

GPU Computing

Laurea Magistrale in Informatica - AA 2024/25

Docente **G. Grossi**

Lezione 2 – Il modello di programmazione CUDA

Sommario

- ✓ Modelli di sistemi di computazione multi-threading (richiami)
- ✓ Modello di programmazione CUDA
- ✓ Gerarchia di thread 1D, 2D, 3D
- ✓ Kernel CUDA
- ✓ Flip di un'immagine 1D
- ✓ Blurring di un'immagine 2D

Programmazione multi-threading

Dal single-core a multi-core aumentando il numero di thread

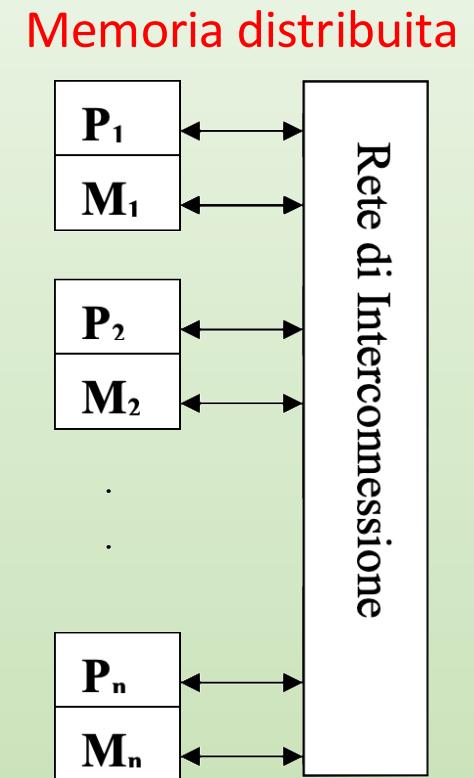
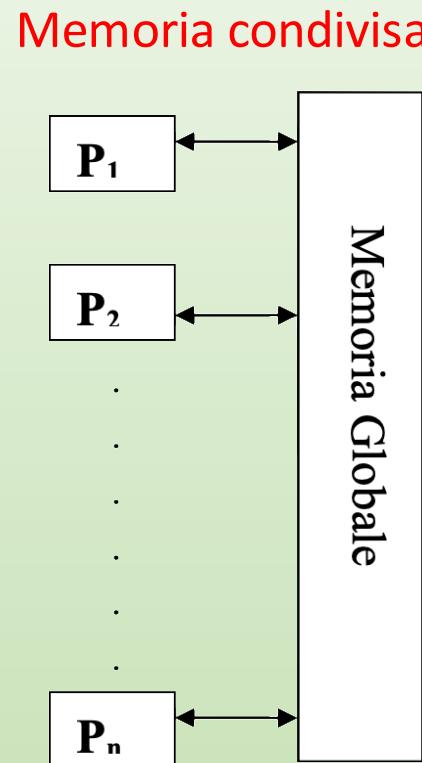
Modelli per sistemi paralleli

Un modello di programmazione parallela rappresenta un'astrazione per un sistema di calcolo parallelo in cui è conveniente esprimere algoritmi concorrenti/paralleli

- ✓ Si possono individuare **diversi livelli di astrazione**, classificabili in 4 tipi base: **modello macchina**, **modello architetturale**, **modello computazionale** e **modello di programmazione**
 - **Modello macchina**: livello più basso che descrive l'hw e il sistema operativo (registri, memoria, I/O), il linguaggio assembly è basato su questo livello di astrazione
 - **Modello architetturale**: rete di interconnessione di piattaforme parallele, organizzazione della memoria e livelli di sincronizzazione tra processi, modalità di esecuzione delle istruzioni di tipo SIMD o MIMD
 - **Modello computazionale**: modello formale di macchina che fornisce metodi analitici per fare predizioni teoriche sulle prestazioni (in base a tempo, uso delle risorse, ...). Per es. il modello RAM descrive il comportamento del modello architetturale di Von Neumann (processore, memoria, operazioni, ...) Il modello PRAM estende RAM per architetture parallele

Modello PRAM (Parallel Random-Access Model)

- ✓ Il più semplice **modello di calcolo parallelo**
 - a **memoria condivisa**
 - con **n-processori**
 - uso di **variabile condivisa** per scambiare valori tra due processori P_i e P_j
- ✓ Il **calcolo** procede **per passi**:
 - Ad ogni passo ogni processore può fare **una operazione** sui dati con possesso esclusivo
 - può **leggere** o **scrivere** nella memoria condivisa
- ✓ E' possibile selezionare un **insieme di processori** che eseguono tutti la **stessa istruzione** (su dati generalmente diversi - **SIMD**)
- ✓ Gli altri **processori** restano **inattivi**
- ✓ I processori attivi sono **sincronizzati** (eseguono la stessa istruzione simultaneamente)

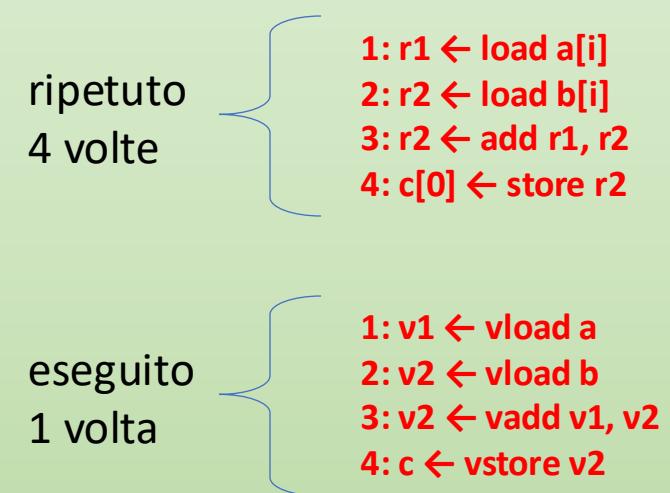
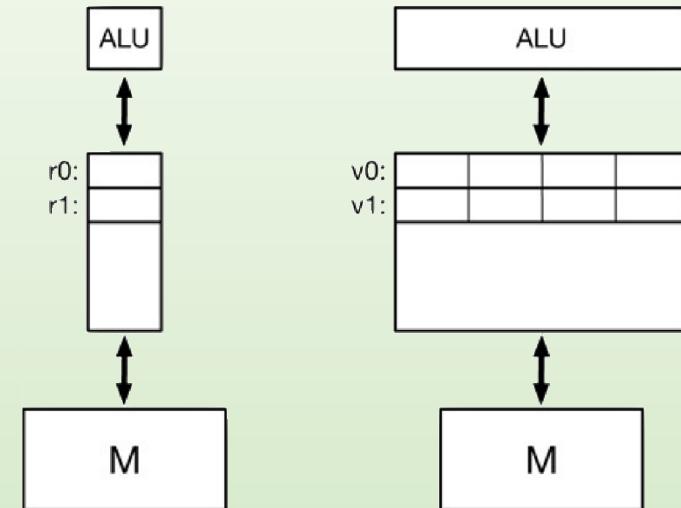


SIMD

- ✓ Modelli SIMD sono basati su **unità funzionali** contenute in processori general purpose

- Le **ALU SIMD** possono effettuare **operazioni multiple** simultaneamente in un ciclo di clock
- Usano **registri** che effettuano **load** e **store** di molteplici elementi di dati in una sola transizione

- ✓ La popolarità SIMD deriva dall'uso esplicito di linguaggi di programmazione parallela sfruttando il **parallelismo dei dati**
- ✓ Possono essere programmati anche in assembler usando **istruzioni vettoriali**



•••

- ✓ **Modello di programmazione parallela:** specifica la “vista” del programmatore del computer parallelo definendo come si possa codificare un algoritmo
 - Comprende la **semantica** del linguaggio di programmazione, librerie, compilatore, tool di profiling
 - Dice di che **tipo** sono le **computazioni** parallele (instruction level, procedural level o parallel loops)
 - Permette di dare **specifiche implicite** o **esplicite** (da parte utente) per il parallelismo
 - Modalità di **comunicazione** tra **unità** di computazione per lo scambio di informazioni (shared variable)
 - Meccanismi di **sincronizzazione** per gestire computazioni e comunicazioni tra diverse unità che operano in parallelo
 - Molti forniscono il concetto di **parallel loop** (iterazioni indipendenti), altri di **parallel task** (moduli assegnati a processori distinti eseguiti in parallelo)
 - Un **programma parallelo** è eseguito da processori in un ambiente parallelo tale che in ogni processore si ha uno o più flussi di esecuzione, quest’ultimi sono detti **processi** o **thread**
 - Ha una organizzazione dello **spazio di indirizzamento**: per esempio, distribuito (no variabili shared quindi uso del message passing) o condiviso (uso di variabili shared per lo scambio di informazioni)

Passi di parallelizzazione... per riassumere

- ✓ Si assume che la parallelizzazione si effettui a partire da un programma o **algoritmo sequenziale**
- ✓ La computazione parallela deve essere suddivisa in **task** dei quali deve essere stabilita la dipendenza
- ✓ Un task è una **sequenza** di computazioni eseguite dallo **stesso processore** o core
- ✓ Dipendentemente dal **modello di memoria**, un task può accedere a **memoria condivisa** o usare tecniche di **message passing**
- ✓ Dipendentemente dall'applicazione i task possono essere **staticamente** fissati all'inizio dell'esecuzione (start) o creati **dinamicamente** durante l'esecuzione
- ✓ il tempo di computazione del task è detto **granularità**: se troppo fine si ha overhead di scheduling, se troppo grezza si può perdere efficienza nell'uso di core
- ✓ I **task** sono assegnati a un processo o **thread** (scheduling) cercando di **bilanciare il carico** tra questi ultimi
- ✓ Il sistema operativo si occupa di **assegnare** fisicamente i processi o thread ai vari core disponibili

Processi

- ✓ Un processo è un **programma in esecuzione** con diverse risorse allocate (stack, heap, registri, prog c.)
- ✓ Un processo con un solo thread (**heavyweight process**) può eseguire un'attività alla volta
- ✓ Può consistere di **vari thread** che condividono lo stesso (esclusivo) **spazio di indirizzamento**
- ✓ La moltitudine di **processi** in esecuzione (round-robin) subiscono un **context switch**
- ✓ I **processi** sono adatti in ambiente con **memoria distribuita i thread su memoria condivisa**
- ✓ Possono essere **creati a runtime** (fork in Unix) ed sono una identica **copia del padre**

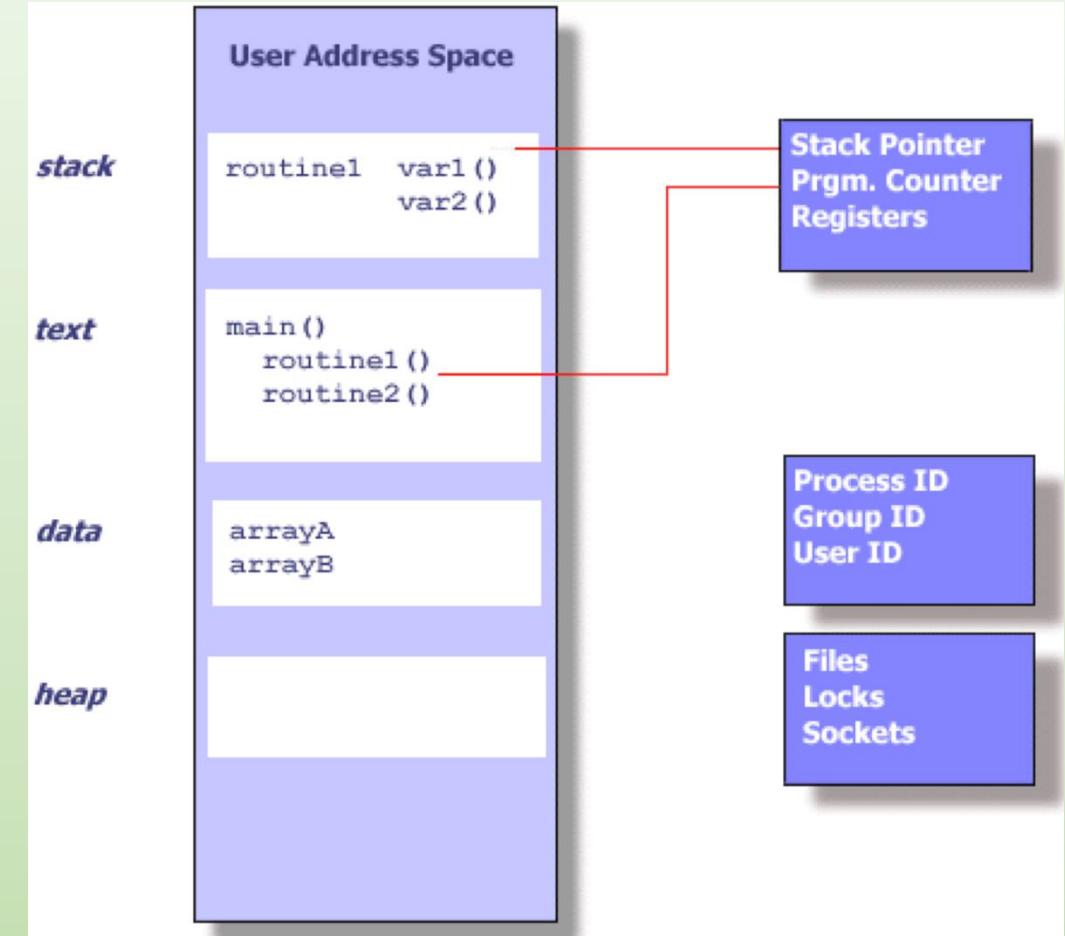
Thread in Unix

- ✓ Un **thread** (su **CPU**) è una **estensione** del modello di processo (**lightweight process** perché possiedono un contesto più snello rispetto ai processi)
- ✓ E' un **flusso di istruzioni** di un programma e viene schedulato come unità **indipendente** nelle code di esecuzione dei processi della CPU (scheduler)
- ✓ Condivide lo **spazio di indirizzamento** con gli altri **thread** del processo: rappresentato da un thread control block (**TCB**) che punta al **PCB** del processo contenitore
- ✓ Dal punto di vista del **programmatore**, l'esecuzione del thread è **sequenziale**, quindi un'istruzione eseguita alla volta, con un **puntatore alla prossima istruzione** da eseguire e verificando costantemente l'accesso ai dati
- ✓ Vi sono meccanismi di **sincronizzazione** tra thread per evitare **race condition** (accesso a variabili condivise o in generale comportamenti non deterministici)

