



Modello di Programmazione CUDA

Sistemi Digitali, Modulo 2

A.A. 2024/2025

Fabio Tosi, Università di Bologna

Panoramica del CUDA Programming Model

➤ Introduzione al Modello di Programmazione

- Concetti base e architettura CUDA
- Ruolo di Host (CPU) e Device (GPU)

➤ Gestione della Memoria in CUDA - Accenni

- Allocazione e trasferimento di memoria
- Tipi di memoria: globale, condivisa (menzioni)

➤ Organizzazione dei Thread

- Gerarchie: Grid, Block, Thread
- Identificazione dei thread

➤ Kernel CUDA

- Definizione e lancio dei kernel
- Configurazione di griglia e blocchi

➤ Tecniche di Mapping e Dimensionamento

- Esempio: Somma di array e mapping degli indici
- Calcolo dinamico delle dimensioni della griglia

➤ Analisi delle Prestazioni

- Correttezza dei risultati e gestione degli errori
- Uso di strumenti di profiling (NVIDIA Nsight)

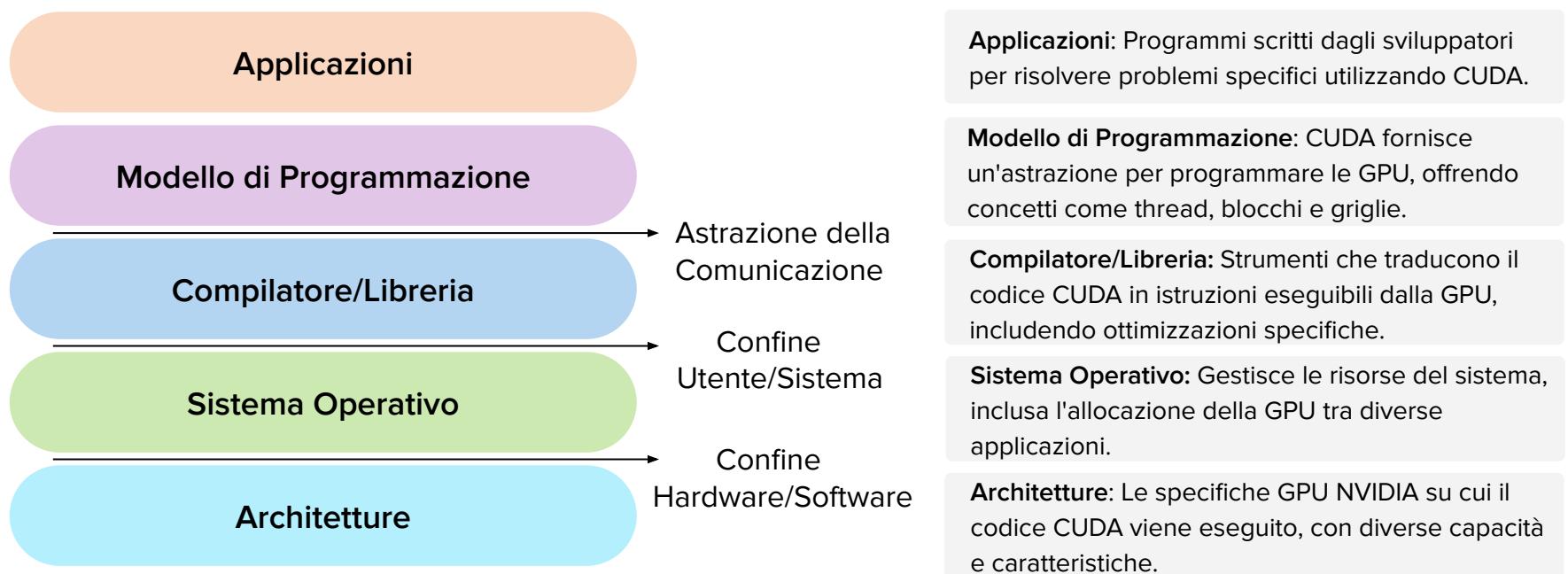
➤ Applicazioni Pratiche

- Operazioni su matrici
- Elaborazione di immagini (es. conversione RGB a grayscale)
- Convoluzione 1D e 2D

La Struttura Stratificata dell'Ecosistema CUDA

Modello CUDA

- L'ecosistema CUDA nel suo complesso può essere visto come una **struttura stratificata** per esprimere algoritmi paralleli su GPU, bilanciando semplicità d'uso e controllo hardware per ottimizzare le prestazioni.



Ruolo del Modello e del Programma

Il Modello di Programmazione:

Definisce la **struttura** e le **regole** per sviluppare applicazioni parallele su GPU. Elementi fondamentali:

- **Gerarchia di Thread:** Organizza l'esecuzione parallela in *thread, blocchi e griglie*, ottimizzando la scalabilità su diverse GPU.
- **Gerarchia di Memoria:** Offre tipi di memoria (*globale, condivisa, locale, costante, texture*) con diverse prestazioni e scopi, per ottimizzare l'accesso ai dati.
- **API:** Fornisce *funzioni* e *librerie* per gestire l'esecuzione del kernel, il trasferimento dei dati e altre operazioni essenziali.

Il Programma:

Rappresenta l'**implementazione concreta** (**il codice**) che specifica come i thread condividono dati e coordinano le loro attività. Nel programma CUDA, si definisce:

- Come i dati verranno **suddivisi** e **elaborati** tra i vari thread.
- Come i thread **accederanno alla memoria** e **condivideranno** dati.
- Quali **operazioni** verranno eseguite in parallelo.
- Quando e come i thread si **sincronizzeranno** per completare un compito.

Livelli di Astrazione nella Programmazione Parallelta CUDA

- Il calcolo parallelo si articola in **tre livelli di astrazione**: dominio, logico e hardware, guidando l'approccio del programmatore.

Livello Dominio

- Focus sulla decomposizione del problema.
- Definizione della struttura parallela di alto livello.

Chiave: Ottimizza la strategia di parallelizzazione.

Livello Logico

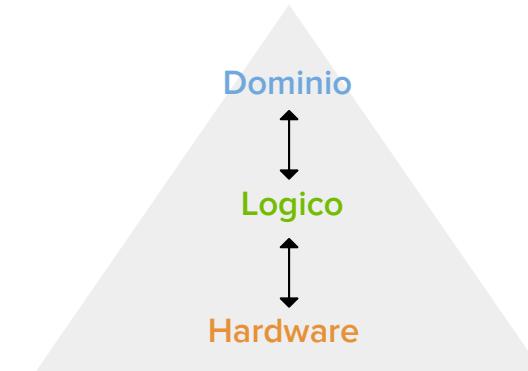
- Organizzazione e gestione dei thread.
- Implementazione della strategia di parallelizzazione.

Chiave: Massimizza l'efficienza del parallelismo.

Livello Hardware

- Mappatura dell'esecuzione sull'architettura GPU.
- Ottimizzazione delle prestazioni hardware.

Chiave: Sfrutta al meglio le risorse GPU.



Esempio: Moltiplicazione di Matrici

- Dominio:** Suddivisione delle matrici.
- Logico:** Organizzazione dei thread per i calcoli.
- Hardware:** Ottimizzazione accesso memoria e esecuzione sui core GPU.

Thread CUDA: L'Unità Fondamentale di Calcolo

Cos'è un Thread CUDA?

- Un thread CUDA rappresenta un'**unità di esecuzione elementare** nella GPU.
- Ogni thread CUDA esegue una porzione di un programma parallelo, chiamato **kernel**.
- Sebbene **migliaia di thread** vengano eseguiti concorrentemente sulla GPU, ogni **singolo thread** segue un percorso di **esecuzione sequenziale** all'interno del suo contesto.



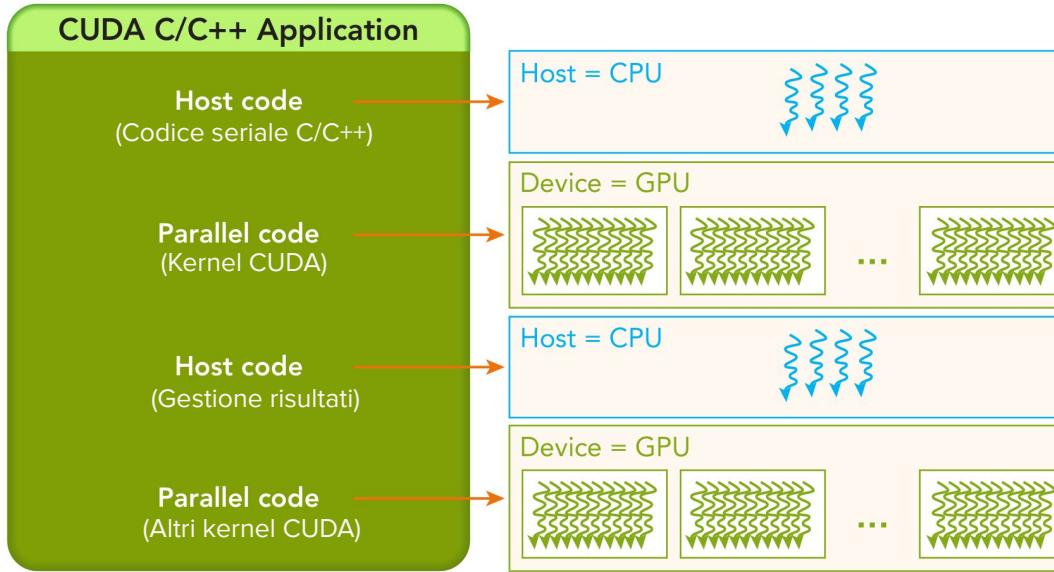
Cosa Fa un Thread CUDA?

