# 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

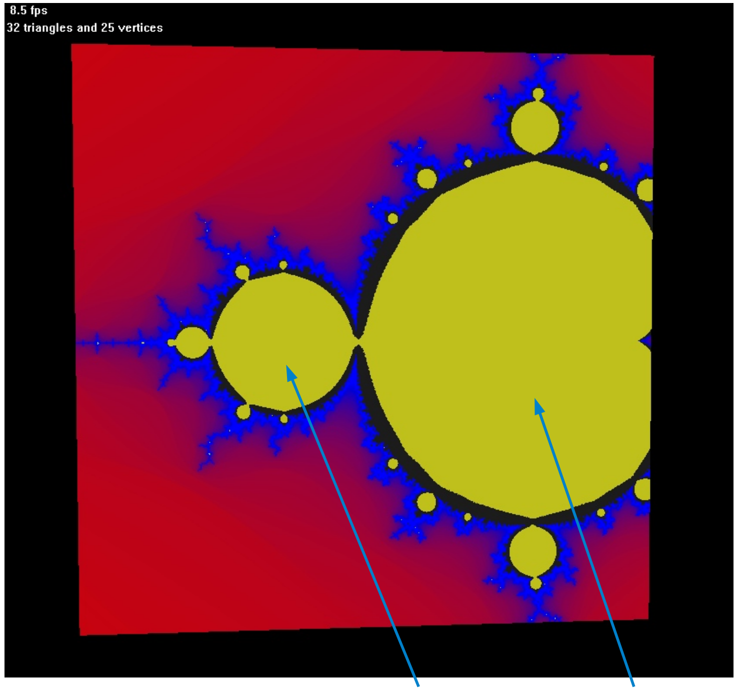
- Example for the latter: make **`cuComplex`** usable on both device and host

```
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. ...
};
```

- An "optimization":
  - 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 (should happen much earlier in most cases)
  - Maintain an array of the *k* most recent elements of the sequence
- Last time I checked: 4x slower than the brute-force version!




All points here converge towards cycle of length 2

All points here converge towards fixed point

# 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

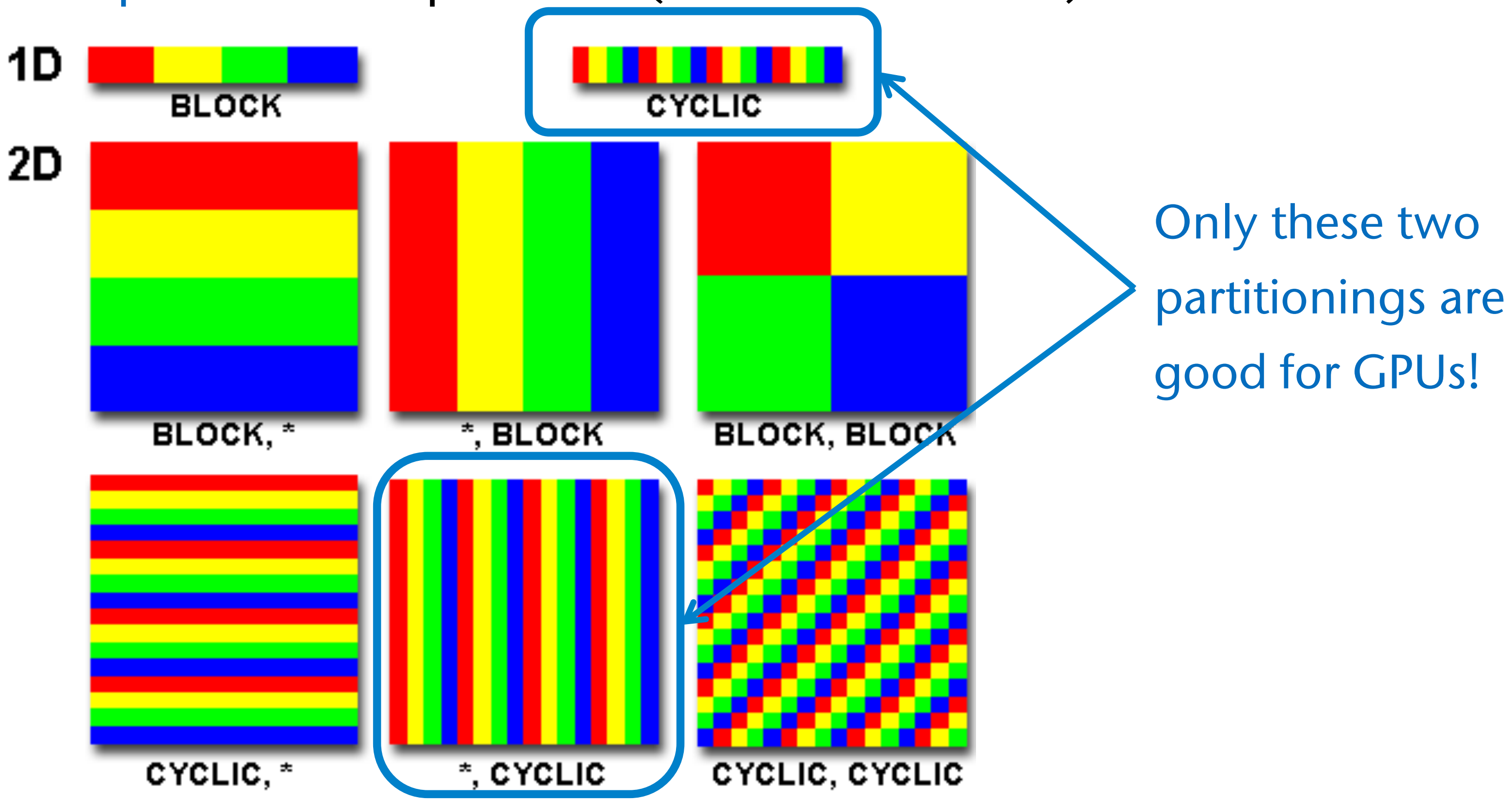
| DEVICE PROPERTY | DESCRIPTION |
|---|---|
| `char name[256];` | An ASCII string identifying the device (e.g., `"GeForce GTX 280"`) |
| `size_t totalGlobalMem` | The amount of global memory on the device in bytes |
| `size_t sharedMemPerBlock` | The maximum amount of shared memory a single block may 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

