

GPU Computing

Laurea Magistrale in Informatica - AA 2019/20

Docente **G. Grossi**

Lezione 2 – Il modello di programmazione CUDA

Sommario

- ✓ Modelli di sistemi di computazione paralleli (richiami)
- ✓ CPU multi-core e programmazione multi-threading
- ✓ Modello di programmazione CUDA per GPU
- ✓ Allocazione e uso della memoria in CUDA
- ✓ Flipping di un'immagine (CPU e GPU)

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 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)

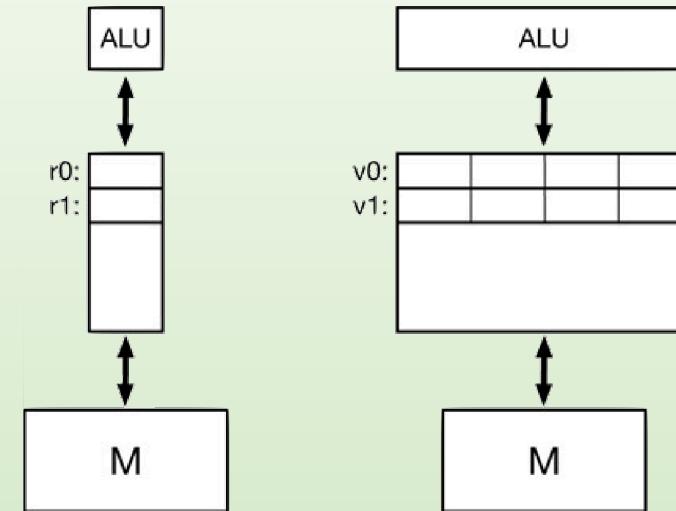
SIMD

- ✓ Modelli SIMD sono basati su **unità funzionali** 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**



ripetuto
4 volte

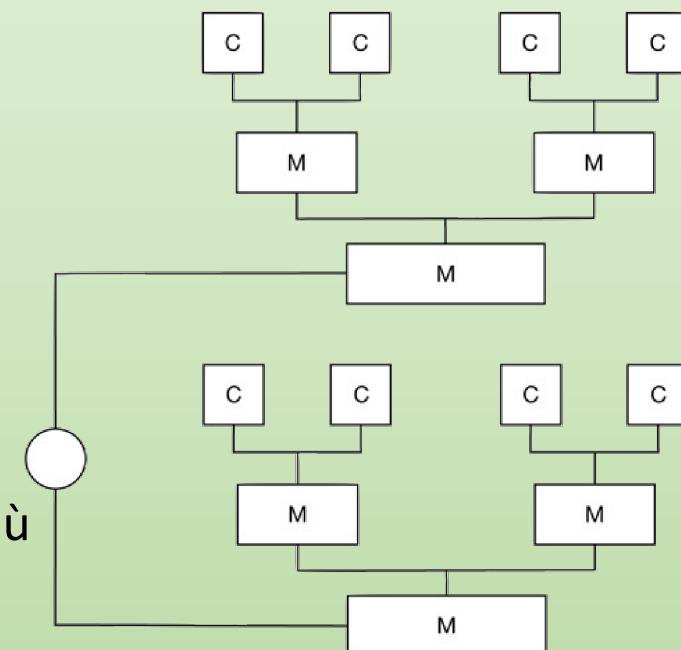
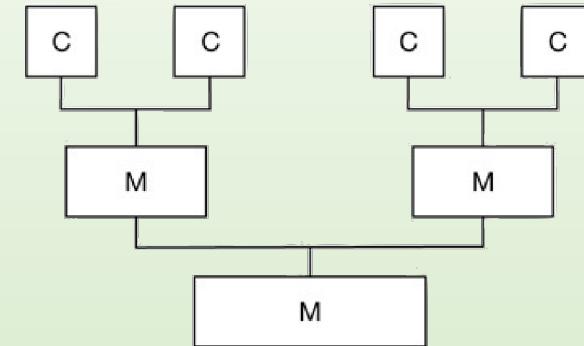
1: $r1 \leftarrow \text{load } a[i]$
2: $r2 \leftarrow \text{load } b[i]$
3: $r2 \leftarrow \text{add } r1, r2$
4: $c[0] \leftarrow \text{store } r2$

eseguito
1 volta

1: $v1 \leftarrow \text{vload } a$
2: $v2 \leftarrow \text{vload } b$
3: $v2 \leftarrow \text{vadd } v1, v2$
4: $c \leftarrow \text{vstore } v2$

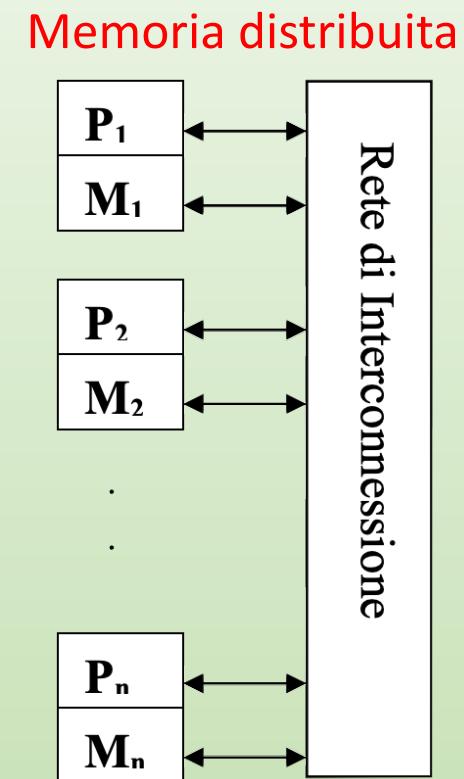
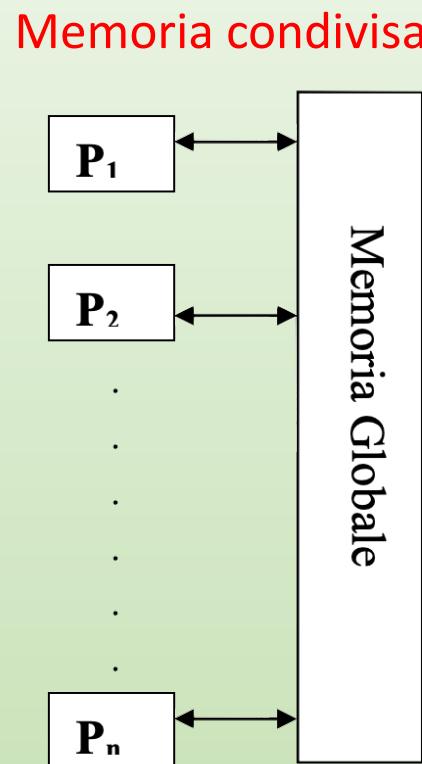
MIMD

- ✓ Molti modelli paralleli sono **MIMD** in cui ogni unità di esecuzione (processi o thread) esegue **diverse istruzioni di dati differenti**
- ✓ **Differiscono** nei modi con cui sono **connessione** tra loro e nei **modelli** di gerarchia di **memoria**
- ✓ La **potenza** di calcolo è ottenuta **connettendo** tra loro i **processori**
 - **Multiprocessor:** memory address space comune + an on-board network che connette i memory modules di ogni processore
 - **Multicomputer:** implementa una architettura NUMA (Non Uniform Memory Architecture) con tempi di accesso alla mem non uniformi (un core può ottenere un dato molto più veloci. di un altro)



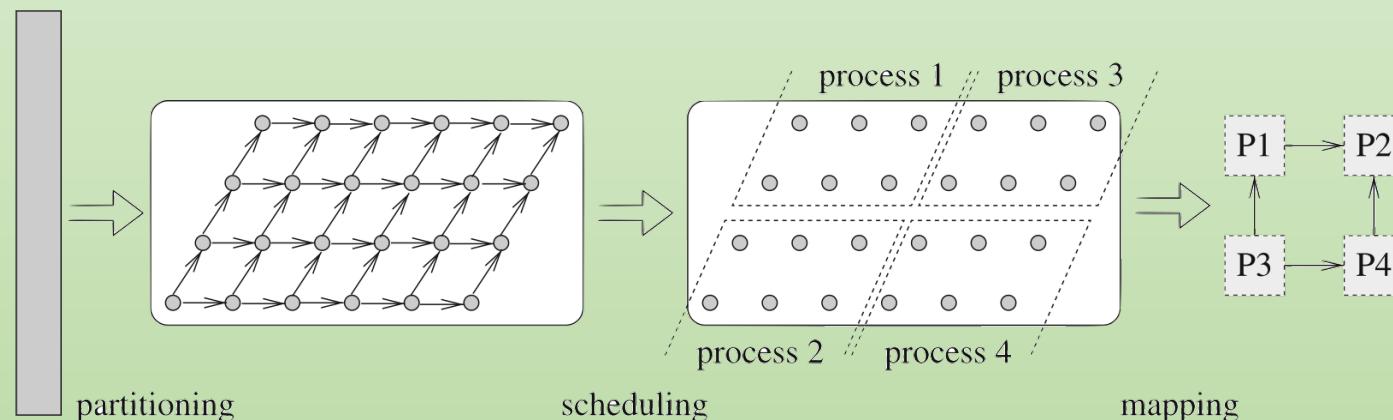
Modello PRAM

- ✓ 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)



Parallelizzazione di algoritmi

1. **Decomposizioni delle computazioni:** suddivisione in **task** e determinazione delle loro (in)dipendenze
 - in linea di principio un task è una **sequenza di istruzioni** eseguite da un singolo **processore**
 - il tempo di computazione di un task è in genere detta **granularità**
2. **Assegnamento di task a thread:** un thread rappresenta un flusso in esecuzione cui viene assegnato un dato task
 - quando il numero eccede i processori si opera un **load balancing** dei task su processori
3. **Mapping dei thread su processori:** in genere si mappa (molti a uno) i thread su processori fisici mediante algoritmi di scheduling (emandato a SO)



