



W • S E N S E
INTEGRATED CABLELESS SOLUTIONS



SAPIENZA
UNIVERSITÀ DI ROMA

Programmazione di sistemi multicore **CUDA**

Fabrizio Gattuso

Avvisi

12 Dicembre **NO LEZIONE PER IT MEETING**

Il secondo esonero sarà svolto insieme al primo appello

Date esame completo e primo esonero:

- **14 Gennaio** ore 16-18, Aula 2L - Via del Castro Laurenziano 7A
- **4 Febbraio** ore 16-18, Aula 2L - Via del Castro Laurenziano 7A

È uscito il codice OPIS per la **valutazione del corso**

È importante che cominciate ad **inviare la mail con la richiesta materiale per il progetto**

CUDA

L'architettura **CUDA** è composta da due componenti:

1. L'**hardware** CUDA (le GPU NVIDIA)
2. Le **librerie** CUDA che estendono il **C** standard.

Ovviamente per eseguire il codice è necessario avere una scheda dedicata NVIDIA. In queste lezioni parleremo in modo molto generale del paradigma e del framework.

CUDA (2)

Per chi è munito di scheda video NVIDIA:

<https://developer.nvidia.com/cuda-downloads>

- Online esistono diversi **emulatori** ma non sono molto semplici da utilizzare o installare.
- Se siete in possesso di buoni **Amazon Cloud** allora potete utilizzare una macchina dotata di scheda video.
- Per possessori di schede video di altri brand **Open CL** è un framework alternativo ma molto simile sia nella sintassi che nel paradigma CUDA.

Calcolo eterogeneo

Il codice sviluppato per la piattaforma CUDA è definito eterogeneo in quanto:

- Alcune parti del codice sono eseguite sulla **CPU**
- Altre parti vengono eseguite direttamente sulla **GPU**

La CPU è chiamata **Host** mentre la GPU **Device**

Calcolo eterogeneo (2)

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

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

    // Store the result
    out[gidex] = 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);

