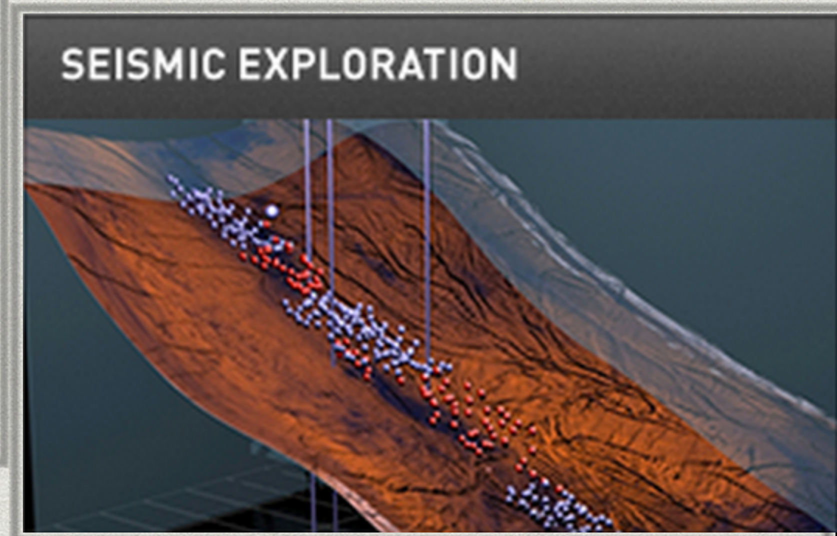
HIGH PERFORMANCE COMPUTING WITH CUDA

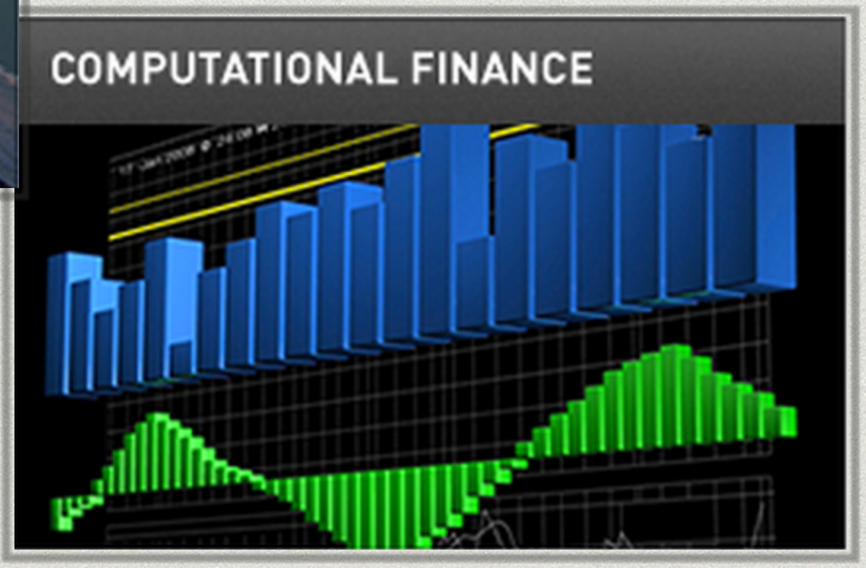
2018/2019

LECTURER **ANDRÉ PEREIRA**

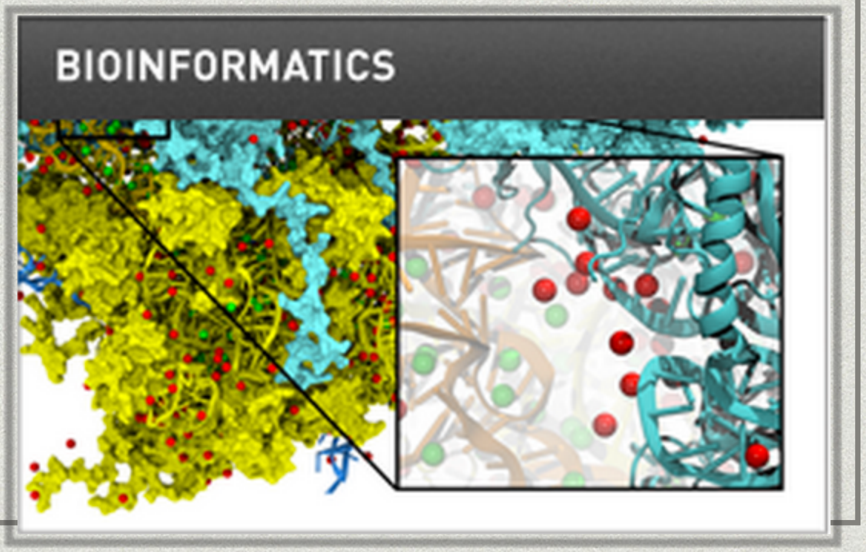
CONTACT **ampereira@di.uminho.pt**

Motivation





1000's x speedup



Pitfalls

Pitfalls

- General misconception

  Scientist: “MORE POWERRRR”