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

CPU multi-core & programmazione multi-threading

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

Processi e thread

Processo

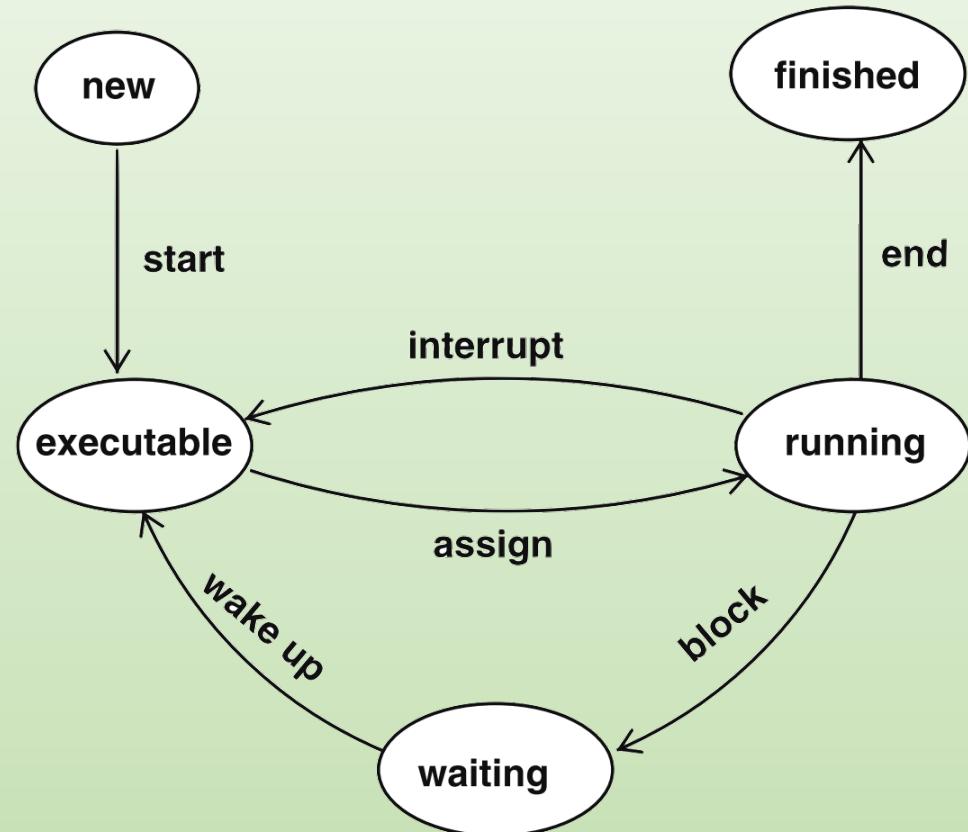
- ✓ Un processo è un **programma in esecuzione** con diverse risorse allocate (stack, heap, registri, prog c.)
- ✓ Consiste di **vari thread** che condividono lo stesso (esclusivo) **spazio di indirizzamento**
- ✓ La moltitudine di **processi** in esecuzione (round-robin) subiscono un **context switch**
- ✓ 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** (stesso program.)

Thread

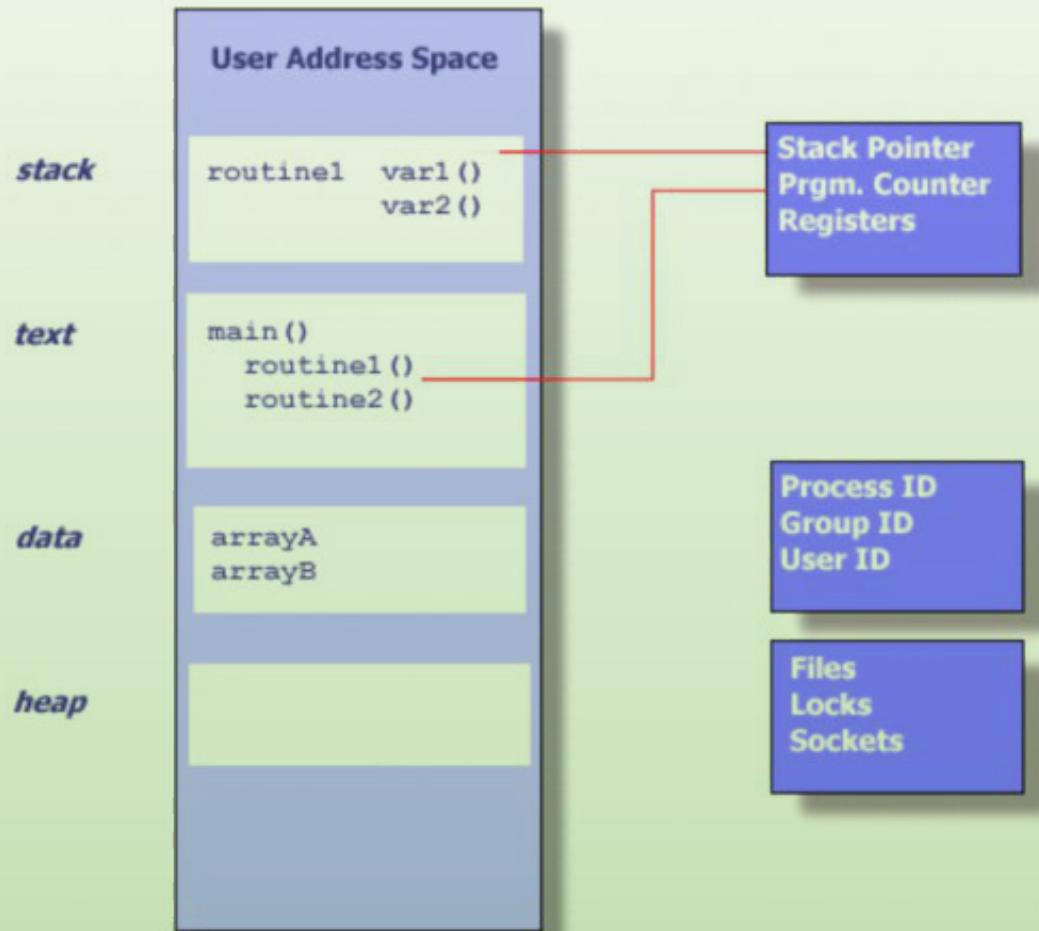
- ✓ Un thread (su CPU) è una **estensione** del modello di processo
- ✓ E' un **flusso di istruzioni** di un programma e viene schedulato come unità **indipendente** nelle code di esecuzione dei processi della CPU (scheduler)
- ✓ Dal punto di vista del **programmatore**, l'esecuzione del thread è **sequenziale**, 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 conditions** (accesso a var condivise o in generale comportamenti non deterministici)

Stati di un thread

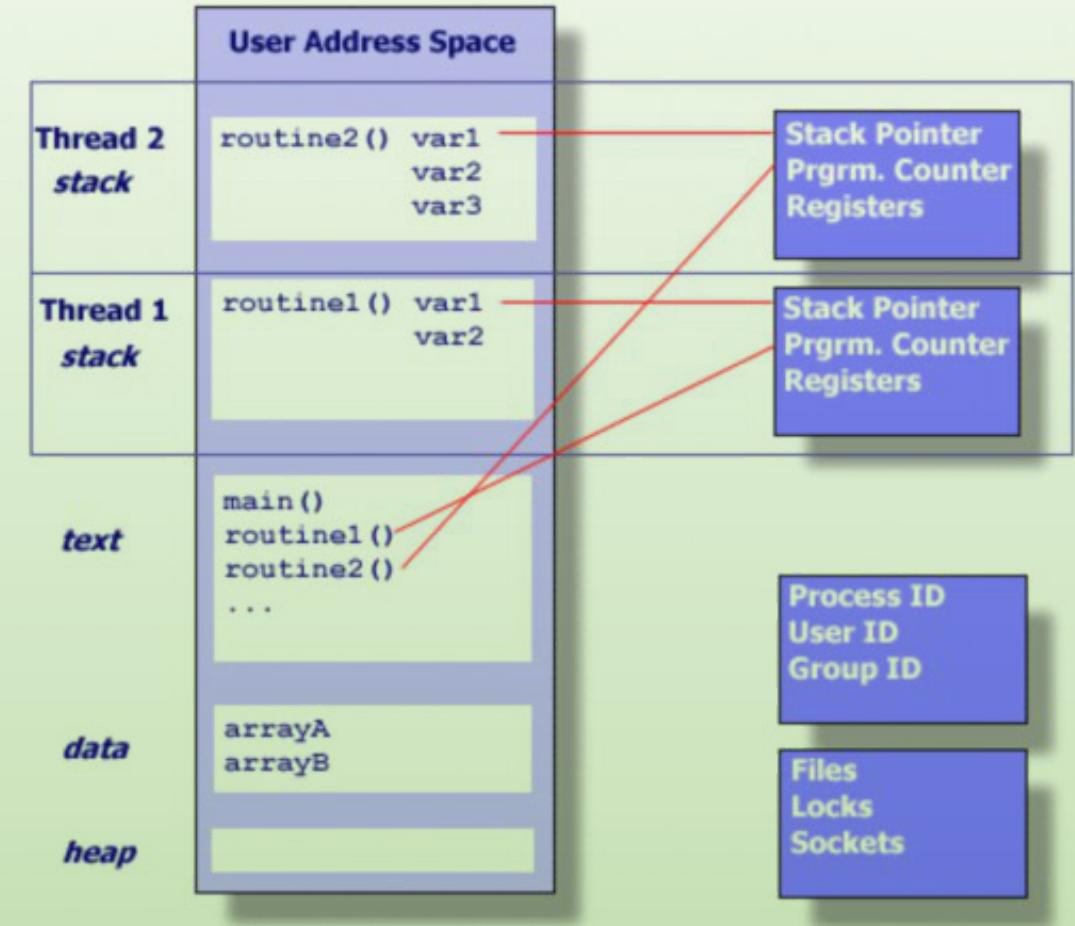
- ✓ **Newly generated:** il thread è stato generato e non ha ancora eseguito operazioni
- ✓ **Executable:** il thread è pronto per l'esecuzione, ma al momento non è assegnato a nessuna unità di calcolo
- ✓ **Running:** il thread è in esecuzione
- ✓ **Waiting:** il thread è in attesa di un evento esterno (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