- Problem: your input, e.g. the vectors, is larger than the maximally allowed size along one dimension?
  - I.e., what if `vec_len > maxThreadsDim[0] * maxGridSize[0]`?
- Solution: partition the problem (color = thread ID)



# Example: Adding Huge Vectors

- Vectors of size 100,000,000 are not uncommon in high-performance computing (HPC) …
- 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  nb_sqrt = (int)( ceilf( sqrtf( n_blocks ) ) );
dim3 blocks( nb_sqrt, nb_sqrt );
```

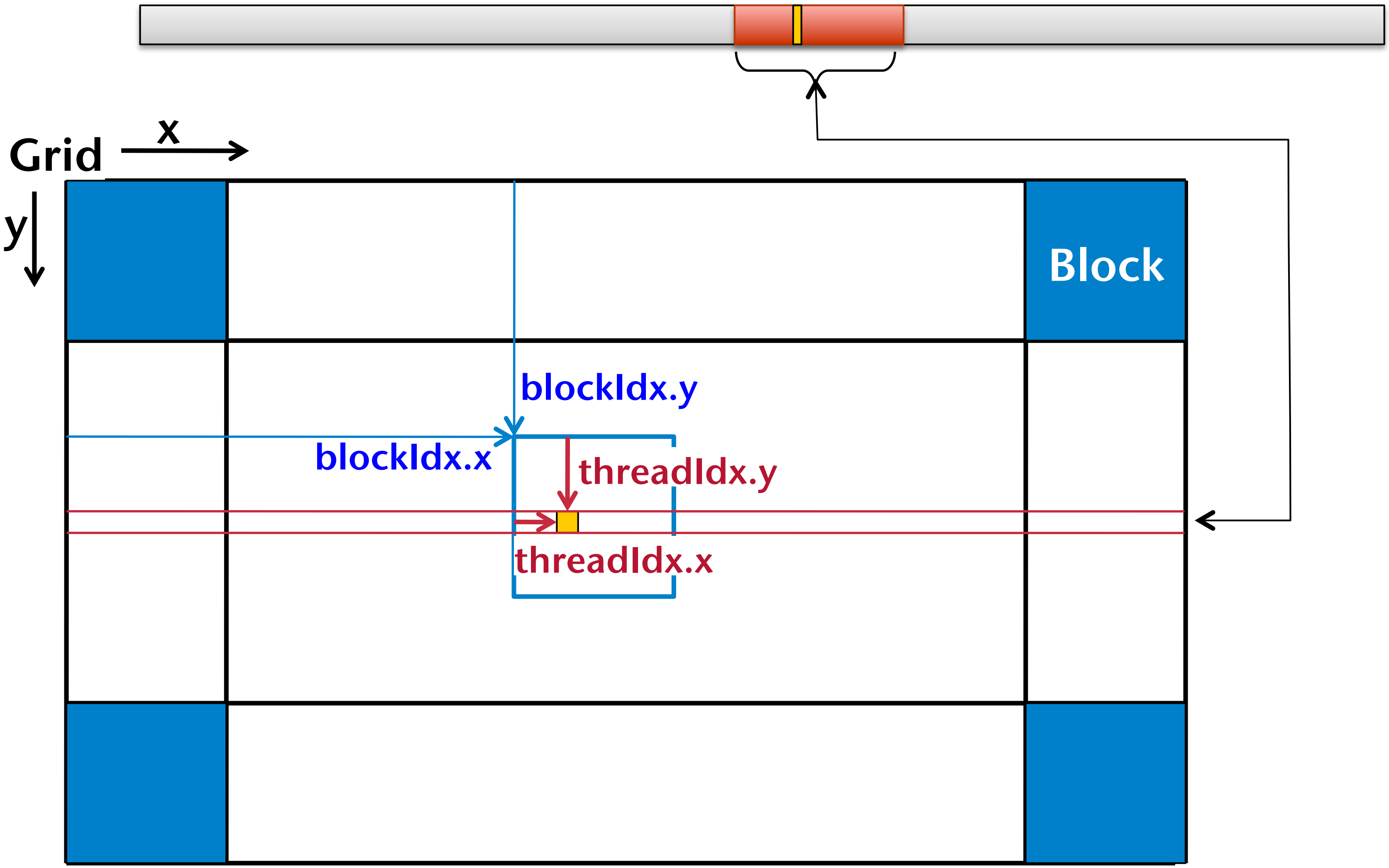
- Kernel launch:

```
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 this index computation:



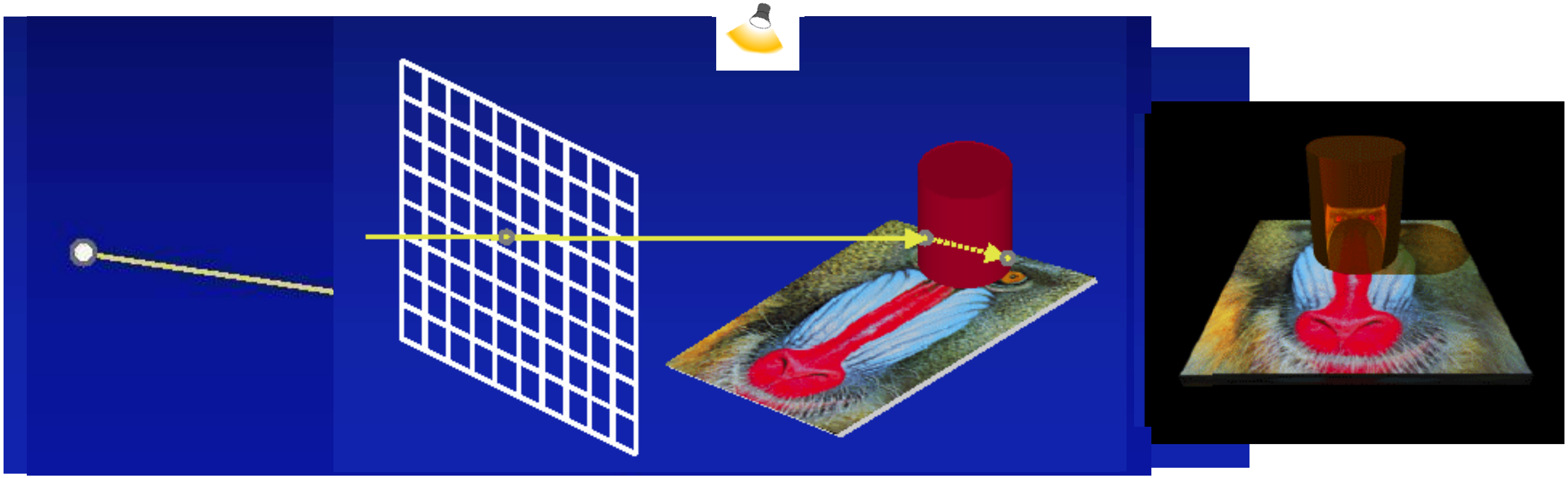
# Constant Memory

- 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

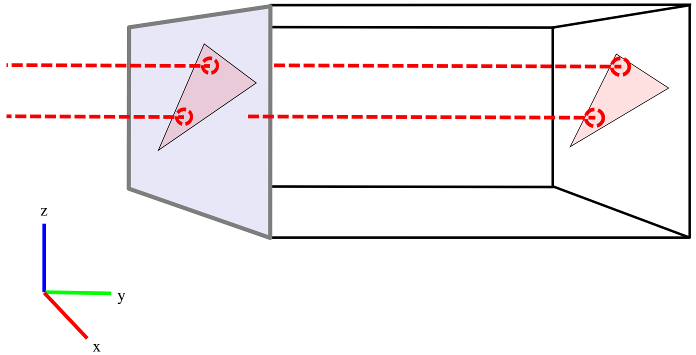
# Example: a Simple Raytracer

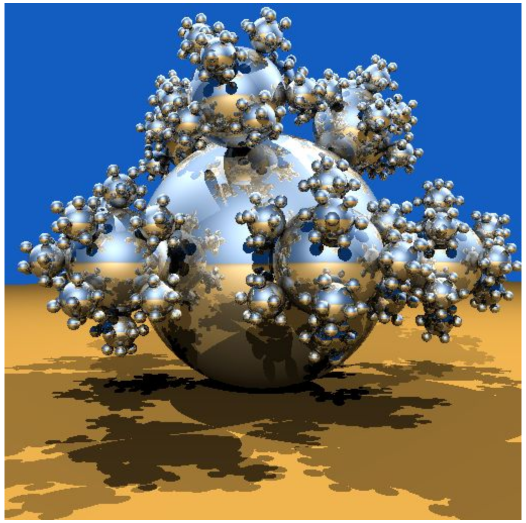
- The ray-tracing principle:

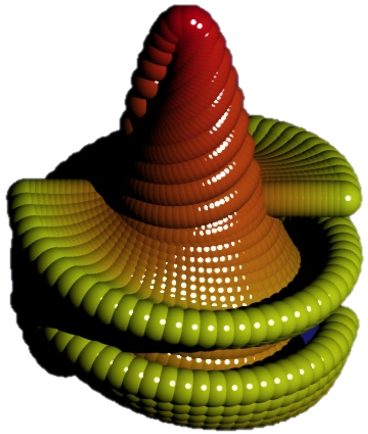
1. Shoot rays from camera through every pixel into scene (primary rays)
2. If the rays hits more than one object, then consider only the first hit
3. From there, shoot rays to all light sources (*shadow feelers*)
4. If a shadow feeler hits another obj → point is in shadow w.r.t. that light source. Otherwise,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

