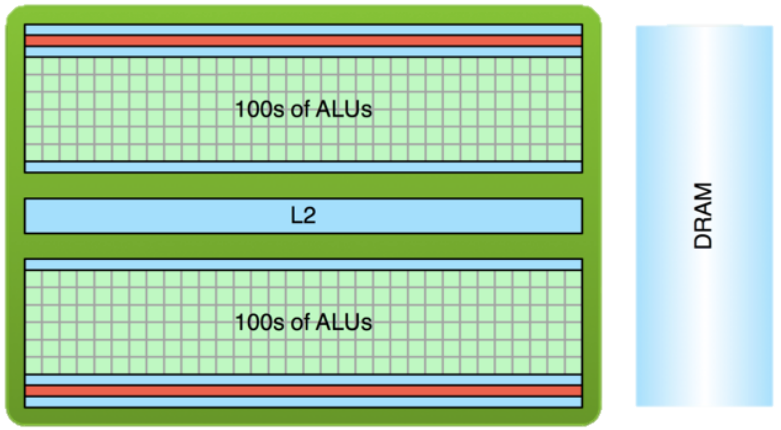
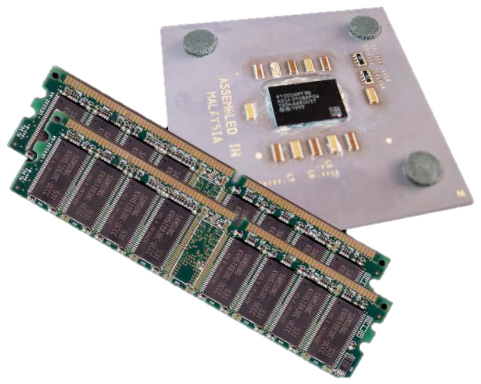
Massively Parallel Algorithms
Introduction to CUDA
and Many Fundamental Concepts
of Parallel Programming

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=sm_30 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( "hello world!\n" );
}

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

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

New Terminology, New Syntax

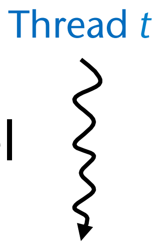
- Kernel := function/program code that is executed on the *device*
  - Syntax for definition by keyword **`__global__`** :