Processi e thread in Unix



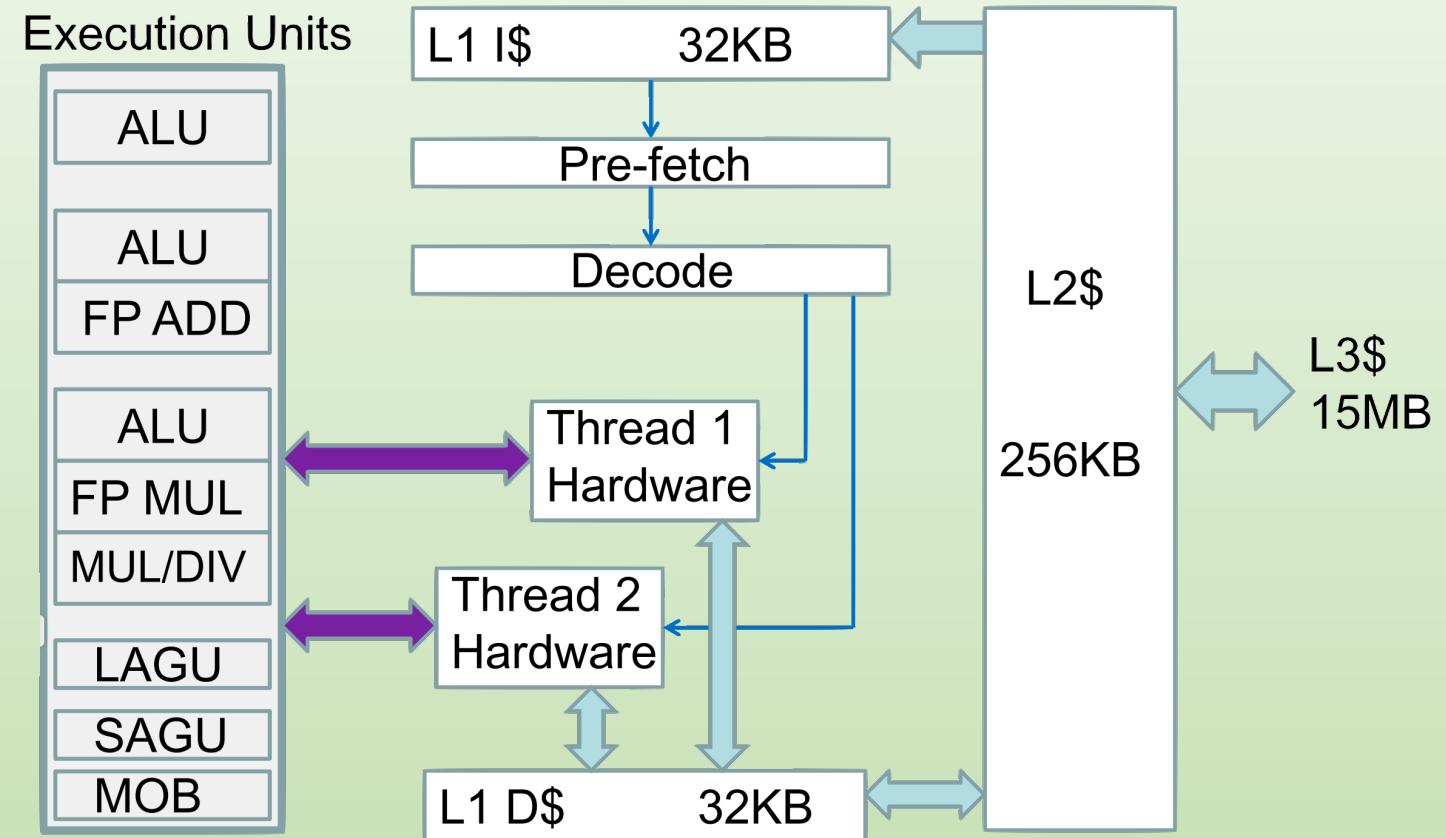
UNIX PROCESS



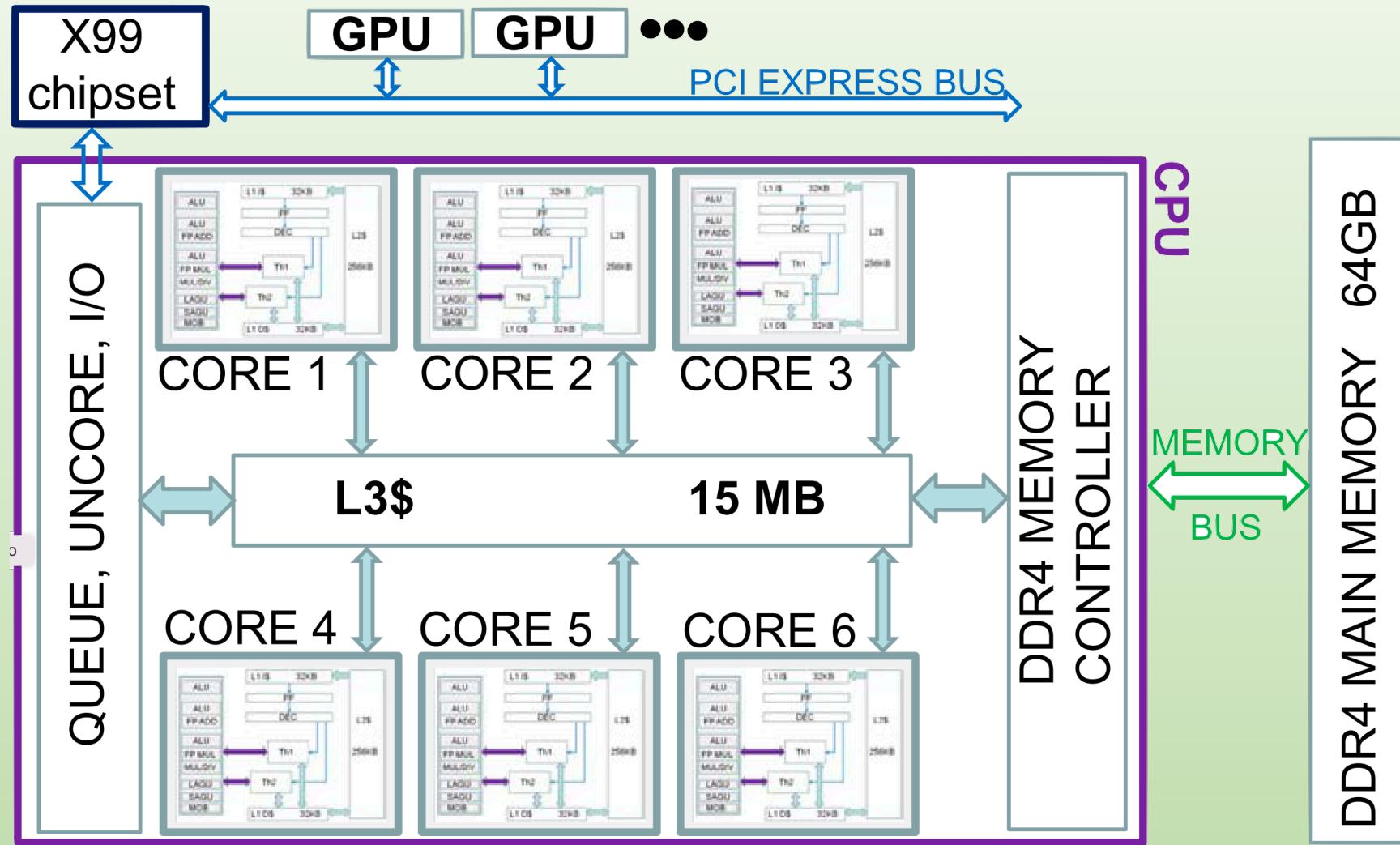
THREADS WITHIN A UNIX PROCESS

Architettura di un core della CPU i7-5930K

- ✓ A **core** is capable of executing **two threads (hyper-threading)**
- ✓ These two threads **share most** of the **core resources**, but have their **own register file**
- ✓ Each core has a **64KB L1** and **256KB L2 cache** memory
- ✓ L1\$ is broken into a **32KB instruction cache (L1I\$)** and a **32KB data cache (L1D\$)**
- ✓ L1I\$ stores the most recently used CPU instructions
- ✓ L1D\$ caches a copy of the data elements
- ✓ The L2\$ is used to **cache either instructions or data**



Architettura della CPU (Intel) i7-5930K

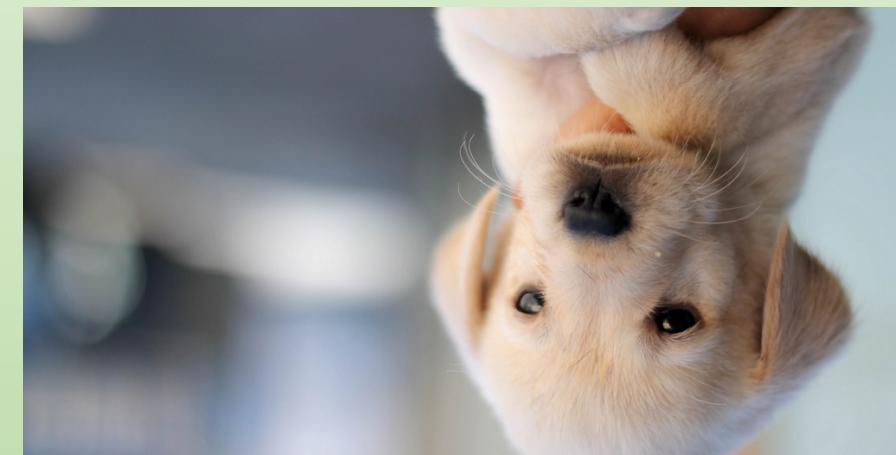


Esempio: flipping di un'immagine

✓ Vedi esercitazione 2



flip orizzontale



flip verticale

CUDA Zone...

Modello di programmazione di CUDA

Che cosa significa programmare in CUDA C?

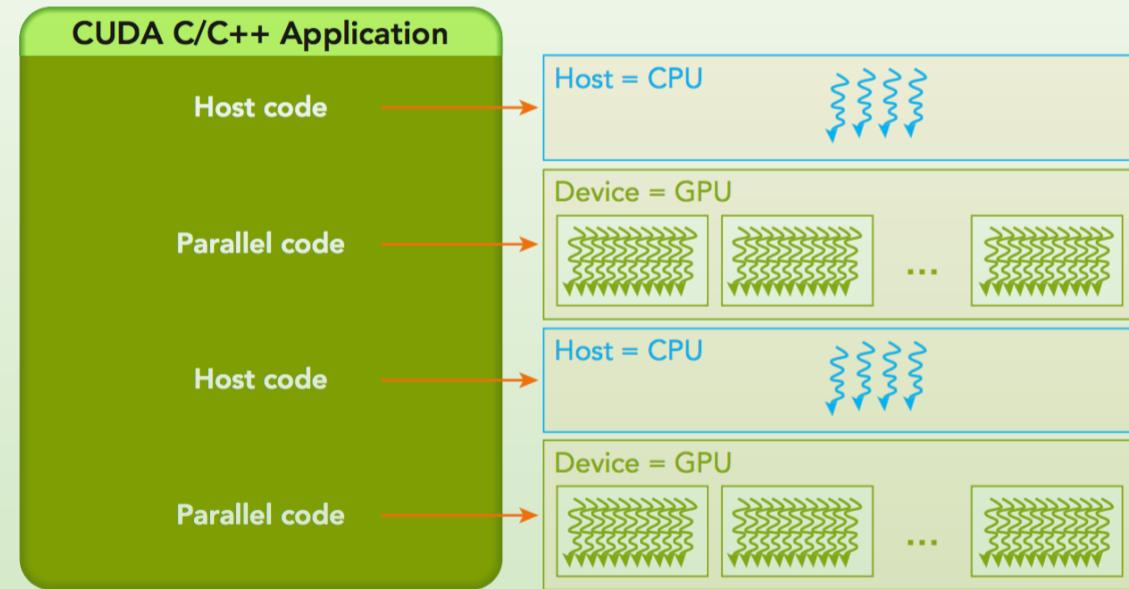
- ✓ Pensare in parallelo significa avere chiaro quali **feature** la **GPU** espone al programmatore
- ✓ Conoscere l'**architettura della GPU** per scalare su migliaia di thread come fosse uno
 - ES.: **gestione basso livello cache** permette di sfruttare principio di località
 - Conoscere lo **scheduling** 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 in analogia a **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 miglia di thread (permette di pensare 'ancora' in sequenziale)

