Example: Sobel Operator (Edge Detection)

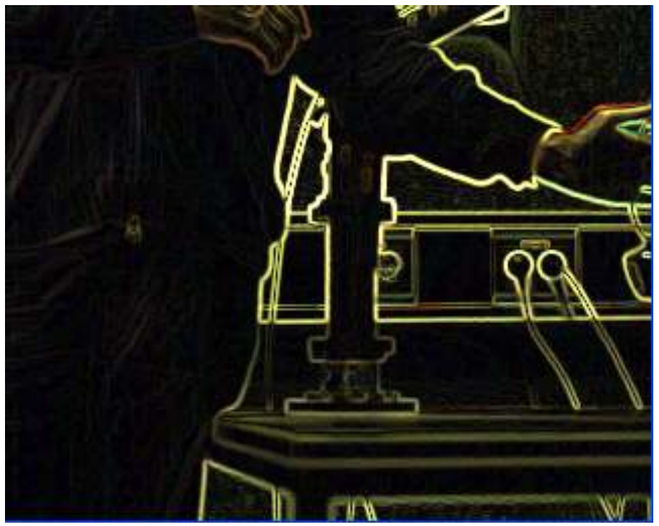
- The convolution kernel:

$$S_x = \frac{1}{8}\begin{pmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{pmatrix} \star I\ , \quad S_y = \frac{1}{8}\begin{pmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix} \star I\ , \quad E = \sqrt{S_x^2 + S_y^2}$$

  where I is the input image, E is the output image

- Example images:

Input image



Edge image

Digression: Derivation of the Sobel Operator

FYI

- Edge = sudden change in image when going left-right/up-down
- Equivalent formulation: large spatial derivative $\frac{\partial I}{\partial x}$ or $\frac{\partial I}{\partial y}$ indicates edge
- Approximate derivative by finite differences:

$$\frac{\partial I}{\partial x} \approx \frac{I(x,y)-I(x+h,y)}{h} \approx \frac{I(x-h,y)-I(x+h,y)}{2h}$$

  with h = small distance

- Choosing $h = 1$ pixel, we *could* compute the spatial derivatives in x and y direction by these two convolution kernels:

$$D_x = \frac{1}{2}\begin{pmatrix} -1 & 0 & +1 \end{pmatrix} \quad , \quad D_y = \frac{1}{2}\begin{pmatrix} +1 \\ 0 \\ -1 \end{pmatrix}$$

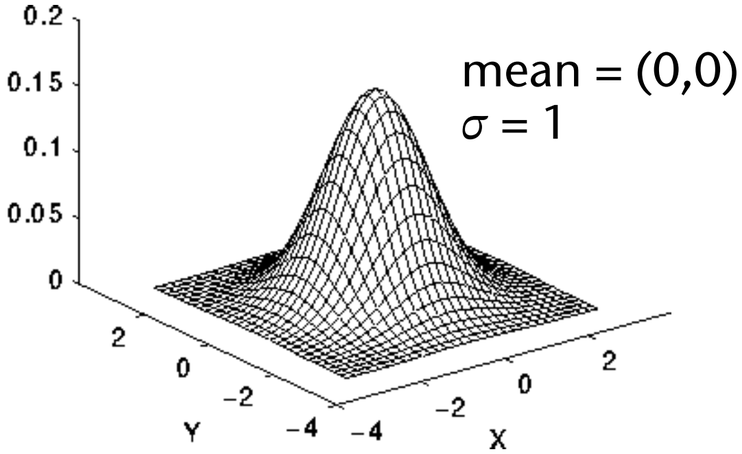
- Problem: noise
  - Taking the derivative actually "amplifies" noise in the image

- Simple trick to reduce noise: convolve image with Gaussian kernel G

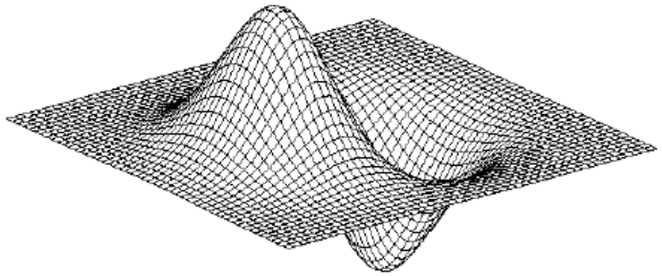
$$I' = I \star G$$

where

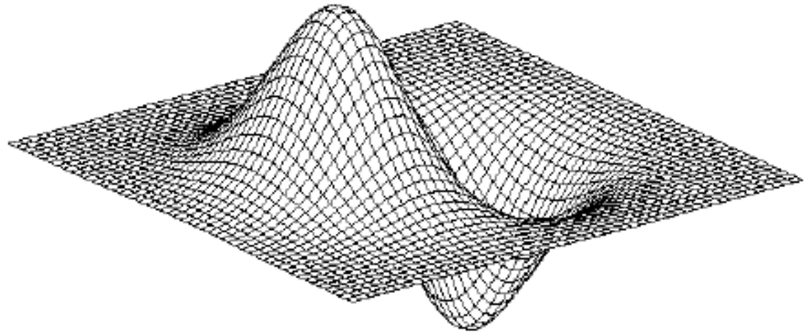
$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$



- Utilize the fact that $\frac{\partial}{\partial x}(I \star G) = I \star \frac{\partial}{\partial x} G$