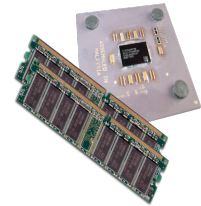
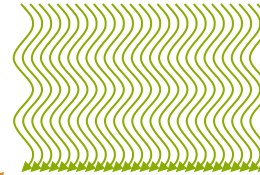
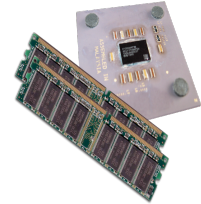
    // Cleanup
    free(in); free(out);
    cudaFree(d_in); cudaFree(d_out);
    return 0;
}
```

parallel fn

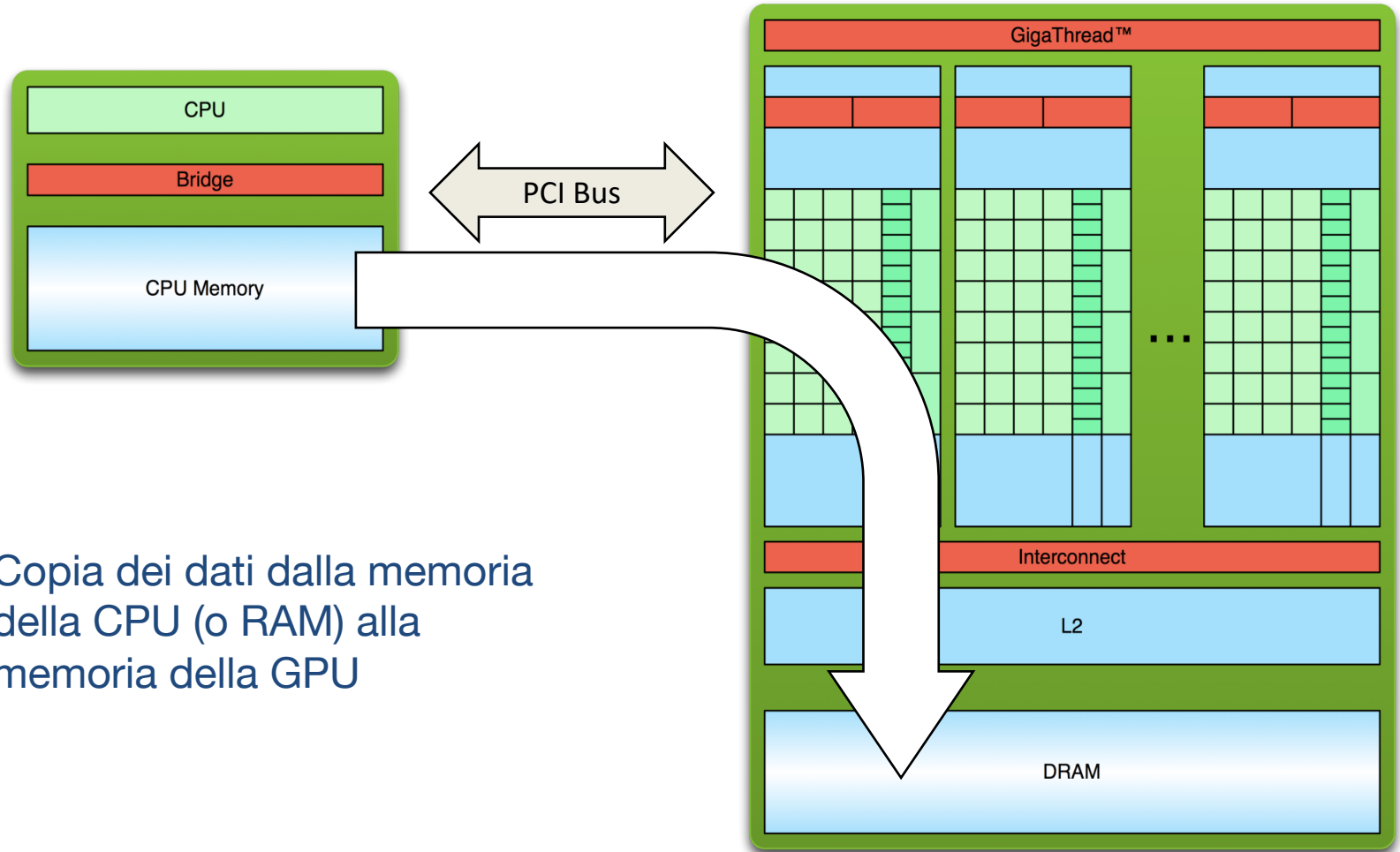
serial code

parallel code

serial code



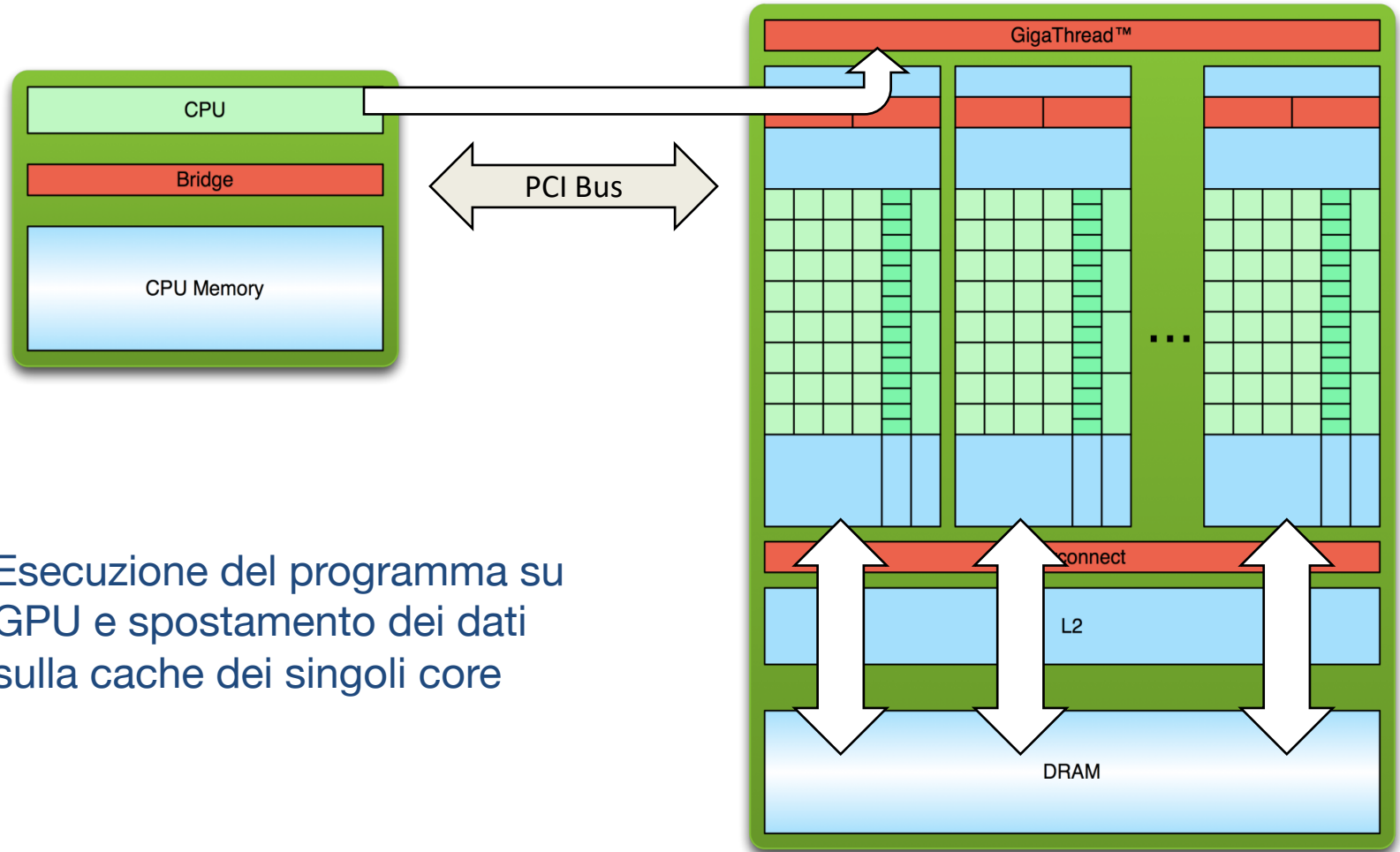
Flusso di computazione