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

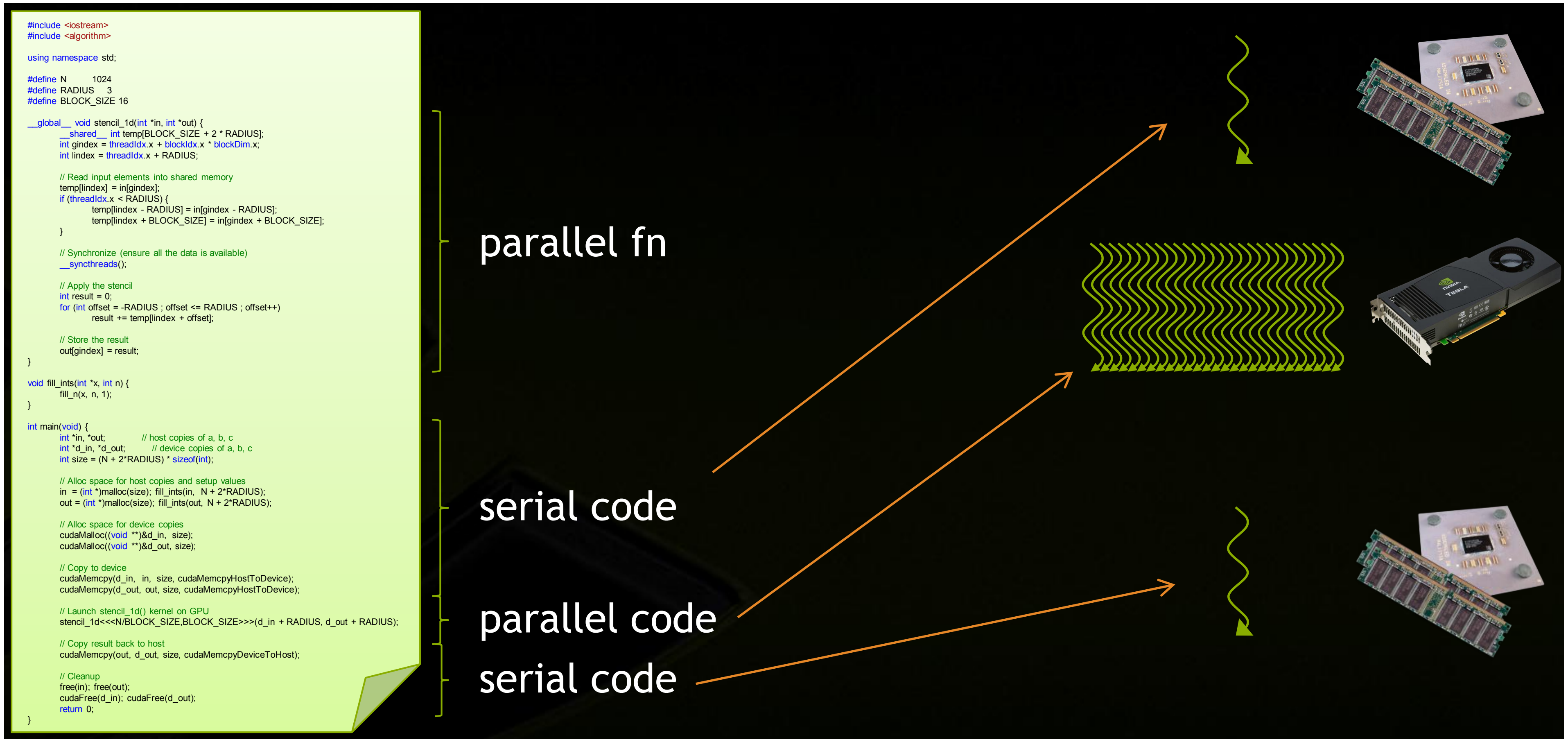
    - Note: kernels cannot return a value! → void
    - Kernels can take arguments (using regular C syntax)
  - Syntax for calling kernels:

```
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)

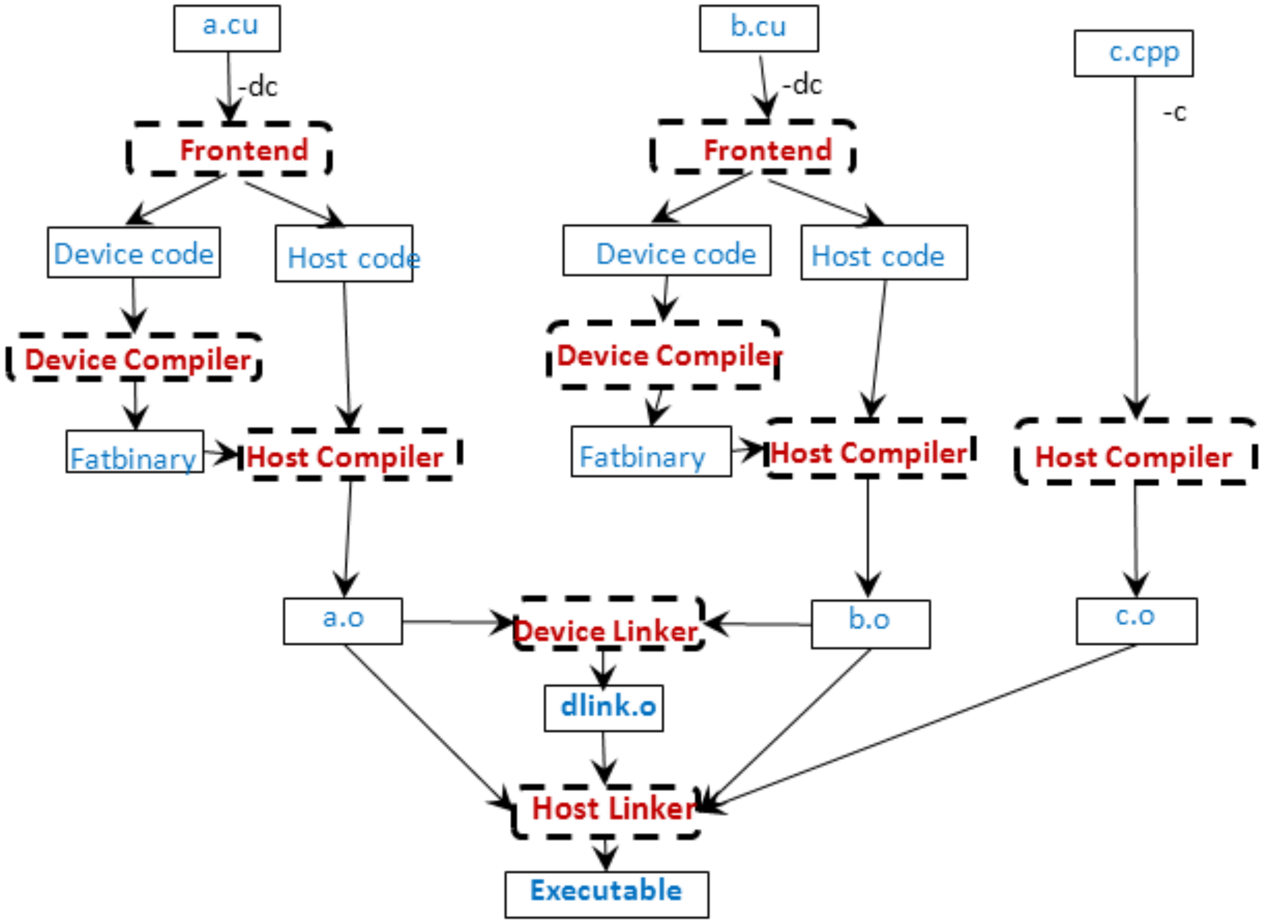


Typical Control Flow in Heterogeneous Computing



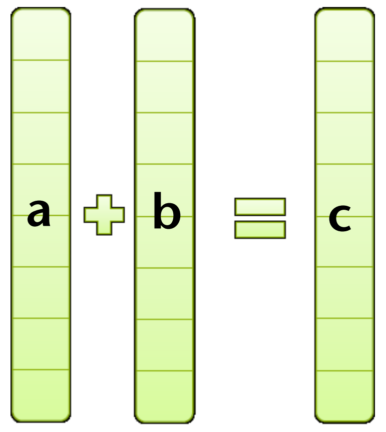
```
#include <iostream>
#include <algorithm>