- How to compute the approximation of the partial derivative of the Gaussian kernel over a pixel grid
  - Integrate the value of the function over the whole pixel (by summing it at 0.001 increments)
  - Rescale the array so that the corners have a value = 1
  - Round all values to nearest integer (if integer arithmetic)
  - Matrix scaling value = 1 / sum of all kernel values



- The 3x3 and 5x5 kernels:

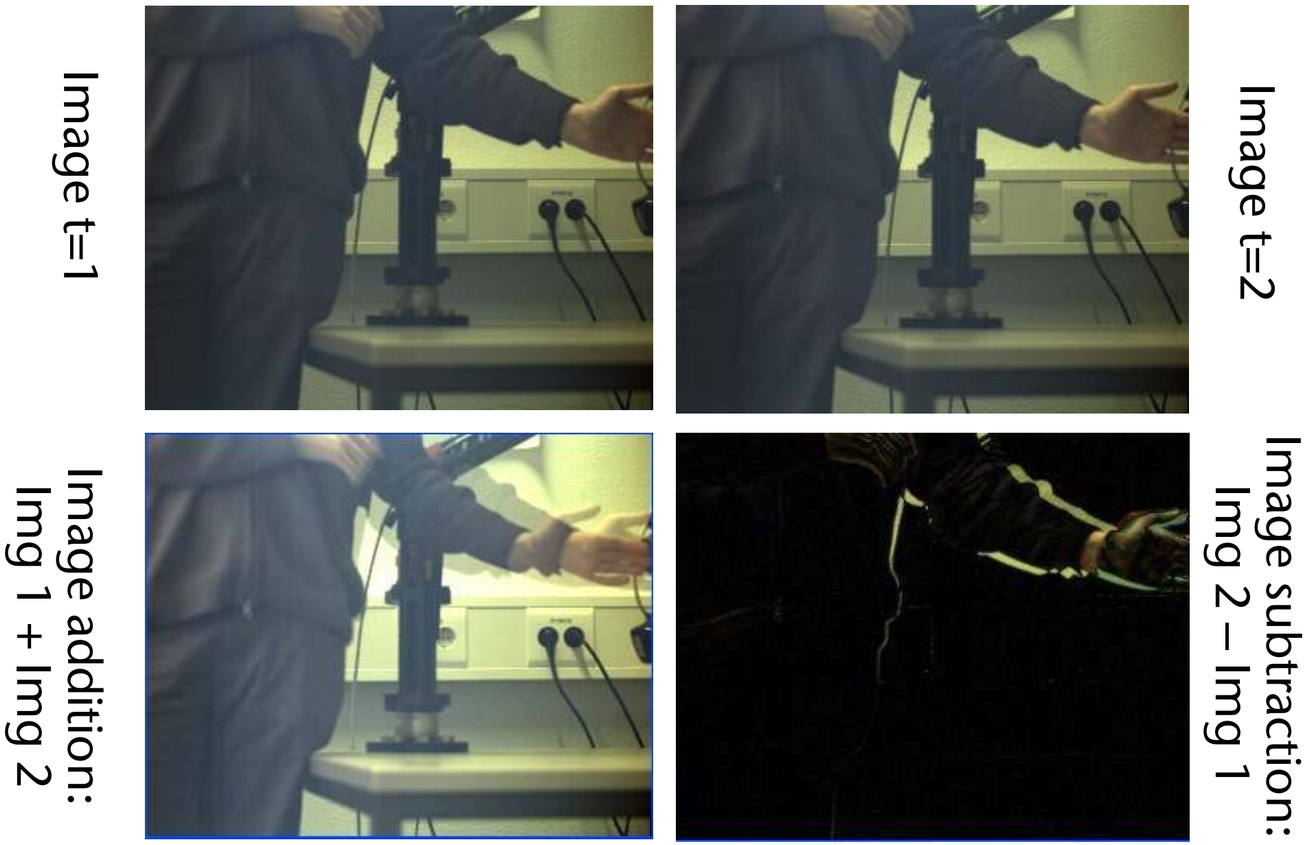
$$G_x = \frac{1}{8}\begin{pmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{pmatrix}$$