- **Elaborazione di Dati:** Ogni thread CUDA si occupa di un piccolo pezzo del problema complessivo, eseguendo calcoli su un sottoinsieme di dati.
- **Esecuzione di Kernel:** Ogni thread esegue lo stesso codice del kernel ma opera su dati diversi, determinati dai suoi identificatori univoci (**threadIdx, blockIdx**).
- **Stato del Thread:** Ogni thread ha il proprio stato, che include il program counter, i registri, la memoria locale e altre risorse specifiche del thread.

Thread CUDA vs Thread CPU

- GPU: Parallelismo Massivo (tanti Core), CPU: Parallelismo Limitato (pochi Core).
- Thread CUDA: Efficienza e Basso Overhead, Thread CPU: Maggior Overhead di Gestione.

Struttura di Programmazione CUDA



Caratteristiche Principali

- Codice Seriale e Parallelo:** Alternanza tra sezioni di codice seriale e parallelo (stesso file).
- Struttura Ibrida Host-Device:** Alternanza tra codice eseguito sulla CPU e sulla GPU.
- Esecuzione Asincrona:** Il codice host può continuare l'esecuzione mentre i kernel GPU sono in esecuzione.
- Kernel CUDA Multipli:** Possibilità di lanciare più kernel GPU all'interno della stessa applicazione.
- Gestione dei Risultati sull'Host:** Fase dedicata all'elaborazione dei risultati sulla CPU dopo l'esecuzione dei kernel.

Flusso Tipico di Elaborazione CUDA

1. Inizializzazione e Allocazione Memoria (Host)

- Prepara dati e alloca memoria su CPU e GPU.

2. Trasferimento Dati (Host → Device)

- Copia input dalla memoria CPU alla GPU.

3. Esecuzione del Kernel (Device)

- GPU esegue calcoli paralleli.

4. Recupero Risultati (Device → Host)

- Copia output dalla memoria GPU alla CPU.

5. Post-elaborazione (Host)

- Analizza o elabora ulteriormente i risultati sulla CPU.

6. Liberazione Risorse

- Libera memoria allocata su CPU e GPU.

***Nota:** I passi 2-5 possono essere ripetuti più volte in un'applicazione complessa.

Panoramica del CUDA Programming Model

- **Introduzione al Modello di Programmazione**
 - Concetti base e architettura CUDA
 - Ruolo di Host (CPU) e Device (GPU)
- **Gestione della Memoria in CUDA - Accenni**
 - Allocazione e trasferimento di memoria
 - Tipi di memoria: globale, condivisa (menzioni)
- **Organizzazione dei Thread**
 - Gerarchie: Grid, Block, Thread
 - Identificazione dei thread
- **Kernel CUDA**
 - Definizione e lancio dei kernel
 - Configurazione di griglia e blocchi
- **Tecniche di Mapping e Dimensionamento**
 - Esempio: Somma di array e mapping degli indici
 - Calcolo dinamico delle dimensioni della griglia
- **Analisi delle Prestazioni**
 - Correttezza dei risultati e gestione degli errori
 - Uso di strumenti di profiling (NVIDIA Nsight)
- **Applicazioni Pratiche**
 - Operazioni su matrici
 - Elaborazione di immagini (es. conversione RGB a grayscale)
 - Convoluzione 1D e 2D

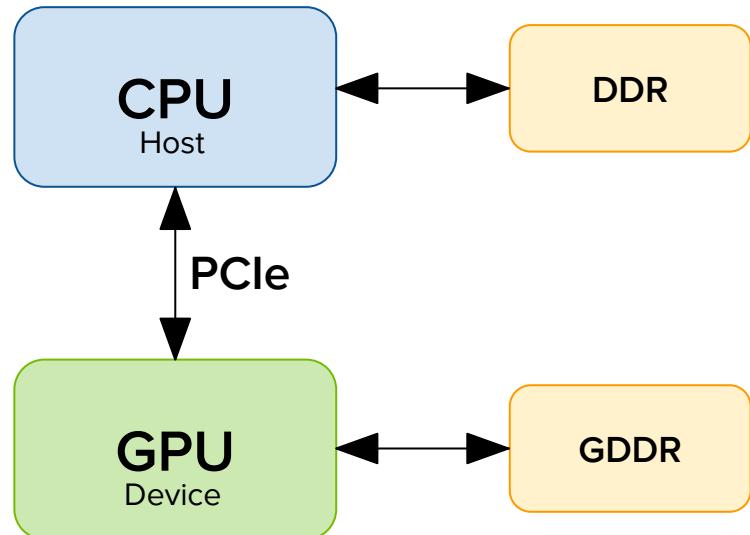
Gestione della Memoria in CUDA

Modello di Memoria CUDA

- Il modello CUDA presuppone un sistema con un **host** e un **device**, ognuno con la propria memoria.
- La **comunicazione** tra la memoria dell'host e quella del device avviene tramite il bus seriale **PCIe** (Peripheral Component Interconnect Express), che permette di trasferire dati tra CPU e GPU.

Caratteristiche PCIe

- Lane:** Ogni lane (canale di trasmissione) è costituito da due coppie di segnali differenziali (quattro fili), una per ricevere e una per trasmettere dati.
- Full-Duplex:** Trasmette e riceve dati simultaneamente in entrambe le direzioni.
- Scalabilità:** La larghezza di banda varia a seconda del numero di lane: x1, x2, x4, x8, x16.
- Bassa Latenza:** Garantisce comunicazioni rapide e reattive nei trasferimenti frequenti.
- Collo di Bottiglia:** Può diventare un collo di bottiglia in trasferimenti di grandi volumi tra CPU e GPU.



Gestione della Memoria in CUDA

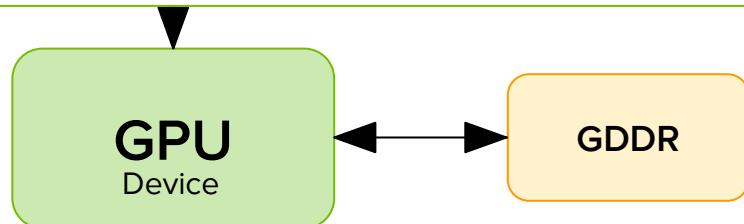
Modello di Memoria CUDA

- Il modello CUDA presuppone un sistema con un **host** e un **device**, ognuno con la propria memoria.
- La **comunicazione** tra la memoria dell'host e quella del device avviene tramite il **bus seriale PCIe**.

Confronto Larghezza di Banda per Direzione PCIe

Numero di Lanes	PCIe 1.0 (2003)	PCIe 2.0 (2007)	PCIe 3.0 (2010)	PCIe 4.0 (2017)	PCIe 5.0 (2019)	PCIe 6.0 (2021)
x1	250 MB/s	500 MB/s	1 GB/s	2 GB/s	4 GB/s	8 GB/s
x2	500 MB/s	1 GB/s	2 GB/s	4 GB/s	8 GB/s	16 GB/s
x4	1 GB/s	2 GB/s	4 GB/s	8 GB/s	16 GB/s	32 GB/s
x8	2 GB/s	4 GB/s	8 GB/s	16 GB/s	32 GB/s	64 GB/s
x16	4 GB/s	8 GB/s	16 GB/s	32 GB/s	64 GB/s	128 GB/s

- Bassa Latenza:** Garantisce comunicazioni rapide e reattive nei trasferimenti frequenti.
- Collo di Bottiglia:** Può diventare un collo di bottiglia in trasferimenti di grandi volumi tra CPU e GPU.