1. Copia dei dati dalla memoria della CPU (o RAM) alla memoria della GPU



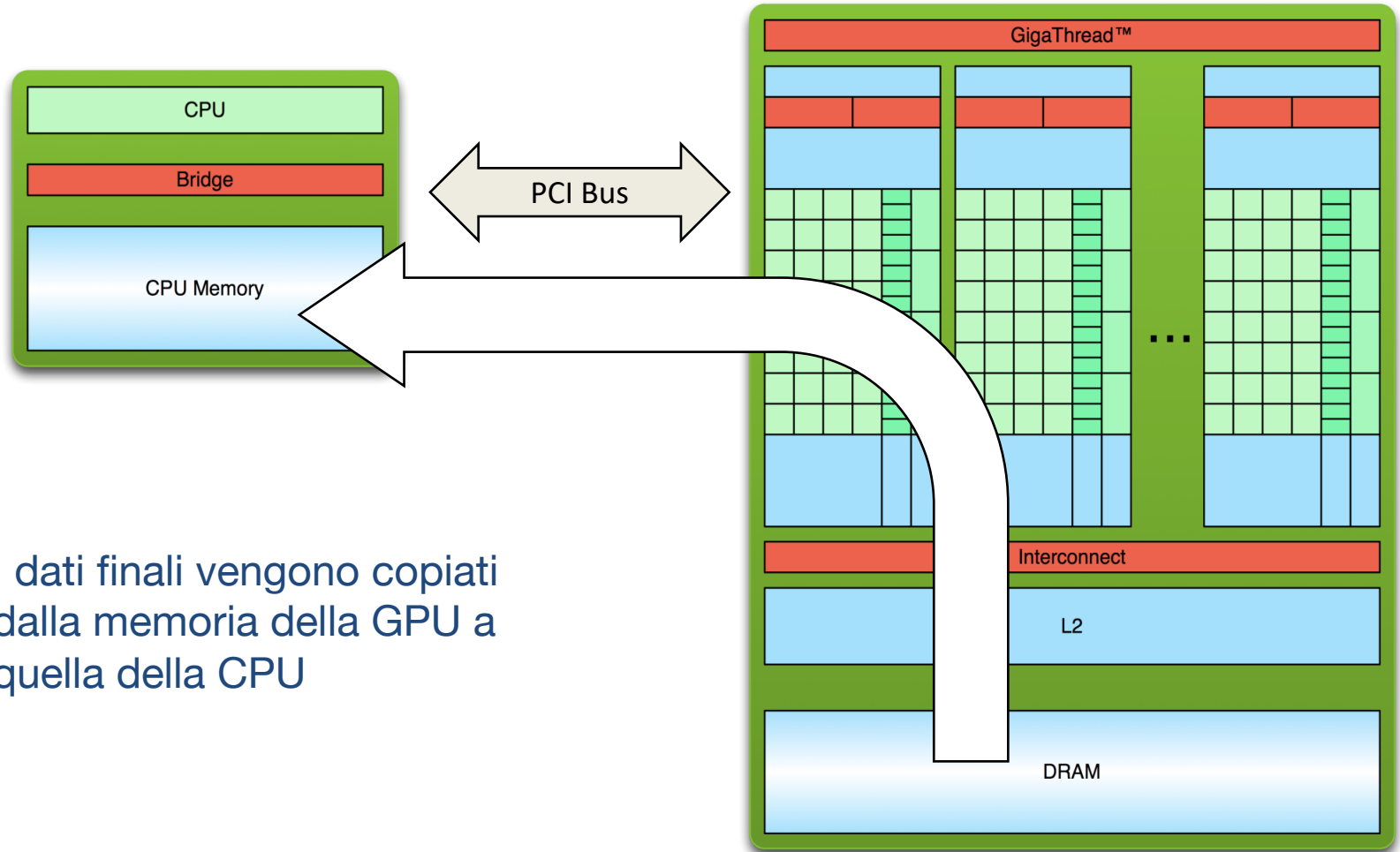
Flusso di computazione (2)



2. Esecuzione del programma su GPU e spostamento dei dati sulla cache dei singoli core



Flusso di computazione (3)



3. I dati finali vengono copiati dalla memoria della GPU a quella della CPU



Hello World in CUDA