  NVidia: “Hey, here’s the new 2000€ Tesla”

Pitfalls

- General misconception

  Scientist: “MORE POWERRRR”

  NVidia: “Hey, here’s the new 2000€ Tesla”

- However, scientists may run into two problems
  - The code is not faster
  - The code is not correct

Concepts

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared Memory
- __syncthreads()
- Asynchronous Operation
- Handling Errors
- Efficiently Managing Memory/Dynamic Parallelism
- Unified Memory
- Profiling

Kepler K20

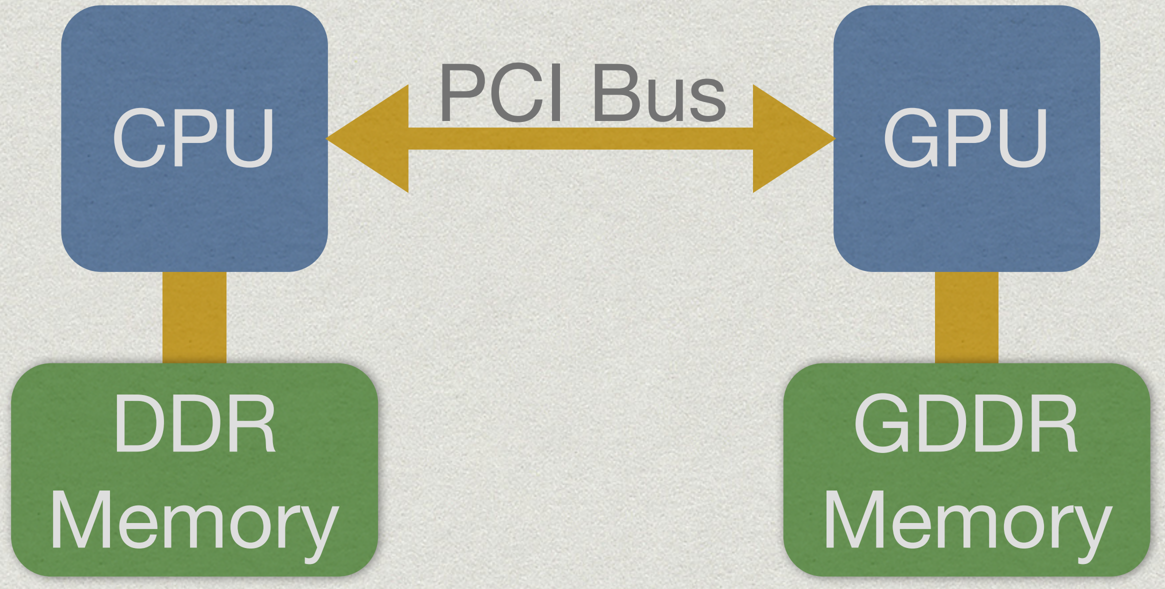
- SP/DP peak performance: 3.52/1.17 TFLOPS
- 5GB GDDR RAM @208 Gbytes/sec
  - 64K 32-bit registers (max 255 per thread)
  - 64KB L1/shared memory and 48KB read only cache
  - 1536 KB shared L2 cache

Heterogeneous Computing

* Host: CPU and its memory (host memory)
* Device: GPU and its memory (device memory)