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;
}
```

- The compilation process:

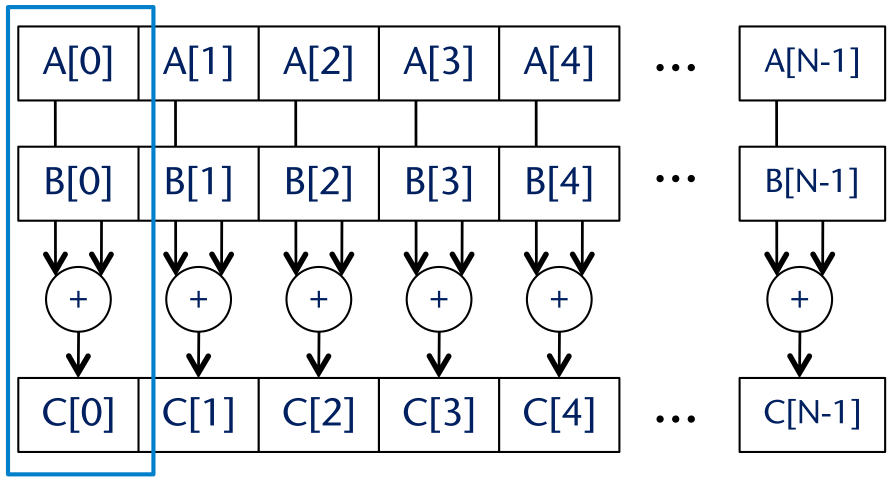


Transferring Data between GPU and CPU

- 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 *one thread per element* in the vector

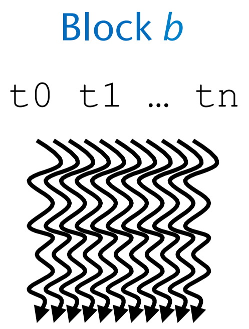
```
__global__
void addVectors( const float *a, const float *b,
                 float *c, unsigned int n       )
{
    unsigned int i = threadIdx.x;
    if ( i < n )
        c[i] = a[i] + b[i];
}
```

- **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*

New Concept: Blocks of Threads

- 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
- Important: within a block, the execution traces should not diverge too much, i.e., all of them should take the same branches, do the same number of loop iterations, as much as possible!
  - If they do diverge, this is called thread divergence → severe performance penalty!