Processi e thread in Unix

PROCESSI

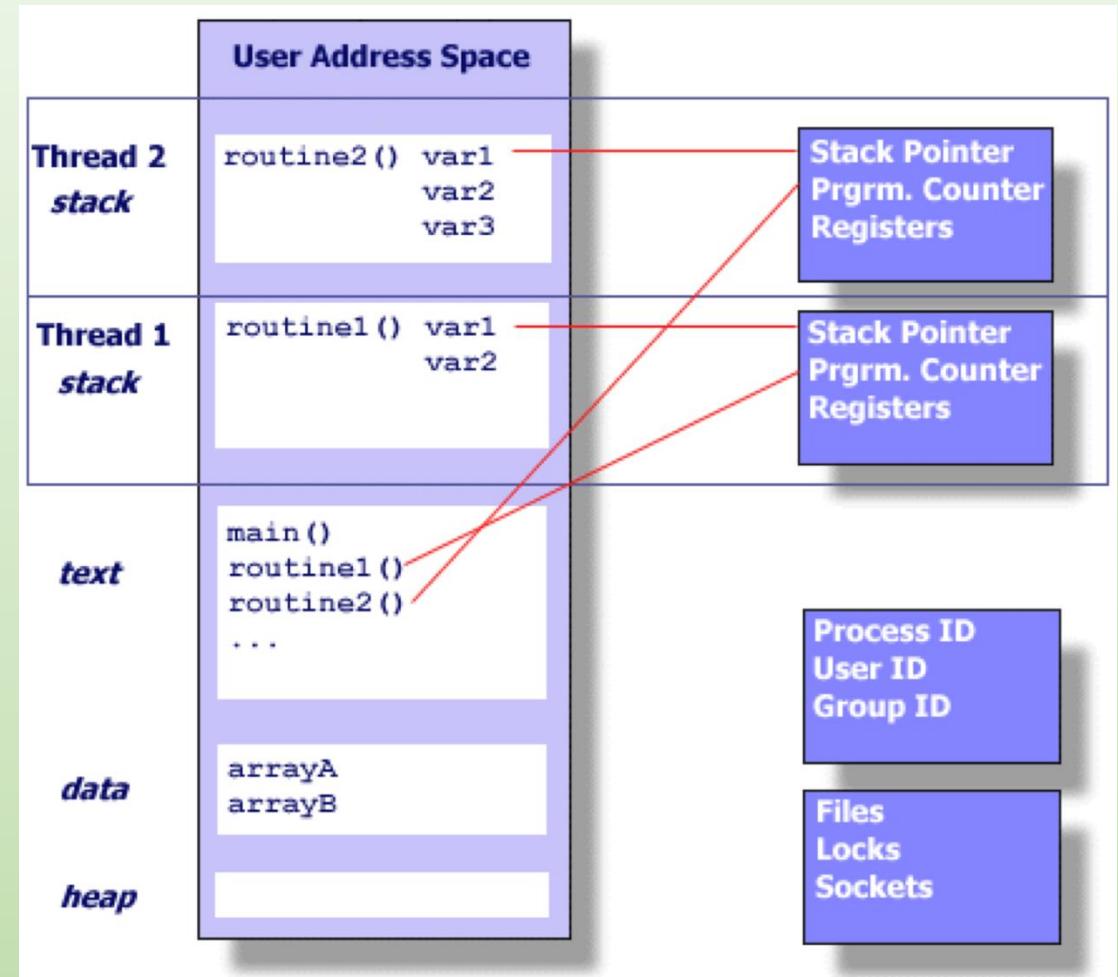
- ✓ Il processo ha il proprio **contesto**:
 - process ID, Program Counter, Stato dei Registri, Stack, Codice, Dati, File descriptors, IO devices,... (descrittore PCB – Process Control Block)
- ✓ E' pensato per eseguire codice **sequenzialmente**
- ✓ L'astrazione dei thread vuole consentire di eseguire **procedure concorrentemente** (in parallelo) definiti da specifici costrutti
- ✓ Ciascuna procedura da eseguire in **parallelo** sarà un **thread** (libreria **pthread**)



Processi e thread in Unix

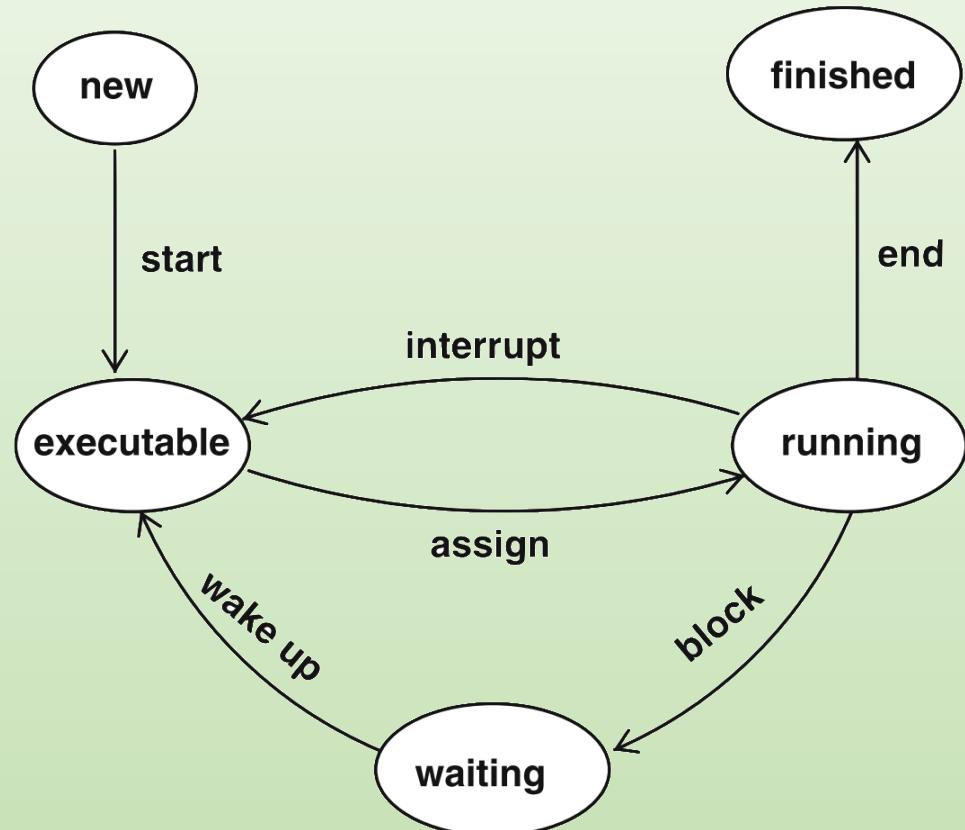
THREAD

- ✓ Un thread è un **singolo flusso** di istruzioni, all'interno di un processo, che lo scheduler può fare **eseguire separatamente** e **concorrentemente** con il resto del processo
- ✓ Per fare questo uno thread deve possedere **strutture dati** per realizzare un proprio flusso di controllo
- ✓ Un thread può essere pensato come una **procedura** che lavora in **parallelo** con altre procedure



Stati di un thread

- ✓ **Newly generated:** il thread è stato generato e non ha ancora eseguito operazioni
- ✓ **Executable:** il thread è pronto per l'esecuzione, ma al momento non è assegnato a nessuna unità di calcolo
- ✓ **Running:** il thread è in esecuzione
- ✓ **Waiting:** il thread è in attesa di un evento esterno (es. I/O) quindi non può andare in esecuzione fino a che l'evento non si verifica
- ✓ **Finished:** il thread ha terminato tutte le operazioni



I thread in CUDA

Gerarchia astratta 1D, 2D, 3D di thread

CUDA multithreading...

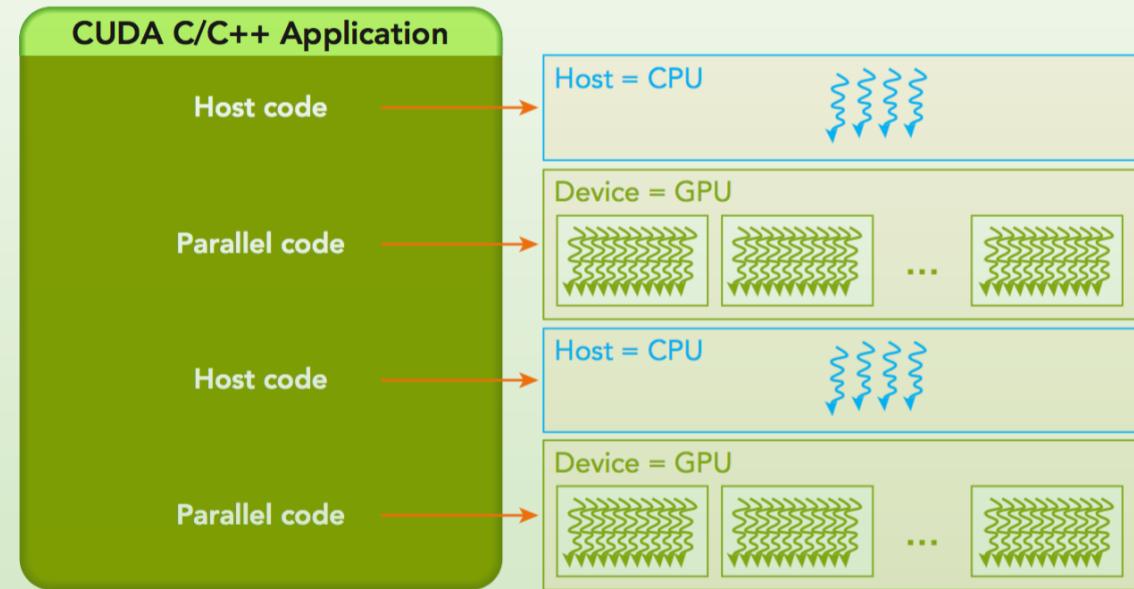


Che cosa significa programmare in CUDA C?

