Parallel I/O

Ivan Girotto – igirotto@ictp.it

Information & Communication Technology Section (ICTS)
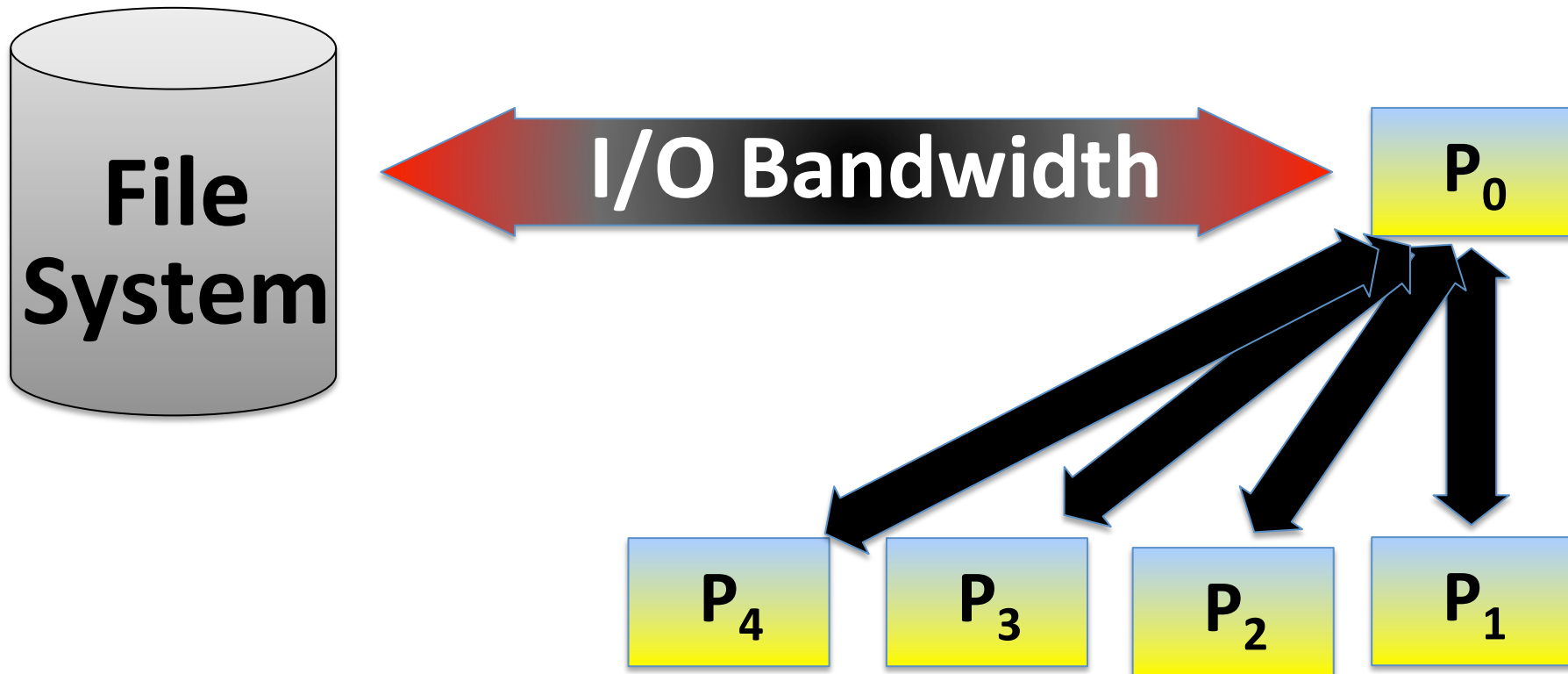