```
__global__ void mykernel(void) { }
```

→ Codice **Device**

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

↘ Codice **Host**

Il codice host viene compilato con **GCC** mentre il codice device con **NVCC**

Hello World in CUDA (2)

La compilazione viene eseguita una sola volta con NVCC. Sarà lui ad occuparsi della compilazione tramite GCC delle componenti host.

Compilazione: `nvcc -o mycode mycode.cu`

Esecuzione: `./mycode`

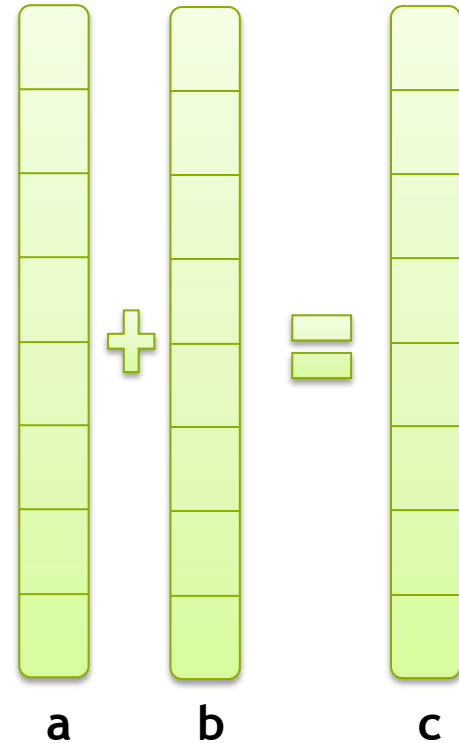
La compilazione può essere eseguita anche senza la necessità di avere l'hardware.

Somma di due vettori

Partiamo con la somma di due interi

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