$\frac{1}{273}$

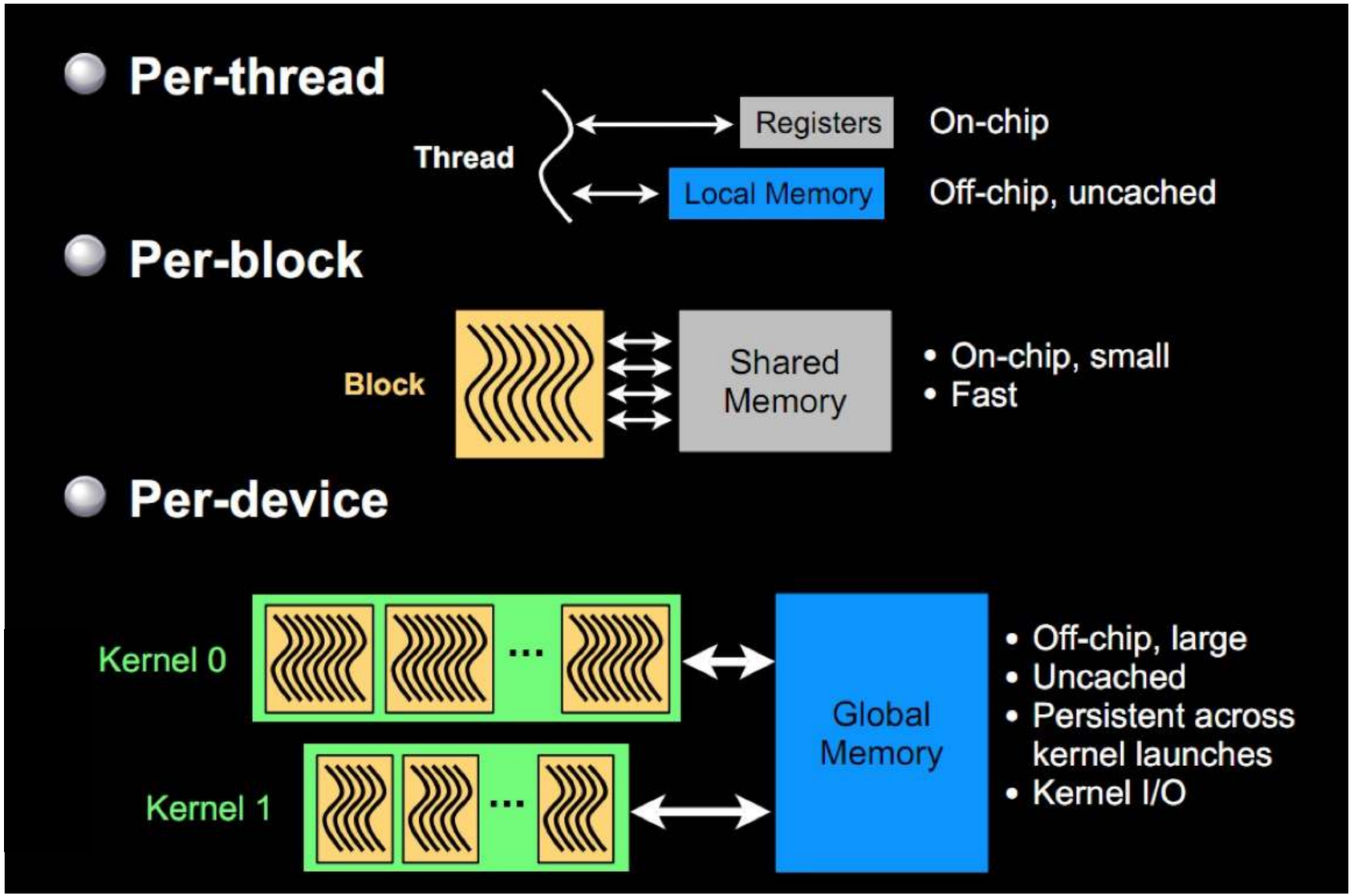
1	4	7	4	1
4	16	26	16	4
7	26	41	26	7
4	16	26	16	4
1	4	7	4	1

Other Applications of Texture Memory

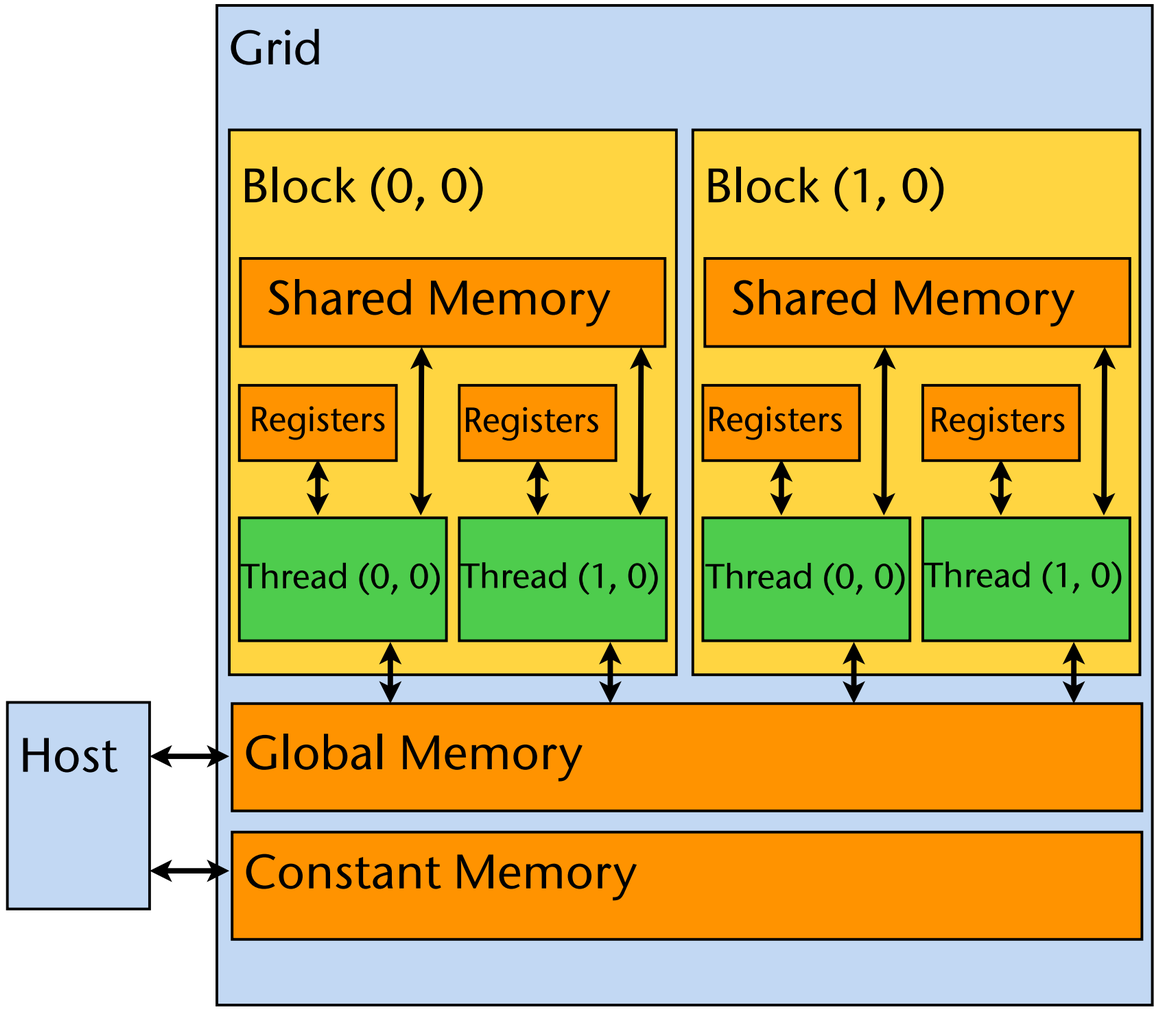
- Most image processing algorithms exhibit this kind of locality
- Trivial examples:



CUDA's Memory Hierarchy



Managed 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

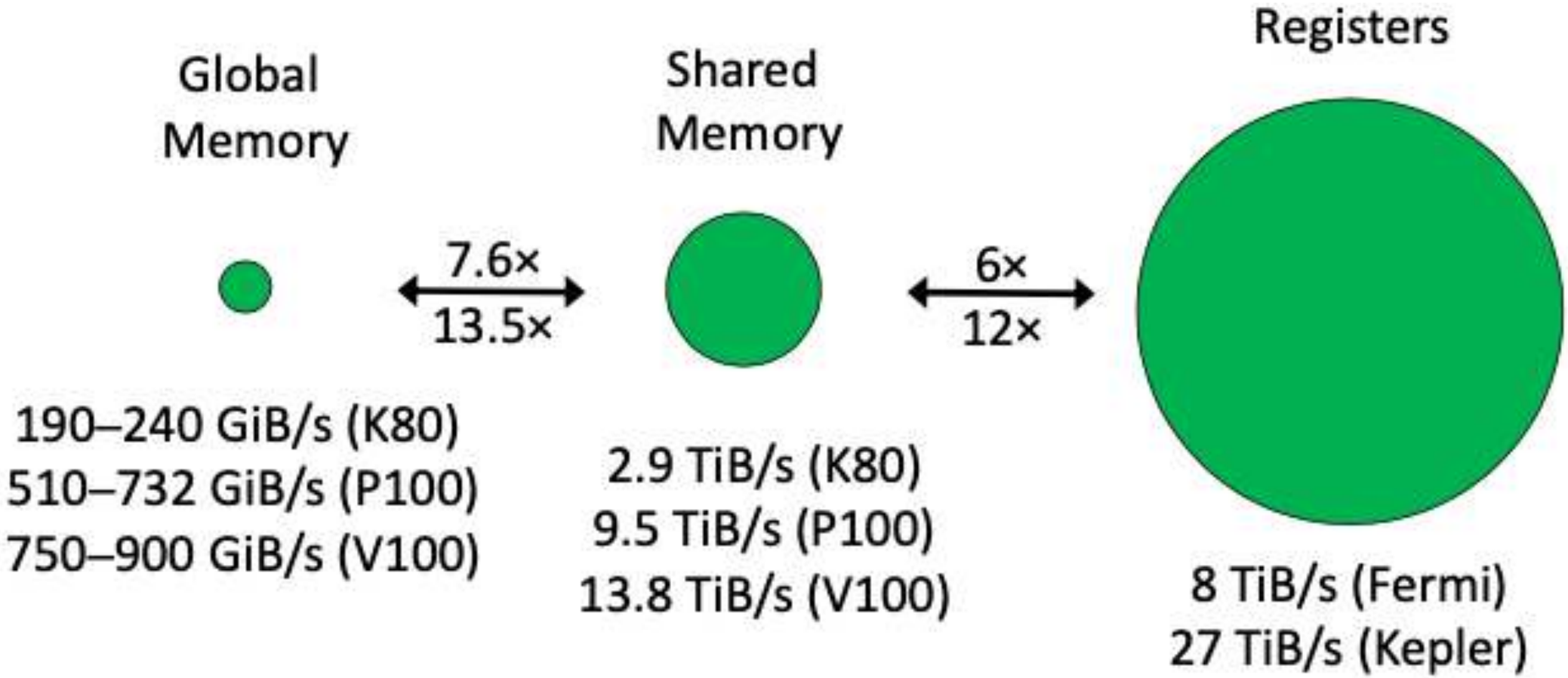
- Remarks:
  - **`__device__`** is optional when used with **`__local__`**, **`__shared__`**, or **`__constant__`**
  - Beware: Automatic variables without any qualifier reside in a register
    - Except arrays, which reside in local memory (slow)