International Centre for Theoretical Physics (ICTP)
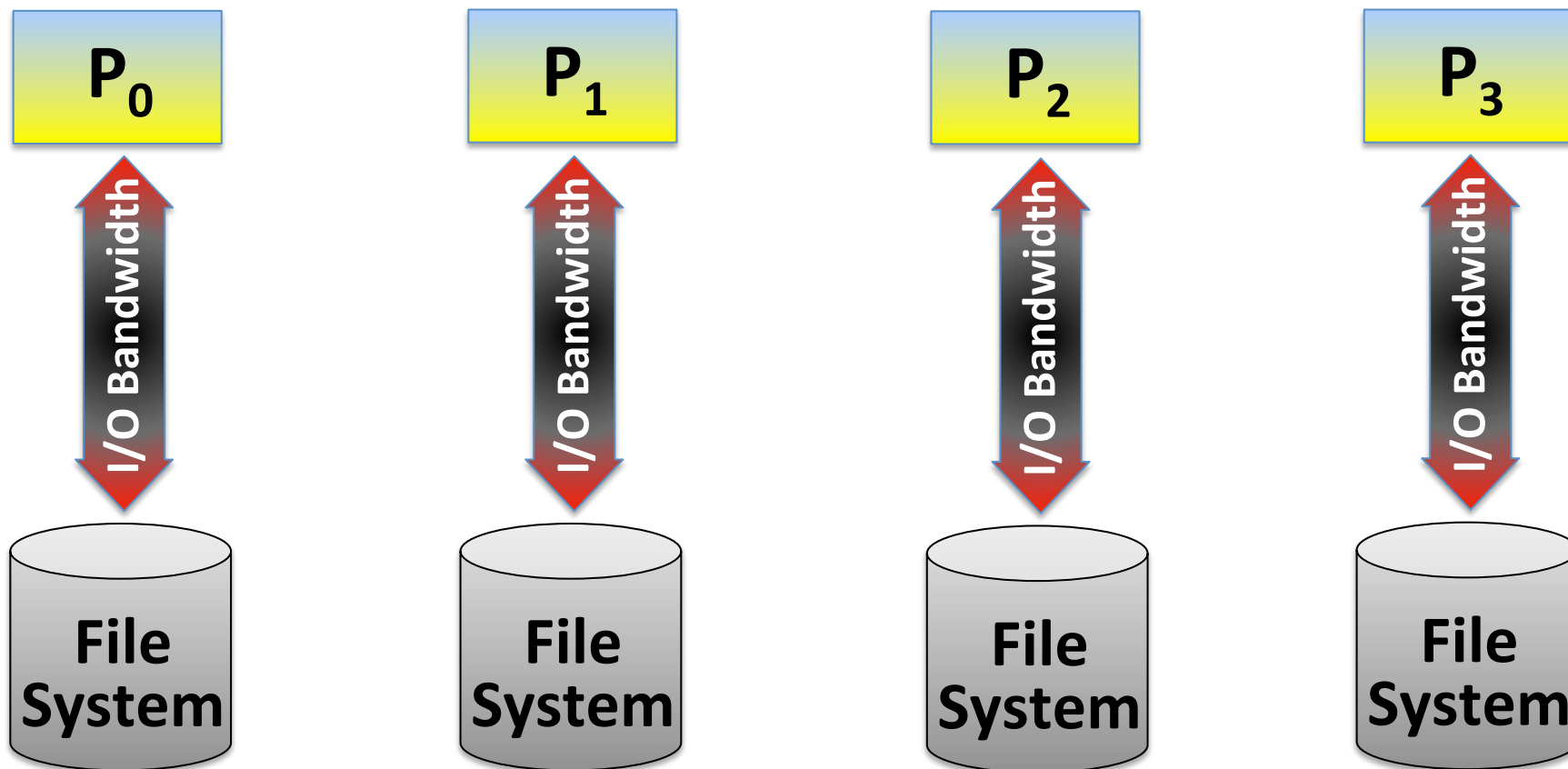
SISSA
Scuola Internazionale Superiore
di Studi Avanzati