```
int main(void) {  
    int a, b, c;  
    add<<<1,1>>>(&a, &b, &c);  
}
```



Qualche problema?



Gestione della memoria

La **memoria** dell'**host** (CPU) e del **device** (GPU) sono sempre **separate**. È sconsigliata la condivisione con i puntatori.

È sempre consigliata:

- L'allocazione sulla CPU
- e la successiva copia sulla GPU

A questo scopo esistono delle funzioni CUDA simili a quelle standard C:

cudaMalloc() cudaFree() cudaMemcpy()

Gestione della memoria (3)

```
cudaError_t cudaMalloc ( void** devPtr, size_t size )
```

```
cudaError_t cudaFree ( void* devPtr )
```

```
cudaError_t cudaMemcpy ( void* dst, const void* src,  
size_t count, cudaMemcpyKind kind )
```

cudaMemcpyKind:

cudaMemcpyHostToHost = 0

cudaMemcpyHostToDevice = 1

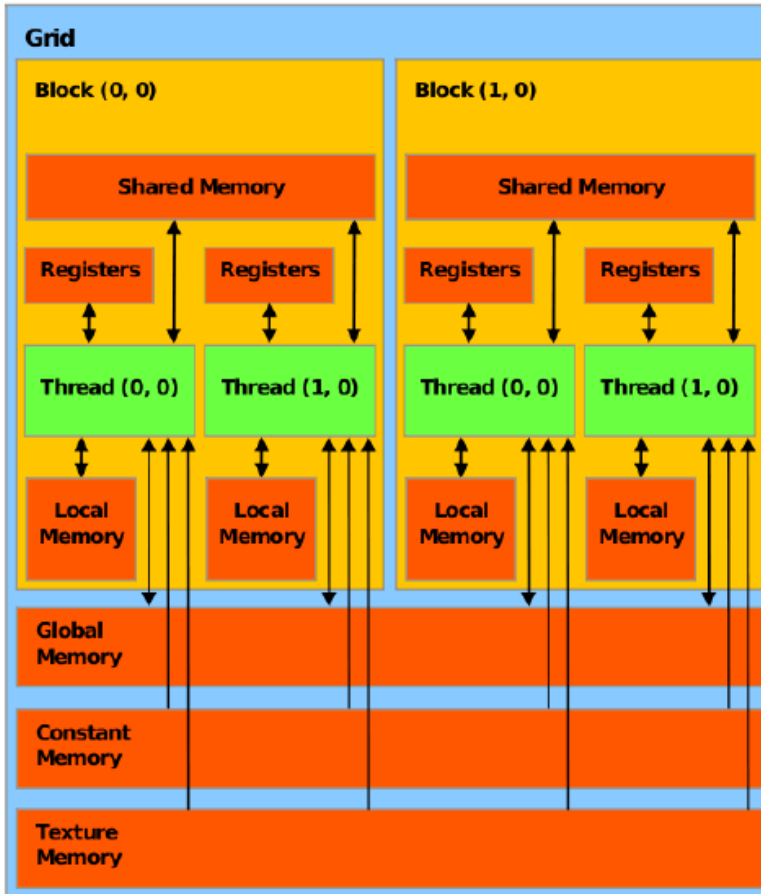
cudaMemcpyDeviceToHost = 2

cudaMemcpyDeviceToDevice = 3

cudaMemcpyDefault = 4



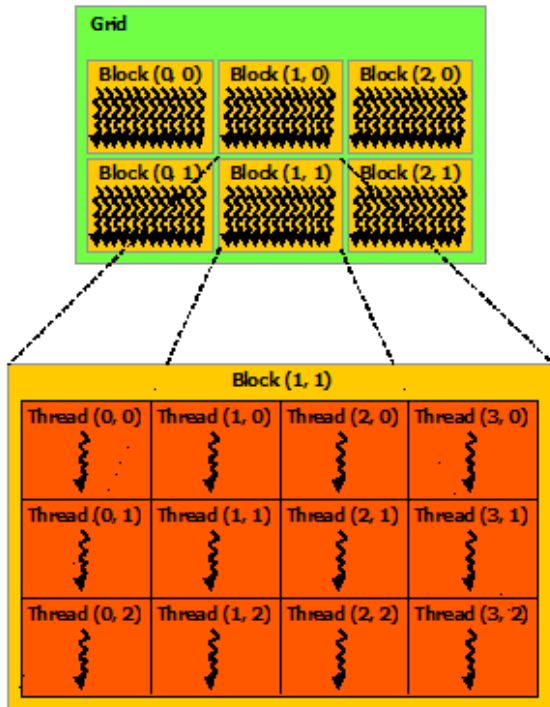
Gestione della memoria (4)



Variable	Memoria	Scope
<code>int var</code>	register	thread
<code>int array[10]</code>	local	thread
<code>__shared__</code>	shared	block
<code>__device__</code>	global	grid
<code>__constant__</code>	constant	grid

Il parallelismo

Fino ad adesso abbiamo eseguito del codice sulla GPU ma in modalità sequenziale



`add<<<1,1>>>`

`add<<<dimGrid, dimBlock>>>`

`add<<<N, 1>>>`

Esegui il codice su N block e
1 thread



Concentriamoci sui blocchi

Una **GRID** è un insieme di **BLOCK**.

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

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