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

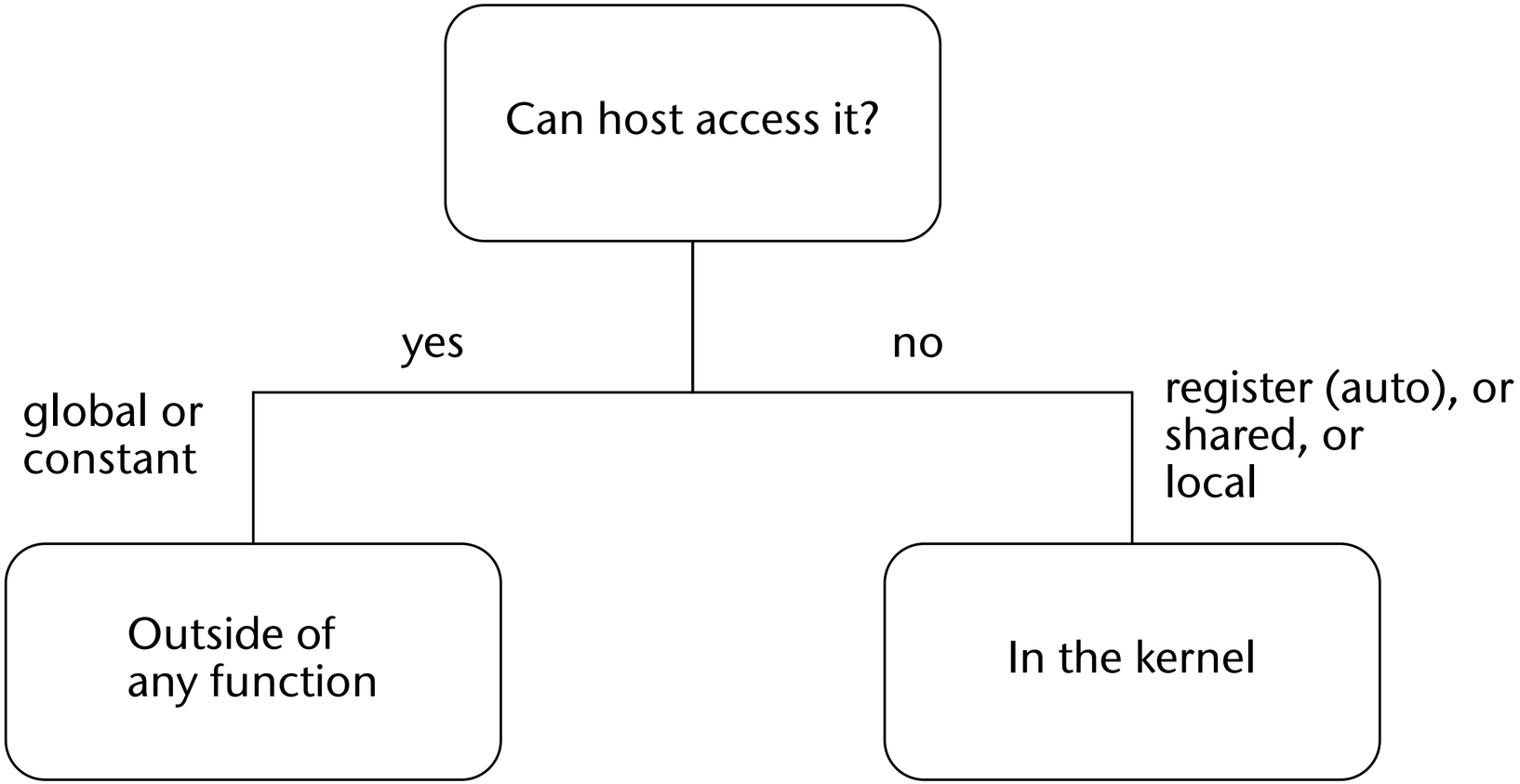
Transfer Bandwidths



Hardware Implementation: Execution Model

- Each thread block of a grid is split into warps, each warp gets executed by one multiprocessor (SM)
- Each thread block is executed on a *single* multiprocessor
  - The shared memory resides in the on-chip memory
- A multiprocessor can execute multiple blocks concurrently
  - Shared memory and registers are partitioned among the threads of all concurrent blocks
  - So, decreasing shared memory usage (per block) and register usage (per thread) increases number of blocks that can run concurrently!
- At compile time: compiler gathers these numbers per thread & block
- At runtime: scheduler does the "scheduling" of blocks and resources

Where to Declare Variables?



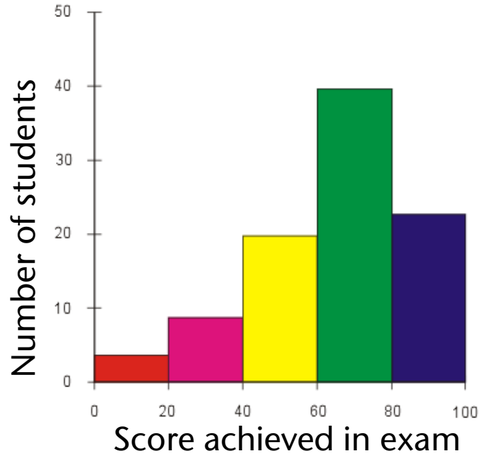
Massively Parallel Histogramm Computation