ICTP
The Abdus Salam
International Centre
for Theoretical Physics

Parallel I/O

P₀ P₁ P₂ P₃

I/O Bandwidth

File System   File System   File System   File System

Ivan Girotto
igirotto@ictp.it

Introduction to CUDA

... curtsey of Dr. Massimo Bernaschi (CNR - http://www.iac.cnr.it/~massimo)

Ivan Girotto – igirotto@ictp.it

Information & Communication Technology Section (ICTS)

International Centre for Theoretical Physics (ICTP)
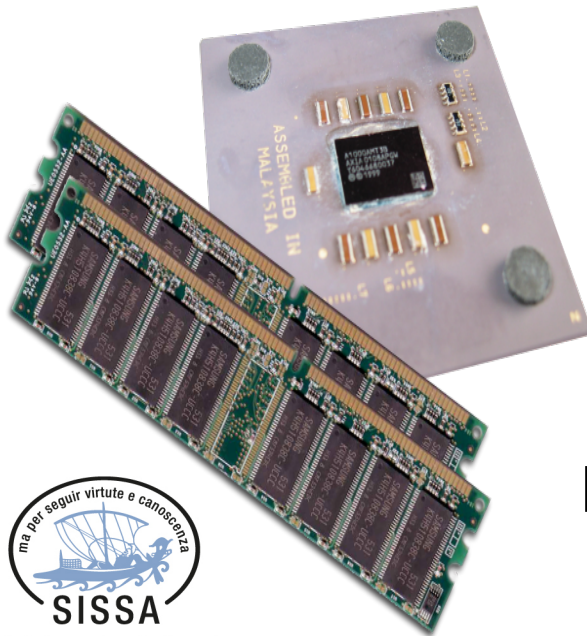
What is CUDA?

- **CUDA** = Compute Unified Device Architecture
  - Expose general-purpose GPU computing as first-class capability
  - Retain traditional DirectX/OpenGL graphics performance

- **CUDA C**
  - Based on industry-standard C
  - A handful of language extensions to allow heterogeneous programs
  - Straightforward APIs to manage devices, memory, etc.

CUDA Programming Model

- **The GPU is viewed as a compute device that:**
  - has its own RAM (device memory)
  - runs data-parallel portions of an application as kernels by using many threads

- **GPU vs. CPU threads**
  - GPU threads are extremely lightweight
  - Very little creation overhead
  - GPU needs 1000s of threads for full efficiency
  - A multi-core CPU needs only a few (basically one thread per core)

Ivan Girotto
igirotto@ictp.it

CUDA C Jargon: The Basics

- The CPU and its memory (host memory)
- The GPU and its memory (device memory)

Host

Device

What Programmer Expresses in CUDA

HOST (CPU)

P P

M

Interconnect between devices and memories

DEVICE (GPU)