Block *b*
t0 t1 … tn

On Memory Management on the GPU

- The API function:

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

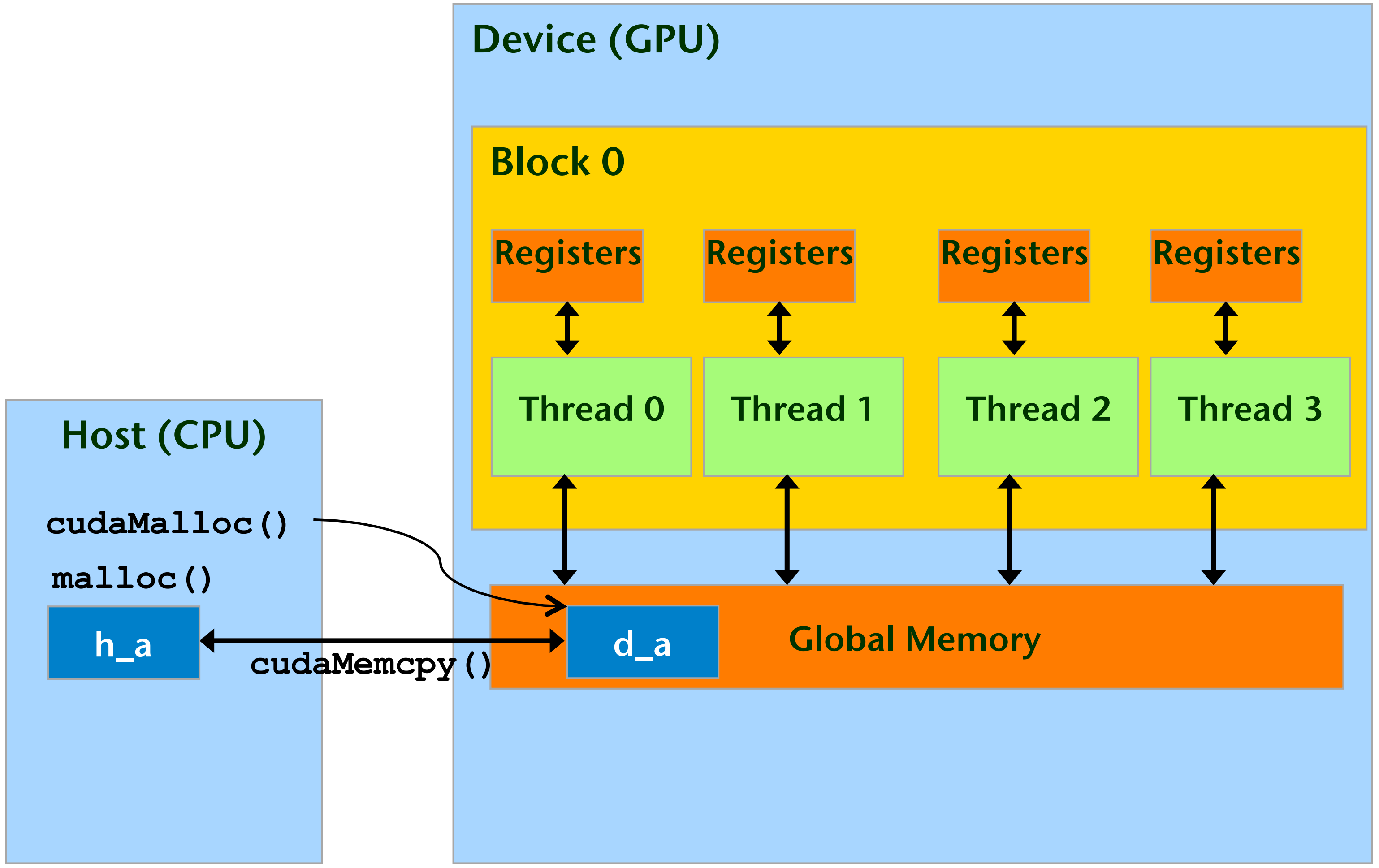
  - Mnemonic: like **`memcpy()`** from Unix/Linux

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

Terminology

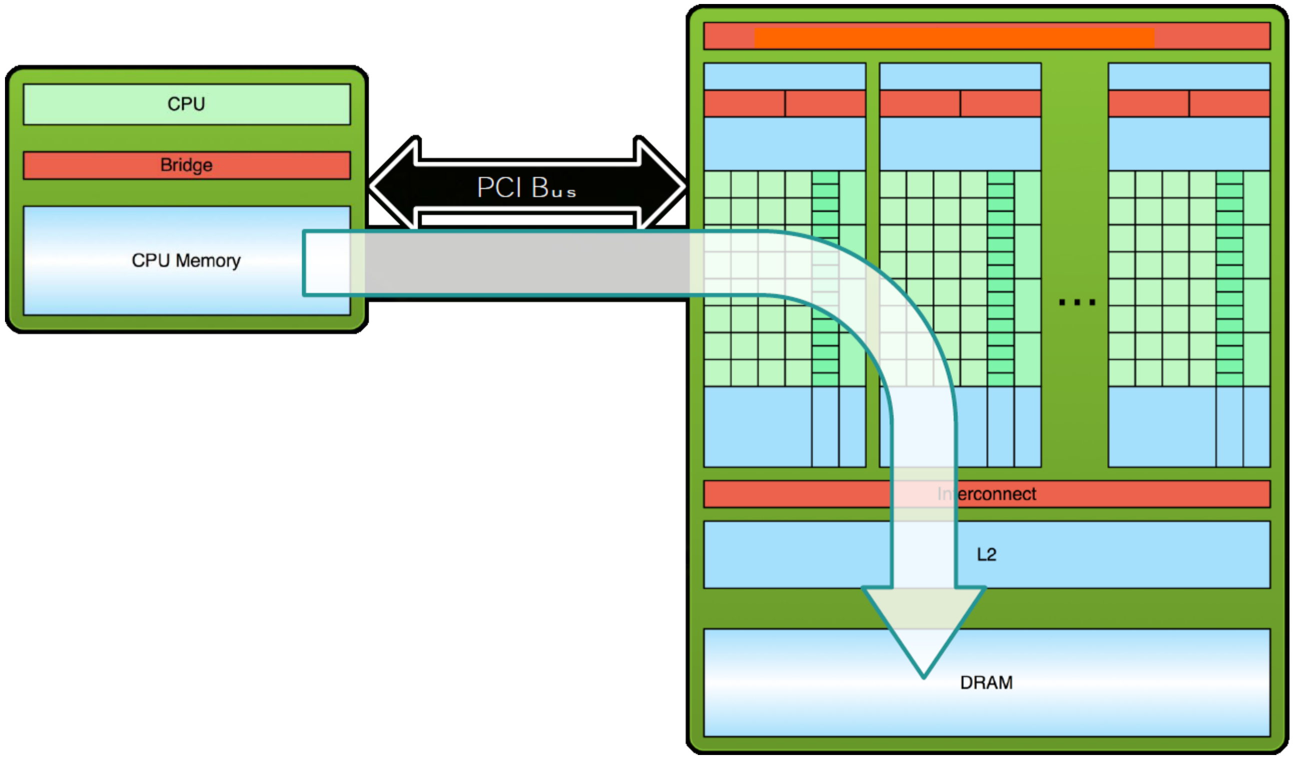
- This memory is called global memory



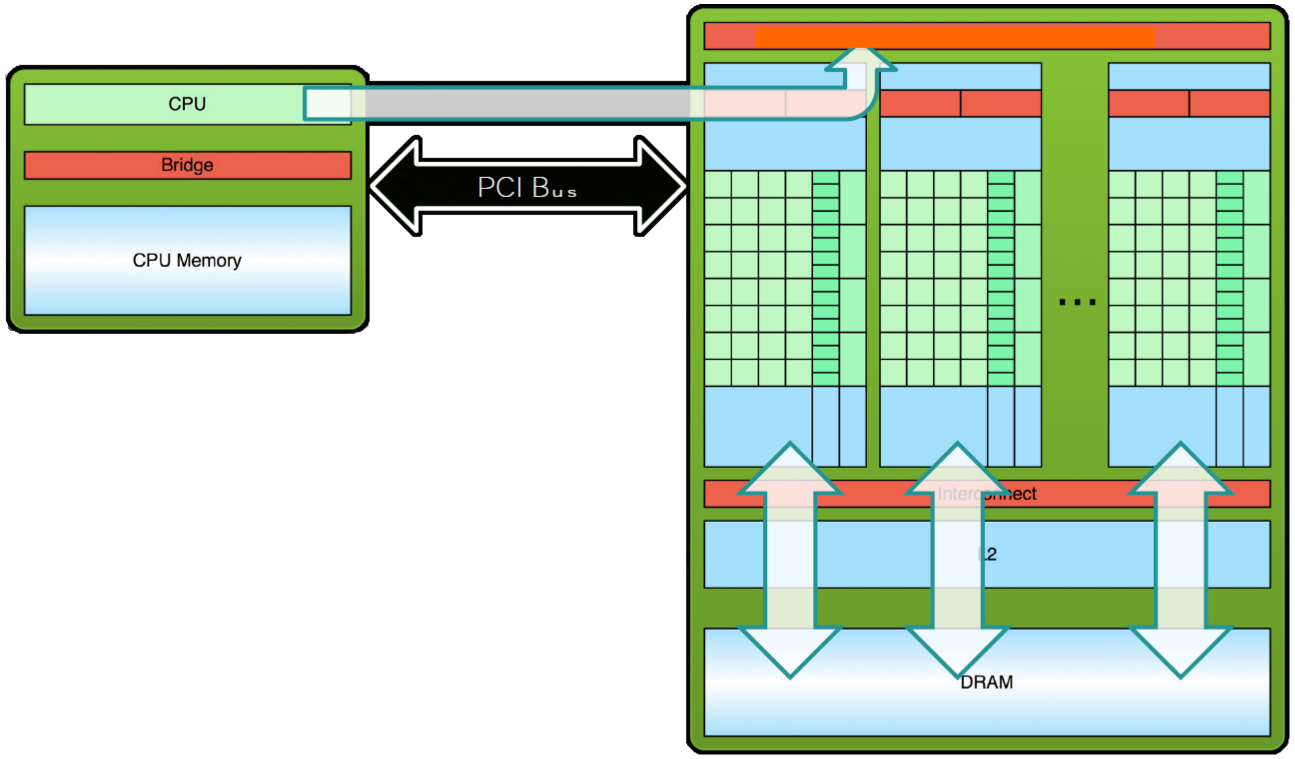
- The API is extremely simple:
  - **`cudaMalloc()`**, **`cudaFree()`**, **`cudaMemcpy()`**
  - Modeled after **`malloc()`**, **`free()`**, **`memcpy()`** from Unix/Linux
- Note: there are two different kinds of pointers!
  - Host memory pointers (obtained from **`malloc()`**)
  - Device memory pointers (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

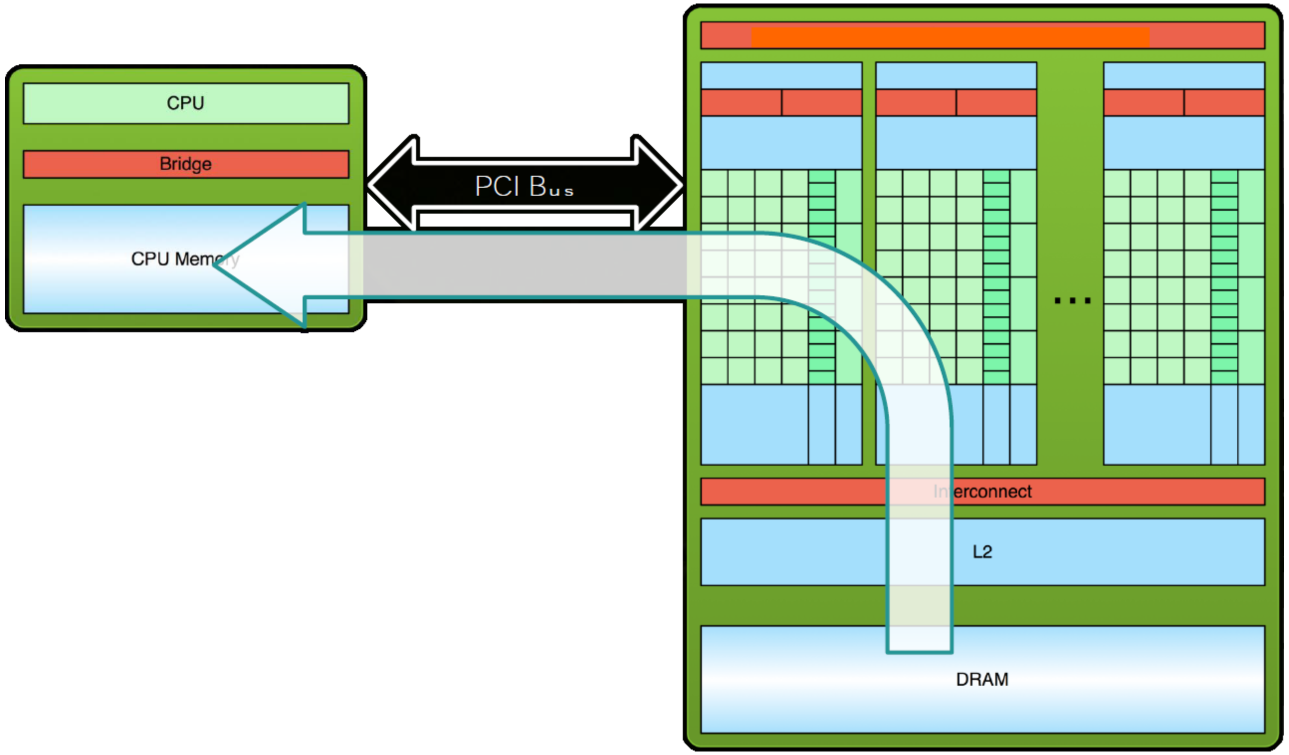
1. Copy input data from CPU memory to GPU memory



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