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

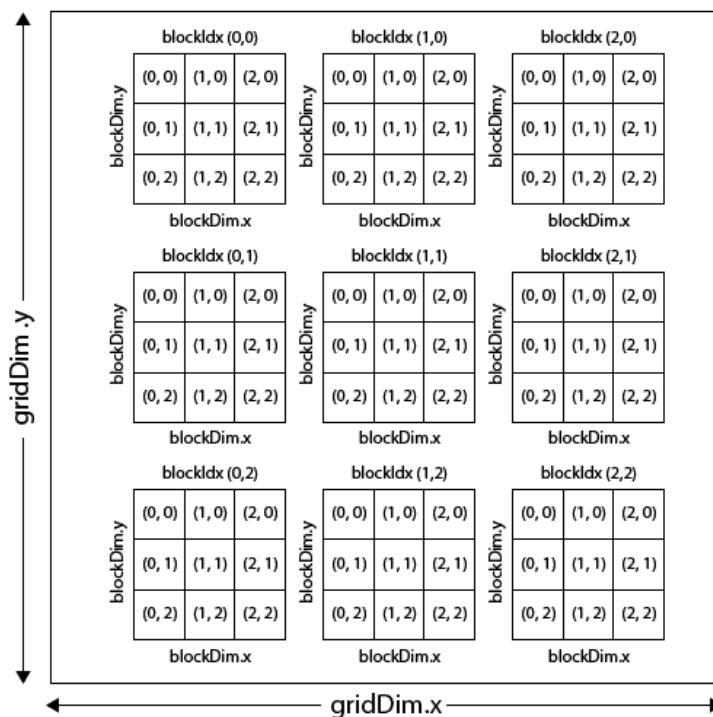


Thread

Un **BLOCK** può essere diviso in più **THREAD**.

I thread di una GPU sono molti di più e molto più leggeri.

CUDA Grid



Thread (2)

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

add<<<1, N>>>

Esegui questo codice su 1 block e N threads

Thread 0

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

Thread 1

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

Thread 2

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

Thread 3

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

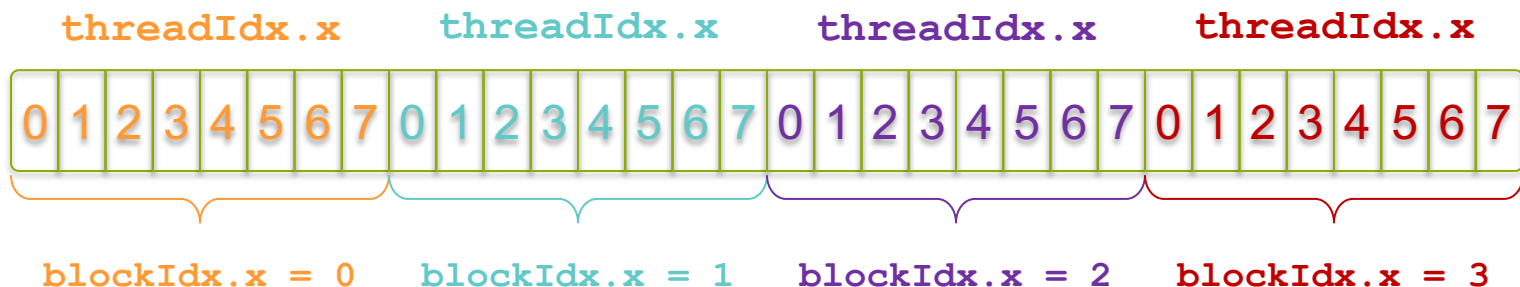


Combiniamo blocchi e thread

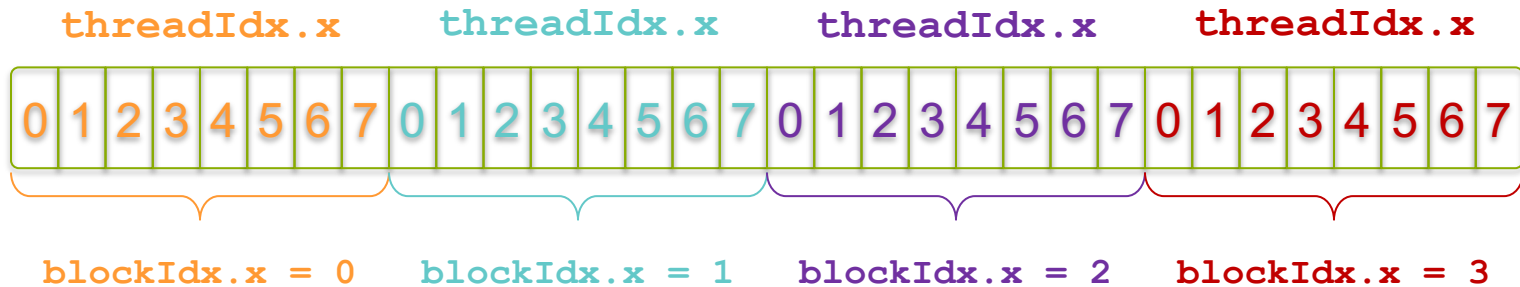
La potenza computazionale di una GPU deriva dalla possibilità di utilizzare **blocchi**, **thread** e **grid** modellati in base al problema da risolvere.

Questo fornisce una potenza di calcolo molto superiore rispetto a una CPU.

Il problema principale è la **CAPACITÀ DI INDEXING**.



Indexing



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

`M = blockDim.x` (in questo caso)



Indexing (2)



$M = 8$

$threadIdx.x = 5$



$blockIdx.x = 2$

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

Thread (3)

Come avete visto i thread introducono un livello di complessità in più. Perché non è possibile calcolare tutto con i soli blocchi?

Perché tra threads è molto più facile:

- **Comunicare**
- **Sincronizzare**

Compute Capability

È una tabella che mostra le capacità tecniche di ogni singola GPU. Interrogabile tramite l'utility **deviceQuery**.