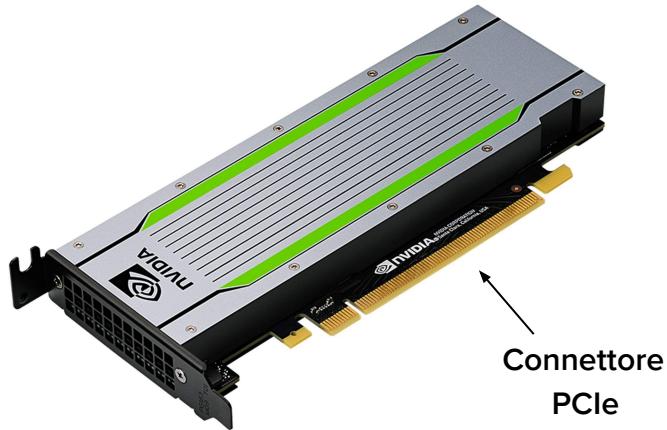


Collegamento Fisico della GPU tramite PCIe

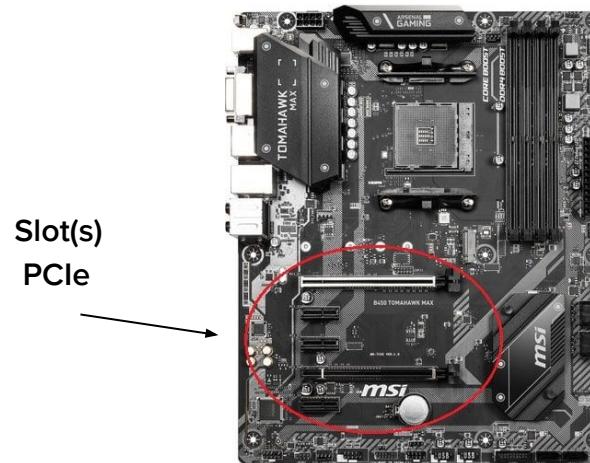
Connessione Fisica GPU

- La GPU si connette **fisicamente** alla scheda madre tramite un **connettore PCIe**.
- Questo connettore è una serie di contatti metallici dorati situati lungo il bordo della GPU, che si inseriscono nello **slot PCIe** della scheda madre.
- La maggior parte delle schede madri moderne ha **uno o più slot PCIe**, con almeno uno slot PCIe x16 destinato alla GPU.

Graphic Processing Unit (GPU)



Scheda Madre



Gestione della Memoria in CUDA

Modello di Memoria CUDA

- I kernel CUDA operano **sulla memoria del device**.
- CUDA Runtime fornisce funzioni per:
 - **Allocare memoria** sul device.
 - **Rilasciare memoria** sul device quando non più necessaria.
 - **Trasferire dati** bidirezionalmente tra la memoria dell'host e quella del device.

Standard C	CUDA C	Funzione
malloc	cudaMalloc	Alloc memoria dinamica
memcpy	cudaMemcpy	Copia dati tra aree di memoria
memset	cudaMemset	Inizializza memoria a un valore specifico
free	cudaFree	Libera memoria allocata dinamicamente

Nota Importante: È responsabilità del programmatore gestire correttamente l'allocazione, il trasferimento e la deallocazione della memoria per ottimizzare le prestazioni.

Gestione della Memoria in CUDA

Gerarchia di Memoria

In CUDA, esistono **diversi tipi di memoria**, ciascuno con caratteristiche specifiche in termini di accesso, velocità, e visibilità. Per ora, ci concentriamo su due delle più importanti:

Global Memory

- Accessibile da tutti i thread su tutti i blocchi
- Più grande ma più lenta rispetto alla shared memory
- Persiste per tutta la durata del programma CUDA
- È adatta per memorizzare dati grandi e persistenti

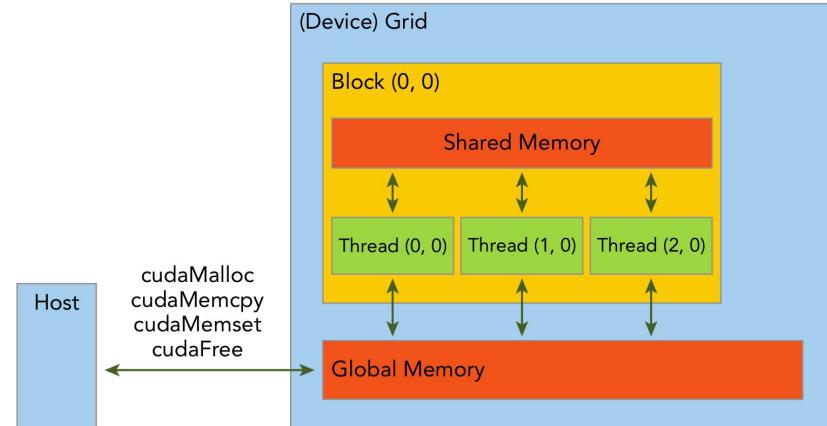
Shared Memory

- Condivisa tra i thread all'interno di un singolo blocco
- Più veloce, ma limitata in dimensioni
- Esiste solo per la durata del blocco di thread
- Utilizzata per dati temporanei e intermedi

Funzioni

- **cudaMalloc**: Alloca memoria sulla GPU.
- **cudaMemcpy**: Trasferisce dati tra host e device.
- **cudaMemset**: Inizializza la memoria del device.
- **cudaFree**: Libera la memoria allocata sul device.

Nota: Queste funzioni operano principalmente sulla Global Memory.



Allocazione della Memoria sul Device

Ruolo della Funzione

- **cudaMalloc** è una funzione CUDA utilizzata per allocare memoria sulla GPU (device).

Firma della Funzione ([Documentazione Online](#))

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

Parametri

- **devPtr**: Puntatore doppio che conterrà l'indirizzo della memoria allocata sulla GPU.
- **size**: Dimensione in byte della memoria da allocare.

Valore di Ritorno

- **cudaError_t**: Codice di errore (**cudaSuccess** se l'allocazione ha successo).

Note Importanti

- **Allocazione**: Riserva memoria lineare contigua sulla GPU a **runtime**.
- **Puntatore**: Aggiorna puntatore CPU con indirizzo memoria GPU.
- **Stato iniziale**: La memoria allocata non è inizializzata.

Allocazione della Memoria sul Device

Ruolo della Funzione

- **cudaMemset** è una funzione CUDA utilizzata per impostare un valore specifico in un blocco di memoria allocato sulla GPU (device).

Firma della Funzione [\(Documentazione Online\)](#)

```
cudaError_t cudaMemset(void* devPtr, int value, size_t count)
```

Parametri

- **devPtr**: Puntatore alla memoria allocata sulla GPU.
- **value**: Valore da impostare in ogni byte della memoria.
- **count**: Numero di byte della memoria da impostare al valore specificato.

Valore di Ritorno

- **cudaError_t**: Codice di errore (**cudaSuccess** se l'inizializzazione ha successo).

Note Importanti

- **Utilizzo**: Comunemente utilizzata per azzerare la memoria (impostando **value** a 0).
- **Gestione**: L'inizializzazione deve avvenire dopo l'allocazione della memoria tramite **cudaMalloc**.
- **Efficienza**: È preferibile usare **cudaMemset** per grandi blocchi di memoria per ridurre l'overhead.

Allocazione della Memoria sul Device

Esempio di Allocazione di Memoria sulla GPU

- Mostra come allocare memoria sulla GPU utilizzando `cudaMalloc`.

```
float* d_array; // Dichiarazione di un puntatore per la memoria sul device (GPU)

size_t size = 10 * sizeof(float); // Calcola la dimensione della memoria da allocare (10 float)

// Allocazione della memoria sul device
cudaError_t err = cudaMalloc((void**)&d_array, size);

// Controlla se l'allocazione della memoria ha avuto successo
if (err != cudaSuccess) {
    // Se c'è un errore, stampa un messaggio di errore con la descrizione dell'errore
    printf("Errore nell'allocazione della memoria: %s\n", cudaGetErrorString(err));
} else {
    // Se l'allocazione ha successo, stampa un messaggio di conferma
    printf("Memoria allocata con successo sulla GPU.\n");
}
```

Trasferimento Dati

Ruolo della Funzione

- **cudaMemcpy** è una funzione CUDA per il trasferimento di dati tra la memoria dell'host e del device, o all'interno dello stesso tipo di memoria.

Firma della Funzione ([Documentazione Online](#))

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

Parametri

- **dst**: Puntatore alla memoria di destinazione.
- **src**: Puntatore alla memoria sorgente.
- **count**: Numero di byte da copiare.
- **kind**: Direzione della copia.

Tipi di Trasferimento (kind)

- **cudaMemcpyHostToHost**: Da host a host
- **cudaMemcpyHostToDevice**: Da host a device
- **cudaMemcpyDeviceToHost**: Da device a host
- **cudaMemcpyDeviceToDevice**: Da device a device

Valore di Ritorno

- **cudaError_t**: Codice di errore (**cudaSuccess** se il trasferimento ha successo).

Note importanti

- **Funzione sincrona**: blocca l'host fino al completamento del trasferimento.
- Per prestazioni ottimali, minimizzare i trasferimenti tra host e device.

Trasferimento Dati

Ruolo della Funzione

- `cudaMemcpy`: trasferisce dati dal device all'host.

Firma della Funzione

`cudaError_t cudaMemcpy(...)`

Parametri

- `dst`: destinazione (host o device).
- `src`: sorgente (host o device).
- `count`: numero di byte da trasferire.
- `kind`: tipo di memoria (può essere specificato).

Valore di Ritorno

- `cudaError_t`: codice di errore.

Note importanti

- **Funzione sincrona:** blocca l'host fino al completamento del trasferimento.
- Per prestazioni ottimali, minimizzare i trasferimenti tra host e device.