M

Host Memory

~ 30/40 GBytes

CPU

1. Copy Data

~ 8/16 GBytes

2. Launch Kernel

4. Copy Result

GPU

~ 110/120 GBytes

Device Memory

3. Execute GPU kernel

Ivan Girotto
igirotto@ictp.it
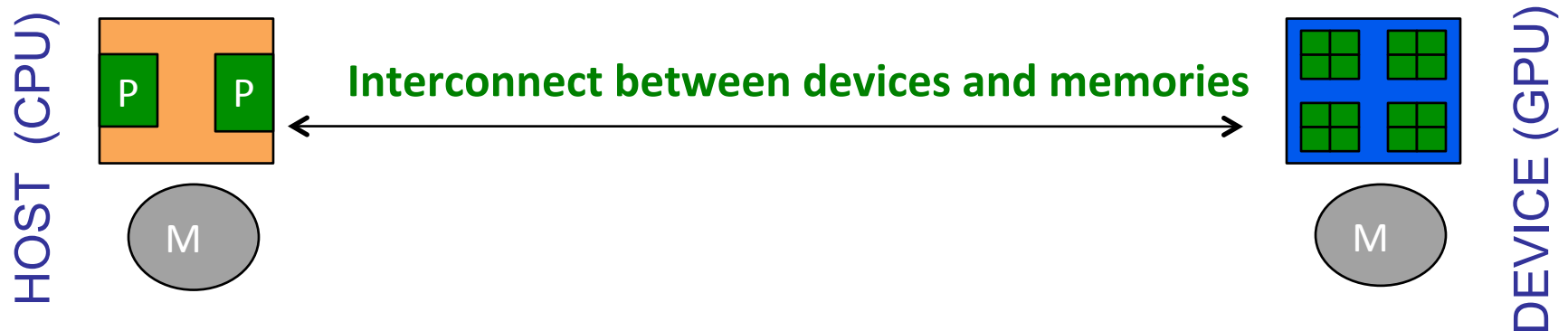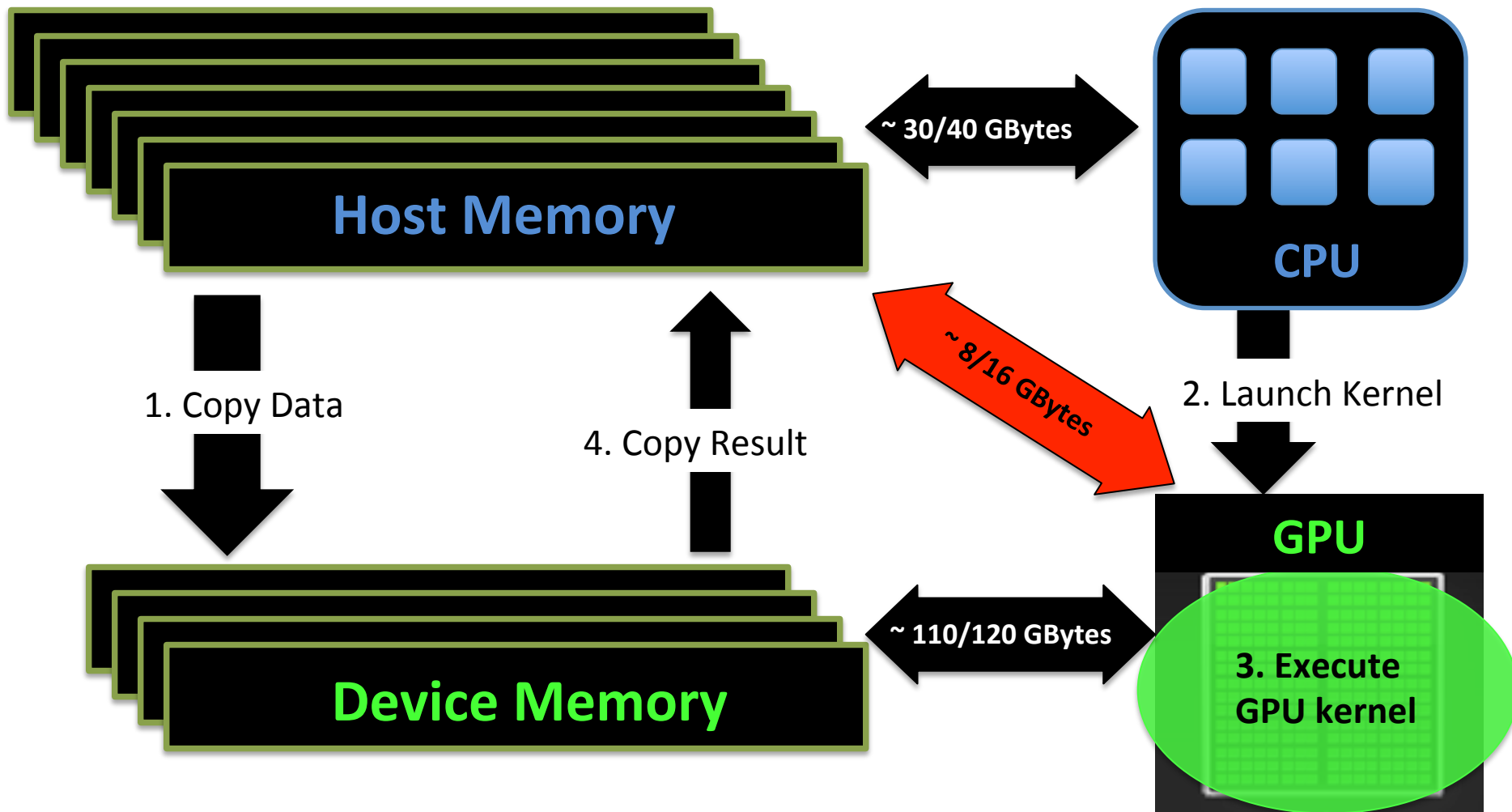
What Programmer Expresses in CUDA

✓ Computation partitioning (where does computation occur?)