CUDA DEVELOPMENT ENVIRONMENT

- ✓ NVIDIA Nsight™ integrated development environment
- ✓ CUDA-GDB command line debugger
- ✓ Visual and command line profiler for performance analysis
- ✓ CUDA-MEMCHECK memory analyzer
- ✓ GPU device management tools

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)

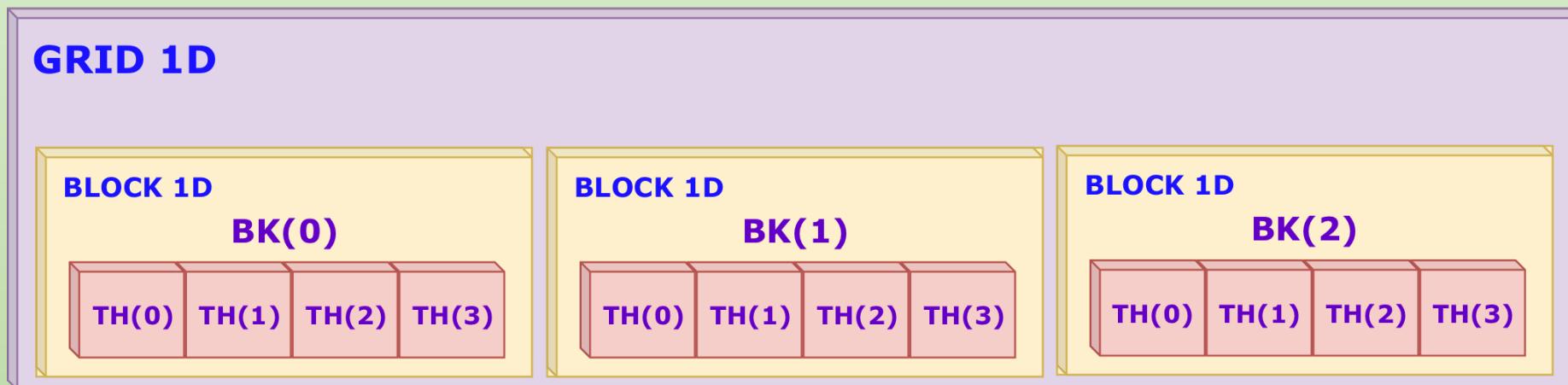
La programmazione in CUDA

La “Formula” di base

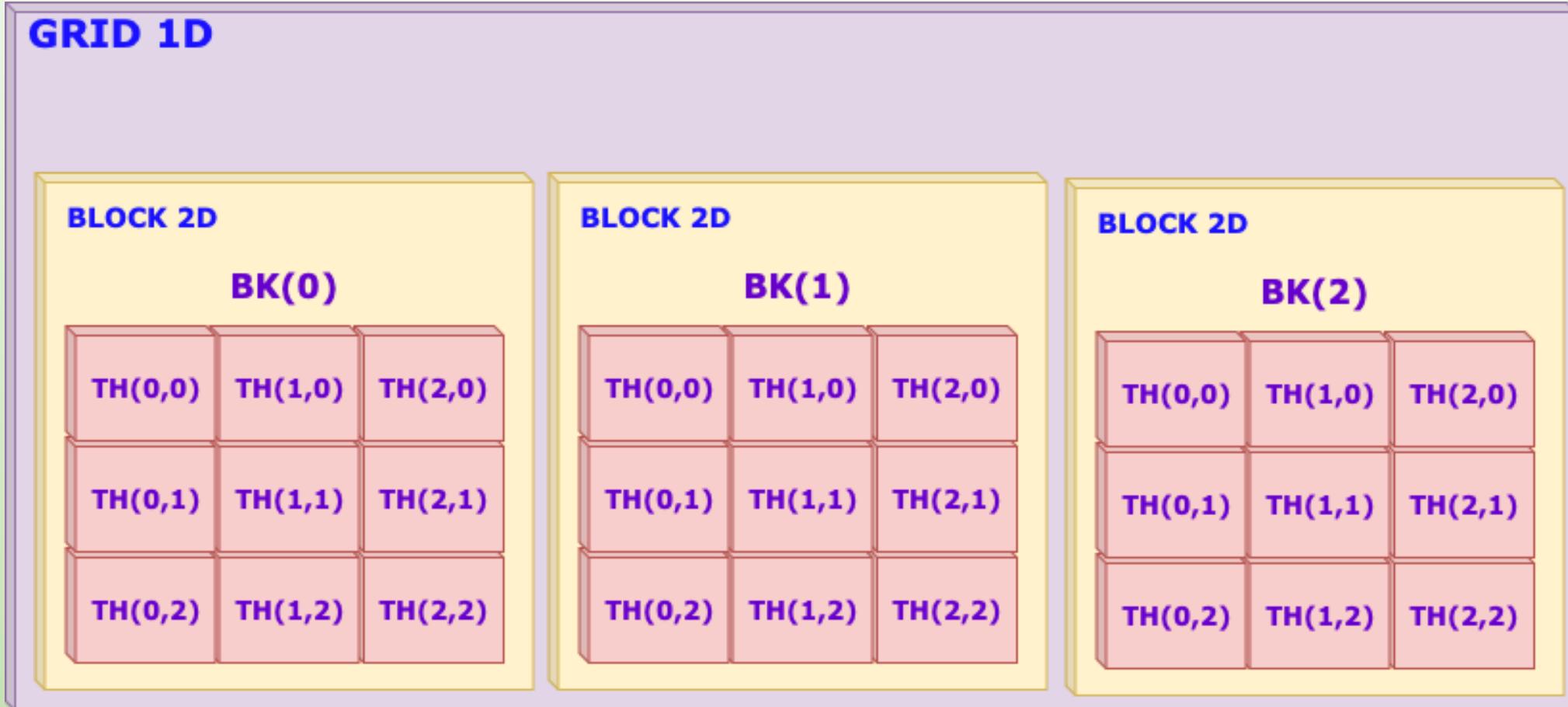
1. Setup dei dati su host (CPU-accessible memory)
2. Alloca memoria per i dati sulla GPU
3. Copia i dati da host a GPU
4. Alloca memoria per output su host
5. Alloca memoria per output su GPU
6. Lancia il kernel su GPU
7. Copia output da GPU a host
8. Cancella le memorie

Organizzazione dei thread

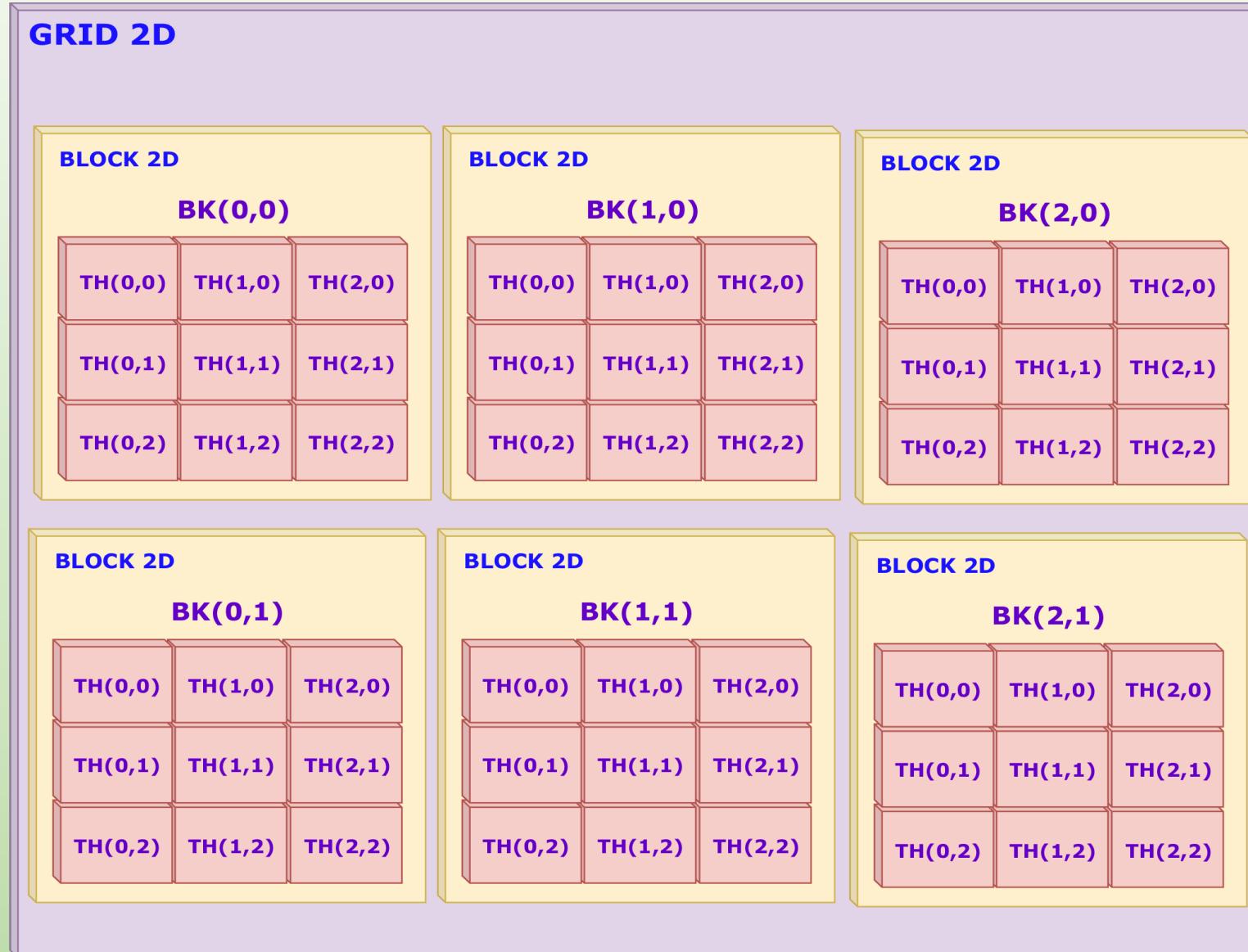
- ✓ CUDA presenta una **gerarchia astratta** di thread strutturata su **due livelli** che si decomponete in
 - **grid**: una griglia ordinata di blocchi
 - **block**: una collezione ordinata di thread
- ✓ Struttura: grid e block possono avere dimensioni
 - **grid**: 1D, 2D e 3D
 - **block**: 1D, 2D e 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