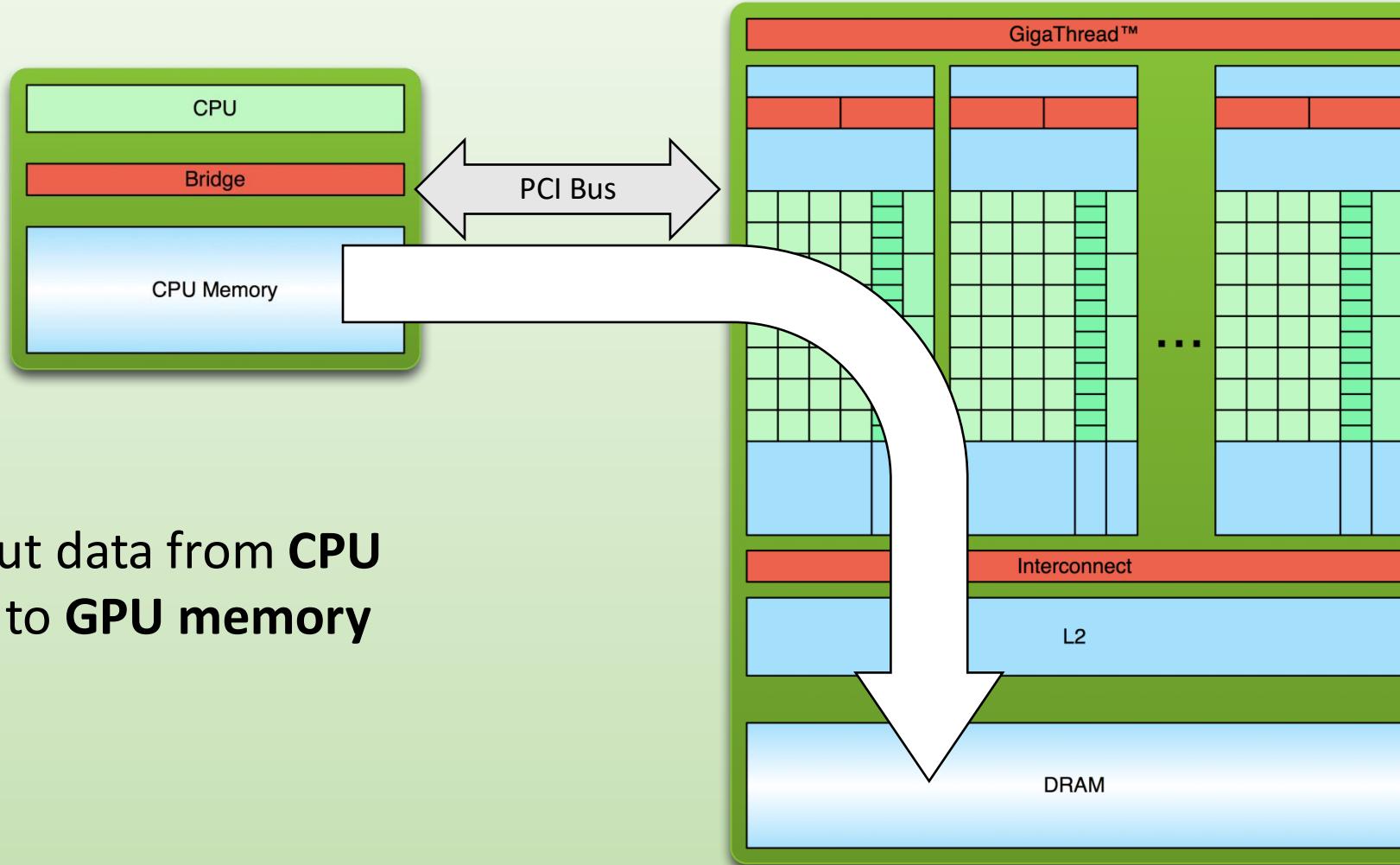
- ✓ Pensare in parallelo significa avere chiaro quali **feature** la **GPU** espone al programmatore
 - Conoscere l'**architettura della GPU** per scalare su migliaia di thread come fosse uno
 - gestione basso livello **cache** permette di sfruttare principio di località
 - Conoscere lo **scheduling** di blocchi di thread e la gerarchia di thread e di memoria (ridurre latenze)
 - Fare impiego diretto della **shared memory** (riduce latenze come le cache)
 - Gestire direttamente le **sincronizzazioni** (barriere tra thread)
- ✓ Si lavora come su **pthread** o **OpenMP**, tecniche di supporto alla programmazione parallela su CPU
- ✓ Si scrive una porzione di **CUDA C** (semplice estensione di C) per l'esecuzione sequenziale e lo si estende a migliaia di thread (permette di pensare 'ancora' in sequenziale)

Elementi chiave di CUDA

- ✓ **Controllo:** gestione dei thread e della memoria dati è nelle mani del programmatore
- ✓ **Kernel:** programma sequenziale eseguito dalla GPU
- ✓ **Host:** opera indipendentemente dal device (per molta parte delle op.)
- ✓ **Host code:** programma in ANSI C
- ✓ **Device Code:** programma in CUDA C
- ✓ **Asincrone:** computazioni (kernel) GPU e CPU
- ✓ **Sincrone:** trasferimenti tra memorie CPU e GPU
- ✓ **Compilatore:** nvcc NVIDIA genera codice eseguibile per host e device (fat-binary)