    ✓ Declarations on functions __host__, __global__, __device__

    ✓ Mapping of thread programs to device: **compute <<<gs, bs>>>(<args>)**

✓ Data partitioning (where does data reside, who may access it and how?)

    ✓ Declarations on data __shared__, __device__, __constant__, …

✓ Data management and orchestration

    ✓ Copying to/from host:
    e.g., cudaMemcpy(h_obj,d_obj, size, cudaMemcpyDevicetoHost)

✓ Concurrency management

    ✓ *e.g.* __synchthreads()

Hello, World!

```c
int main( void ) {
  printf( "Hello, World!\n" );
  return 0;
}
```

- To compile: **nvcc –o hello_world hello_world.cu**

- To execute: **./hello_world**

- This basic program is just standard C that runs on the *host*

- NVIDIA's compiler (`nvcc`) will not complain about CUDA programs with no *device* code

- At its simplest, CUDA C is just C!

Ivan Girotto
igirotto@ictp.it

Hello, World! with Device Code

```
__global__ void kernel( void ) {
}


int main( void ) {

    kernel<<<1,1>>>();
    printf( "Hello, World!\n" );
    return 0;
}
```

To compile: **nvcc –o simple_kernel simple_kernel.cu**

To execute: **./simple_kernel**

Ivan Girotto
igirotto@ictp.it

Hello, World! with Device Code

```
__global__ void kernel( void ) {
}
```

- CUDA C keyword **__global__** indicates that a function
    - Runs on the device
    - Called from host code

- **nvcc** splits source file into host and device components
    - NVIDIA's compiler handles device functions like **kernel**()
    - Standard host compiler handles host functions like **main**()
        - **gcc, icc,** …
        - **Microsoft Visual C**

Hello, World! with Device Code

```
int main( void ) {
    kernel<<< 1, 1 >>>();
    printf( "Hello, World!\n" );
    return 0;
}
```

- Triple angle brackets mark a call from *host* code to *device* code
  - A "kernel launch" in CUDA jargon
  - We'll discuss the parameters inside the angle brackets later

- This is all that's required to execute a function on the GPU!

A More Complex Example

- A kernel to add two integers:

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

- As before, `__global__` is a CUDA C keyword meaning
    - `add()` will execute on the device
    - `add()` will be called from the host

A More Complex Example

- Notice that now we use *pointers* for all our variables:

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

- `add()` runs on the device…so `a`, `b`, and `c` must point to device memory

- How do we allocate memory on the GPU?

Memory Management

- Up to CUDA 4.0 host and device memory were distinct entities from the programmers' viewpoint

    - Device pointers point to GPU memory
        - May be passed to and from host code
        - (In general) May not be dereferenced from host code

    - Host pointers point to CPU memory
        - May be passed to and from device code
        - (In general) May not be dereferenced from device code
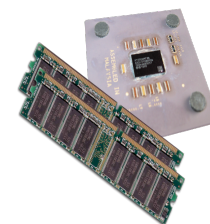
Starting on CUDA 4.0 there is a **Unified Virtual Addressing** feature.

Ivan Girotto
igirotto@ictp.it

Memory Management

- Basic CUDA API for dealing with device memory

  - **cudaMalloc**(&p, size),**cudaFree**(p),
    cudaMemcpy(t, s, size, direction)