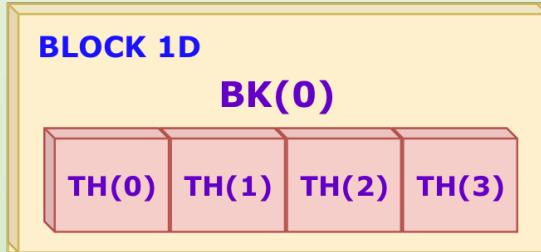


Esempio di grid 1D e block 2D



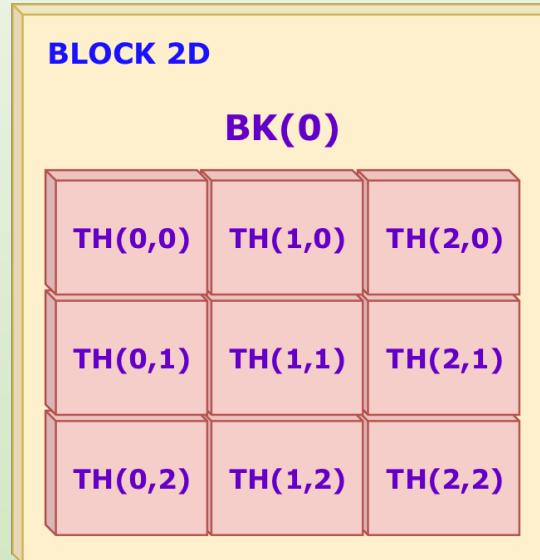
Numero di thread per block

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



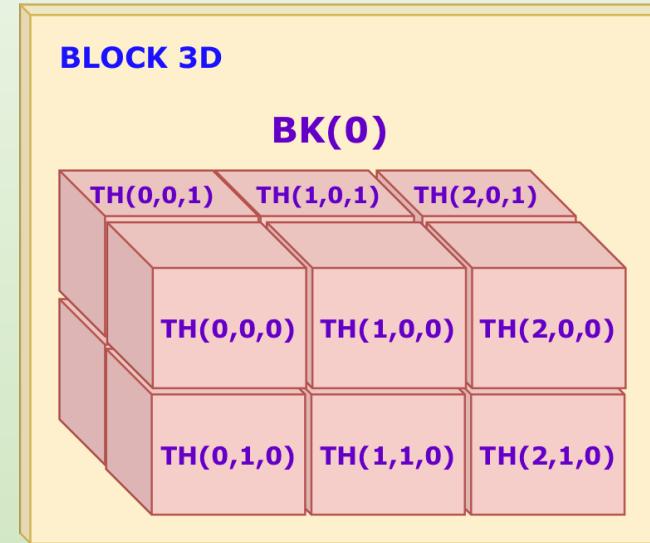
Esempi 1D:

- (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 **non possono** cooperare
- ✓ Tutti i thread in una **grid** condividono la 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**

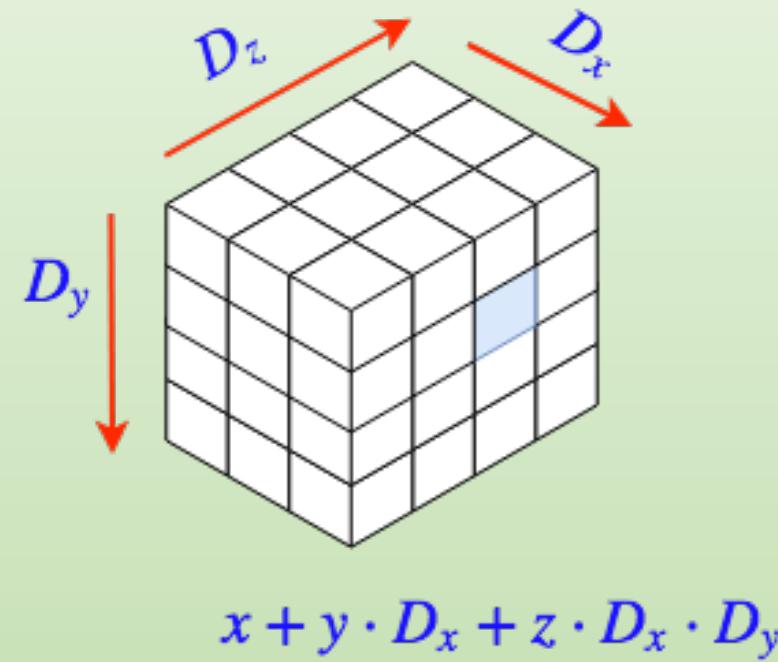
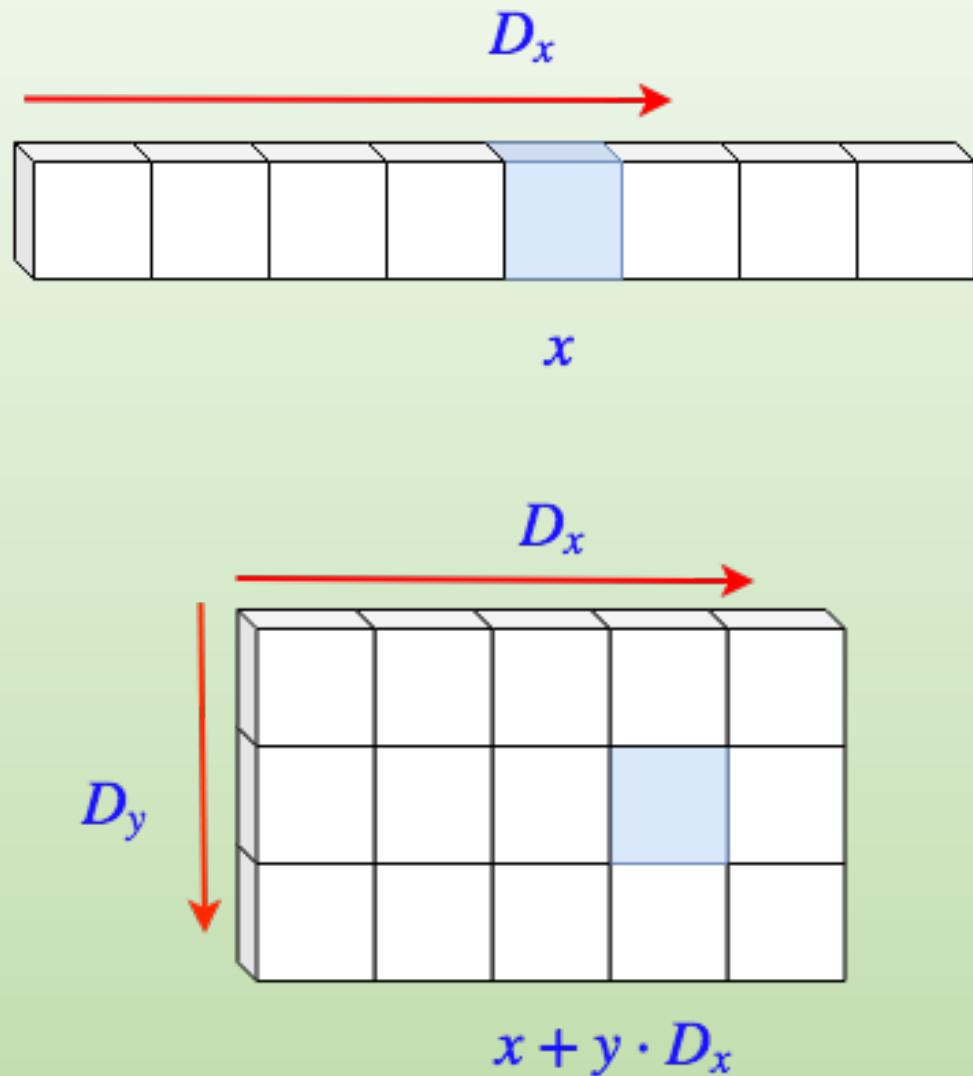
dimensioni
di blocco

blockDim.x
blockDim.y
blockDim.z

dimensioni
di griglia

gridDim.x
gridDim.y
gridDim.z

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



ID di thread unico nella griglia 1D

✓ grid 1D block 1D

$ID_{th} = blockIdx.x * blockDim.x + threadIdx.x$ (indice di thread)

✓ grid 1D block 2D

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

✓ grid 1D block 3D

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

ID unico nella griglia 2D

✓ grid 2D block 1D

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

$ID_{th} = blockIdx.x * blockDim.x + threadIdx.x$ (indice di thread)

✓ grid 2D block 2D

$ID_{th} = ID_{bk} * (blockDim.x * blockDim.y) +$
 $threadIdx.y * blockDim.x +$
 $threadIdx.x$

✓ grid 2D block 3D

$ID_{th} = ID_{bk} * (blockDim.x * blockDim.y * blockDim.z) +$
 $threadIdx.z * (blockDim.x * blockDim.y) +$
 $threadIdx.y * blockDim.x +$
 $threadIdx.x$

ID unico nella griglia 3D

✓ grid 3D block 1D

```
IDbk = blockIdx.x + blockIdx.y * blockDim.x +      (indice di blocco)  
        blockDim.x * blockDim.y * blockIdx.z;  
  
IDth = IDbk * blockDim.x + threadIdx.x;           (indice di thread)
```

✓ grid 3D block 2D

```
IDth = IDbk * (blockDim.x * blockDim.y) +  
        threadIdx.y * blockDim.x +  
        threadIdx.x;
```

✓ grid 3D block 3D

```
IDth = IDbk * (blockDim.x * blockDim.y * blockDim.z) +  
        threadIdx.z * (blockDim.x * blockDim.y) +  
        threadIdx.y * blockDim.x +  
        threadIdx.x;
```