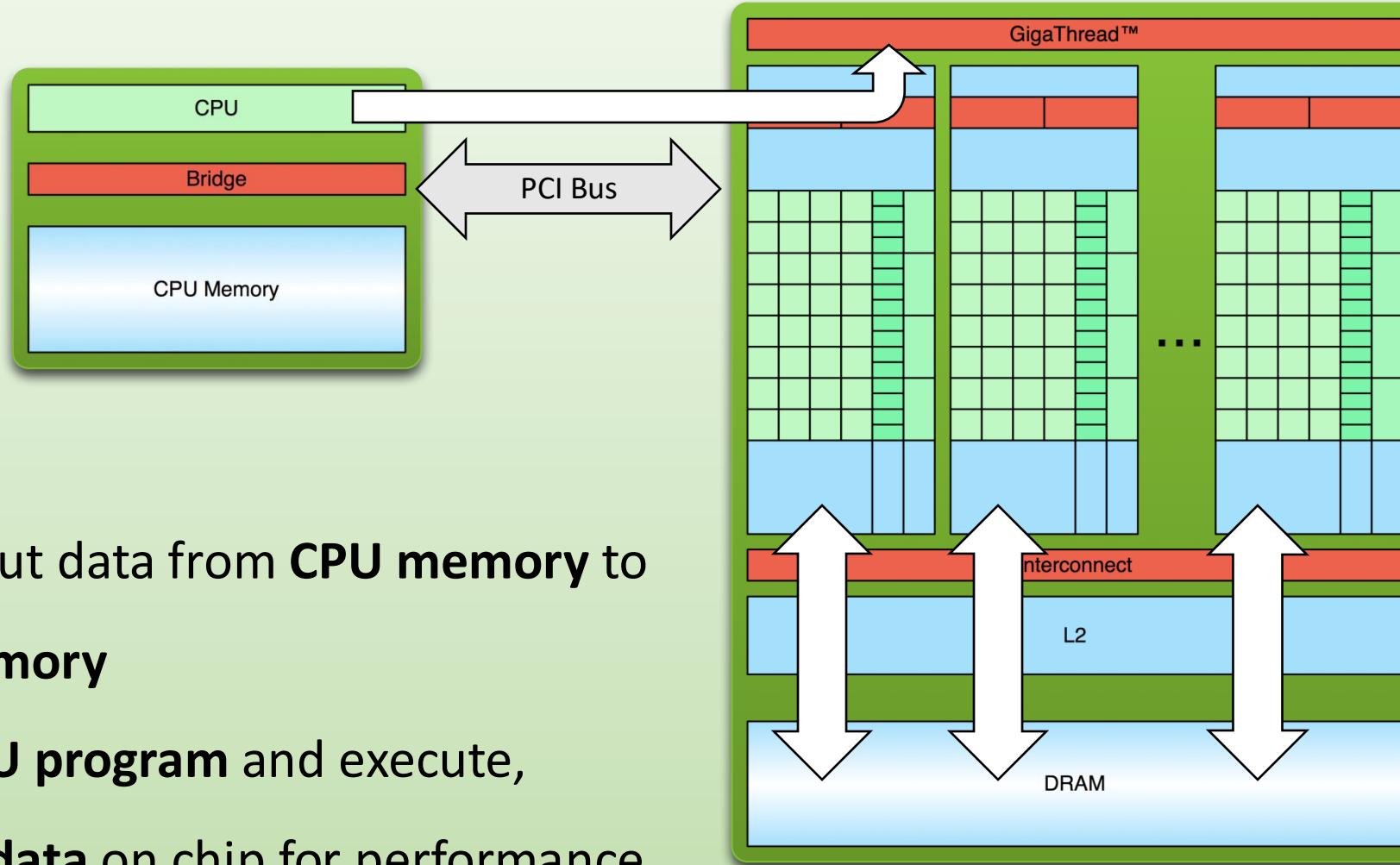


Simple Processing Flow

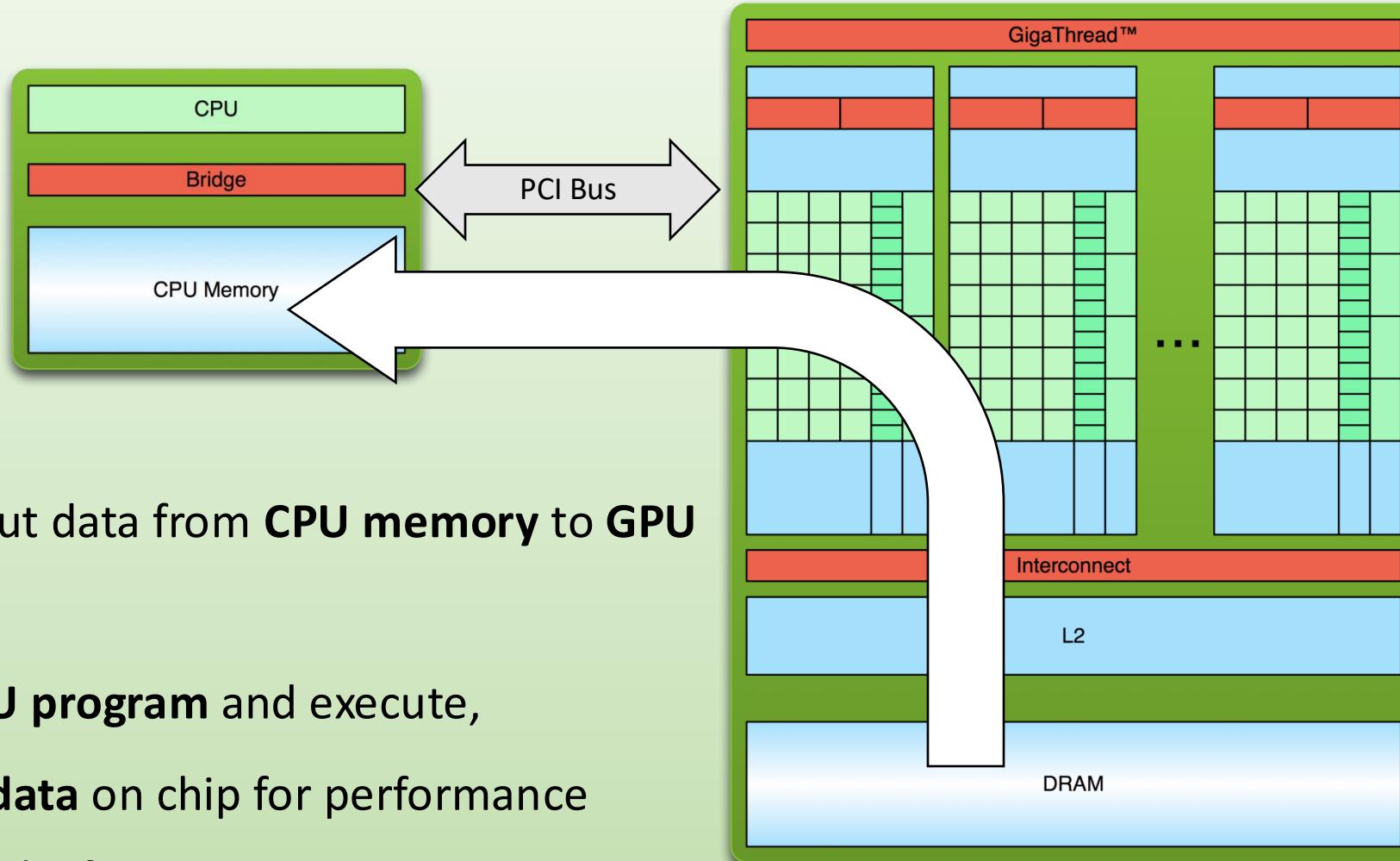


1. Copy input data from **CPU memory** to **GPU memory**

Simple Processing Flow



Simple Processing Flow



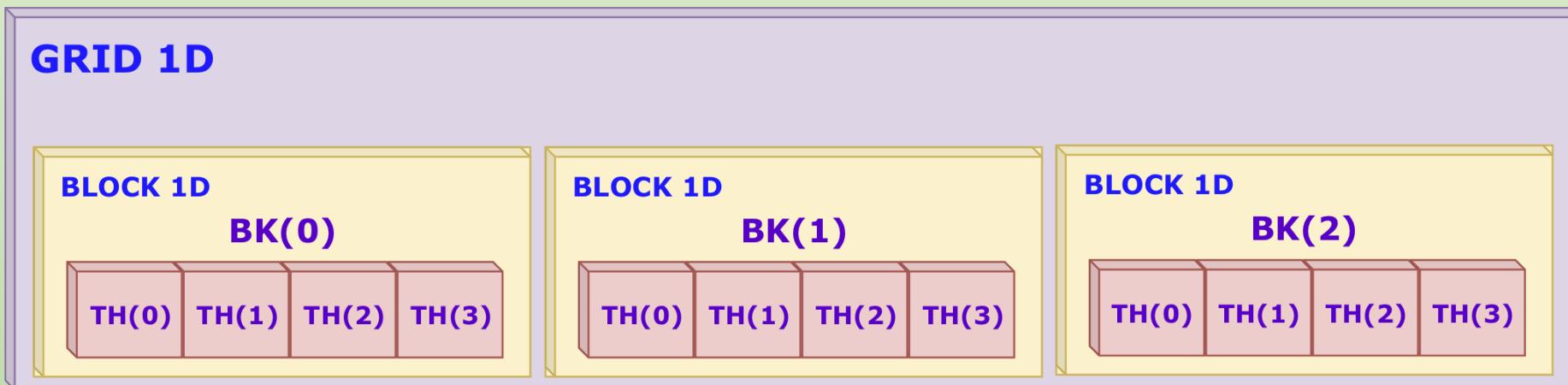
La programmazione in CUDA

La “Formula” di base

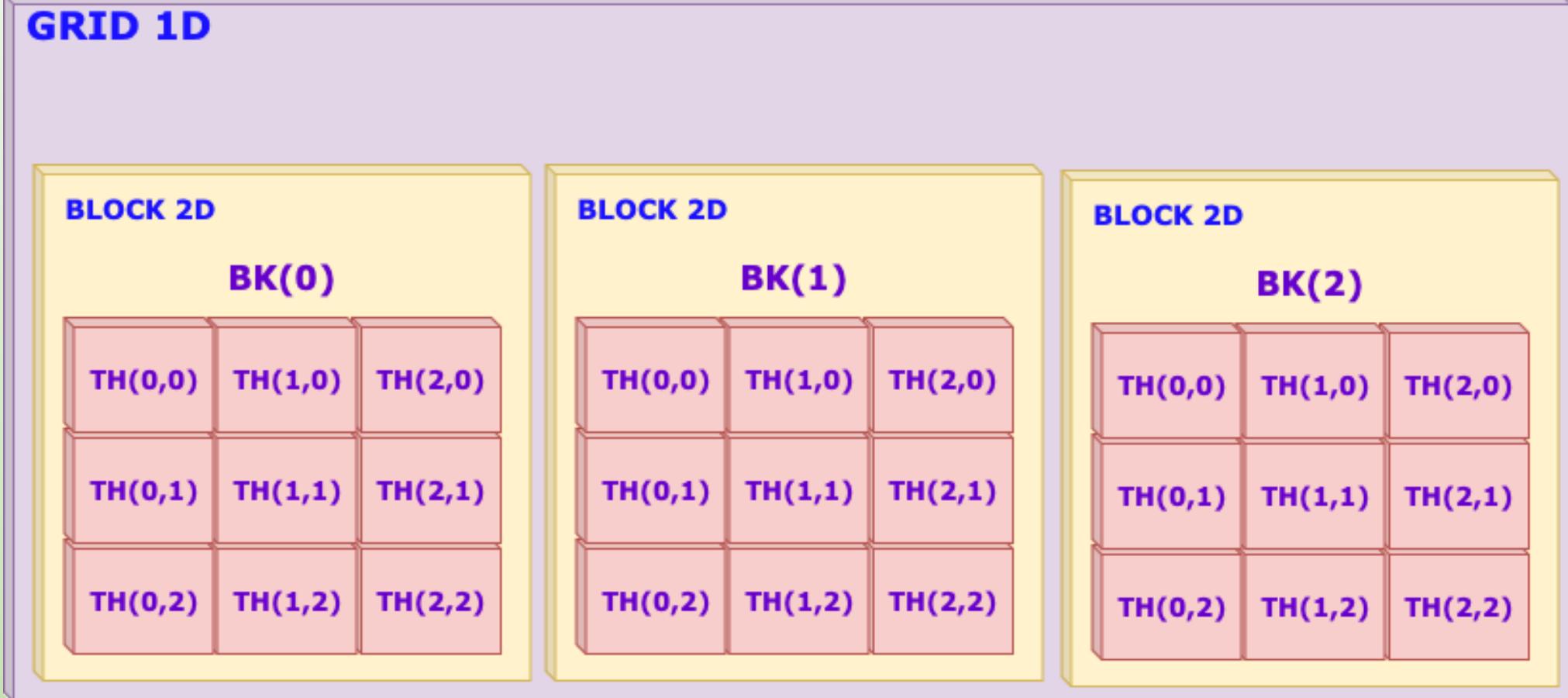
1. Setup dei dati su host (CPU-accessible memory)
2. Alloca memoria per i dati sulla GPU
3. Copia i dati da host a GPU
4. Alloca memoria per output su host
5. Alloca memoria per output su GPU
6. Lancia il kernel su GPU
7. Copia output da GPU a host
8. Cancella le memorie

Organizzazione dei thread

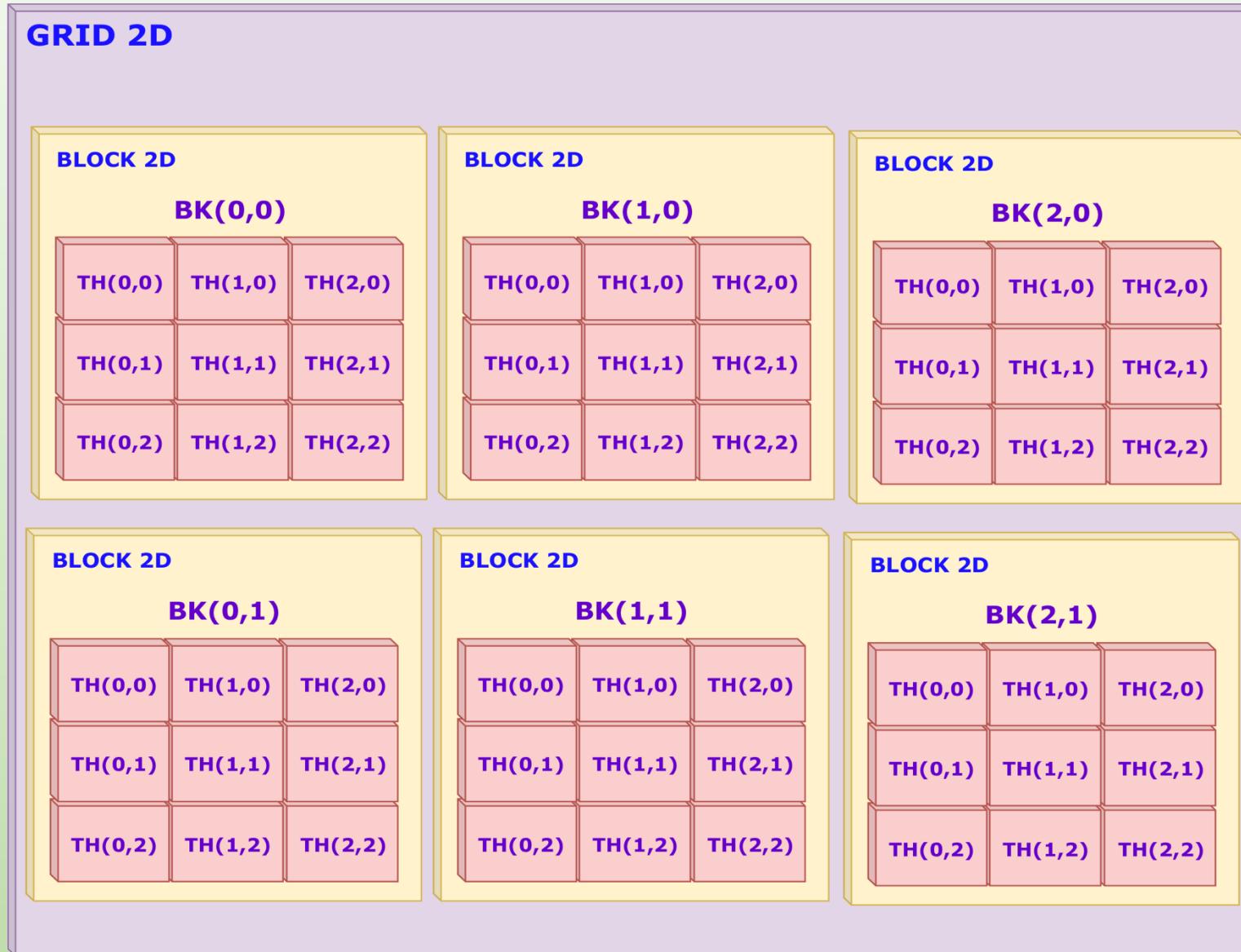
- ✓ CUDA presenta una **gerarchia astratta** di thread strutturata su **due livelli** che si decomponete in
 - **grid**: una griglia ordinata di blocchi
 - **block**: una collezione ordinata di thread
- ✓ Struttura: grid e block possono avere dimensioni
 - **grid**: 1D, 2D e 3D
 - **block**: 1D, 2D e 3D
- ✓ **9 combinazioni** in tutto anche se in genere si usa la stessa per grid e block
- ✓ la scelta delle **dimensioni** è da definire a seconda delle dim della struttura dei dati del **task**



Esempio di grid 1D e block 2D

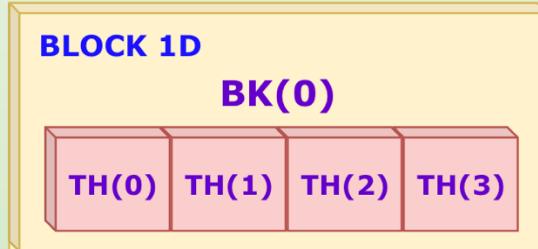


Esempio di grid 2D e block 2D



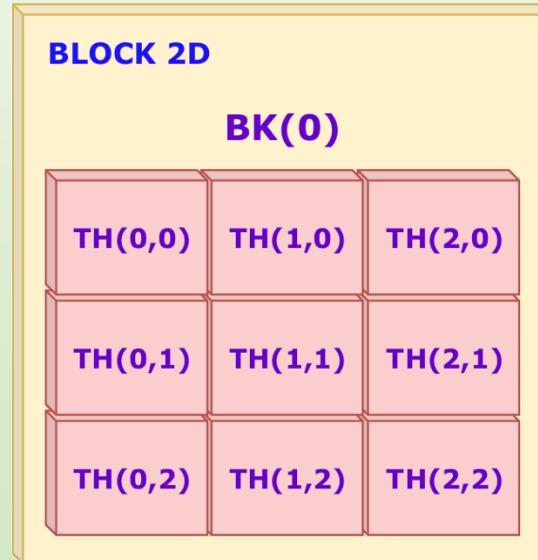
Numero di thread per block

Un **blocco** può contenere **al più 1024 thread!**



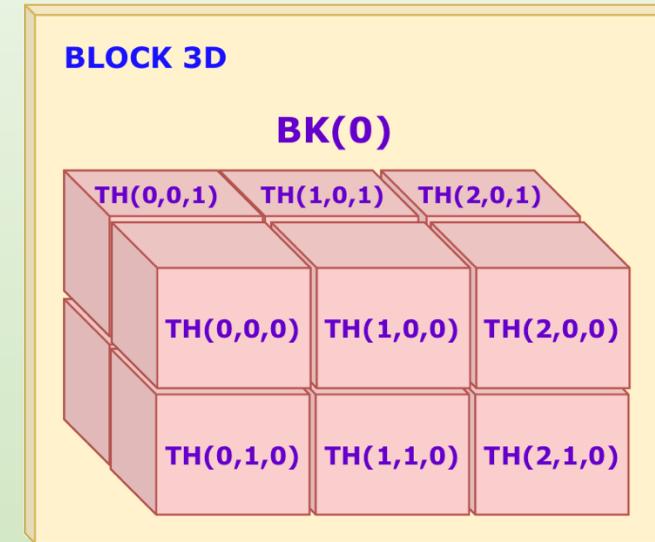
Esempi 1D:

- (32,1,1)
- (96,1,1)
- (512,1,1)
- . . .
- (1024,1,1)
- (2048,1,1) NO



Esempi 2D:

- (16,4,1)
- (128,2,1)
- . . .
- (32,32,1)
- (64,32,1) NO



Esempi 3D:

- (16,4,8)
- (8,8,8)
- . . .
- (64,4,2)
- (32,32,2) NO

Thread block

- ✓ Un blocco di thread è un gruppo di thread che possono **cooperare** tra loro mediante:
 - **Block-local synchronization**
 - **Block-local shared memory**
- ✓ I thread di differenti blocchi **possono** cooperare come **Cooperative Groups** (a partire da CUDA 9.0 e Compute Capability 3.0+)
- ✓ Tutti i thread in una **grid** condividono lo stesso spazio di **global memory**
- ✓ I thread vengono identificati **univocamente** dalle seguenti due coordinate:
 - **blockIdx** (indice di blocco nella **grid**)
 - **threadIdx** (indice di thread nel **blocco**)