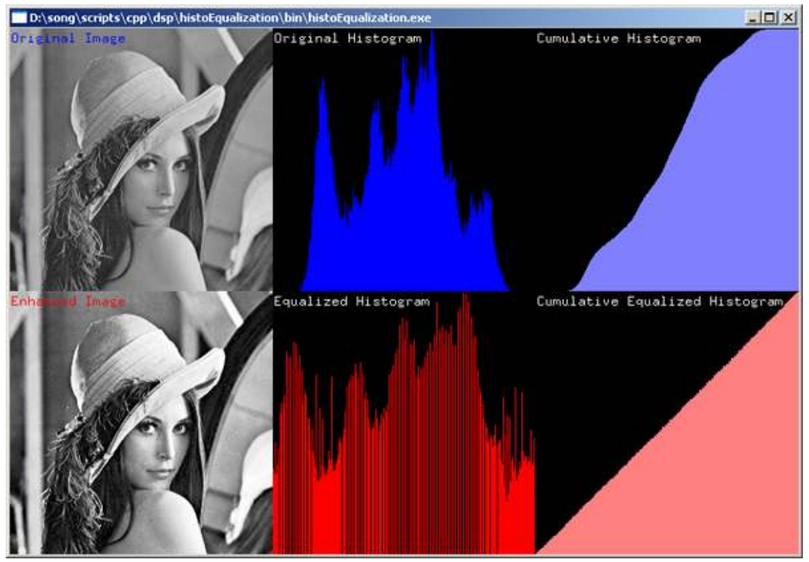
- Definition (for images):

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

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

- Applications: many!
  - Huffman compression (see computer science 2nd semester)
  - Histogram equalization (see Advanced Computer Graphics)





The Sequential Algorithm

```
unsigned char input[MAX_INP_SIZE];// e.g. image
int input_size;                   // # valid pixls in input
unsigned int histogram[256];      // here, 256 levels

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

// verify histogram (kind of a unit test)
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" intensity level in the image
  - At the end, each thread stores its level count in its histogram slot
- Disadvantage: this algo is not so massively parallel ...

- New approach: "each thread visits only a few pixels"
- The setup on the host side:

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

- Notes:
  - Letting **`threadsPerBlock`** = 256 will make things much easier in our case
  - Letting

    `nBlocks` = (#multiprocessors [SMs] on the device) * 2

    is a good rule of thumb
  - On current hardware, this yields ~ 16384 threads

- The kernel on the device side:

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

- Prototype of atomicAdd():

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

where **`T`** can be **`int`**, **`float`** (and a few other types)

- Semantics: *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

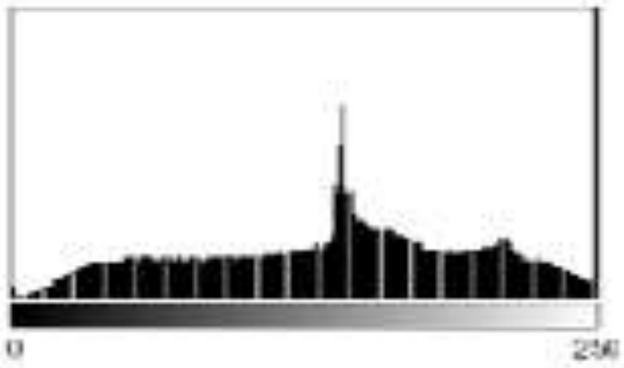
```
computeHistogram( unsigned char * input,
                  long int input_size,
                  unsigned int histogram[256] )
{
   __shared__ unsigned int partial_histo[256];

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

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

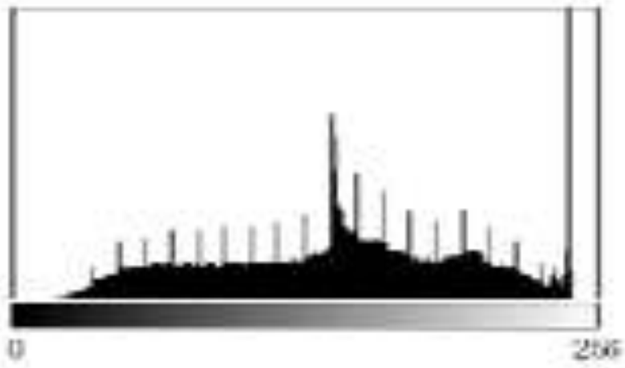
Note: now it's obvious why we chose 256 threads/block
(and in real programs,
we should turn 256 into a
constant variable and write
a comment about it!)

Further Applications Histograms: Detecting Image Manipulation

Original

Manipulated





More Atomic Operations

- All programming languages / libraries / environments providing for some kind of parallelism/concurrency have one or more of the following atomic operations:
  - **`int atomicExch( int* address, int val ):`**
    Read old value at address, store **`val`** in address, return old value
  - Atomic AND: performs the following in one atomic operation

```
int atomicAnd( int* address, int val )
{
    int old = *address;
    *address = old & val;
    return old;
}
```

  - Atomic minimum/maximum operation (just analogous to AND)
  - Atomic compare-and-swap (CAS), and several more ...