Spazi di Memoria Differenti

- **Attenzione:** I puntatori del device non devono essere dereferenziati nel codice host (spazi di memoria CPU e GPU differenti).
- **Esempio:** Assegnazione errata come:

`host_array = dev_ptr`

invece di

`cudaMemcpy(host_array, dev_ptr, nBytes, cudaMemcpyDeviceToHost)`

- **Conseguenza dell'errore:** L'applicazione potrebbe bloccarsi durante l'esecuzione a causa del tentativo di accesso a uno spazio di memoria non valido.
- **Soluzione:** CUDA 6 ha introdotto la Memoria Unificata (Unified Memory), che consente di accedere sia alla memoria CPU che GPU utilizzando un unico puntatore (lo vedremo).

e, o

ice

Deallocazione della Memoria sul Device

Ruolo della Funzione

- **cudaFree** è una funzione CUDA utilizzata per liberare la memoria precedentemente allocata sulla GPU (device).

Firma della Funzione ([Documentazione Online](#))

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

Parametri

- **devPtr**: Puntatore alla memoria sul device che deve essere liberata. Questo puntatore deve essere stato precedentemente restituito tramite la chiamata **cudaMalloc**.

Valore di Ritorno

- **cudaError_t**: Codice di errore (**cudaSuccess** se la deallocazione ha successo).

Note Importanti

- **Gestione**: È responsabilità del programmatore assicurarsi che ogni blocco di memoria allocato con **cudaMalloc** sia liberato per evitare perdite di memoria (memory leaks) sulla GPU.
- **Efficienza**: La deallocazione della memoria può avere un overhead significativo, pertanto è consigliato minimizzare il numero di chiamate.

Allocazione e Trasferimento Dati sul Device

Esempio di Allocazione e Trasferimento Dati (1/2)

- Mostra come **allocare** e **trasferire** dati dalla memoria host alla memoria device.

```
size_t size = 10 * sizeof(float); // Calcola la dimensione della memoria da allocare (10 float)
float* h_data = (float*)malloc(size); // Alloca memoria sull'host (CPU) per memorizzare i dati
for (int i = 0; i < 10; ++i) h_data[i] = (float)i; // Inizializza ogni elemento di h_data

float* d_data; // Dichiarazione di un puntatore per la memoria sulla GPU (device)
cudaMalloc((void**)&d_data, size); // Allocazione della memoria sulla GPU

// Copia dei dati dalla memoria dell'host (CPU) alla memoria del device (GPU)
cudaError_t err = cudaMemcpy(d_data, h_data, size, cudaMemcpyHostToDevice);

// Controlla se la copia è avvenuta con successo
if (err != cudaSuccess) {
    // Se c'è un errore, stampa un messaggio di errore e termina il programma
    fprintf(stderr, "Errore nella copia H2D: %s\n", cudaGetErrorString(err));
    exit(EXIT_FAILURE);
}
// continua
```

Allocazione e Trasferimento Dati sul Device

Esempio di Allocazione e Trasferimento Dati (2/2)

- Mostra come **allocare** e **trasferire** dati dalla memoria host alla memoria device

```
// Esegui operazioni sulla memoria della GPU (d_data)
// (Le operazioni specifiche da eseguire non sono mostrate in questo esempio)

// Copia dei risultati dalla memoria della GPU (device) alla memoria dell'host (CPU)
err = cudaMemcpy(h_data, d_data, size, cudaMemcpyDeviceToHost);

// Controlla se la copia è avvenuta con successo
if (err != cudaSuccess) {
    fprintf(stderr, "Errore nella copia D2H: %s\n", cudaGetErrorString(err));
    exit(EXIT_FAILURE);
}

free(h_data); // Libera la memoria allocata sull'host
cudaFree(d_data); // Libera la memoria allocata sulla GPU
```

Panoramica del CUDA Programming Model

- **Introduzione al Modello di Programmazione**
 - Concetti base e architettura CUDA
 - Ruolo di Host (CPU) e Device (GPU)
- **Gestione della Memoria in CUDA - Accenni**
 - Allocazione e trasferimento di memoria
 - Tipi di memoria: globale, condivisa (menzioni)
- **Organizzazione dei Thread**
 - Gerarchie: Grid, Block, Thread
 - Identificazione dei thread
- **Kernel CUDA**
 - Definizione e lancio dei kernel
 - Configurazione di griglia e blocchi
- **Tecniche di Mapping e Dimensionamento**
 - Esempio: Somma di array e mapping degli indici
 - Calcolo dinamico delle dimensioni della griglia
- **Analisi delle Prestazioni**
 - Identificazione dei colli di bottiglia
 - Uso di strumenti di profiling (NVIDIA Nsight)
- **Applicazioni Pratiche**
 - Operazioni su matrici
 - Elaborazione di immagini (es. conversione RGB a grayscale)
 - Convoluzione 1D e 2D

Organizzazione dei Thread in CUDA

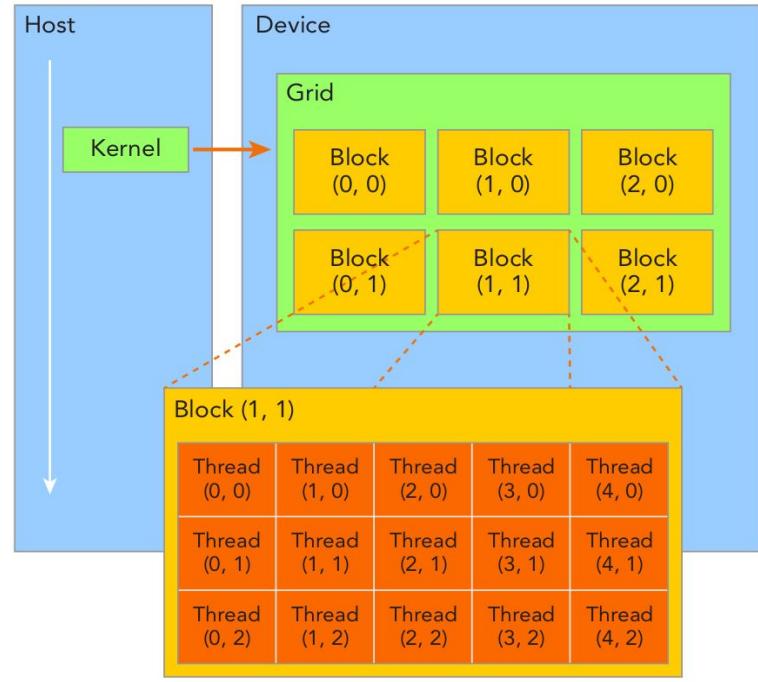
- CUDA adotta una gerarchia a due livelli per organizzare i thread basata su **blocchi di thread** e **griglie di blocchi**.

Struttura Gerarchica

- Grid (Griglia)**
 - Array di thread blocks.
 - È organizzata in una struttura **1D, 2D o 3D**.
 - Rappresenta l'intera **computazione** di un kernel.
 - Contiene **tutti i thread** che eseguono il **singolo kernel**.
 - Condivide lo stesso spazio di memoria globale.
- Block (Blocco)**
 - Un thread block è un gruppo di thread eseguiti **logicamente in parallelo**.
 - Ha un **ID** univoco all'interno della sua griglia.
 - I blocchi sono organizzati in una struttura **1D, 2D o 3D**.
 - I thread di un blocco possono **sincronizzarsi** (non automaticamente) e **condividere memoria**.
 - I **thread di blocchi diversi non possono cooperare**.

Thread

- Ha un proprio **ID** univoco all'interno del suo blocco.
- Ha accesso alla propria memoria privata (**registri**).



Perché una Gerarchia di Thread?

➤ Mappatura Intuitiva

- La gerarchia di thread (grid, blocchi, thread) permette di **scomporre problemi complessi** in unità di lavoro parallele più piccole e gestibili, rispecchiando spesso la struttura intrinseca del problema stesso.

➤ Organizzazione e Ottimizzazione

- Il programmatore può **controllare la dimensione** dei blocchi e della griglia per adattare l'esecuzione alle caratteristiche specifiche dell'hardware e del problema, ottimizzando l'utilizzo delle risorse.

➤ Efficienza nella Memoria

- I thread in un blocco condividono dati tramite memoria on-chip veloce (es. shared memory), **riducendo gli accessi alla memoria globale** più lenta, migliorando dunque significativamente le prestazioni.

➤ Scalabilità e Portabilità

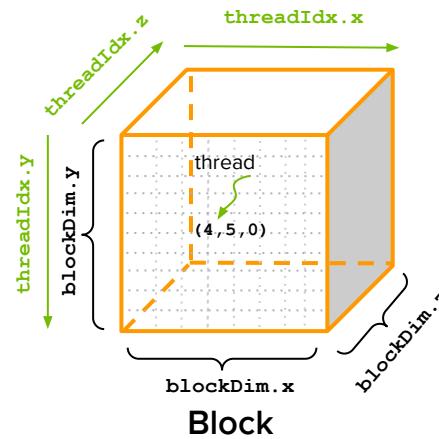
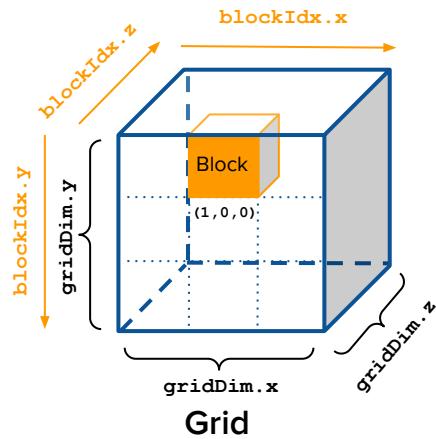
- La gerarchia è **scalabile** e permette di **adattare l'esecuzione** a GPU con diverse capacità e numero di core. Il codice CUDA, quindi, risulta più **portabile** e può essere eseguito su diverse architetture GPU.