Indici di blocchi e thread

- ✓ Gli indici di **grid** e **block** sono specificati dalle seguenti variabili **built-in**:
 - **blockIdx** (indice del blocco nella griglia)
 - **threadIdx** (indice di thread nel blocco)
- ✓ Le coordinate sono di tipo **uint3** pre-inizializzate e possono essere accedute all'interno del kernel
- ✓ Quando un kernel viene eseguito, **blockIdx** e **threadIdx** vengono assegnate a ogni thread da **CUDA runtime**
- ✓ Ogni componente in una variabile di tipo **uint3** è accessibile attraverso i campi **x**, **y**, **z**

Indici di
blocco

blockIdx.x
blockIdx.y
blockIdx.z

Indici di
thread

threadIdx.x
threadIdx.y
threadIdx.z

Dimensioni di blocchi e thread

- ✓ Le dimensioni di **grid** e **block** sono specificate dalle seguenti variabili **built-in**:
 - **blockDim** (dimensione di blocco, misurata in thread)
 - **gridDim** (dimensione della griglia, misurata in blocchi)
- ✓ Queste variabili sono di tipo **dim3**, un tipo di vettore di interi basato su **uint3**
- ✓ Quando si definisce una variabile di tipo **dim3**, ogni componente non specificata è inizializzata a 1
- ✓ Ogni componente in una variabile di tipo **dim3** è accessibile attraverso i campi **x**, **y**, **z**

dimensioni
di blocco

blockDim.x
blockDim.y
blockDim.z

dimensioni
di griglia

gridDim.x
gridDim.y
gridDim.z

Dim3

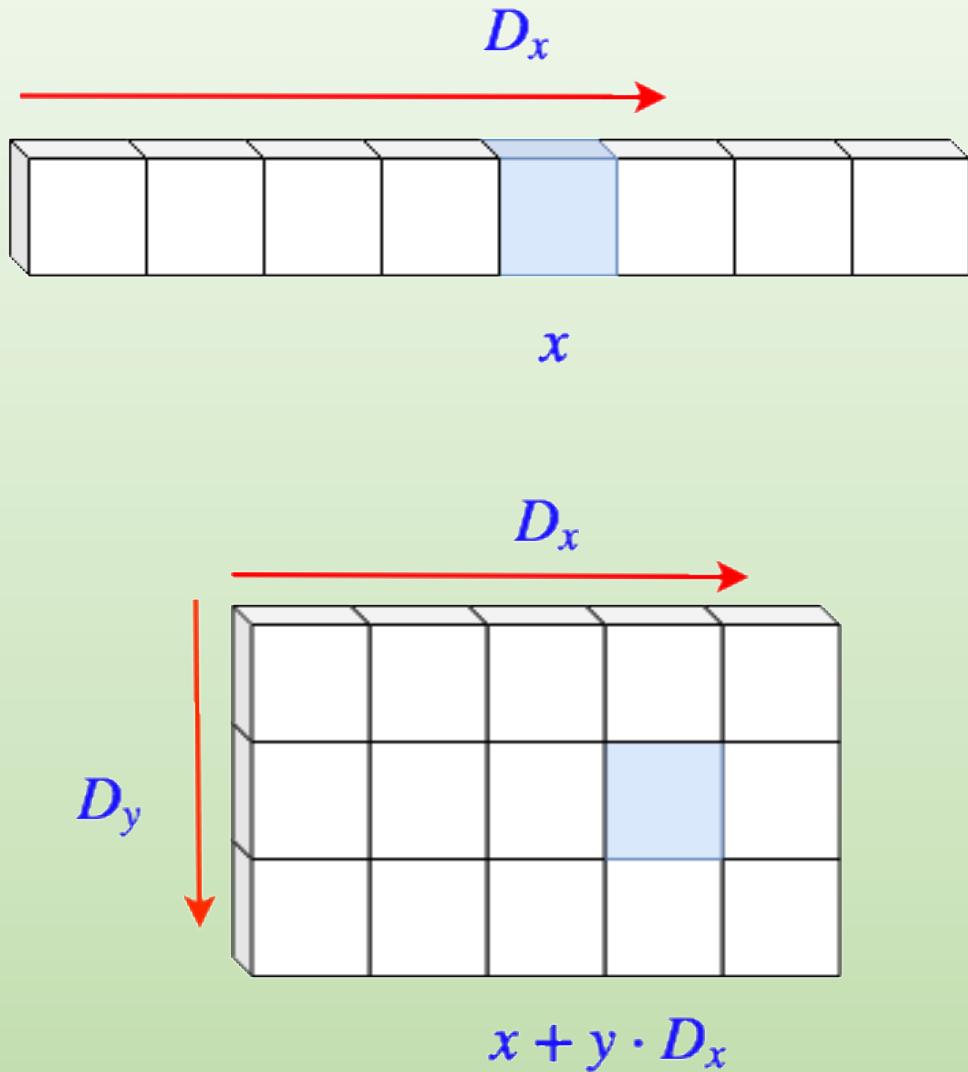
dim3 is a struct (defined in **vector_types.h**) to define your Grid and Block dimensions

```
struct __device_builtin__ dim3
{
    unsigned int x, y, z;
#ifndef __cplusplus
    __host__ __device__ dim3(unsigned int vx = 1, unsigned int vy = 1, unsigned int vz = 1) : x(vx), y(vy), z(vz) {}
    __host__ __device__ dim3(uint3 v) : x(v.x), y(v.y), z(v.z) {}
    __host__ __device__ operator uint3(void) { uint3 t; t.x = x; t.y = y; t.z = z; return t; }
#endif /* __cplusplus */
};
```

Works for dimensions 1, 2, and 3:

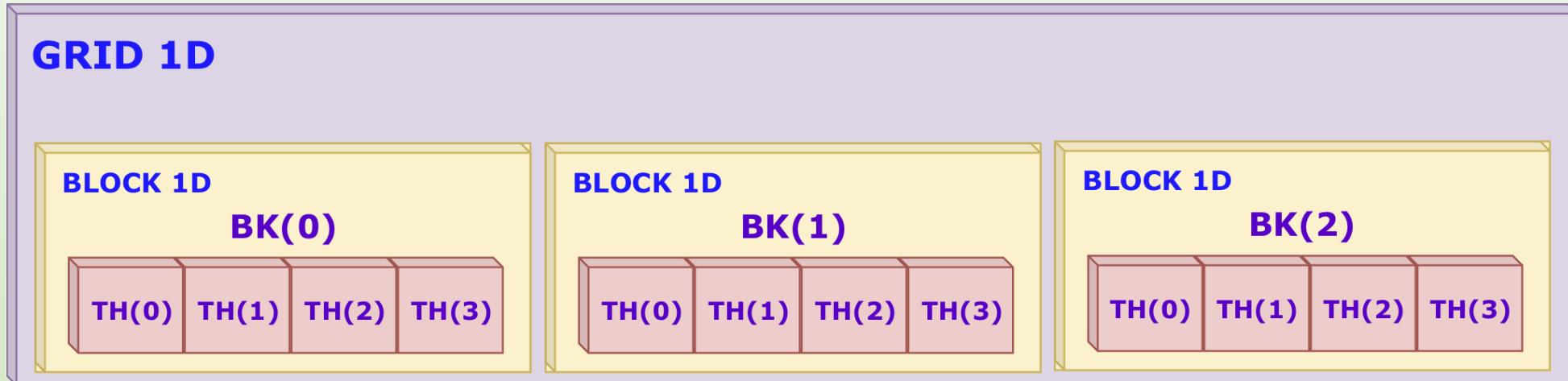
- **dim3 grid(256);** // defines a grid of 256 x 1 x 1 blocks
- **dim3 block(512, 512);** // defines a block of 512 x 512 x 1 threads
- **foo<<<grid, block>>>(...);**

Indice unico nei blocchi 1D, 2D, 3D



$$x + y \cdot D_x + z \cdot D_x \cdot D_y$$

ID unico di thread: grid1D – block1D



$$ID_{th} = blockIdx.x * blockDim.x + threadIdx.x$$

Grid 1D - coordinate 1D

4 blocchi con 4
thread ciascuno



Calcolo
dell'indice
lineare

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

Somma di vettori: kernel

Block 1D: Indice lineare

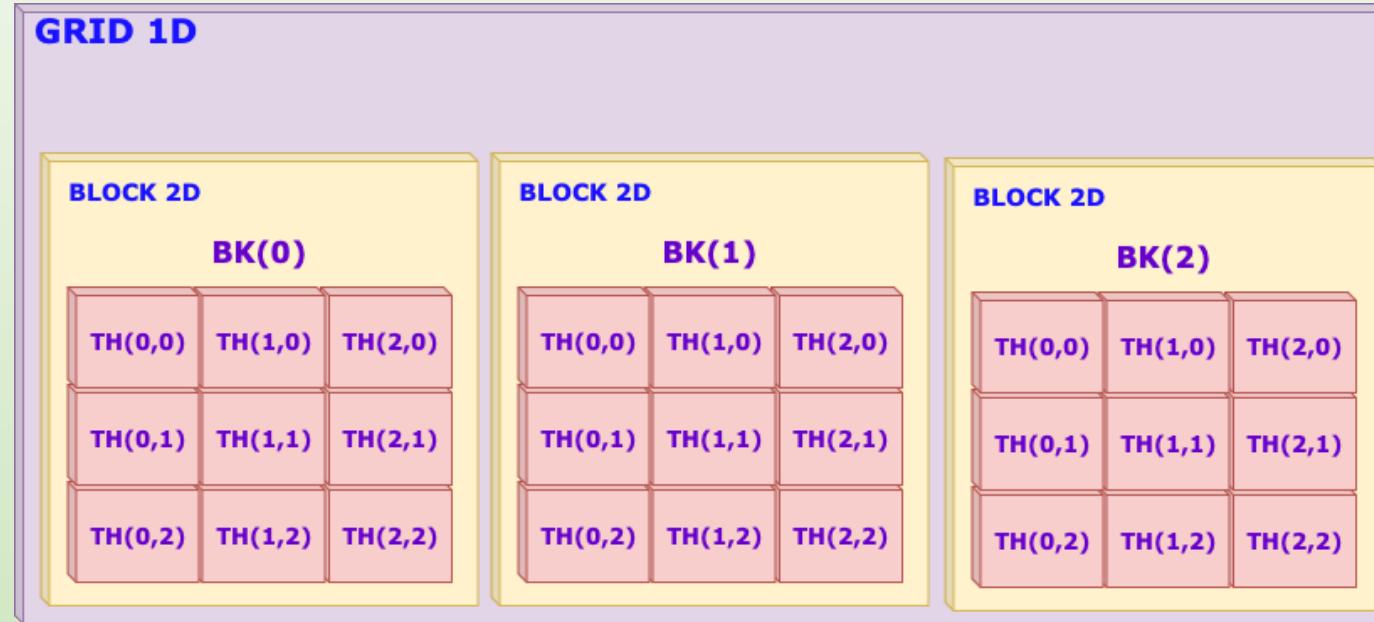
Controllo su thread effettivi
(num di thread potrebbe essere diverso da dimensione dei dati!)

```
#include <stdio.h>
#include <cuda_runtime.h>
#define N 1024*1024 // vector size
#define TxB 32      // threads x block

/* kernel: vector add */
_global_ void add_vect(int *a, int *b, int *c) {
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    if (idx < N)
        c[idx] = a[idx] + b[idx];
}
```

ID unico di thread: grid1D – block2D

grid(3,1)



$$\begin{aligned} ID_{th} = & \text{blockIdx.x} * \text{blockDim.x} * \text{blockDim.y} + \\ & \text{threadIdx.y} * \text{blockDim.x} + \\ & \text{threadIdx.x} \end{aligned}$$

ID di thread unico nella griglia 1D

✓ grid 1D block 1D

```
IDth = blockIdx.x * blockDim.x +  
        threadIdx.x
```