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



Blocks and Grids

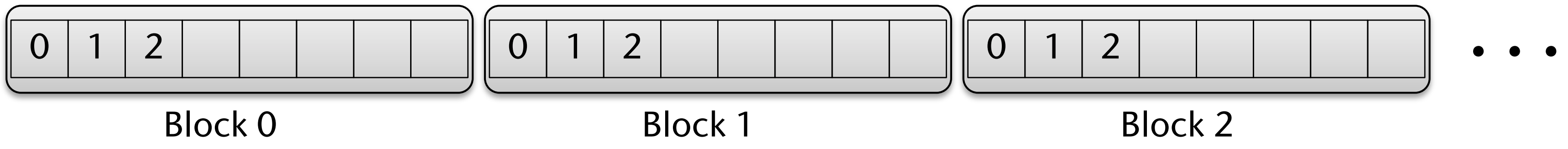
- What if we want to handle vectors larger than **`maxThreadsPerBlock`** ?
- We launch several blocks of our kernel!

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

$\Downarrow$

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

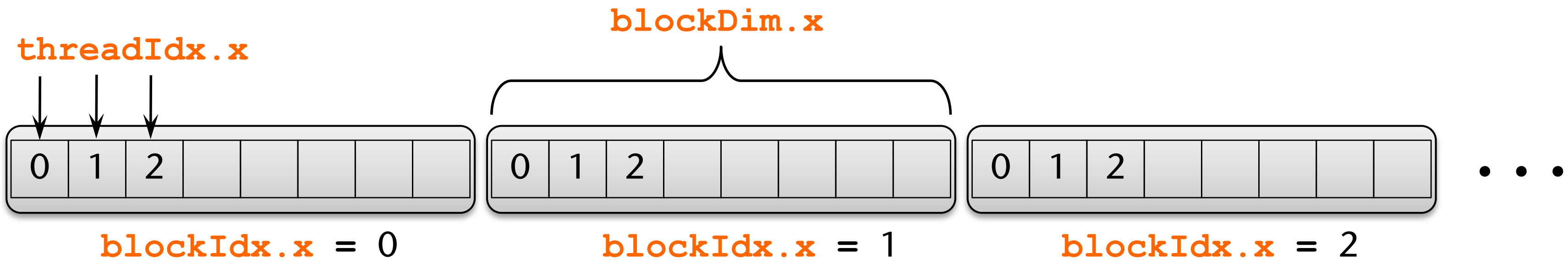
- This gives the following threads layout:



- 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 Should We Launch?

- Number of threads per block should be multiple of 32
- Number of threads must be a multiple of 'number of threads per block'
- The C idiom to do this:

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

- 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 … (we'll get back to this)

Thread Layouts for 2D Computational Problems

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