  - Similar to their C equivalents: **malloc**(), **free**(),
    memcpy()

  pointer to pointer

```
int main( void ) {
    int a, b, c;                    // host copies of a, b, c
    int *dev_a, *dev_b, *dev_c; // device copies of a, b, c
    int size = sizeof( int );    // we need space for an integer
    // allocate device copies of a, b, c
    cudaMalloc( (void**)&dev_a, size );
    cudaMalloc( (void**)&dev_b, size );
    cudaMalloc( (void**)&dev_c, size );
    a = 2;
    b = 7;
// copy inputs to device
    cudaMemcpy( dev_a, &a, size, cudaMemcpyHostToDevice );
    cudaMemcpy( dev_b, &b, size, cudaMemcpyHostToDevice );
    // launch add() kernel on GPU, passing parameters
    add<<< 1, 1 >>>( dev_a, dev_b, dev_c );
    // copy device result back to host copy of c
    cudaMemcpy( &c, dev_c, size, cudaMemcpyDeviceToHost );
    cudaFree( dev_a ); cudaFree( dev_b ); cudaFree( dev_c )
    return 0;
}
```

Parallel Programming in CUDA C

- But wait…GPU computing is about **massive** parallelism

- So how do we run code *in parallel* on the device?

- Solution lies in the parameters between the triple angle brackets:

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

add<<< N, 1 >>>( dev_a, dev_b, dev_c );
```

- Instead of executing `add()` once, `add()` executed `N` times in parallel

Ivan Girotto
igirotto@ictp.it

Parallel Programming in CUDA C

- With **add()** running in parallel…let's do *vector* addition

- Terminology: Each parallel invocation of **add()** referred to as a ***block***

- Kernel can refer to its block's index with the variable **blockIdx.x**

- Each block adds a value from **a[]** and **b[]**, storing the result in **c[]** :

```
__global__ void add( int *a, int *b, int *c ) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- By using **blockIdx.x** to index arrays, each block handles a different index

- **blockIdx.x** is the first example of a CUDA predefined variable.

Ivan Girotto
igirotto@ictp.it

Parallel Programming in CUDA C

- We write this code:

```
__global__ void add( int *a, int *b, int *c ) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- This is what runs in parallel on the device:

Block 0

```
c[0]=a[0]+b[0];
```

Block 1

```
c[1]=a[1]+b[1];
```

Block 2

```
c[2]=a[2]+b[2];
```

Block 3

```
c[3]=a[3]+b[3];
```

Ivan Girotto
igirotto@ictp.it

Parallel Addition: `main()`

```
#define N   512
int main( void ) {
    int *a, *b, *c;              // host copies of a, b, c
    int *dev_a, *dev_b, *dev_c;  // device copies of a, b, c
    int size = N * sizeof( int ); // we need space for 512
                                  // integers

    // allocate device copies of a, b, c
    cudaMalloc( (void**)&dev_a, size );
    cudaMalloc( (void**)&dev_b, size );
    cudaMalloc( (void**)&dev_c, size );

    a = (int*)malloc( size );
    b = (int*)malloc( size );
    c = (int*)malloc( size );

    random_ints( a, N );
    random_ints( b, N );
```

Parallel Addition: `main()` (cont)

```
// copy inputs to device
cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice );
cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice );

// launch add() kernel with N parallel blocks
add<<< N, 1 >>>( dev_a, dev_b, dev_c );

// copy device result back to host copy of c
cudaMemcpy( c, dev_c, size,   cudaMemcpyDeviceToHost );

free( a ); free( b ); free( c );
cudaFree( dev_a );
cudaFree( dev_b );
cudaFree( dev_c );
return 0;
}
```

Ivan Girotto
igirotto@ictp.it

Review

- Difference between "host" and "device"
  - Host = CPU
  - Device = GPU

- Using **__global__** to declare a function as device code
  - Runs on device
  - Called from host

- Passing parameters from host code to a device function

Review (cont)

- Basic device memory management
  - **cudaMalloc**()
  - **cudaMemcpy**()
  - **cudaFree**()

- Launching parallel kernels
  - Launch **N** copies of **add()** with: **add <<< N, 1 >>>();**
  - **blockIdx.x** allows to access block's index

Exercise: look at, compile and run the *add_simple_blocks.cu* code

Ivan Girotto
igirotto@ictp.it

Threads

- Terminology: A block can be split into parallel *threads*

- Let's change vector addition to use parallel threads instead of parallel blocks:

```
__global__ void add( int *a, int *b, int *c ) {
c[threadIdx.x] = a[threadIdx.x]+ b[threadIdx.x];
}
```

- We use **threadIdx.x** instead of **blockIdx.x** in **add()**

- **main()** will require one change as well…

Parallel Addition (Threads): `main()`