- 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_val ← value in memory location address
   if compare == old_val:
      store new_val → memory location address
   return old_val
```

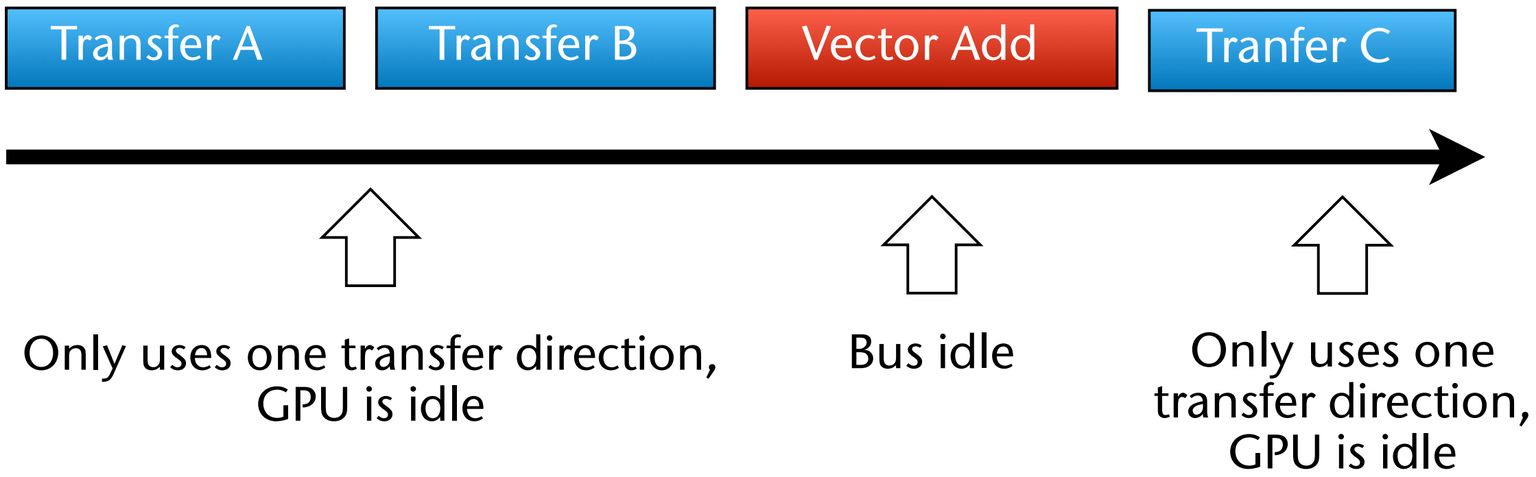
- Theorem (w/o proof):
  All other atomic operations can be implemented using atomic compare-and-swap.
- (In practice, they are implemented directly in hardware)

- Example:

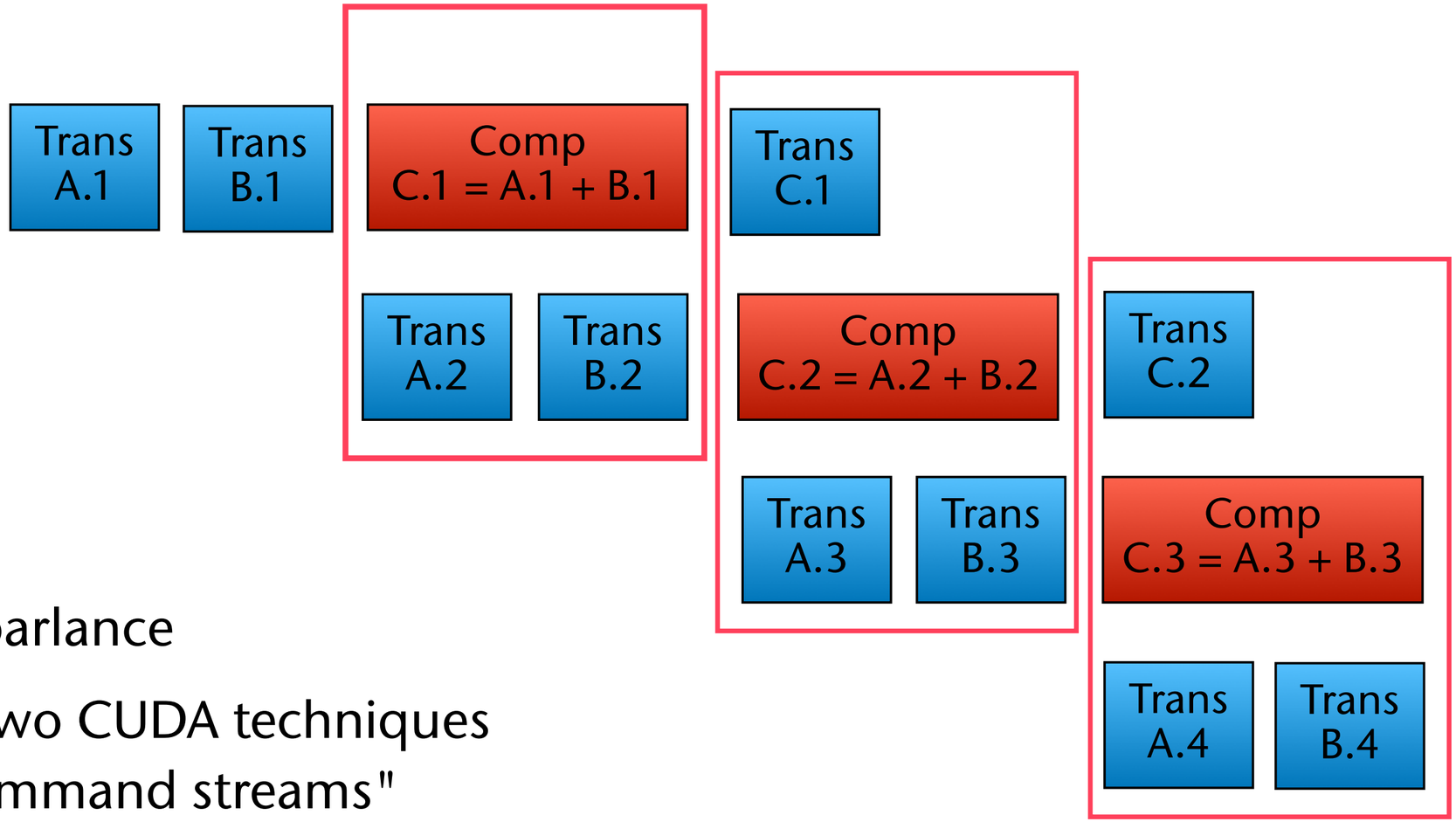
```
atomic_add( address, incr ):
   current_val := value in memory location address
   repeat
      new_val     := current_val + incr
      assumed_val := current_val
      current_val := compare_and_swap( address,
                                       assumed_val,
                                 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