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);
```

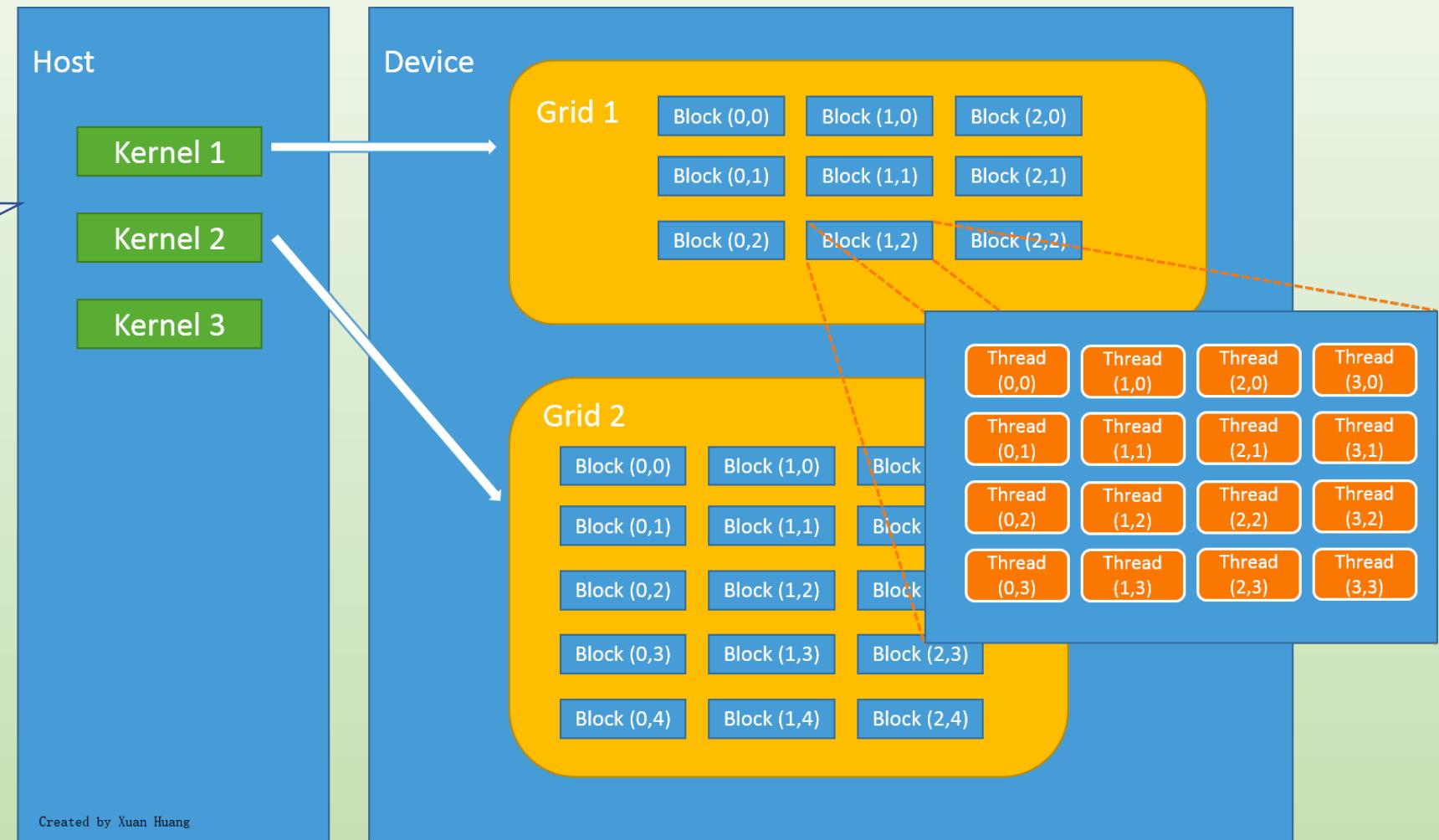
```
kernel_name <<<1, 32>>>(argument list);
```

- ✓ Poiché i dati sono memorizzati linearmente in global memory, possiamo usare le variabili built-in **blockIdx.x** e **threadIdx.x** per:
 - Identificare un unico thread nella grid
 - Stabilire una mappa tra i thread e i singoli elementi dei dati

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)



Kernel: indici e dimensioni

Kernel function: uso delle vars builtin

Dimensionare blocchi e griglia...
Identificatore = variabile libera

La CPU non aspetta... prosegue

Runtime: distruzione contesto associato a processo

```
#include <stdio.h>

/*
 * Mostra DIMs e IDs di grid, block e thread
 */
__global__ void checkIndex(void) {
    printf("threadIdx:(%d, %d, %d) blockIdx:(%d, %d, %d) "
        "blockDim:(%d, %d, %d) gridDim:(%d, %d, %d)\n",
        threadIdx.x, threadIdx.y, threadIdx.z,
        blockIdx.x, blockIdx.y, blockIdx.z,
        blockDim.x, blockDim.y, blockDim.z,
        gridDim.x,gridDim.y,gridDim.z);
}

int main(int argc, char **argv) {
    // definisce grid e struttura dei blocchi
    dim3 block(4);
    dim3 grid(3);
    // controlla dim. dal lato host
    printf("grid.x %d grid.y %d grid.z %d\n", grid.x, grid.y, grid.z);
    printf("block.x %d block.y %d block.z %d\n", block.x, block.y, block.z);
    // controlla dim. dal lato device
    checkIndex<<<grid, block>>>();
    // reset device
    cudaDeviceReset();
    return(0);
}
```

Esecuzione

Compilazione

```
$ nvcc -arch=sm_20 grid1D.cu -o grid1D
```

Esecuzione con parametri **gridDim = 3** e **blockDim = 4**