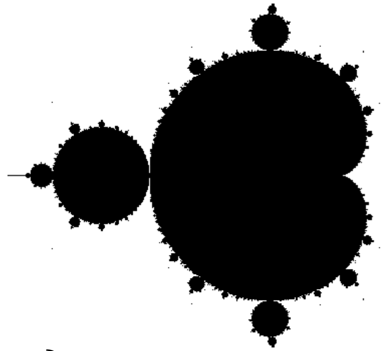
Example: Mandelbrot Set Computation

- 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

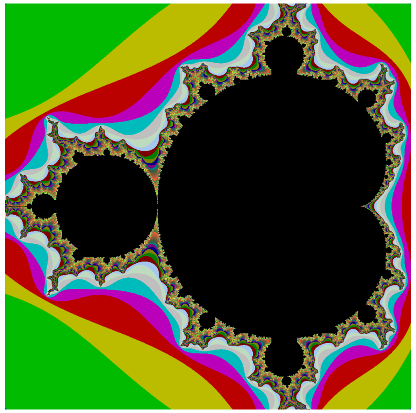
$$\mathbb{M} = \{ c \in \mathbb{C} \mid \text{sequence } (z_i) \text{ remains bounded } \}$$



- Theorem (w/o proof):

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

- Visualizing $\mathbb{M}$ nicely:
  - Color pixel $c = (x,y)$ black, if $|z|$ remains <2 after "many" iterations
  - Color c depending on the number of iterations necessary to make $|z_t| > 2$



- A few interesting facts about $\mathbb{M}$
  (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")
  - There is an infinite number of "mini Mandelbrot sets", i.e., smaller copies of $\mathbb{M}$(self similarity)

Computing the Mandelbrot Set on the GPU

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

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