```c
#define N  512
int main( void ) {
    int *a, *b, *c;            // host copies of a, b, c
    int *dev_a, *dev_b, *dev_c;  // device copies of a, b, c
    int size = N * sizeof( int ); // we need space for 512
                                  // integers

    // allocate device copies of a, b, c
    cudaMalloc( (void**)&dev_a, size );
    cudaMalloc( (void**)&dev_b, size );
    cudaMalloc( (void**)&dev_c, size );

    a = (int*)malloc( size );
    b = (int*)malloc( size );
    c = (int*)malloc( size );

    random_ints( a, N );
    random_ints( b, N );
```

Parallel Addition (Threads): `main()` (cont)

```
// copy inputs to device
cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice );
cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice );

// launch add() kernel with N parallel threads
add<<< 1, N >>>( dev_a, dev_b, dev_c );

// copy device result back to host copy of c
cudaMemcpy( c, dev_c, size,    cudaMemcpyDeviceToHost );

free( a ); free( b ); free( c );
cudaFree( dev_a );
cudaFree( dev_b );
cudaFree( dev_c );
return 0;
}
```

Ivan Girotto
igirotto@ictp.it

Using Threads _And_ Blocks

- We've seen parallel vector addition using
  - Many blocks with 1 thread apiece
  - 1 block with many threads

- Let's adapt vector addition to use lots of **both** blocks and threads

- After using threads and blocks together, we'll talk about **why** threads

- First let's discuss data indexing…

Ivan Girotto
igirotto@ictp.it

Indexing Arrays With Threads & Blocks

- No longer as simple as just using **`threadIdx.x`** or **`blockIdx.x`** as indices

- To index array with 1 thread per entry (using 8 threads/block)
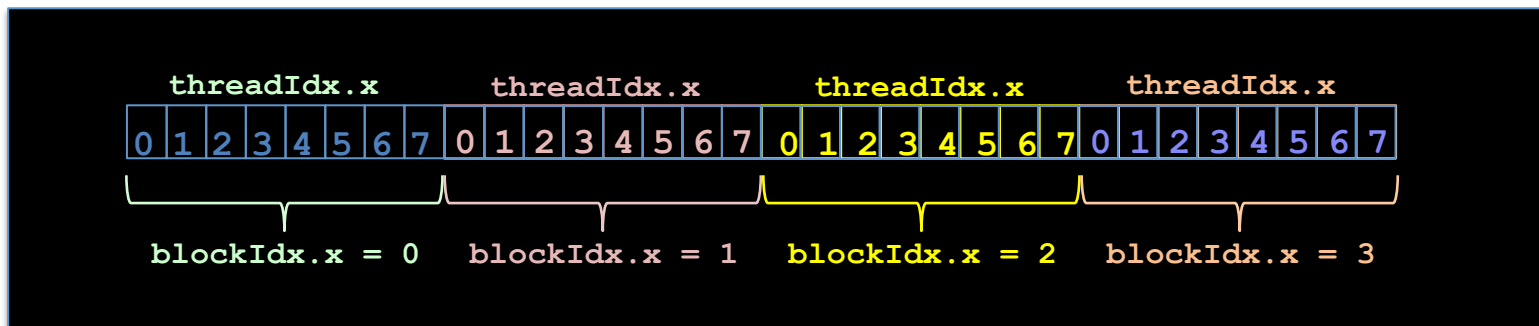
- If we have **M** threads/block, a unique array index for each entry is given by

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