```
$ ./ grid1D
grid.x = 3  grid.y = 1  grid.z = 1
block.x = 4  block.y = 1  block.z = 1
threadIdx:(0, 0, 0) blockIdx:(1, 0, 0) blockDim:(4, 1, 1) gridDim:(3, 1, 1)
threadIdx:(1, 0, 0) blockIdx:(1, 0, 0) blockDim:(4, 1, 1) gridDim:(3, 1, 1)
threadIdx:(2, 0, 0) blockIdx:(1, 0, 0) blockDim:(4, 1, 1) gridDim:(3, 1, 1)
threadIdx:(3, 0, 0) blockIdx:(1, 0, 0) blockDim:(4, 1, 1) gridDim:(3, 1, 1)
threadIdx:(0, 0, 0) blockIdx:(0, 0, 0) blockDim:(4, 1, 1) gridDim:(3, 1, 1)
threadIdx:(1, 0, 0) blockIdx:(0, 0, 0) blockDim:(4, 1, 1) gridDim:(3, 1, 1)
threadIdx:(2, 0, 0) blockIdx:(0, 0, 0) blockDim:(4, 1, 1) gridDim:(3, 1, 1)
threadIdx:(3, 0, 0) blockIdx:(0, 0, 0) blockDim:(4, 1, 1) gridDim:(3, 1, 1)
threadIdx:(0, 0, 0) blockIdx:(2, 0, 0) blockDim:(4, 1, 1) gridDim:(3, 1, 1)
threadIdx:(1, 0, 0) blockIdx:(2, 0, 0) blockDim:(4, 1, 1) gridDim:(3, 1, 1)
threadIdx:(2, 0, 0) blockIdx:(2, 0, 0) blockDim:(4, 1, 1) gridDim:(3, 1, 1)
threadIdx:(3, 0, 0) blockIdx:(2, 0, 0) blockDim:(4, 1, 1) gridDim:(3, 1, 1)
```

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](#)

Verifica del codice (debugging by printf)

- ✓ Invece di 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

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](#)

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

- `__device__` e `__host__` possono essere **usati insieme** e in questo caso la funzione viene compilata sia per **host** che per **device**

Restrizioni sul kernel

1. Accede alla sola memoria device
2. Deve restituire un tipo void
3. Non supporta il numero variabile di argomenti
4. Non supporta variabili statiche
5. Non supporta puntatori a funzioni
6. Esibisce un comportamento asincrono rispetto all'host

Runtime API vs Device API

The driver and runtime APIs are very similar and can for the most part be used interchangeably

Complexity vs. control

- ✓ The **runtime API** eases device code management by providing:
 - **implicit initialization, context management, and module management**
 - **simpler code**, but it also lacks the level of **control** that the driver API has
- ✓ The **driver API** offers more **fine-grained control**, especially over contexts and module loading
 - **Kernel launches** are much **more complex** to implement, as the execution configuration and kernel parameters must be specified with **explicit function calls**
 - However, unlike the **runtime**, where all the kernels are **automatically loaded** during initialization and stay loaded for as long as the program runs, with the **driver API** it is possible to only keep the modules that are **currently needed loaded**, or even dynamically reload modules
 - driver API is also **language-independent** as it only deals with **cubin objects**

Runtime API vs Device API

The driver and runtime APIs are very similar and can for the most part be used interchangeably

Complexity vs. control

- ✓ The **runtime API** eases device code management by providing:
 - **implicit initialization, context management, and module management**
 - **simpler code**, but it also lacks the level of **control** that the driver API has
- ✓ The **driver API** offers more **fine-grained control**, especially over contexts and module loading
 - **Kernel launches** are much **more complex** to implement, as the execution configuration and kernel parameters must be specified with **explicit function calls**
 - However, unlike the **runtime**, where all the kernels are **automatically loaded** during initialization and stay loaded for as long as the program runs, with the **driver API** it is possible to only keep the modules that are **currently needed loaded**, or even dynamically reload modules
 - driver API is also **language-independent** as it only deals with **cubin objects**

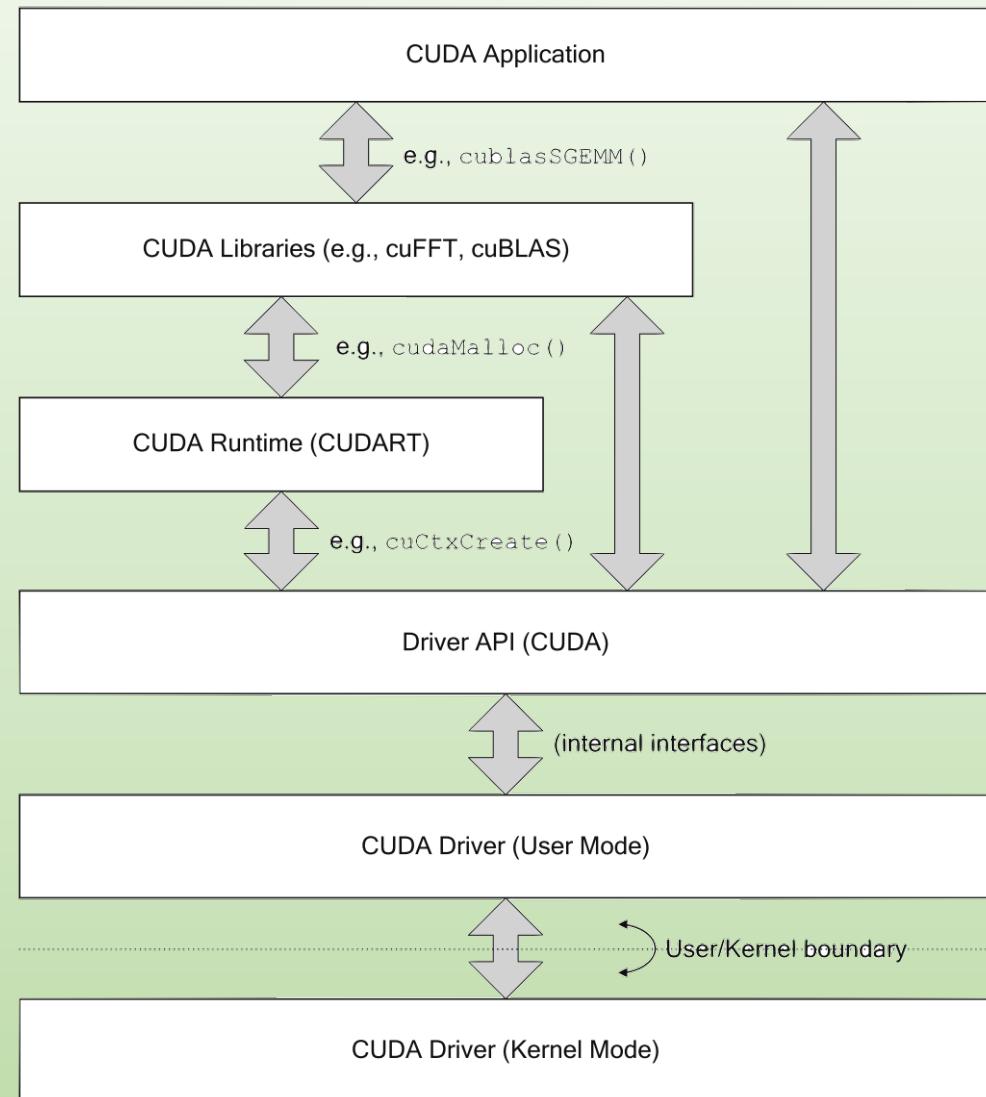
Runtime API vs Device API

Context management

- ✓ **Context management** can be done **through the driver API**, but is **not exposed** in the **runtime API**
- ✓ Runtime API decides itself which **context** to use for a **thread**: if a context has been made current to the calling thread through the driver API, the runtime will use that, but if there is no such context, it uses a "**primary context**"
- ✓ **Primary contexts** are **created as needed**, one per device per process, are reference-counted, and are then destroyed when there are no more references to them
- ✓ Within one process, all **users** of the runtime API will **share** the **primary context**, unless a context has been made **current** to each thread. The context that the runtime uses, i.e, either the current context or primary context, can be synchronized with **cudaDeviceSynchronize()**, and destroyed with **cudaDeviceReset()**
- ✓ Using the runtime API with primary contexts has its tradeoffs, however. It can cause trouble for users writing plug-ins for larger software packages, for example, because if all plug-ins run in the same process, they will all share a context but will likely have no way to communicate with each other.
- ✓ So, if one of them calls **cudaDeviceReset()** after finishing all its CUDA work, the other plug-ins will fail because the context they were using was destroyed without their knowledge.
- ✓ To avoid this issue, CUDA clients can use the driver API to create and set the current context, and then use the runtime API to work with it.
- ✓ However, contexts may consume significant resources, such as device memory, extra host threads, and performance costs of context switching on the device. This runtime-driver context sharing is important when using the driver API in conjunction with libraries built on the runtime API, such as cuBLAS or cuFFT.

Livelli SW di CUDA

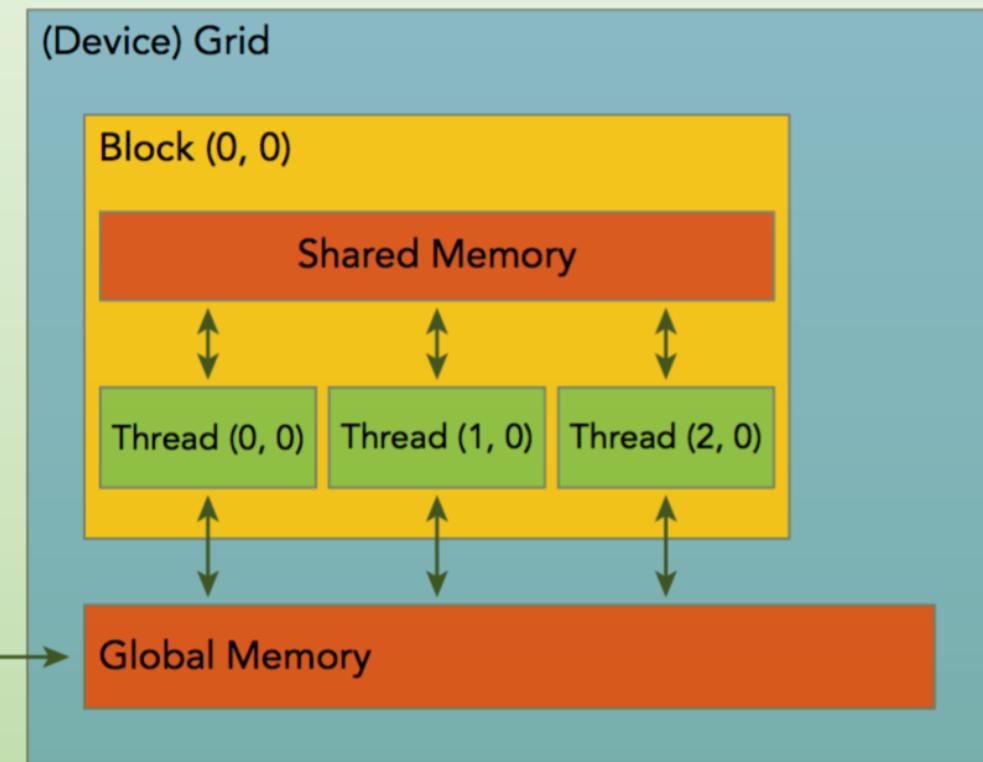
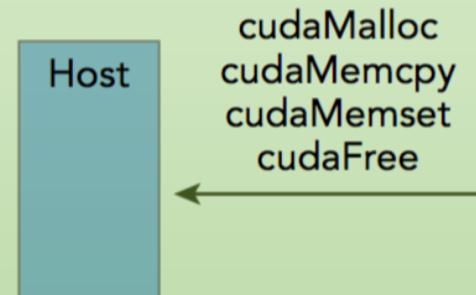
- ✓ Tutti i livelli operano con i **privilegi utente** eccetto il livello finale del **driver** che opera in **kernel mode**
- ✓ In CUDA host e device memory allocati a un programma CUDA non possono essere acceduti da un altro programma CUDA
- ✓ The **CUDA runtime** è la libreria su cui poggia la toolchain C++/GPU integrata in CUDA
- ✓ Il compilatore **nvcc** suddivide i **.cu** file in porzioni di codice host e device: la porzione ridiretta all'host genera automaticamente chiamate a CUDA runtime per facilitare operazioni come il lancio di un kernel
- ✓ The **CUDA driver** preserva la compatibilità verso il basso, supportando programmi indipendentemente dalla versione di CUDA o vecchi programmi
- ✓ Esporta “**driver API**” (in `cuda.h`) di basso livello consentendo l’uso diretto di risorse che si interfacciano al kernel mode



Gestione della Memoria (preview)

Funzioni C standard	Funzioni CUDA C
malloc	cudaMalloc
memcpy	cudaMemcpy
memset	cudaMemset
free	cudaFree

- ✓ When you use **cudaMemcpy** to copy data between the host and device, **implicit synchronization** at the host side is performed and the host application must wait for the data copy to complete



cudaMalloc e cudaMemcpy

Segnatura->

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

- `devPtr` è un puntatore a un puntatore in Global memory del device

Segnatura->

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

Kind-->

```
cudaMemcpyHostToHost, cudaMemcpyHostToDevice  
cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice
```

- Questa funzione esibisce un comportamento sincrono che blocca il programma host fino a che il trasferimento non viene completato
- Per capire se `destination` e `src` sono puntatori a memoria CPU o GPU si guarda la variabile `kind`

Errore-->

cudaError_t

return->

success

cudaErrorMemoryAllocation

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

Errore-->

```
char* cudaGetString(cudaError_t error)
```

- Conversione del messaggio di errore in forma leggibile

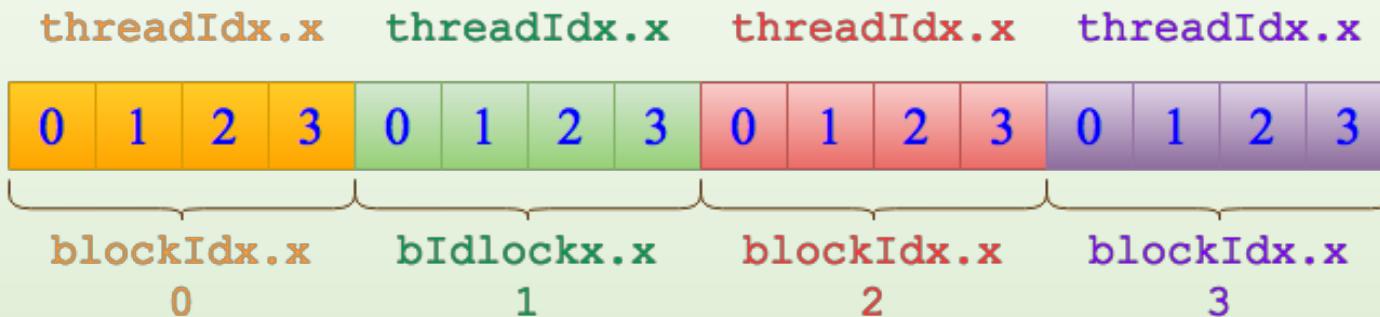
Gestione errori

- ✓ 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());`

Griglia 1D - coordinate 1D



Calcolo dell'indice lineare:

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

Kernel finale:

```
/*
 * kernel: somma di vettori
 */
__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];
}
```

Esempio: allocazione su host e device

Alloc. CPU->

```
float *h_A, *h_B, *h_C;  
  
h_A = (float *) malloc(nBytes);  
  
h_B = (float *) malloc(nBytes);  
  
h_C = (float *) malloc(nBytes);
```

Alloc. GPU->

```
float *d_A, *d_B, *d_C;  
  
cudaMalloc((void **) &d_A, nBytes);  
  
cudaMalloc((void **) &d_B, nBytes);  
  
cudaMalloc((void **) &d_C, nBytes);
```

➤ Notare l'uso degli identificatori specifici per i dispositivi

- ✓ 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

Esempio: somma di vettori

Schema generale:

```
#include <stdio.h>
. . .
_global_ void add_vect(...)
```

1. allocare la memoria device per h_a, h_b e h_c
2. Lanciare il kernel con argomenti h_a, h_b e h_c
3. Copiare dalla memoria device alla memoria host c il risultato in h_c
4. Liberare la memoria hos e device con cudaFree(*ptr);

Osservazioni:

- ✓ Fissata la dimensione dei dati, il passo generale per determinare le dimensioni di grid e blocco sono:
 - Decidere il **block size** (\leq di 1024)
 - Calcolare la **dimensione di grid** basandoci sulla dimensione dei dati dell'applicazione
 - Per device basati su Fermi (es. le schede Tesla sulla macchina Lagrange) la massima dim. per la grid per ogni x, y, and z è 65,535
- ✓ Per determinare la dimensione del blocco, si considera:
 - **Performance** che caratterizzano il kernel
 - **Limitazioni** sulle risorse della GPU

Il codice della somma vettori

```
#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];
}

int main(void) {
    int *a, *b, *c;
    int *dev_a, *dev_b, *dev_c;
    int nBytes = N * sizeof(int);

    // malloc host memory
    a = (int *) malloc(nBytes);
    b = (int *) malloc(nBytes);
    c = (int *) malloc(nBytes);

    // malloc device memory
    cudaMalloc((void**) &dev_a, nBytes);
    cudaMalloc((void**) &dev_b, nBytes);
    cudaMalloc((void**) &dev_c, nBytes);

    // fill the arrays 'a' and 'b' on the CPU
    for (int i = 0; i < N; i++) {
        a[i] = rand() % 10;
        b[i] = rand() % 10;
    }
}
```

```
// copy the arrays 'a' and 'b' to the GPU
cudaMemcpy(dev_a, a, nBytes, cudaMemcpyHostToDevice);
cudaMemcpy(dev_b, b, nBytes, cudaMemcpyHostToDevice);

add_vect<<<N / TxB, TxB>>>(dev_a, dev_b, dev_c);

// copy the array 'c' back from the GPU to the CPU
cudaMemcpy(c, dev_c, nBytes, cudaMemcpyDeviceToHost);

// display the results
for (int i = 0; i < N; i++) {
    printf("%d + %d = %d\n", a[i], b[i], c[i]);
}

// Free host memory
free(a);
free(b);
free(c);

// free the memory allocated on the GPU
cudaFree(dev_a);
cudaFree(dev_b);
cudaFree(dev_c);

return 0;
}
```

Misurare il tempo con la CPU

Con la chiamata di sistema `gettimeofday` (includere l'header `sys/time.h`) si ottiene il 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

ESERCITAZIONE:

Misurare il tempo di esecuzione del kernel somma vettori al variare della dim. dei vettori (per adesso usare dati di dimensioni minori di $\sim 6 \cdot 10^7$) e confrontarlo con quello della funzione unix time eseguita a livello di shell

Timing con nvprof

Da CUDA 5.0 il tool di profilazione **nvprof** è incluso per avere la timeline dell'applicazione relative all'attività CPU e GPU: esecuzione dl kernel, trasferimenti di memoria, chiamate delle CUDA API, etc...

```
$ nvprof [nvprof_args] <application> [application_args]
```

```
$ nvprof -help
```

Timeline su somma vettori:

```
$ nvprof add_vect
==8132== Profiling application: a.out
==8132== Profiling result:
Time(%)      Time      Calls      Avg      Min      Max  Name
 37.38%  7.3920us        1  7.3920us  7.3920us  7.3920us  add_vect(int*, int*, int*)
 32.36%  6.4000us        2  3.2000us  2.8480us  3.5520us  [CUDA memcpy HtoD]
 30.26%  5.9840us        1  5.9840us  5.9840us  5.9840us  [CUDA memcpy DtoH]

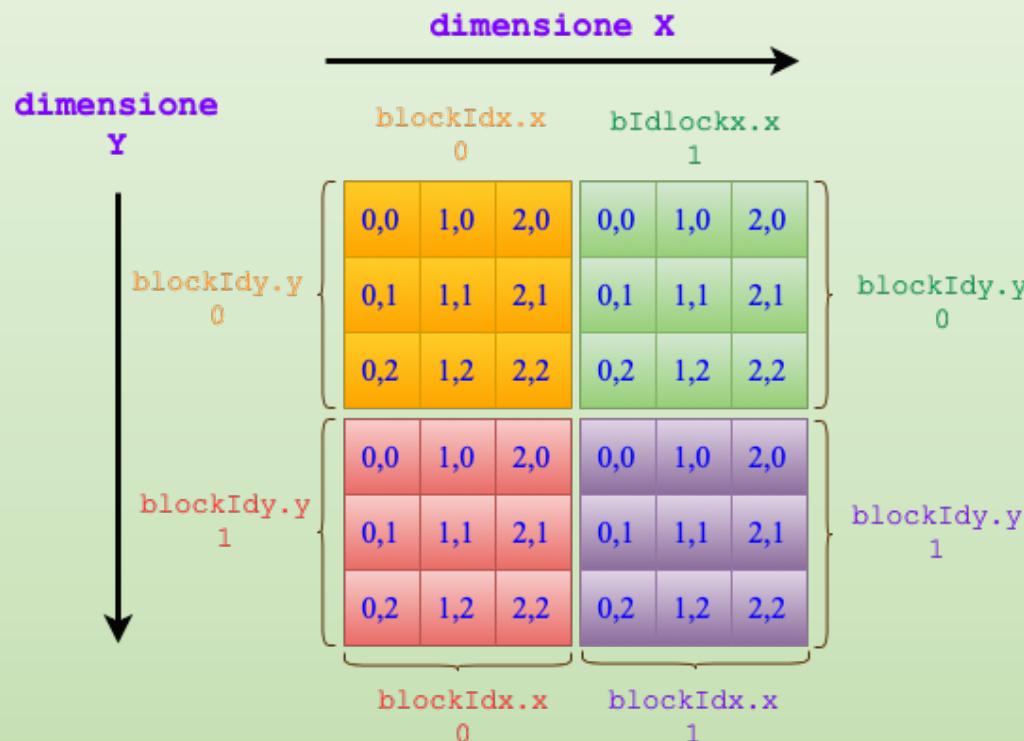
==8132== API calls:
Time(%)      Time      Calls      Avg      Min      Max  Name
 99.73% 174.54ms        3  58.179ms  3.7540us 174.53ms  cudaMalloc
   0.10% 172.88us        3  57.625us  6.9580us 149.96us  cudaFree
   0.07% 115.60us       83  1.3920us    100ns  53.943us  cuDeviceGetAttribute
   0.06% 103.47us        3  34.490us  18.391us  49.043us  cudaMemcpy
   0.03% 46.886us        1  46.886us  46.886us  46.886us  cudaLaunch
   0.01% 17.483us        1  17.483us  17.483us  17.483us  cuDeviceGetName
   0.00% 8.0370us        1  8.0370us  8.0370us  8.0370us  cuDeviceTotalMem
   0.00% 4.7450us        3  1.5810us    267ns  3.8460us  cudaSetupArgument
   0.00% 3.3510us        1  3.3510us  3.3510us  3.3510us  cudaConfigureCall
   0.00% 1.0780us        2    539ns    531ns   547ns  cuDeviceGetCount
   0.00%    697ns        2    348ns    339ns   358ns  cuDeviceGet
```

Griglia 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=5	X=0	X=2	X=2	X=0	X=2	X=5
Y=0	Y=0	Y=0	Y=0	Y=0	Y=0	Y=2	Y=2	Y=2	Y=3	Y=3	Y=2
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=5	X=0	X=2	X=2	X=3	X=5	X=5
Y=3	Y=3	Y=3	Y=3	Y=3	Y=3	Y=3	Y=5	Y=5	Y=5	Y=5	Y=5

Lavorare con le matrici

- ✓ Nel caso 2D ci sono 3 tipi di indici

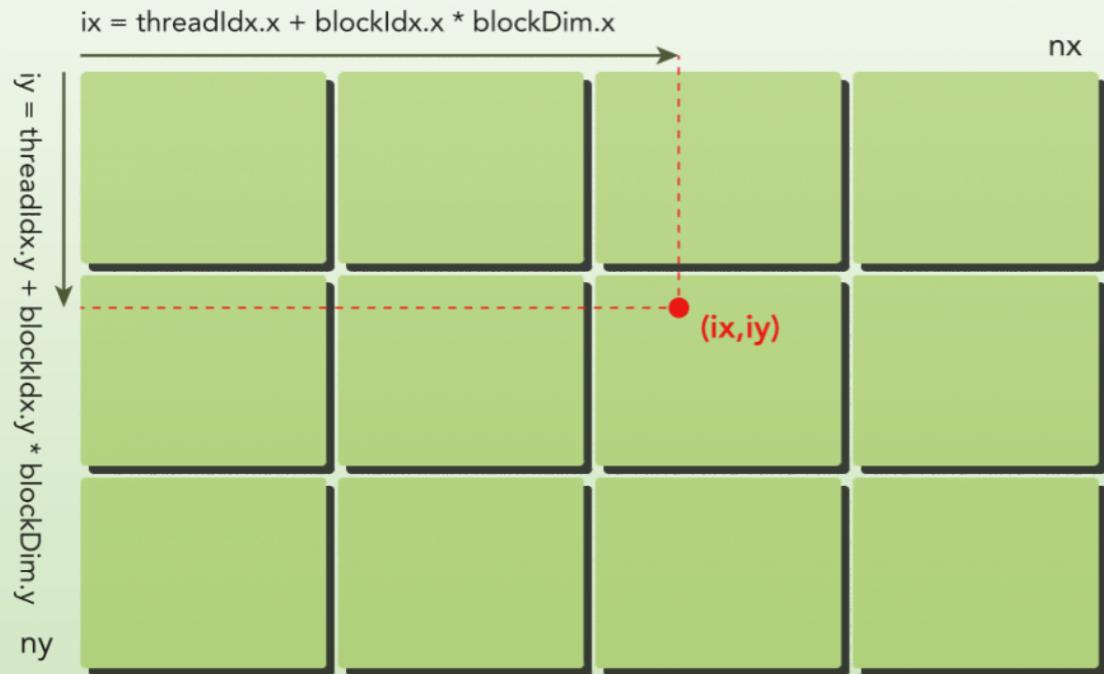
- **indici** di blocchi e di thread
- indice dell'**elemento** della matrice
- indice (**offset**) della memoria lineare globale

- ✓ 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) ...
```

Esercitazione

- Sviluppare un programma CUDA C che effettui la somma di immagini di dimensioni varie e ne normalizzi i dati (intervallo [0,1]) con con configurazione (**grid 2D, block 2D**)
- Calcolare la timeline con nvprof al variare dei params di griglia e blocco

Osservazioni:

- 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
- 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

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