✓ grid 1D block 2D

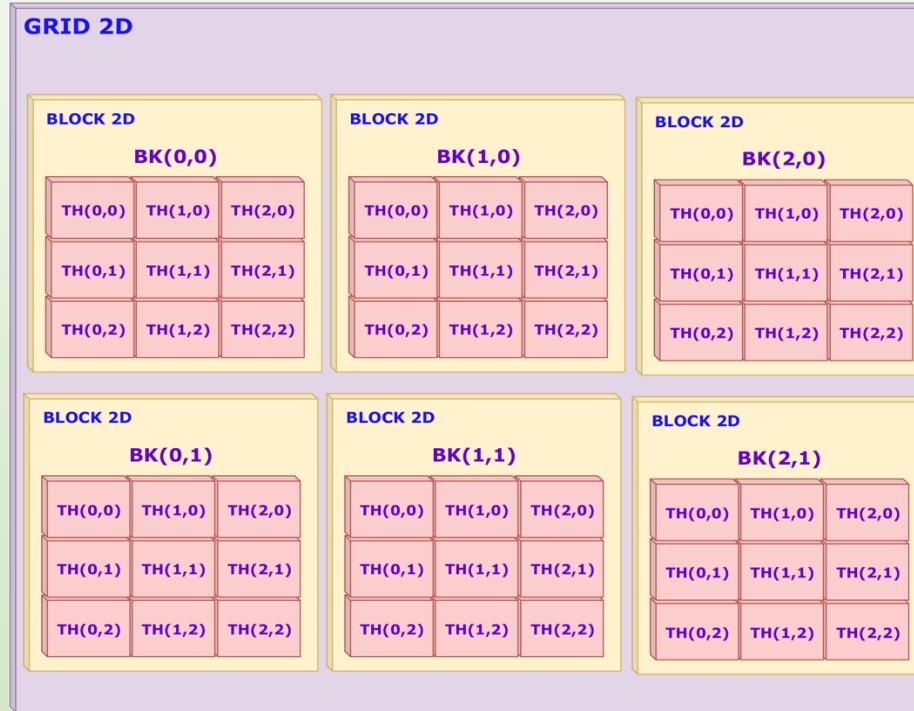
```
IDth = blockIdx.x * blockDim.x * blockDim.y +  
        threadIdx.y * blockDim.x +  
        threadIdx.x
```

✓ grid 1D block 3D

```
IDth = blockIdx.x * blockDim.x * blockDim.y * blockDim.z +  
        threadIdx.z * blockDim.y * blockDim.x +  
        threadIdx.y * blockDim.x +  
        threadIdx.x
```

Esempio di grid 2D e block 2D

grid(3,2)



$$ID_{bk} = blockIdx.y * blockDim.x + blockIdx.x \quad (\text{indice di blocco})$$

$$\begin{aligned} ID_{th} = & ID_{bk} * (\text{blockDim.x} * \text{blockDim.y}) + \\ & threadIdx.y * \text{blockDim.x} + \\ & threadIdx.x \end{aligned}$$

→ lab2

ID di grid e block

I kernel CUDA

Definizione di function parallele

Lancio di un kernel CUDA

- ✓ Un **kernel CUDA** è un diretta estensione della **sintassi** delle funzioni C con in aggiunta i **parametri di configurazione dell'esecuzione** all'interno di **parentesi triplo-angolari**

```
kernel_name <<<grid, block>>>(argument list);
```

- ✓ Il primo valore dei parametri è la **dimensione della grid = numero di blocchi** che racchiudono i thread
- ✓ Il secondo valore è la **dimensione di blocco = numero di thread** all'interno del blocco
- ✓ Per esempio, supponiamo di avere un dato di 32 elementi, possiamo raggruppare 8 elementi in 4 blocchi, oppure 32 in un solo blocco...

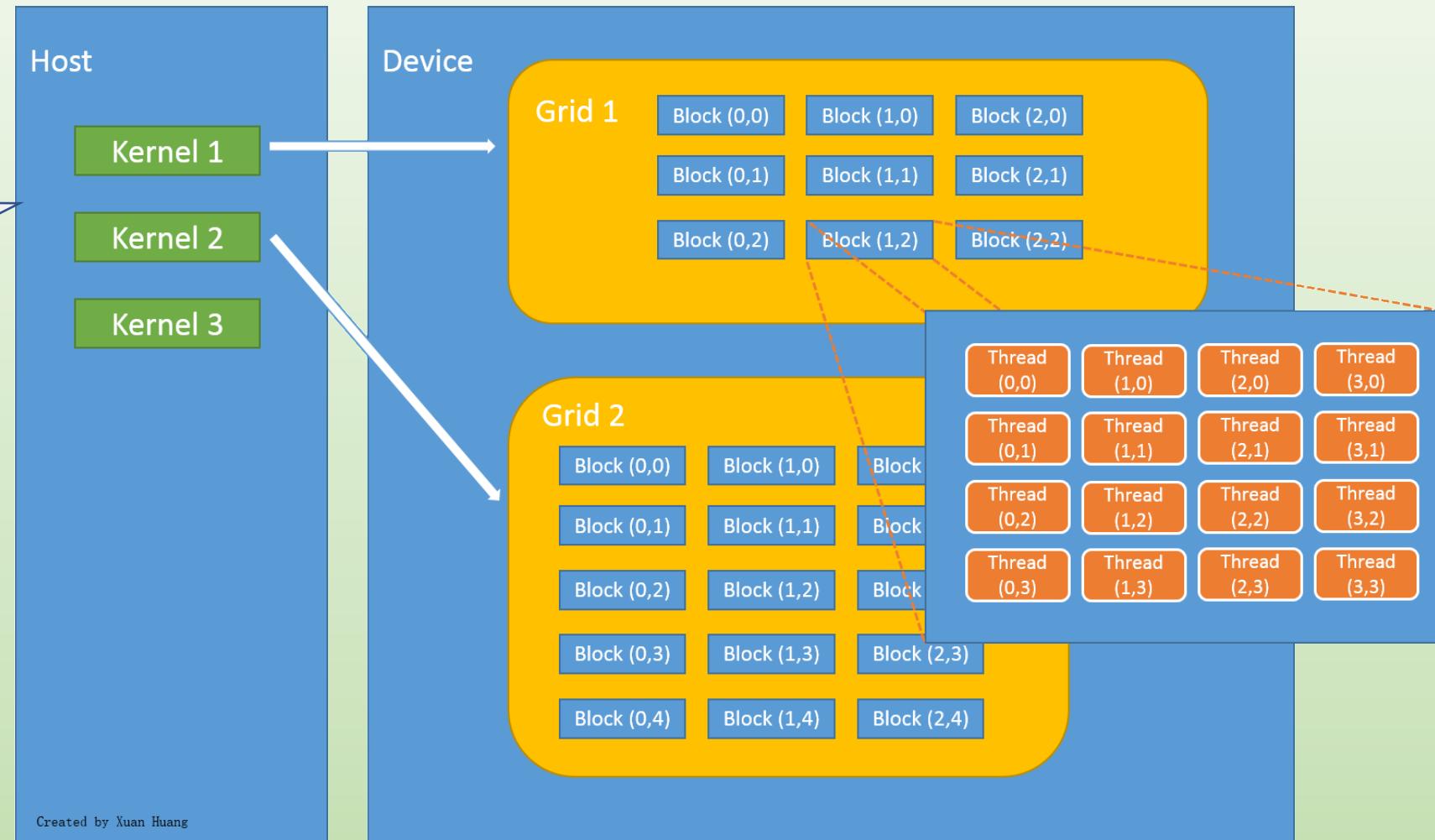
```
kernel_name <<<4, 8>>>(argument list);
```

```
kernel_name <<<1, 32>>>(argument list);
```

Kernel concorrenti

kernel in esecuzione concorrente:
grid 2D e block 2D

- **grid1:**
 - grid (3,3)
 - block (4,4)
- **grid2:**
 - grid (3,5)
 - block (4,4)



Created by Xuan Huang

Runtime API `cudaDeviceReset`

```
__host__ cudaError_t cudaDeviceReset ( void )
```

Destroy all allocations and reset all state on the current device in the current process.

Returns

[cudaSuccess](#)

Description

Explicitly destroys and cleans up all resources associated with the current device in the current process. Any subsequent API call to this device will reinitialize the device.

Note that this function will reset the device immediately. It is the caller's responsibility to ensure that the device is not being accessed by any other host threads from the process when this function is called.

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

[cudaDeviceSynchronize](#)

Runtime API `cudaDeviceSynchronize`

```
__host__ __device__ cudaError_t cudaDeviceSynchronize ( void )
```

Wait for compute device to finish.

Returns

[cudaSuccess](#)

Description

Blocks until the device has completed all preceding requested tasks. [cudaDeviceSynchronize\(\)](#) returns an error if one of the preceding tasks has failed. If the [cudaDeviceScheduleBlockingSync](#) flag was set for this device, the host thread will block until the device has finished its work.

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

[cudaDeviceReset](#), [cuCtxSynchronize](#)

Verifica del codice (debugging by printf)

- ✓ Oltre ad usare tool di debugging avanzati si può ricorrere a due mezzi molto semplici e spesso altrettanto efficaci per verificare correttezza del codice:
 - **Printf**: disponibile nei device a partire dall'architettura Fermi in poi
 - **Kernel<<<1,1>>>**: forzare il kernel ad eseguire con un solo blocco e un solo thread.... cosa che emula il comportamento dell'esecuzione sequenziale su un singolo dato
- ✓ OSS. È utile usare `cudaDeviceSynchronize` o `cudaDeviceReset` per avere la **print** corretta su **std output**:
 - «*Kernel calls are asynchronous, which means control is returned to the calling (CPU) thread before the kernel completes. If the application terminates shortly thereafter, then the kernel will have no opportunity to print*» by NVIDIA

Proprietà del kernel

QUALIFICATORI	ESECUZIONE	CHIAMATA	NOTE
<code>_global_</code>	Eseguito dal device	Dall'host e dalla compute cap. 3 anche dal device	Restituisce un tipo void
<code>_device_</code>	Eseguito dal device	Solo dal device	
<code>_host_</code>	Eseguito dall'host	Solo dall'host	Può essere omesso

✓ **Nota:** I qualificatori `_device_` e `_host_` possono essere **usati insieme** e in questo caso la funzione viene compilata sia per **host** che per **device**

Restrizioni sul kernel

- ✓ Accede alla sola **memoria device**
- ✓ Deve restituire un tipo **void**
- ✓ Non supporta il **numero variabile di argomenti**
- ✓ Non supporta **variabili statiche**
- ✓ Non supporta **puntatori a funzioni**
- ✓ Esibisce un **comportamento asincrono** rispetto all'host

Gestione errori

Segnatura

`cudaError_t`

- Ogni chiamata CUDA (eccetto il kernel) restituisce un tipo enumerativo `cudaError_t`

Return ex.

`success`

`cudaErrorMemoryAllocation`

Segnatura

`cudaError_t cudaGetLastError(void)`

- Restituisce il codice dell'ultimo errore

Segnatura

`char *cudaGetLastError(cudaError_t)`

- Restituisce il codice dell'ultimo errore
- `printf("%s\n", cudaGetStringError(cudaGetLastError()));`

Gestione errori

- ✓ Poiché molte **chiamate** CUDA sono **asincrone** è spesso molto difficile identificare quale function o routine che causa un errore
- ✓ Definire una **macro** per la gestione errori per effettuare **wrap** delle chiamate CUDA API semplifica il processo di individuazione degli errori:

```
#define CHECK(call) {  
    const cudaError_t error = call;  
    if (error != cudaSuccess) {  
        printf("Error: %s:%d, ", __FILE__, __LINE__);  
        printf("code:%d, reason: %s\n", error, cudaGetErrorString(error));  
        exit(1);  
    }  
}
```

- ✓ Usare poi il seguente codice:
`CHECK(cudaMemcpy(d_C, gpuRef, nBytes, cudaMemcpyHostToDevice));`
- ✓ Per motivi di debugging di kernel si può usare il codice:
`kernel_function<<<grid, block>>>(argument list);
CHECK(cudaDeviceSynchronize());`

Puntatori su host e device

- ✓ Un errore comune è quello di confondere i **puntatori** dei due diversi **spazi di memoria** CPU e GPU
- ✓ Un puntatore alla memoria device non può essere **dereferenziato** nel codice host e viceversa
- ✓ Per es., un assegnamento improprio sarebbe:
 - **gpuRef = d_C // ERROR!**
- ✓ Invece di usare:
 - **cudaMemcpy(gpuRef, d_C, nBytes, cudaMemcpyDeviceToHost)**
- ✓ Questi genere di errori non occorrono nella **Unified Memory** (introdotta in CUDA 6) che consente di accedere alle memorie CPU e GPU usando un singolo puntatore

Misurare il tempo con la CPU

- ✓ Chiamata di sistema `gettimeofday` (includere l'header `sys/time.h`)
- ✓ Tempo di clock complessivo (numero di sec da una certa epoca)

```
double cpuSecond() {  
    struct timeval tp;  
    gettimeofday(&tp, NULL);  
    return ((double)tp.tv_sec + (double)tp.tv_usec*1.e-6);  
}  
  
// misura del tempo di esecuzione del kernel  
double iStart = cpuSecond();  
kernel_name<<<grid, block>>>(argument list);  
cudaDeviceSynchronize();  
double iElaps = cpuSecond() - iStart;
```