```
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

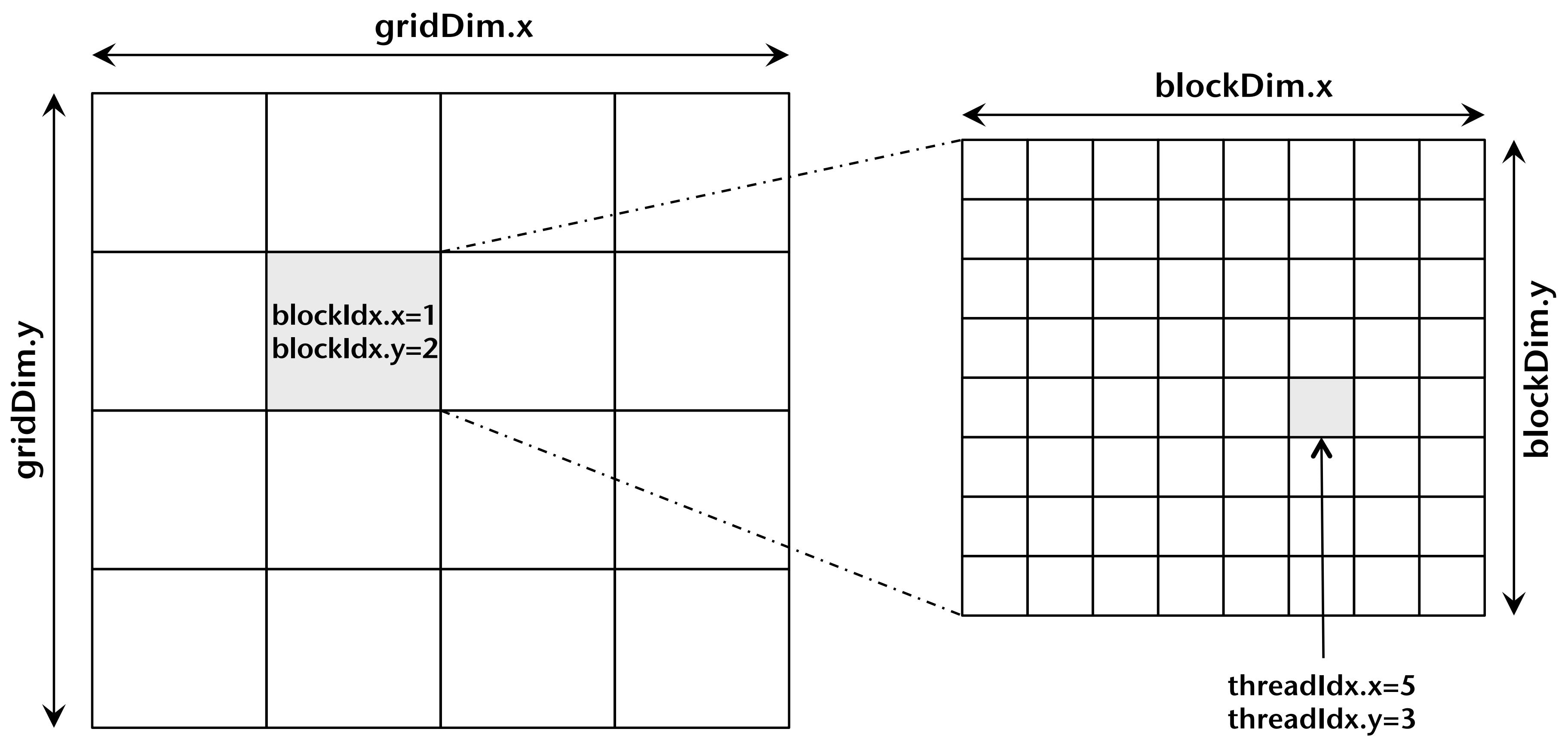
- Launch kernel:

```
mandelImage<<< blocks,threads >>>( d_bitmap, img_size );
```

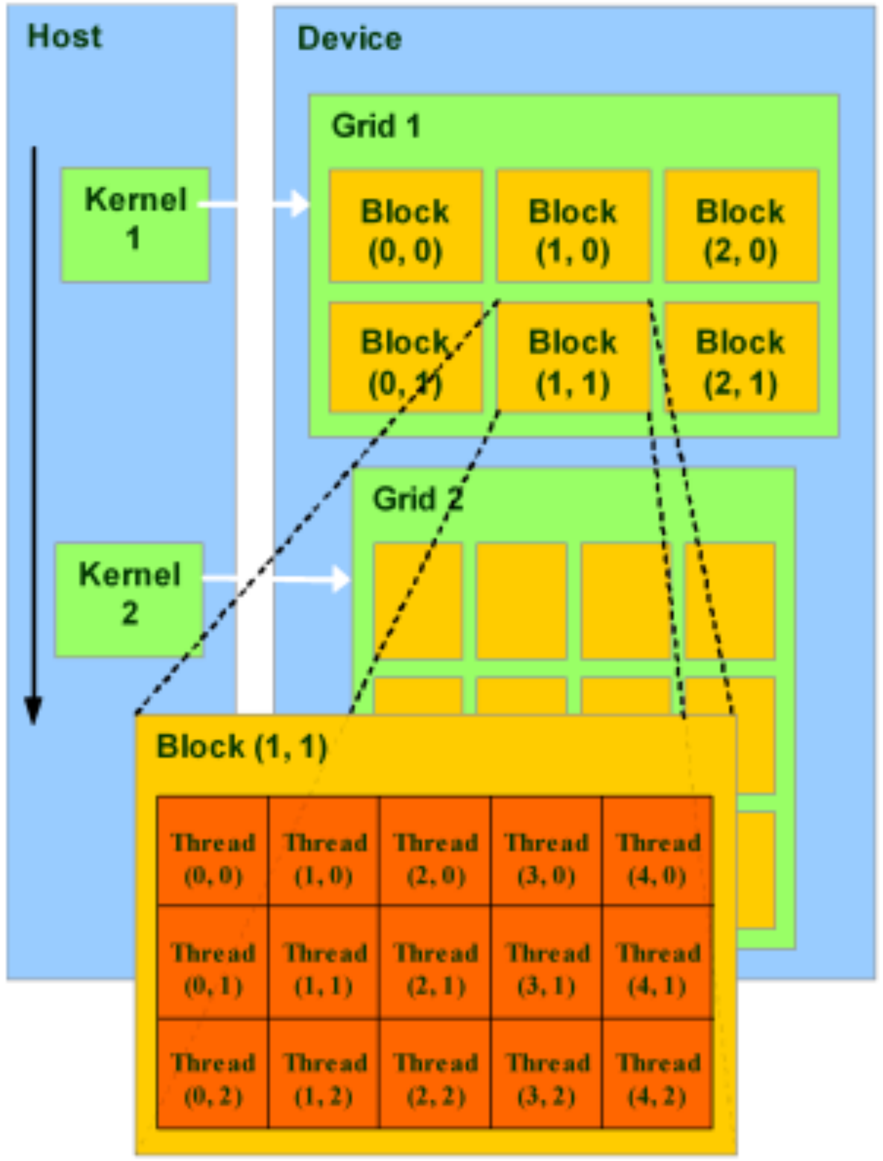
- Implementation of the kernel (simplified):

```
__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 );

  bitmap[offset].x = 255 * isOutsideM;     // red = outside
  bitmap[offset].y = bitmap[offset].z = 0;
  bitmap[offset].w = 255;
}
```