Advanced GPU & Bus Utilization

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



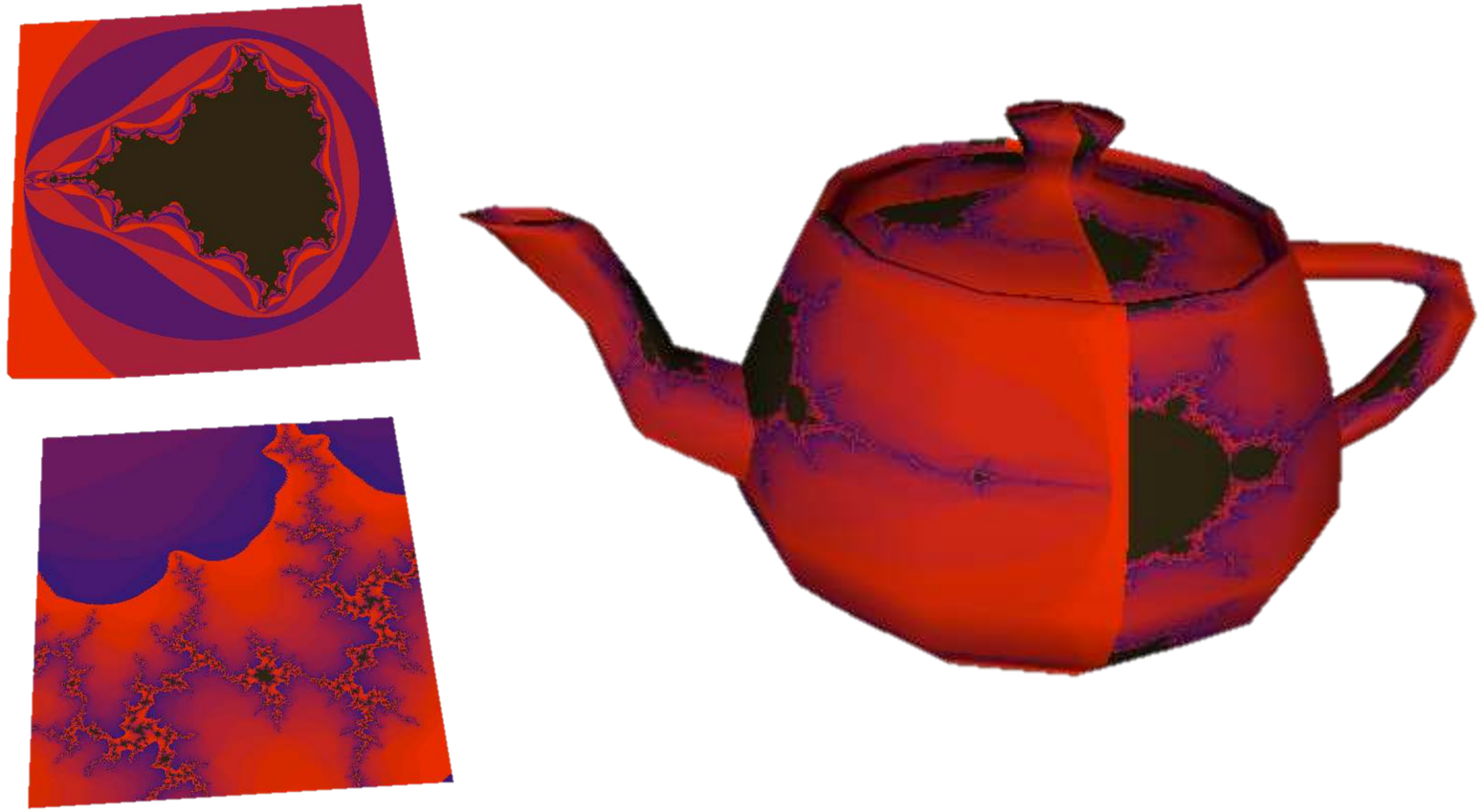
- Solution: pipelining (the "other" *parallelism paradigm*)



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

Graphics Interoperability

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



For More Information on CUDA ...

- *CUDA C Programming Guide* (zur Programmiersprache)
- *CUDA C Best Practices Guide* (zur Performance-Steigerung)