La CPU aspetta fino a che tutti i thread abbiano terminato

Vantaggi e svantaggi dei thread

Vantaggi:

- ✓ Visibilità dei dati globali: condivisione di oggetti semplificata
- ✓ Più flussi di esecuzione
- ✓ **Gestione semplice di eventi asincroni** (I/O per esempio)... comunicazioni veloci
- ✓ Tutti i thread di un processo condividono lo **stesso spazio di indirizzamento**, quindi le comunicazioni tra thread sono più semplici delle comunicazioni tra processi
- ✓ **Context switch veloce**: nel passaggio da un thread ad un altro di uno stesso processo viene mantenuto buona parte dell'ambiente

Svantaggi:

- ✓ **Concorrenza** invece di parallelismo: gestire la mutua esclusione
- ✓ **Difficoltà di ottenere risorse private**

Esempio: flipping di un'immagine



flip orizzontale



flip verticale

→ lab2

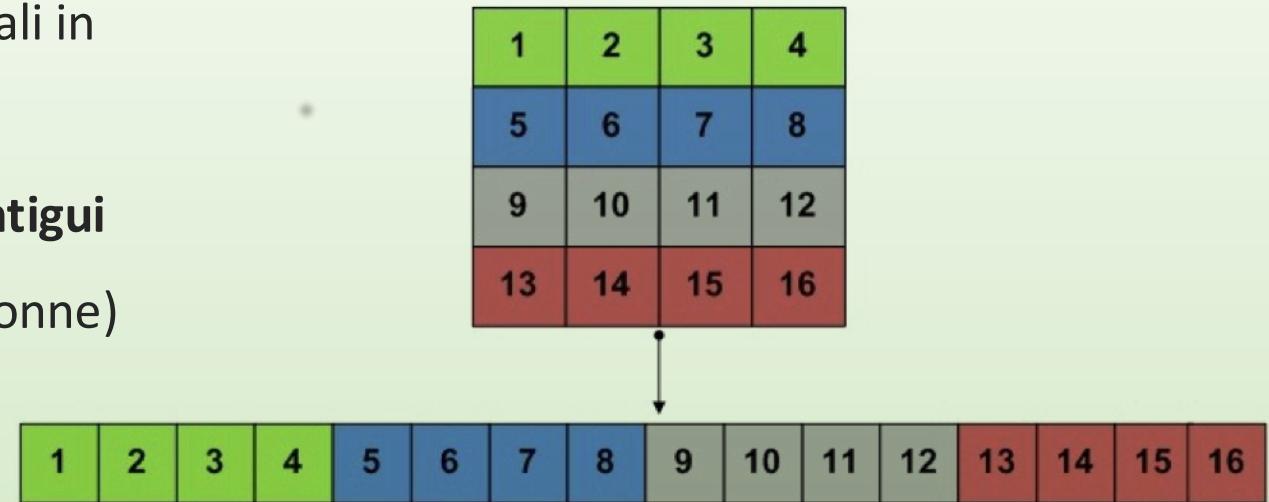
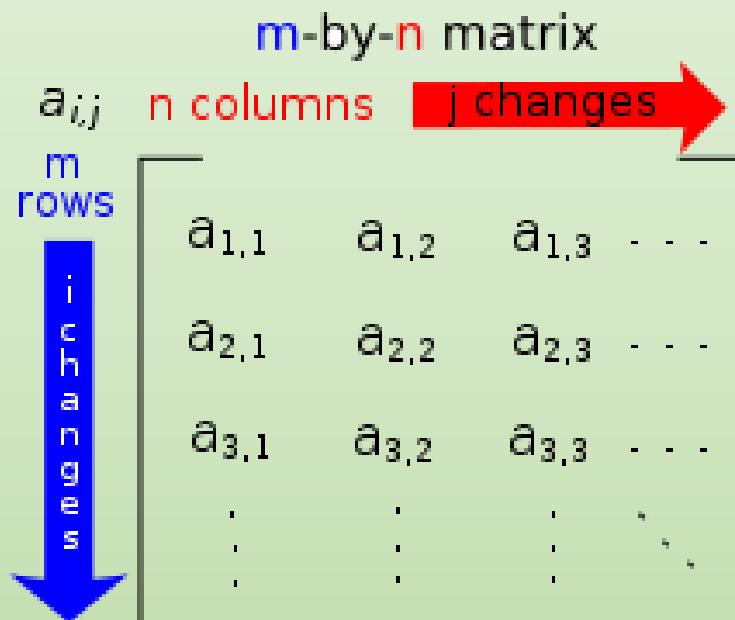
Flip di un'immagine con grid 1D

Array multidimensionali in C

Esercitazione su prodotti di matrici “linearizzate”

Allocazione dinamica della memoria

- ✓ C organizza i dati di array multidimensionali in **row-major order** ("linearizzati")
- ✓ Elementi **consecutivi** delle **righe** sono **contigui**
- ✓ Esempio: matrice **$m \times n$** (**m** righe e **n** colonne)



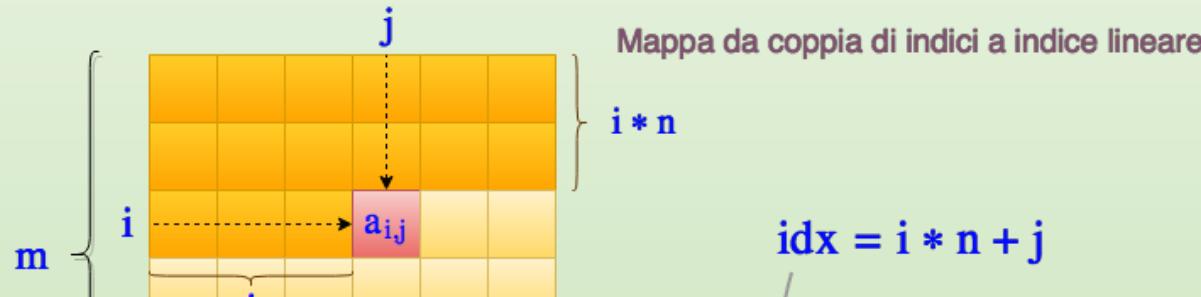
```
// azzeramento della matrice generata dinamicamente
A = (int *) malloc(m * n * sizeof(int));

for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
        // calcola idx (indice linearizzato)
        A[idx] = 0;
    }
}
```

Allocazione dinamica e indicizzazione

- ✓ Allocazione fisica di dati in memoria e accesso logico alle strutture dati in C

Matrice: organizzazione logica

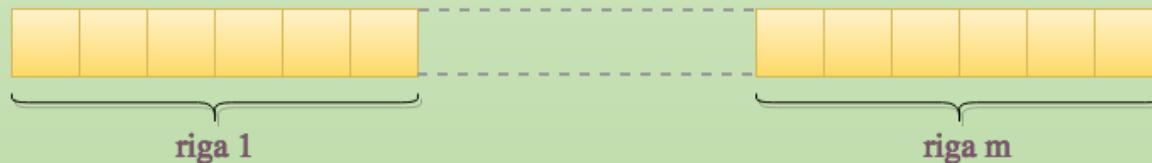


Mappa da coppia di indici a indice lineare

$$i * n$$

$$\text{idx} = i * n + j$$

Matrice: organizzazione "linearizzata" in memoria fisica



- ✓ Codice C per l'azzeramento di dati in una matrice di m righe e n colonne con accesso tramite indice linearizzato

```
// azzeramento della matrice generata
// dinamicamente

A = (int *) malloc(n * m * sizeof(int));

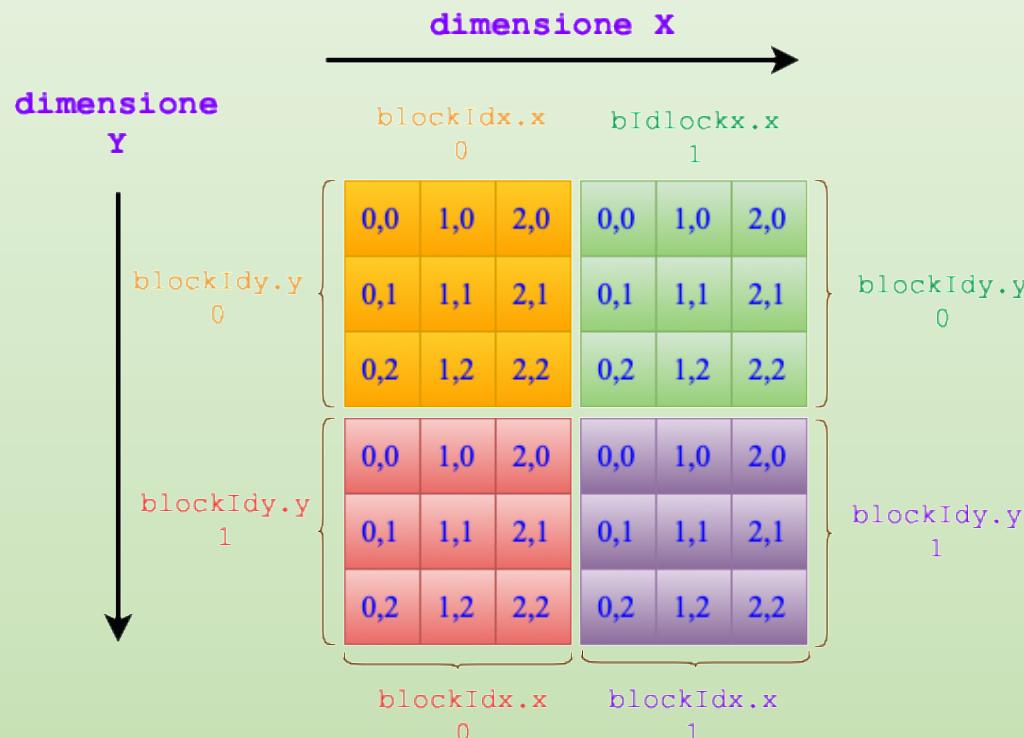
for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
        idx = i * n + j; // indice linearizzato
        A[idx] = func(i,j); // funz di i e j
    }
}
```

Grid 2D - coordinate 2D

grid 2 x 2
block 3 x 3

gridDim = (2,2,1)
blockDim = (3,3,1)

threadIdx.x = 0,1,2
threadIdx.y = 0,1,2



dimensione X = blockDim.x * blockIdx.x + threadIdx.x											
dimensione Y = blockDim.y * blockIdx.y + threadIdx.y											
0,0	1,0	2,0	0,0	1,0	2,0	0,2	1,2	2,2	0,2	1,2	2,2
X=0	X=2	X=4	X=0	X=2	X=4	X=2	X=4	X=6	X=0	X=2	X=4
Y=0	Y=0	Y=0	Y=0	Y=0	Y=0	Y=2	Y=2	Y=2	Y=3	Y=3	Y=3
0,0	1,0	2,0	0,0	1,0	2,0	0,2	1,2	2,2	0,2	1,2	2,2
X=0	X=2	X=4	X=0	X=2	X=4	X=2	X=4	X=6	X=3	X=5	X=7
Y=3	Y=3	Y=3	Y=3	Y=3	Y=3	Y=5	Y=5	Y=5	Y=5	Y=5	Y=5

Lavorare con le matrici

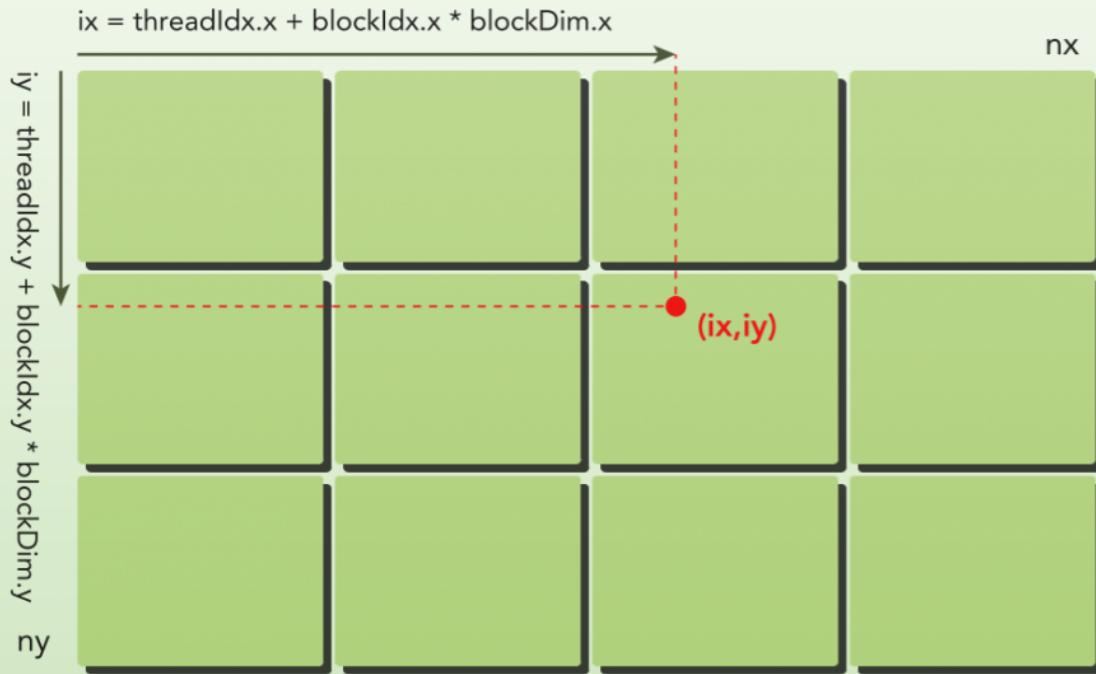
- ✓ Nel caso 2D ci sono 3 tipi di indici
 - **indici** di blocchi e di thread
 - indice dell'**elemento** della matrice
 - indice (**offset**) della memoria lineare globale
- ✓ Il primo passaggio è dalle **coordinate 2D** del **thread** a quelle dell'elemento **(ix, iy)** della matrice