➤ Sincronizzazione Granulare

- I thread possono essere sincronizzati solo **all'interno del proprio blocco**, evitando costose sincronizzazioni globali che possono creare colli di bottiglia.

Identificazione dei Thread in CUDA

- Ogni thread ha un'identità unica definita da **coordinate** specifiche all'interno della gerarchia grid-block. Queste coordinate, **private per ogni thread**, sono essenziali per l'esecuzione dei kernel e l'**accesso corretto ai dati**.



Un singolo thread di calcolo che opera in maniera indipendente

Thread

Variabili di Identificazione (Coordinate)

- blockIdx** (indice del blocco all'interno della griglia)
 - Componenti: `blockIdx.x`, `blockIdx.y`, `blockIdx.z`
- threadIdx** (indice del thread all'interno del blocco)
 - Componenti: `threadIdx.x`, `threadIdx.y`, `threadIdx.z`

Entrambe sono variabili **built-in** di tipo `uint3` **pre-inizializzate** dal CUDA Runtime e accessibili solo **all'interno del kernel**.

Variabili di Dimensioni

- blockDim** (dimensione del blocco in termini di thread)
 - Tipo: `dim3` (lato host), `uint3` (lato device, built-in)
 - Componenti: `blockDim.x`, `blockDim.y`, `blockDim.z`
- gridDim** (dimensione della griglia in termini di blocchi)
 - Tipo: `dim3` (lato host), `uint3` (lato device, built-in)
 - Componenti: `gridDim.x`, `gridDim.y`, `gridDim.z`

`uint3` è un built-in vector type di CUDA con tre campi (x,y,z) ognuno di tipo `unsigned int`

Identificazione dei Thread in CUDA

- Ogni thread ha un'identità unica definita da **coordinate** specifiche all'interno della gerarchia grid-block. Queste coordinate ai dati.



Variabili di Identificazione

1. **blockIdx** (indice del blocco)
 - Componenti: `blockIdx.x`, `blockIdx.y`, `blockIdx.z`
2. **threadIdx** (indice del thread all'interno del blocco)
 - Componenti: `threadIdx.x`, `threadIdx.y`, `threadIdx.z`

Entrambe sono variabili **built-in** di tipo `uint3` pre-inizializzate dal CUDA Runtime e accessibili solo **all'interno del kernel**.

Dimensione delle Griglie e dei Blocchi

- La scelta delle dimensioni ottimali dipende dalla struttura dati del task e dalle capacità hardware/risorse della GPU.
- Le variabili per le dimensioni di griglie e blocchi vengono definite nel codice host **prima di lanciare un kernel**.
- Sia le griglie che i blocchi utilizzano il tipo **dim3** (lato host) con tre campi `unsigned int`. I campi non utilizzati vengono inizializzati a 1 e ignorati.
- 9 possibili configurazioni** in tutto anche se in genere si usa la stessa per grid e block.

un built-in vector type di
con tre campi (x,y,z)
di tipo `unsigned int`

(read)
e, built-in)

1. **blockDim** (dimensione della griglia in termini di blocchi)
 - Componenti: `blockDim.x`, `blockDim.y`, `blockDim.z`
2. **gridDim** (dimensione della griglia in termini di blocchi)
 - Tipo: `dim3` (lato host), `uint3` (lato device, built-in)
 - Componenti: `gridDim.x`, `gridDim.y`, `gridDim.z`

Struttura dim3

Definizione

- **dim3** è una struttura definita in `vector_types.h` usata per specificare le dimensioni di griglia e blocchi.
- Supporta le **dimensioni 1, 2 e 3**:

- **Esempi**

```
dim3 gridDim(256); // Definisce una griglia di 256x1x1 blocchi.
```

```
dim3 blockDim(512, 512); // Definisce un blocco di 512x512x1 threads.
```

- Utilizzato per specificare le dimensioni di griglia e blocchi quando si lancia un kernel dal lato host:

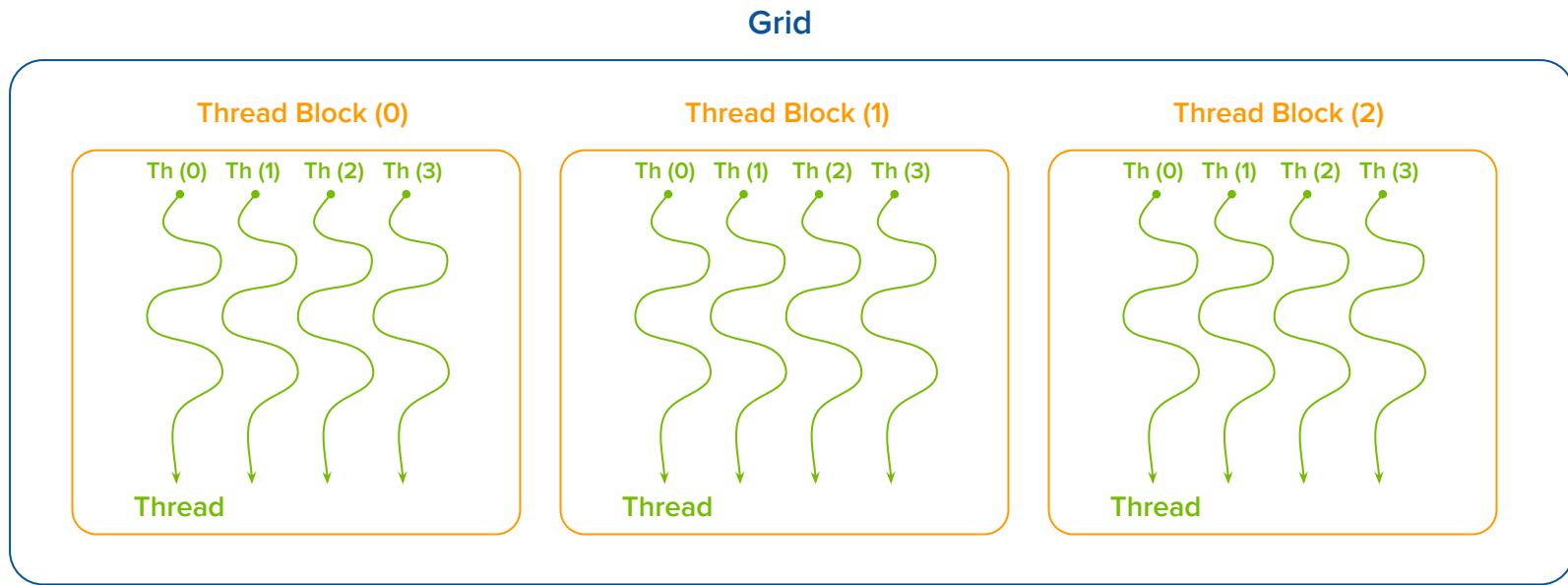
```
kernel_name<<<gridDim, blockDim>>>(....);
```

Codice Originale ([Link](#))

```
struct __device_builtin__ dim3
{
    unsigned int x, y, z;
#if defined(__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 */
};
```

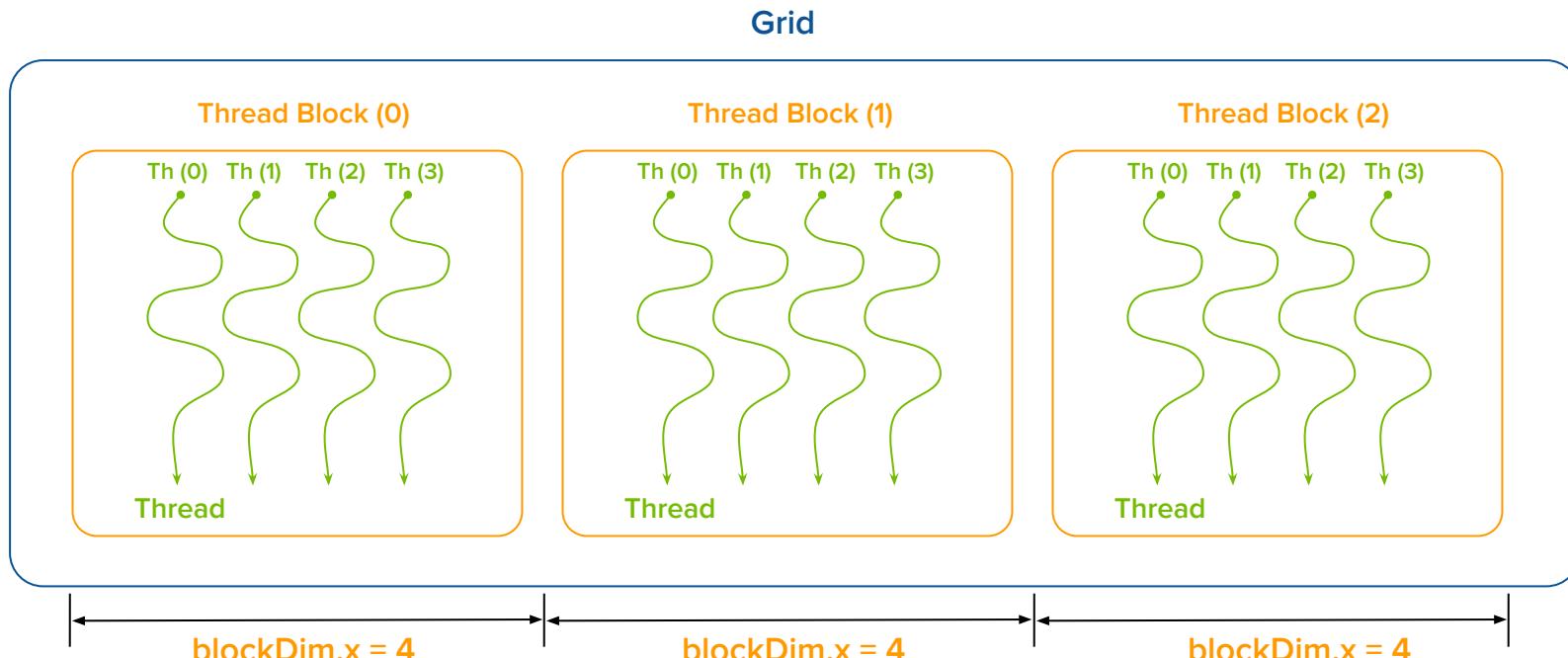
Identificazione dei Thread: Esempio Grid 1D, Block 1D