```
int index =        x     +     y      * width;
```
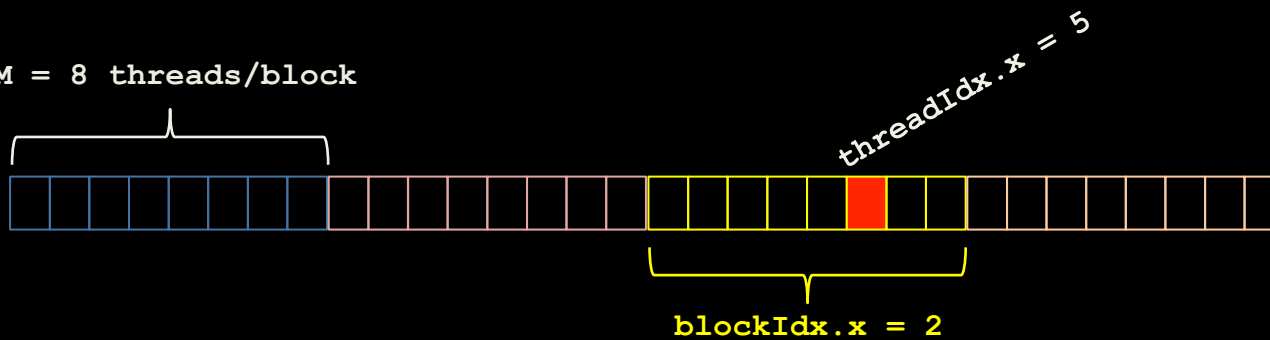
Indexing Arrays: Example

- In this example, the **red** entry would have an index of 21:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 |

M = 8 threads/block

threadIdx.x = 5

blockIdx.x = 2

```
int index    = threadIdx.x + blockIdx.x * M;
             =      5       +     2       * 8;
             = 21;
```

Ivan Girotto
igirotto@ictp.it

Indexing Arrays: other examples (4 blocks with 4 threads *per* block)

```
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = 7;
}
                                Output: 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7
```

```
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = blockIdx.x;
}
                                Output: 0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3
```

```
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = threadIdx.x;
}
                                Output: 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3
```

Ivan Girotto
igirotto@ictp.it

Addition with Threads and Blocks

- **`blockDim.x`** is a built-in variable for threads per block:

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

- **`gridDim.x`** is a built-in variable for blocks in a grid;

- A combined version of our vector addition kernel to use blocks *and* threads:

  ```
  __global__ void add( int *a, int *b, int *c ) {
  int index = threadIdx.x + blockIdx.x * blockDim.x;
      c[ index ] = a[ index ] + b[ index ];
  }
  ```

- So what changes in **`main()`** when we use both blocks and threads?

Ivan Girotto
igirotto@ictp.it

Parallel Addition (Bloks/Threads): `main()`

```c
#define N   (2048 * 2048)
#define THREADS_PER_BLOCK 512
int main( void ) {
    int *a, *b, *c;              // host copies of a, b, c
    int *dev_a, *dev_b, *dev_c;  // device copies of a, b, c
    int size = N * sizeof( int ); // we need space for N integers

    // allocate device copies of a, b, c
    cudaMalloc( (void**)&dev_a, size );
    cudaMalloc( (void**)&dev_b, size );
    cudaMalloc( (void**)&dev_c, size );

    a = (int*)malloc( size );
    b = (int*)malloc( size );
    c = (int*)malloc( size );

    random_ints( a, N );
    random_ints( b, N );
```

Parallel Addition (Threads): `main()` (cont)

```c
// copy inputs to device
cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice );
cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice );

// launch add() kernel with blocks and threads
add<<< N/THREADS_PER_BLOCK, THREADS_PER_BLOCK >>>(dev_a, dev_b, dev_c);

// copy device result back to host copy of c
cudaMemcpy( c, dev_c, size,   cudaMemcpyDeviceToHost );

free( a ); free( b ); free( c );
cudaFree( dev_a );
cudaFree( dev_b );
cudaFree( dev_c );
return 0;
}
```

Ivan Girotto
igirotto@ictp.it

Exercises

- Array reversal: fill an input array d_in and save the content in revers order into d_out. The revers is performed into the GPU.
    - d_in is [100, 110, 200, 220, 300]
      then d_out must be [300, 220, 200, 110, 100]
    - blockDim.x is the number of threads per block
    - gridDim.x is the number of blocks in a grid

- Implement a Matrix Transpose using threads and blocks

- Implement a Matrix Multiplication for Matrix sizes 2048^2.
  Use max 512 threads x block.