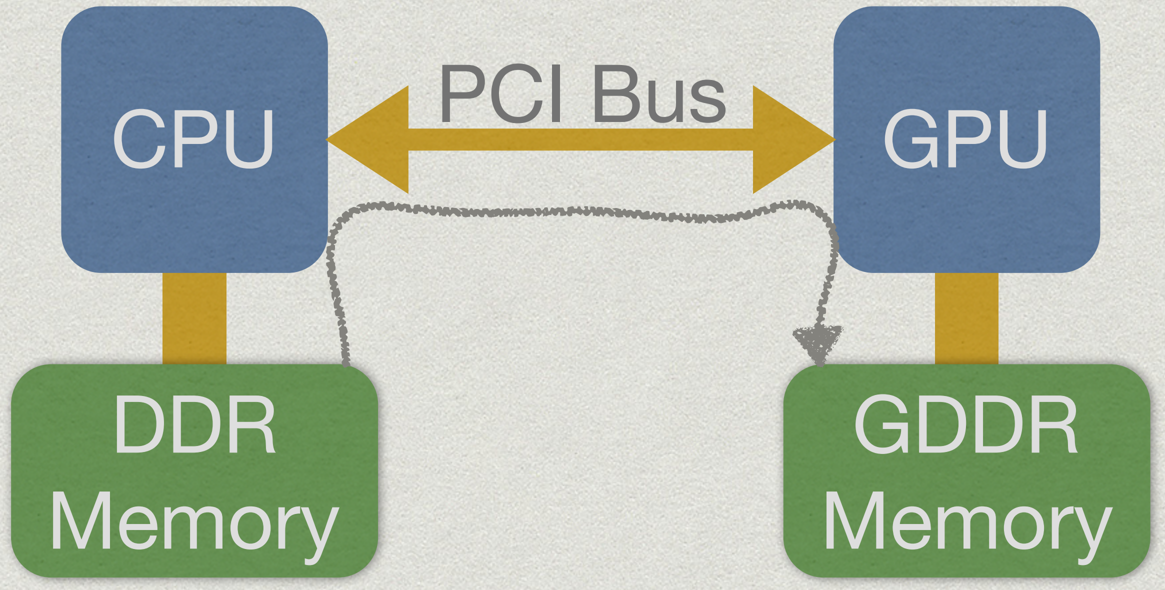
Heterogeneous Computing

- A very simple execution flow



Heterogeneous Computing

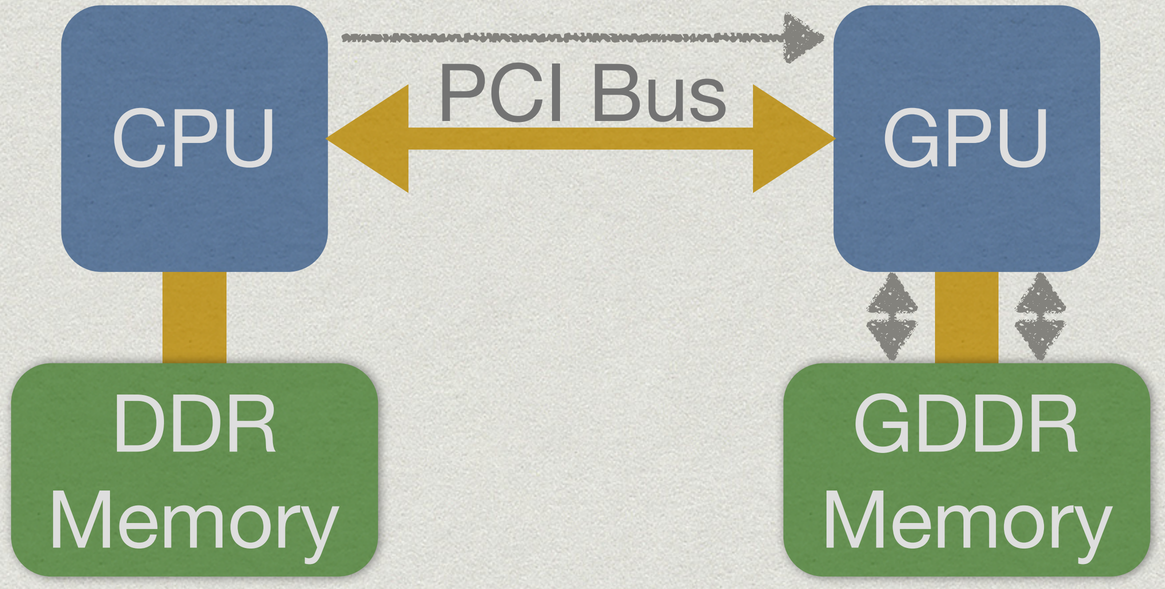
- A very simple execution flow



- Copy input data from CPU to GPU memory

Heterogeneous Computing

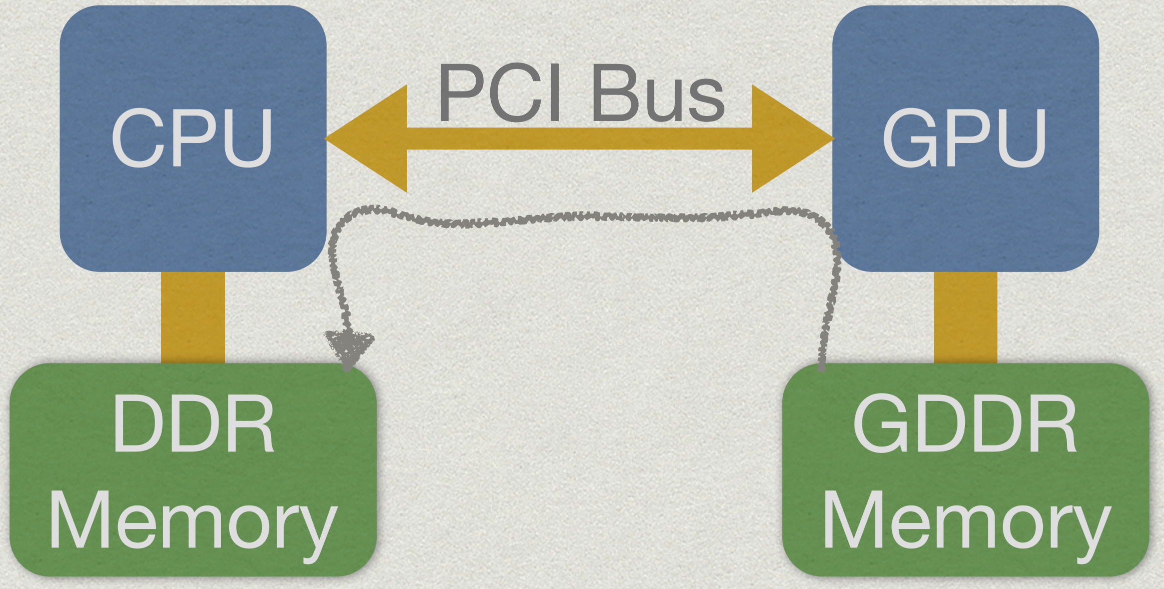
- A very simple execution flow



- Copy input data from CPU to GPU memory
- Load GPU code and execute it

Heterogeneous Computing

- A very simple execution flow



- Copy input data from CPU to GPU memory
- Load GPU code and execute it
- Copy the results from GPU to the CPU memory

Ex.: Addition on the Device

```
__global__ void mykernel (void) {
}

int main (void) {
    mykernel <<< 1, 1 >>> ();
    return 0;
}
```

Ex.: Addition on the Device

- New keywords
  - __global__ runs on the device and is called from the host
  - __host__ runs on the host
  - __device__ runs on the device (inlined inside a __global__ kernel)

```
__global__ void mykernel (void) {
}

int main (void) {
    mykernel <<< 1, 1 >>> ();
    return 0;
}
```

Ex.: Addition on the Device

- New keywords
  - __global__ runs on the device and is called from the host
  - __host__ runs on the host
  - __device__ runs on the device (inlined inside a __global__ kernel)
- Kernel launch
  - Number of threads (to see later)
  - Input parameters

```