gridDim.x: Numero di blocchi nella griglia, in questo caso 3.



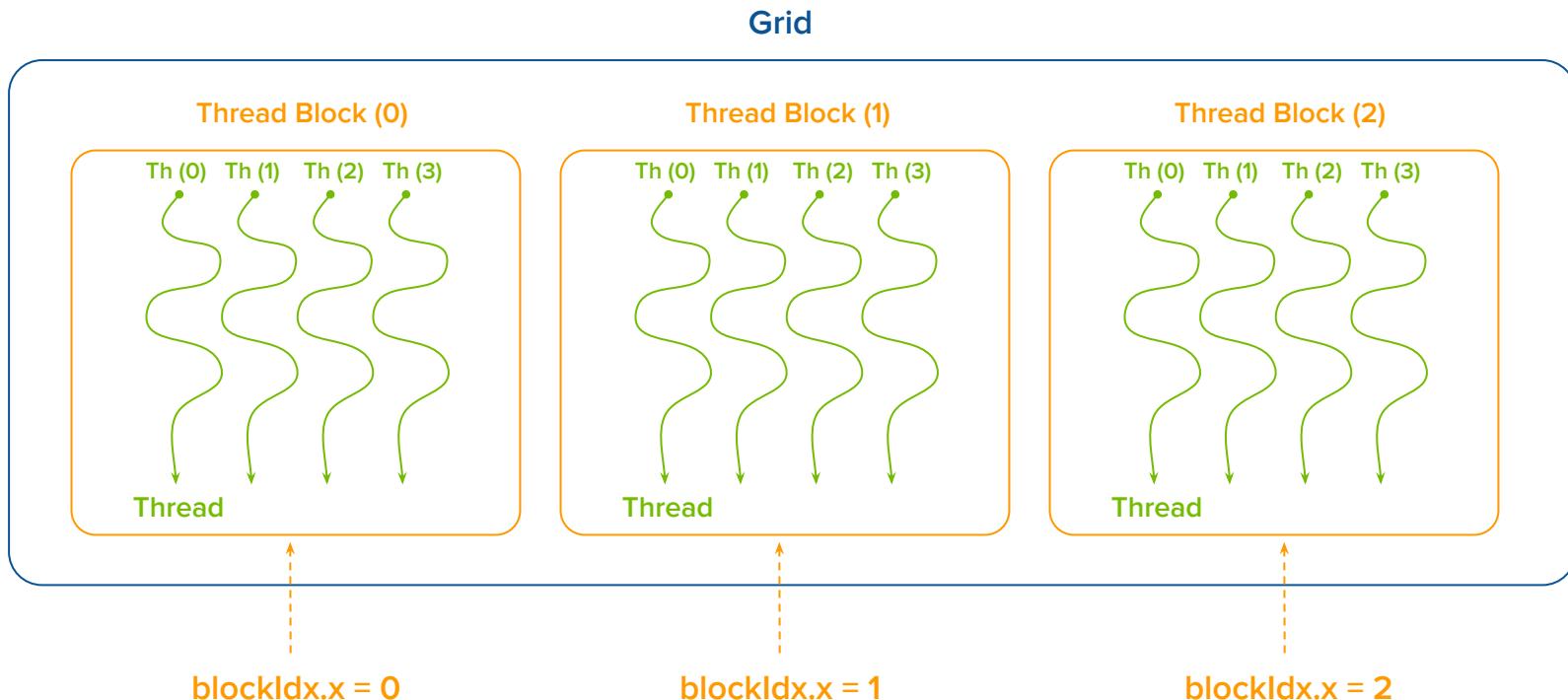
Identificazione dei Thread: Esempio Grid 1D, Block 1D

blockDim.x: Numero di thread per blocco, in questo caso 4.



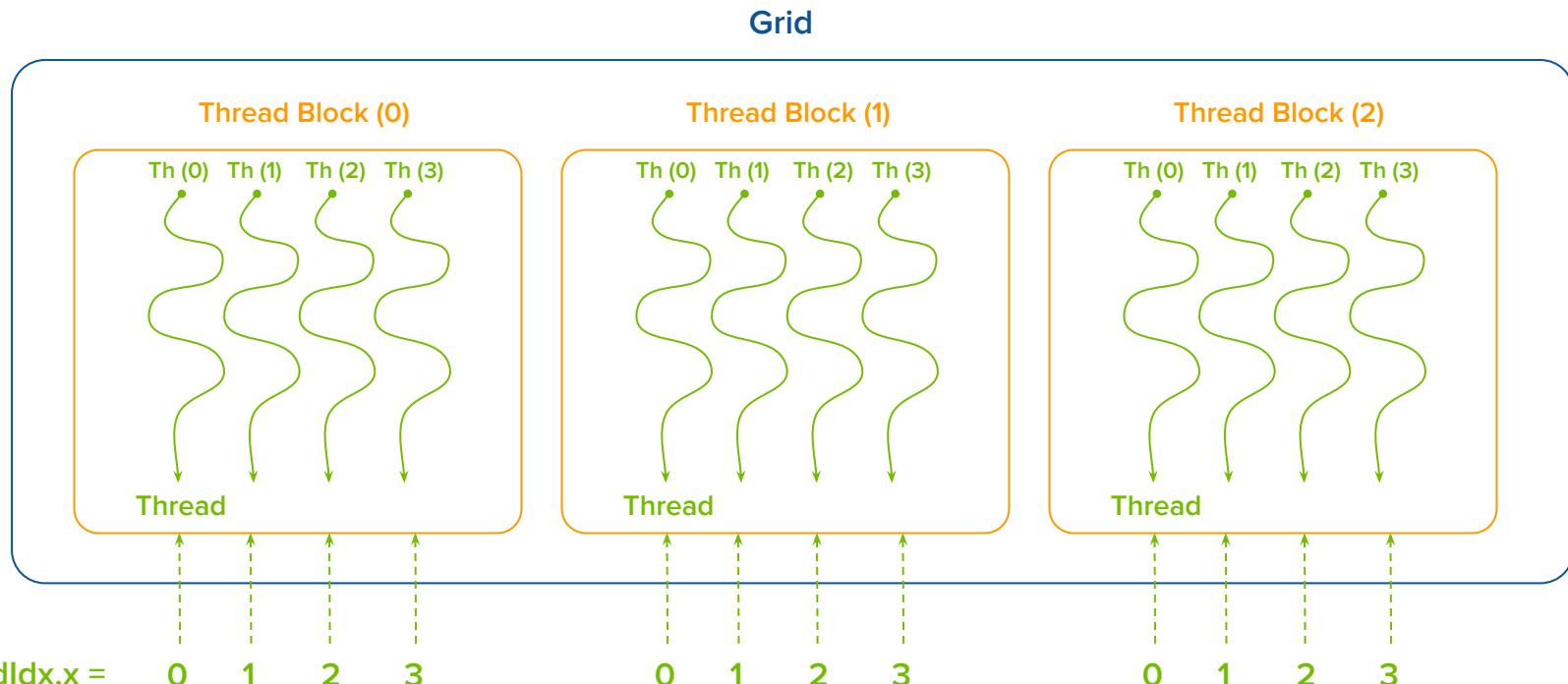
Identificazione dei Thread: Esempio Grid 1D, Block 1D

blockIdx.x: Indice di un blocco nella griglia.