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

```
int iy = blockDim.y * blockIdx.y + threadIdx.y;
```

- ✓ Il secondo passaggio è **mappare** le coordinate della matrice sulla **memoria lineare** globale

```
idx = iy * nx + ix
```



- ✓ Spesso è necessario un **controllo** quando la dim. di **griglia** non collima con quella della **matrice**:

```
if (ix < nx) ...  
if (iy < ny) ...
```

Somma di matrici: kernel

Block 2D, grid 2D

Indice lineare
elementi delle
matrici

Controllo su
thread effettivi

```
/* kernel: matrix add */

_global_ void add_matrix(int *A, int *B, int *C, int nx, int ny) {

    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    int idy = blockDim.y * blockIdx.y + threadIdx.y;

    int id_elem = idy * nx + idx

    if (idx < nx & idy < ny)

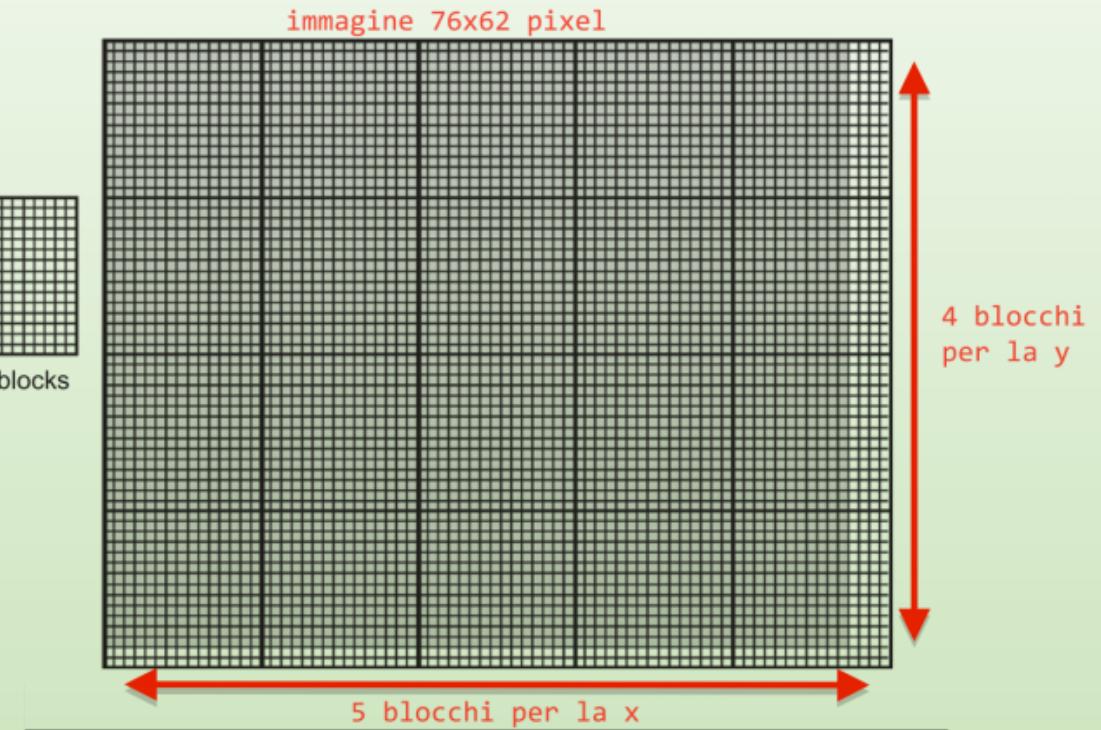
        C[id_elem] = A[id_elem] + B[id_elem];

}
```

Dimensioni di blocchi e matrici

- ✓ Se si deve trattare una matrice che rappresenta l'immagine in figura, si possono organizzare blocchi di dimensione fissata (qui 16×16) e calcolare la griglia di conseguenza (qui 5×4). Il tutto somma a 5120 thread

```
dim3 dimBlock(16,16,1)  
dim3 dimGrid(5,4,1)  
genera 80x64 thread!!
```

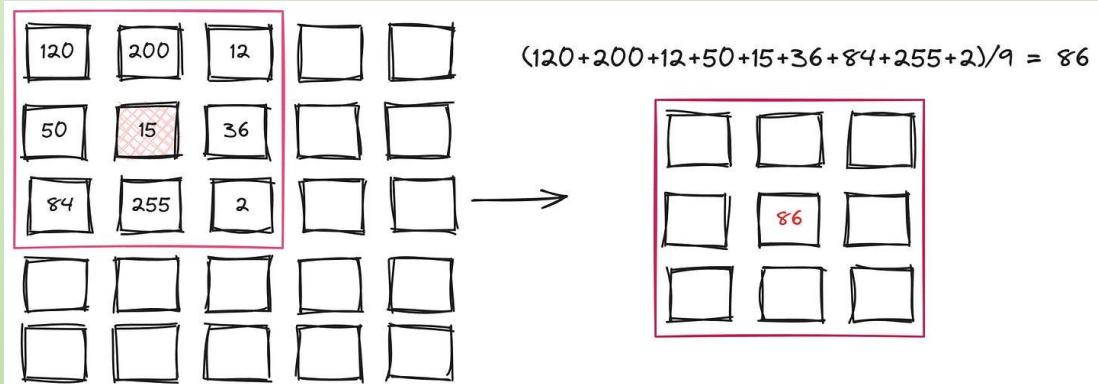
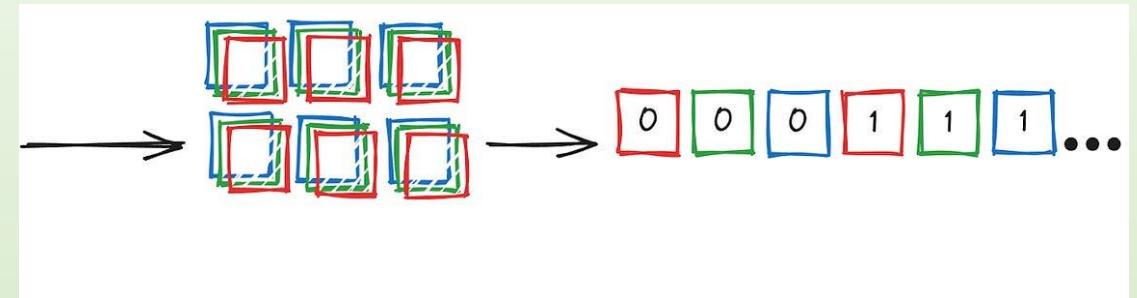


- ✓ Siccome l'immagine è 76×62 pixels (quindi 4712 pixel complessivi), occorre mettere in atto controlli tra indici generati dai thread e quelli effettivi della matrice dell'immagine data

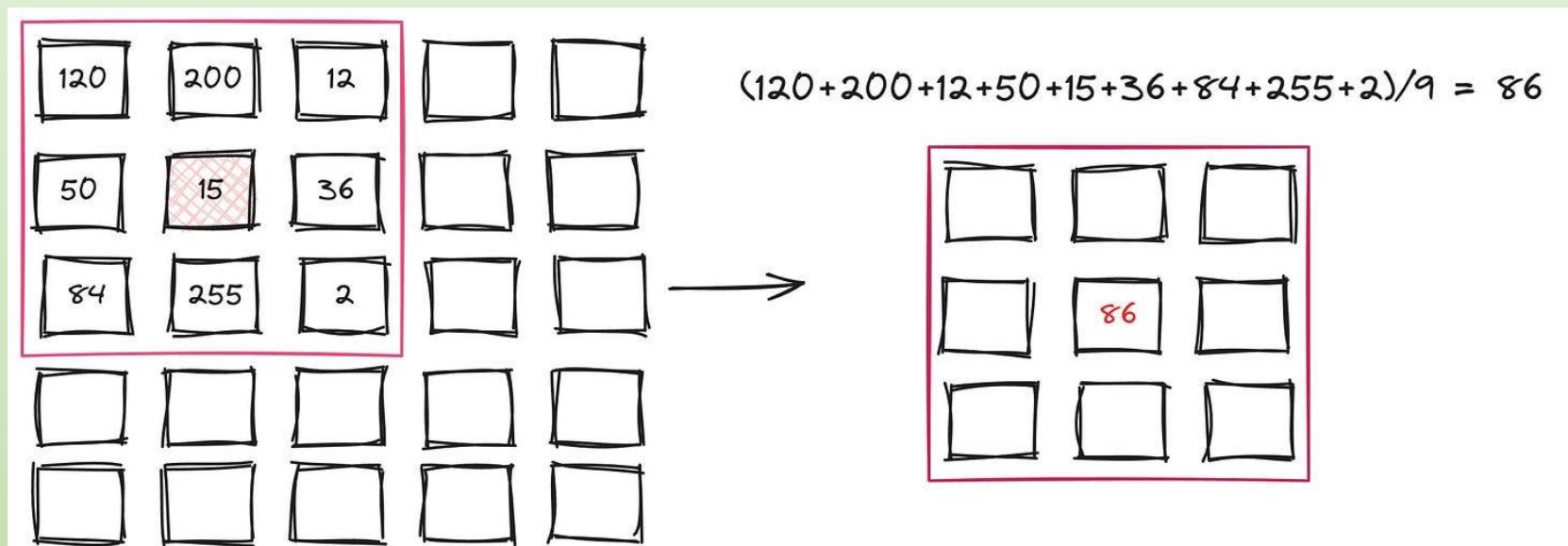
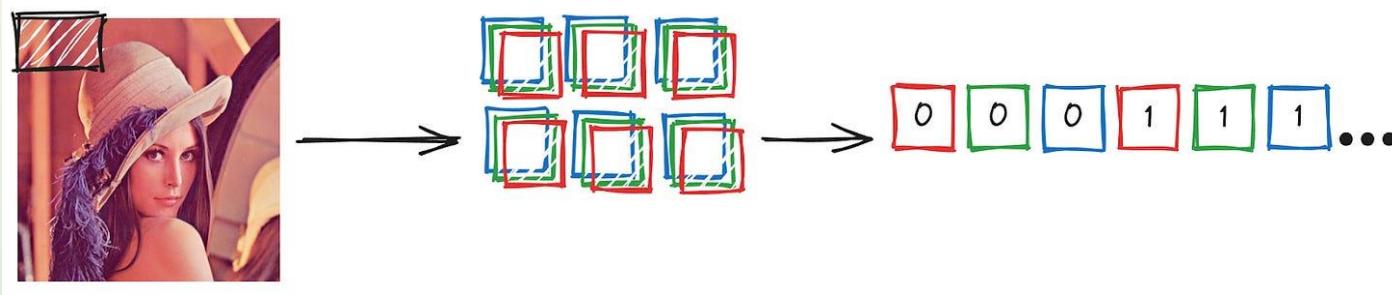
Blurring di un'immagine

Multithreading su pixel

Filtro di blurring



Blurring di immagini



→ lab2

Blurring di un'immagine con grid 2D

Riferimenti bibliografici

Testi generali:

1. J. Cheng, M. Grossman, T. Mckercher, [Professional Cuda C Programming](#), Wrox Pr Inc. (1[^] ed), 2014 (cap 2)
(free available <https://www.cs.utexas.edu/~rossbach/cs380p/papers/cuda-programming.pdf>)
2. David B. Kirk, Wen-Mei W Hwu, Programming Massively Parallel Processors - A Hands-On Approach. Morgan Kaufmann, 2017 (cap 2).

NVIDIA docs:

1. CUDA Programming: <http://docs.nvidia.com/cuda/cuda-c-programming-guide/>
2. CUDA C Best Practices Guide: <http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/>