__global__ void mykernel (void) {
}

int main (void) {
    mykernel <<< 1, 1 >>> ();
    return 0;
}
```

Ex.: Addition on the Device

- New keywords
  - __global__ runs on the device and is called from the host
  - __host__ runs on the host
  - __device__ runs on the device (inlined inside a __global__ kernel)
- Kernel launch
  - Number of threads (to see later)
  - Input parameters
- Compile everything with nvcc
  - *nvcc* compiles the device code
  - *nvcc* calls *gcc/icc* for the rest

```
__global__ void mykernel (void) {
}

int main (void) {
    mykernel <<< 1, 1 >>> ();
    return 0;
}
```

Extending the Kernel

```
__global__ void mykernel (int *a, int *b, int *c) {
    *c = *a + *b;
}

int main (void) {
    mykernel <<< 1, 1 >>> ();
    return 0;
}
```

Extending the Kernel

- It now adds two integers
  - Copy the data into / out of, the device
  - Device and host pointers address different memory spaces (as of CUDA 6.0)

```
__global__ void mykernel (int *a, int *b, int *c) {
    *c = *a + *b;
}

int main (void) {
    mykernel <<< 1, 1 >>> ();
    return 0;
}
```

Extending the Kernel

- It now adds two integers
  - Copy the data into / out of, the device
  - Device and host pointers address different memory spaces (as of CUDA 6.0)
- We must transfer the data!
  - cudaMalloc
  - cudaFree
  - cudaMemcpy