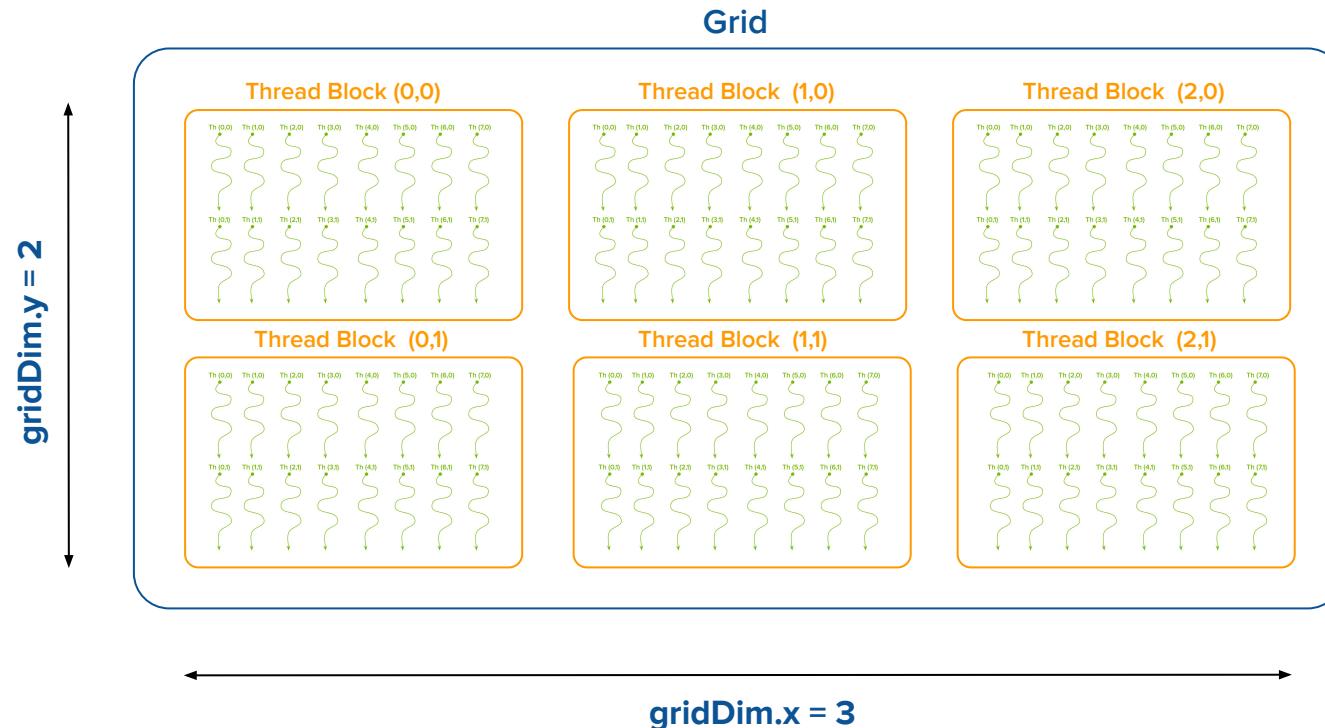
Identificazione dei Thread: Esempio Grid 1D, Block 1D

threadIdx.x: Indice del thread all'interno del blocco.



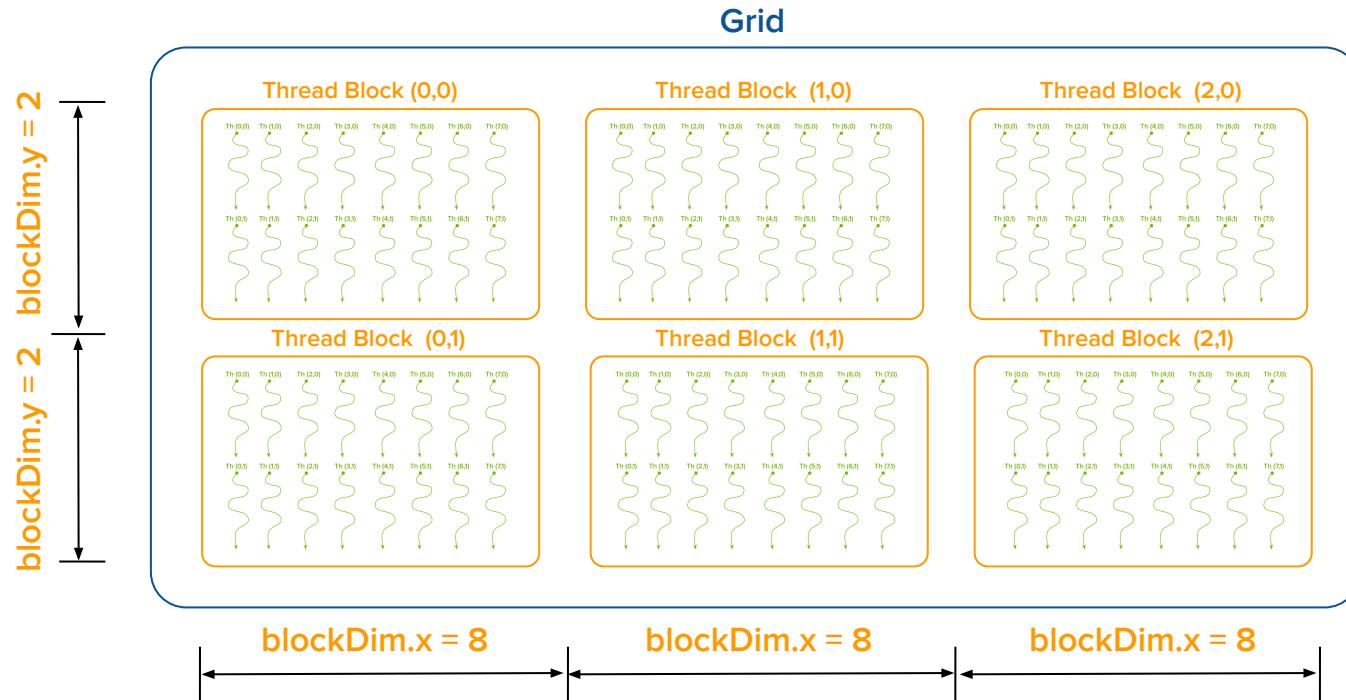
Identificazione dei Thread: Esempio Grid 2D, Block 2D

`gridDim.x, gridDim.y`: Numero di blocchi nella griglia lungo le dimensioni x e y.



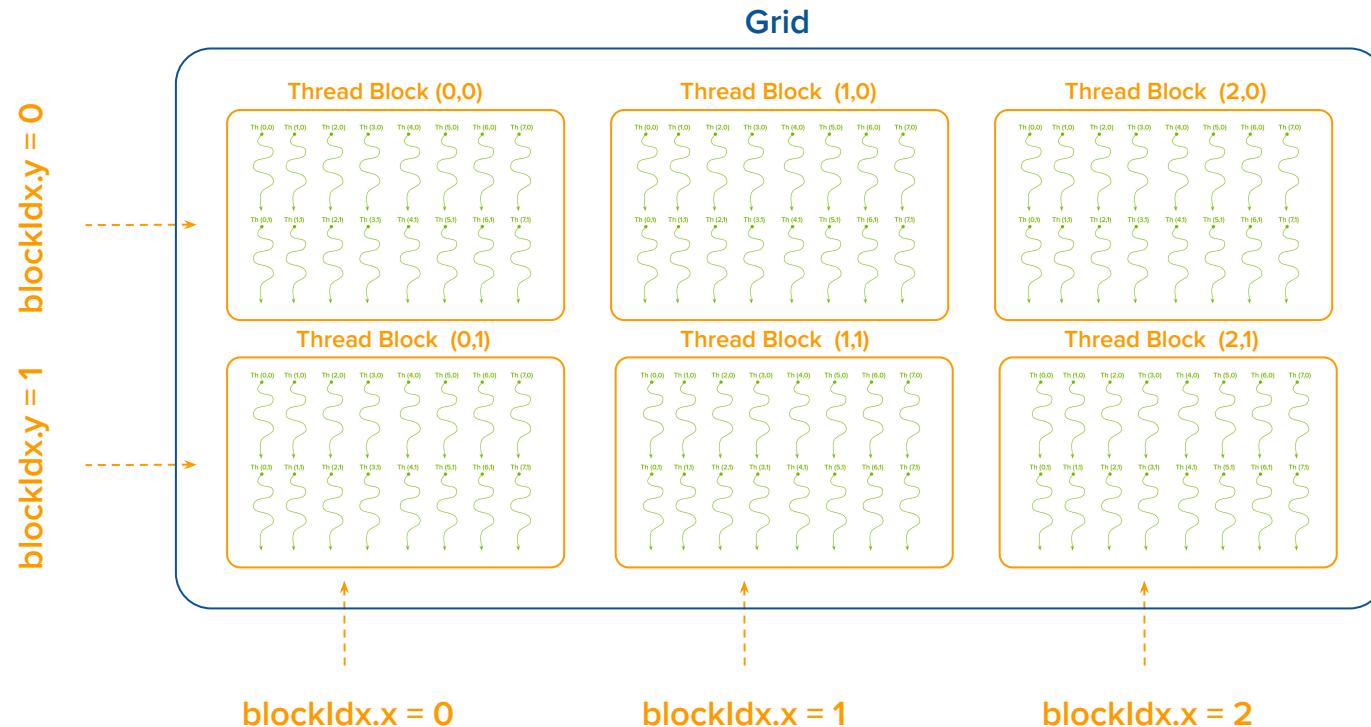
Identificazione dei Thread: Esempio Grid 2D, Block 2D

blockDim.x, blockDim.y: Numero di thread per blocco lungo le dimensioni x e y.