```
deviceQuery — bash — 156x41
./deviceQuery Starting...

CUDA Device Query (Runtime API) version (CUDA static linking)

Detected 1 CUDA Capable device(s)

Device 0: "GeForce GT 650M"
  CUDA Driver Version / Runtime Version      6.0 / 6.0
  CUDA Capability Major/Minor version number: 3.0
  Total amount of global memory:             512 MBytes (536543232 bytes)
  ( 2 ) Multiprocessors, (192) CUDA Cores/MP: 384 CUDA Cores
  GPU Clock rate:                            405 MHz (0.41 GHz)
  Memory Clock rate:                          2000 Mhz
  Memory Bus Width:                           128-bit
  L2 Cache Size:                              262144 bytes
  Maximum Texture Dimension Size (x,y,z)     1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers 1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers 2D=(16384, 16384), 2048 layers
  Total amount of constant memory:           65536 bytes
  Total amount of shared memory per block:    49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                  32
  Maximum number of threads per multiprocessor: 2048
  Maximum number of threads per block:        1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size (x,y,z):  (2147483647, 65535, 65535)
  Maximum memory pitch:                       2147483647 bytes
  Texture alignment:                           512 bytes
  Concurrent copy and kernel execution:       Yes with 1 copy engine(s)
  Run time limit on kernels:                   Yes
  Integrated GPU sharing Host Memory:          No
  Support host page-locked memory mapping:     Yes
  Alignment requirement for Surfaces:          Yes
  Device has ECC support:                       Disabled
  Device supports Unified Addressing (UVA):    Yes
  Device PCI Bus ID / PCI location ID:         1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 6.0, CUDA Runtime Version = 6.0, NumDevs = 1, Device0 = GeForce GT 650M
Result = PASS
```