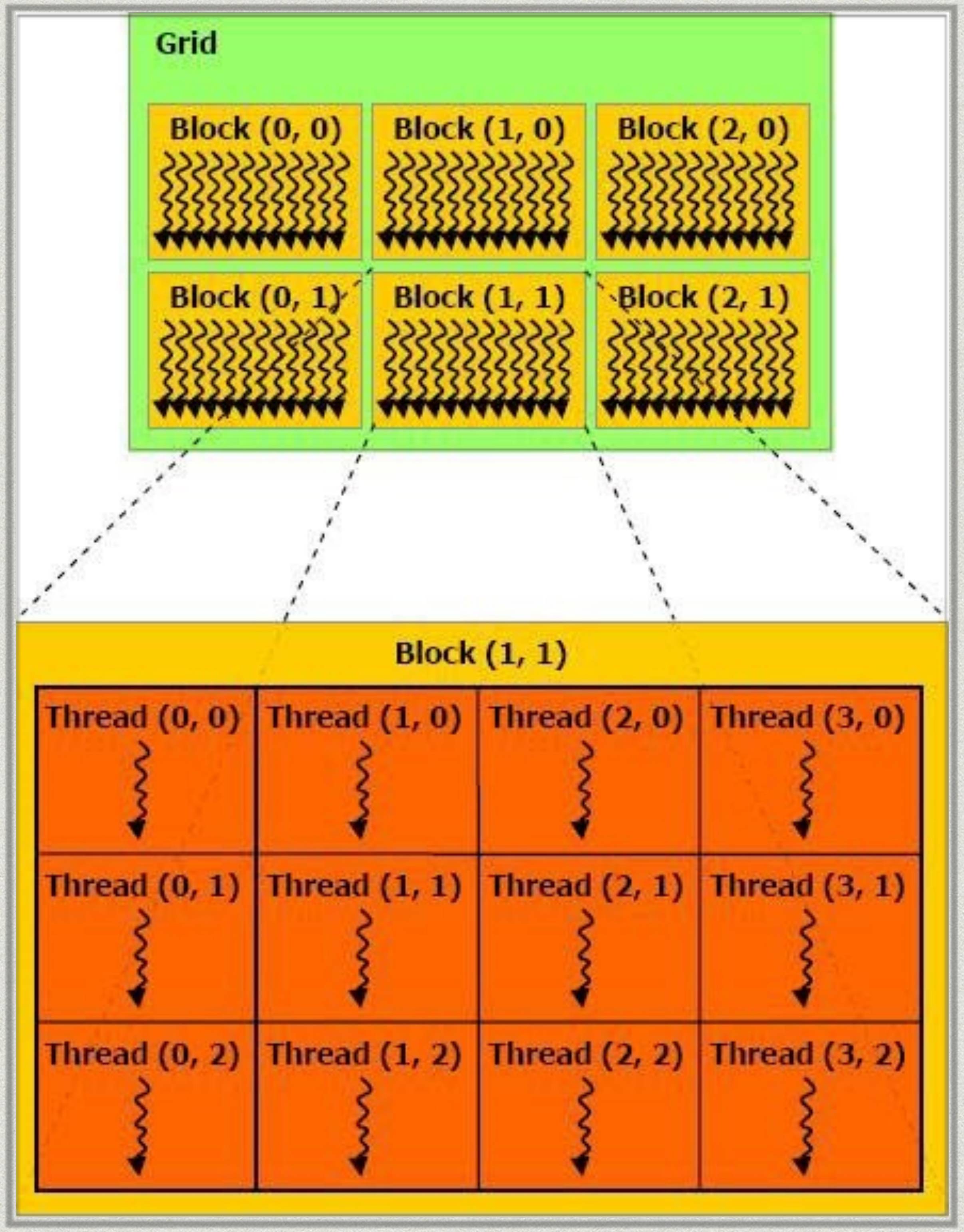
```
__global__ void mykernel (int *a, int *b, int *c) {
    *c = *a + *b;
}

int main (void) {
    mykernel <<< 1, 1 >>> ();
    return 0;
}
```