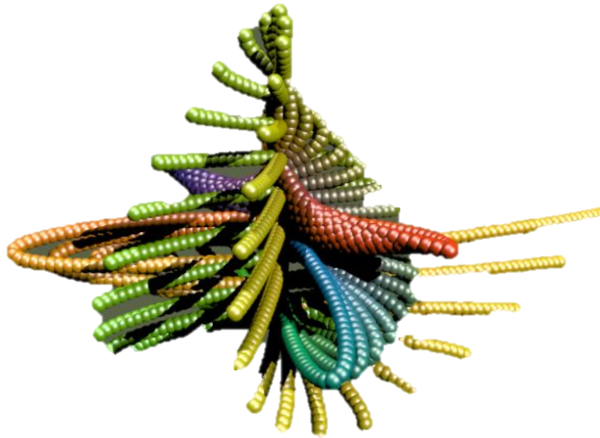


- Simplifications (for now):
  - Only primary rays
  - Camera is at infinity → primary rays are orthogonal to image plane
  - Only spheres
    - They are so easy, every raytracer has them ☺







- 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, clean up, and exit
};
```

# 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 = INF;
   int hit_sphere = MAX_INT;
   Hit hit;
   for ( int i = 0; i < n_spheres; i ++ ) {
      if ( intersect(ray, 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
}
```

# Declaration & Transfer

- Since it is constant memory, we declare it as such:

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

- Transfer now works by a different kind of Memcpy:

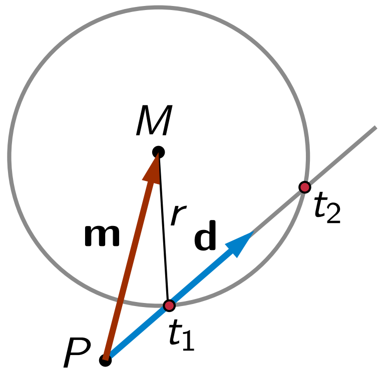
```
int main( void )
{
    ...
    // transfer spheres to device
    cudaMemcpyToSymbol( c_spheres, h_spheres,
                        n_spheres * sizeof(Sphere) );
    ...
};
```

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

```
__constant__ Sphere c_spheres[MAX_NUM_SPHERES];

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

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

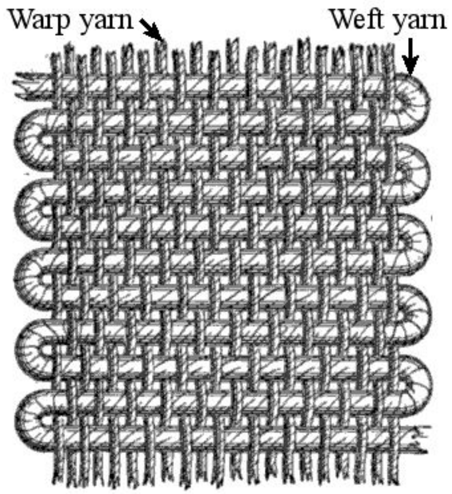


# Some Considerations on Constant Memory

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

# New Terminology

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

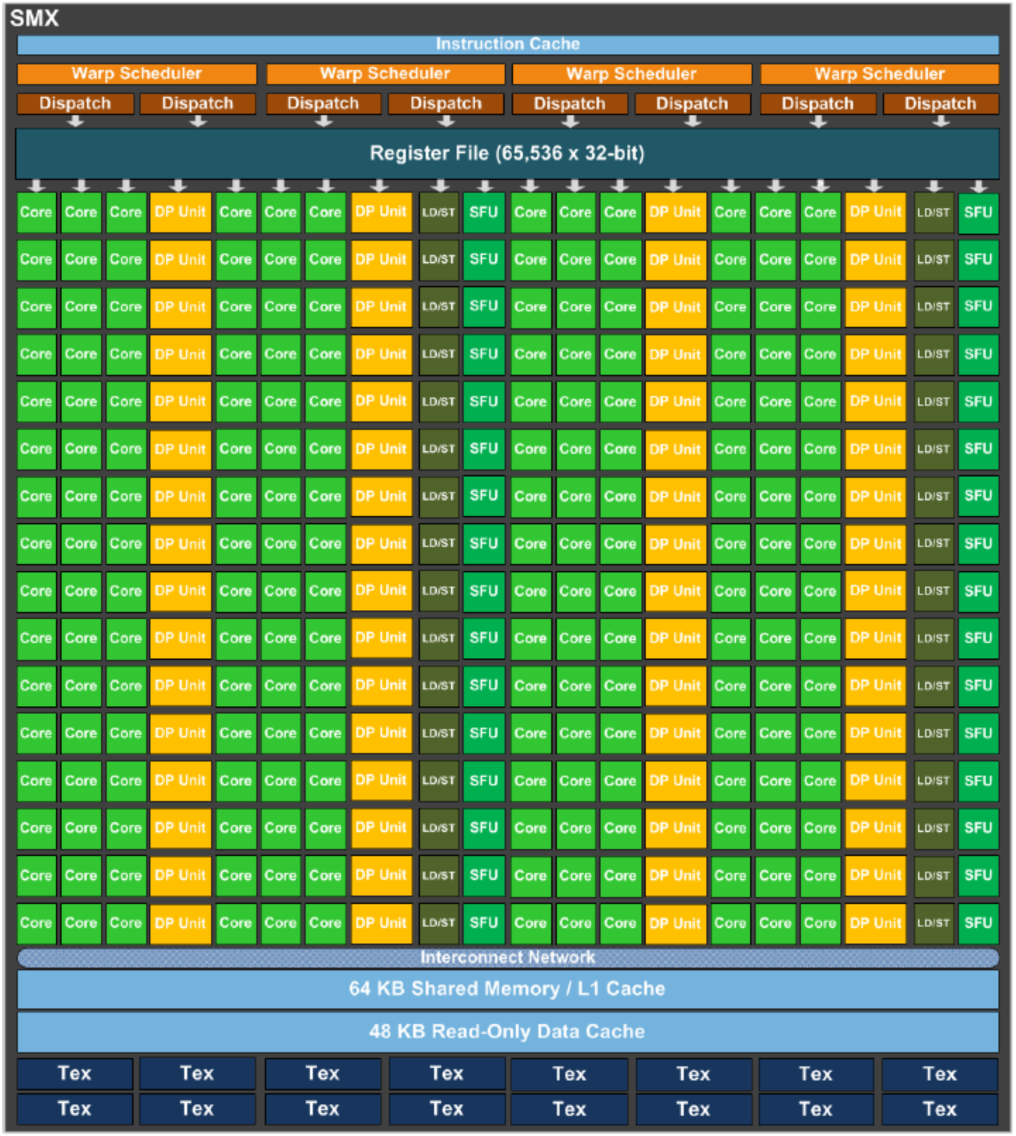



For more details: see "Performance with constant memory" on course web page

# Overview of a GPU's Architecture

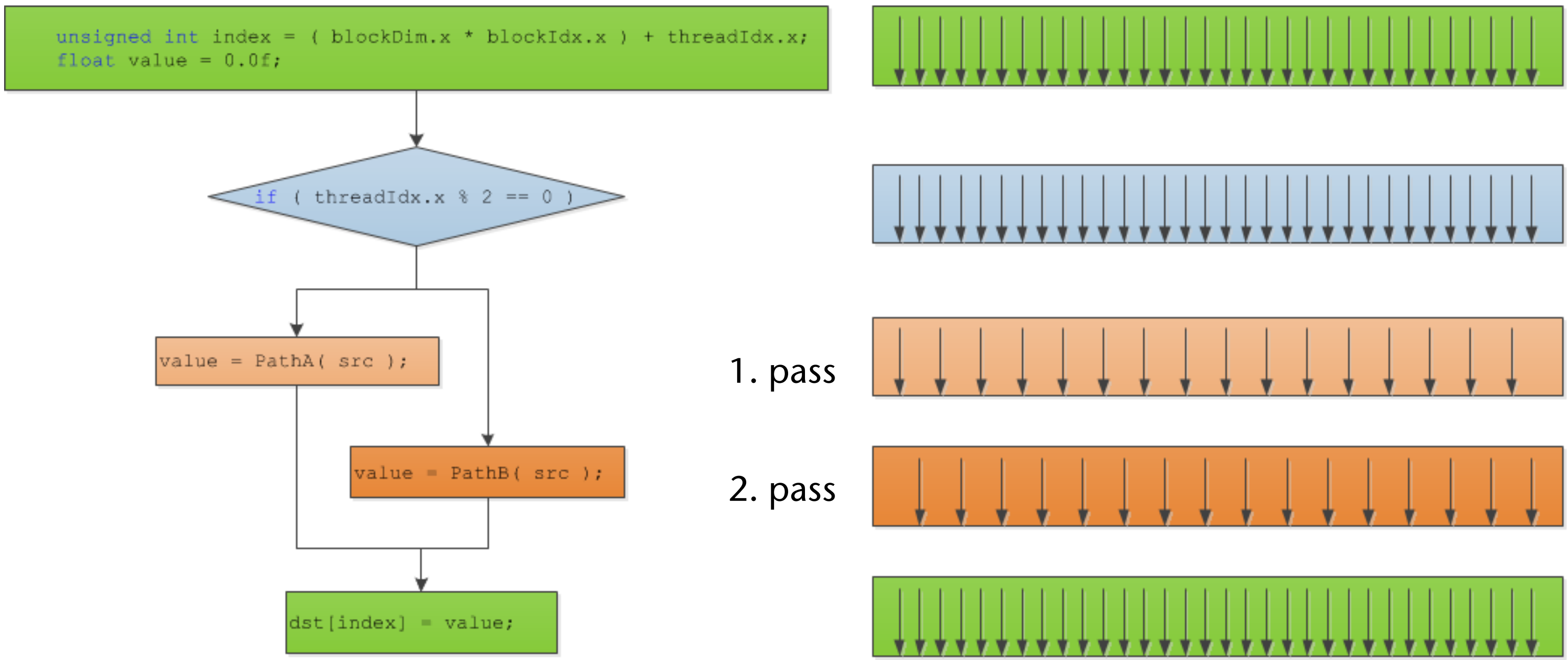