- Visualization of our block/grid layout:



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



A Word About **`dim3`**

- Definition (done by CUDA):

```
struct dim3   // is actually a C++ class
{
   unsigned int x, y, z;
};
```

- Usage:

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

  is equivalent :

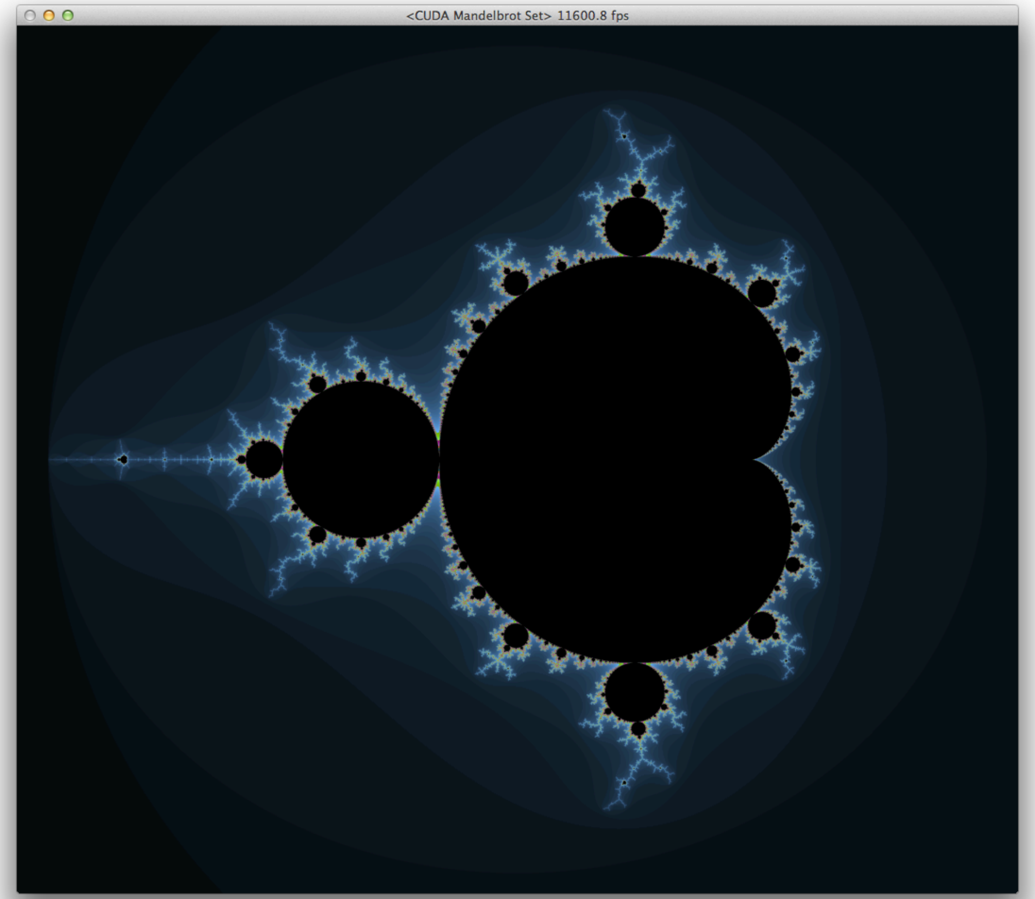
```
dim3 threads(M,1);
dim3 blocks(N,1);
kernel<<<blocks,threads>>>(...);
```

Implementation of the Kernel

```
__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;
    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 (on Macbook)



Keys: e = Reset mouse+drag = pan shift+drag up/down = zoom c = change colors