Adapting main()

```
int main (void) {
    int a, b, c;
    int *dev_a, *dev_b, *dev_c;

    cudaMalloc(void **)&dev_a, sizeof(int));
    cudaMalloc(void **)&dev_b, sizeof(int));
    cudaMalloc(void **)&dev_c, sizeof(int));

    a = 2;
    b = 4;

    cudaMemcpy(dev_a, &a, sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(dev_b, &b, sizeof(int), cudaMemcpyHostToDevice);

    mykernel <<< 1, 1 >>> (dev_a, dev_b, dev_c);

    cudaThreadSynchronize();

    cudaMemcpy(&c, dev_c, sizeof(int), cudaMemcpyDeviceToHost);

    cudaFree(dev_a); cudaFree(dev_b); cudaFree(dev_c);

    return 0;
}
```

GOING PARALLEL

Thread Hierarchy



Thread Hierarchy

Thread Hierarchy

- Blocks must be independent
  - any possible interleaving of blocks should be valid

Thread Hierarchy

- Blocks must be independent
  - any possible interleaving of blocks should be valid
- Blocks may coordinate but never synchronise
  - shared queue pointer **OK**
  - shared lock **NOT OK**

Thread Hierarchy

- Blocks must be independent
  - any possible interleaving of blocks should be valid
- Blocks may coordinate but never synchronise
  - shared queue pointer **OK**
  - shared lock **NOT OK**
- This ensures some scalability

Thread Hierarchy

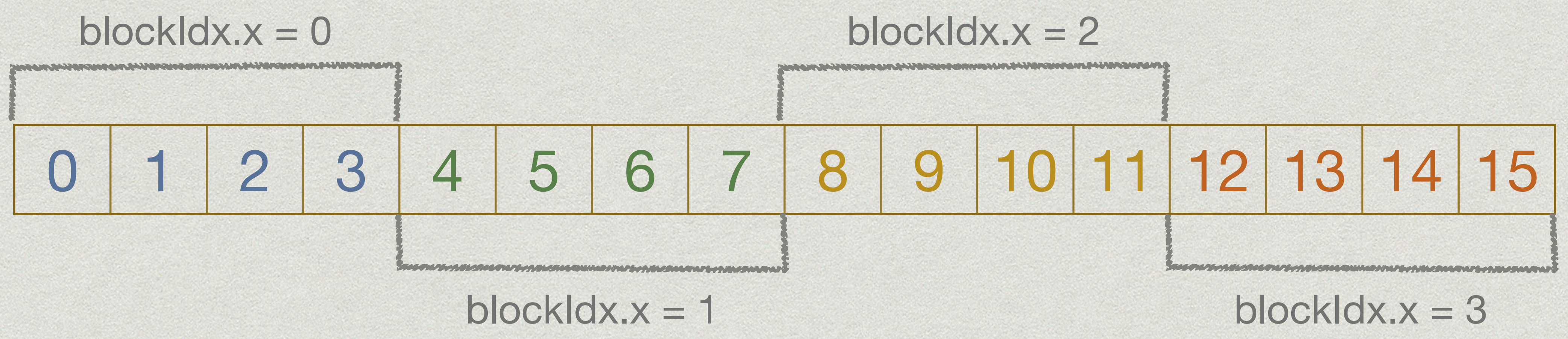
Thread Hierarchy

- Declare a specific type for the dimensions of the grid (in number of blocks) and blocks (in number of threads)
  - dim3 var (x, y, z)

Thread Hierarchy

- Declare a specific type for the dimensions of the grid (in number of blocks) and blocks (in number of threads)
  - dim3 var (x, y, z)
- Access the indexes and dimensions inside the kernel
  - gridDim.(x, y, z) and blockDim.(x, y, z)
  - threadIdx.(x, y, z) and blockIdx.(x, y, z)

Thread Hierarchy



An array position (one dimensional grid and block) is given by:

```
int index = threadIdx.x + blockIdx.x * blockDim.x;
```

Update the Kernel Call

Old:

```
mykernel <<< 1, 1 >>> (dev_a, dev_b, dev_c);
```

New:

```
dim3 dimGrid (NUM_BLOCKS);
dim3 dimBlock (THREADS_PER_BLOCK);
mykernel <<< dimBlock, dimGrid >>> (dev_a, dev_b, dev_c);
```

Compiling CUDA

Compiling CUDA

- nvcc <options> file.cu -o executable

Compiling CUDA

- nvcc <options> file.cu -o executable
- Some useful options
  - -g - compiles with debug symbols
  - -arch=sm_xx - compiles for a specific CUDA compatibility version
  - -ptx - generates the ptx instructions for the GPU
  - -Xptxas -v - displays extra information about the kernel (such as register spills, cache usage, etc)