Nvidia's Kepler architecture as of 2012 (192 single-precision cores / 15 SM)

# One Streaming Multiprocessor

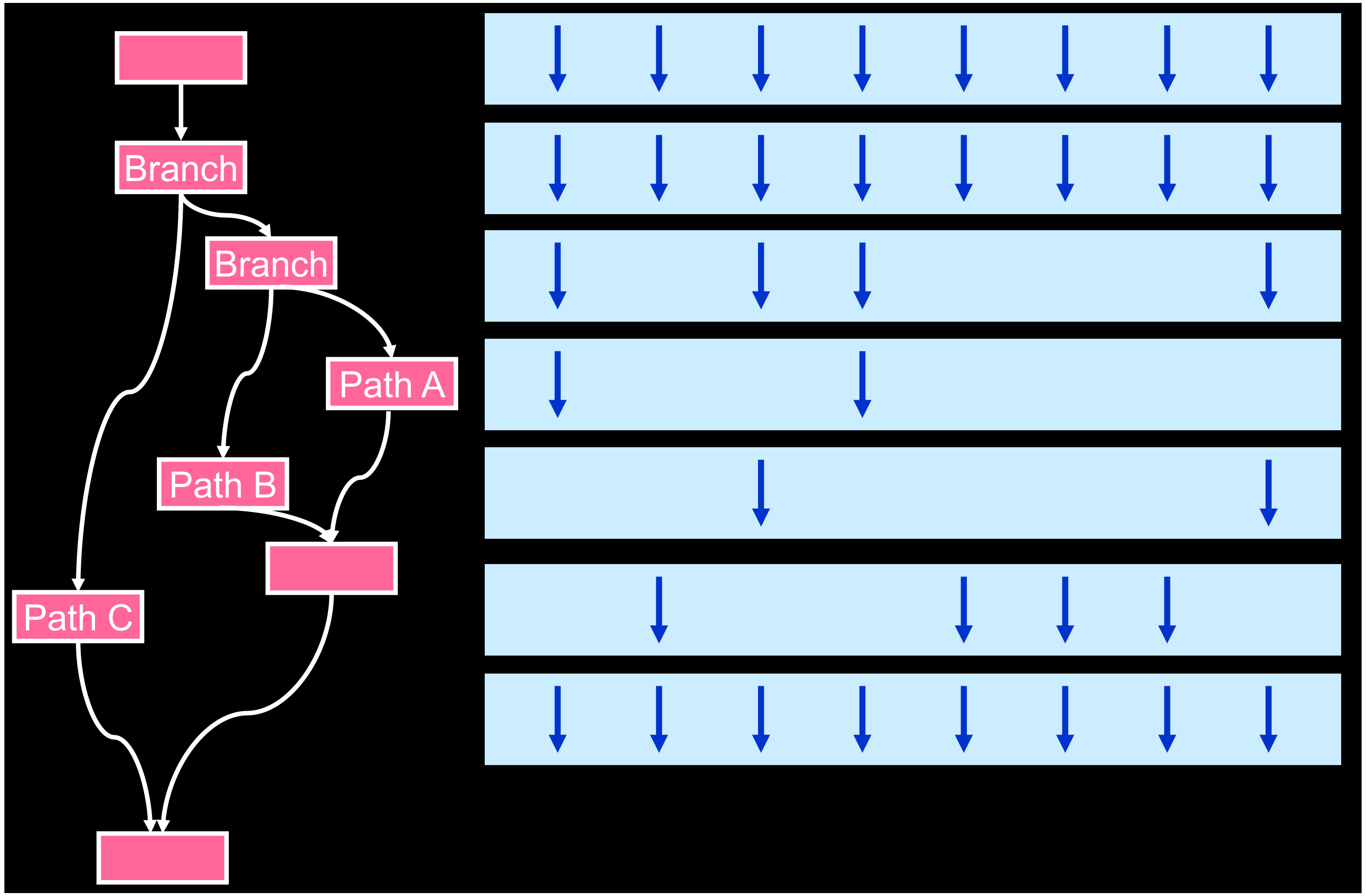


# Thread Divergence Revisited

- 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`** (all flow control statements)
- Example:



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



# Consequences for You as an Algorithm Designer / Programmer

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