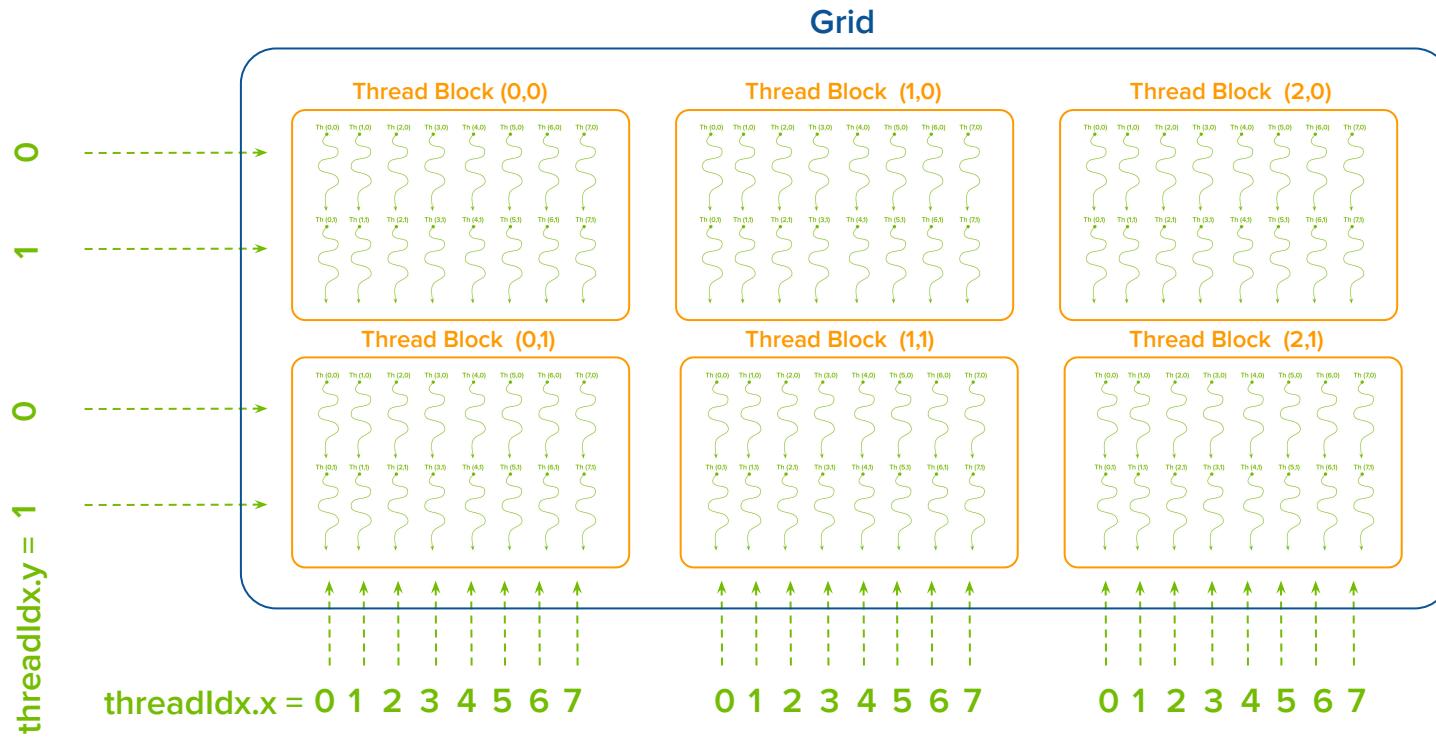
Identificazione dei Thread: Esempio Grid 2D, Block 2D

blockIdx.x, blockIdx.y: Indici del blocco lungo le dimensioni x e y della griglia.



Identificazione dei Thread: Esempio Grid 2D, Block 2D

`threadIdx.x, threadIdx.y`: Indici x e y del thread nel blocco 2D.



Panoramica del CUDA Programming Model

- **Introduzione al Modello di Programmazione**
 - Concetti base e architettura CUDA
 - Ruolo di Host (CPU) e Device (GPU)
- **Gestione della Memoria in CUDA - Accenni**
 - Allocazione e trasferimento di memoria
 - Tipi di memoria: globale, condivisa (menzioni)
- **Organizzazione dei Thread**
 - Gerarchie: Grid, Block, Thread
 - Identificazione dei thread
- **Kernel CUDA**
 - Definizione e lancio dei kernel
 - Configurazione di griglia e blocchi
- **Tecniche di Mapping e Dimensionamento**
 - Esempio: Somma di array e mapping degli indici
 - Calcolo dinamico delle dimensioni della griglia
- **Analisi delle Prestazioni**
 - Correttezza dei risultati e gestione degli errori
 - Uso di strumenti di profiling (NVIDIA Nsight)
- **Applicazioni Pratiche**
 - Operazioni su matrici
 - Elaborazione di immagini (es. conversione RGB a grayscale)
 - Convoluzione 1D e 2D

Esecuzione di un Kernel CUDA

Cos'è un Kernel CUDA?

- Un kernel CUDA è una **funzione** che viene eseguita in parallelo sulla GPU da **migliaia o milioni** di thread.
- Rappresenta il **nucleo computazionale** di un programma CUDA.
- Nei kernel viene definita la **logica di calcolo** per un singolo thread e l'**accesso ai dati** associati a quel thread.
- Ogni thread esegue lo **stesso codice kernel**, ma opera su **diversi elementi** dei dati.

Sintassi della chiamata Kernel CUDA

```
kernel_name <<<gridSize,blockSize>>>(argument list);
```

- **gridSize**: Dimensione della griglia (num. di blocchi).
- **blockSize**: Dimensione del blocco (num. di thread per blocco).
- **argument list**: Argomenti passati al kernel.

Sintassi Standard C

```
function_name (argument list);
```

- Con **gridSize** e **blockSize** si definisce:
 - Numero **totale** di thread per un kernel.
 - Il **layout** dei thread che si vuole utilizzare.

Come Eseguiamo il Codice in Parallelo sul Dispositivo?

Sequenziale (non ottimale): `kernel_name<<<1, 1>>>(args); // 1 blocco, 1 thread per blocco`

Parallelo: `kernel_name<<<256, 64>>>(args); // 256 blocchi, 64 thread per blocco`

Qualificatori di Funzione in CUDA

- I **qualificatori** di funzione in CUDA sono essenziali per specificare **dove una funzione verrà eseguita e da dove può essere chiamata**.

Qualificatore	Esecuzione	Chiamata	Note
<code>__global__</code>	Sul Device	Dall'Host	Deve avere tipo di ritorno <code>void</code>
<code>__device__</code>	Sul Device	Solo dal Device	
<code>__host__</code>	Sull'Host	Solo dall'Host	Può essere omesso

```
__global__ void kernelFunction(int *data, int size);
```

- Funzione kernel (eseguita sulla GPU, chiamabile solo dalla CPU).

```
__device__ int deviceHelper(int x);
```

- Funzione device (eseguita sulla GPU, chiamabile solo dalla GPU).

```
__host__ int hostFunction(int x);
```

- Funzione host (eseguibile su CPU).

Combinazione dei qualificatori host e device

In CUDA, combinando `__host__` e `__device__`, una funzione può essere eseguita sia sulla CPU che sulla GPU.

```
__host__ __device__ int hostDeviceFunction(int x);
```

Permette di scrivere una sola volta funzioni che possono essere utilizzate in entrambi i contesti.

Restrizioni dei Kernel CUDA

- 1. Esclusivamente Memoria Device (`__global__` e `__device__`)**
 - Accesso consentito solo alla memoria della GPU. Niente puntatori a memoria host.
- 2. Ritorno `void` (`__global__`)**
 - I kernel non restituiscono valori direttamente. La comunicazione con l'host avviene tramite la memoria.
- 3. Nessun supporto per argomenti variabili (`__global__` e `__device__`)**
 - Il numero di argomenti del kernel deve essere definito staticamente al momento della compilazione.
- 4. Nessun supporto per variabili statiche (`__global__` e `__device__`)**
 - Tutte le variabili devono essere passate come argomenti o allocate dinamicamente.
- 5. Nessun supporto per puntatori a funzione (`__global__` e `__device__`)**
 - Non è possibile utilizzare puntatori a funzione all'interno di un kernel.
- 6. Comportamento asincrono (`__global__`)**
 - I kernel vengono lanciati in modo asincrono rispetto al codice host, salvo sincronizzazioni esplicite.

Configurazioni di un Kernel CUDA

Griglie e Blocchi 1D, 2D e 3D

- La configurazione di **griglia** e **blocchi** può essere **1D**, **2D** o **3D** (9 combinazioni in totale), permettendo una mappatura efficiente (ed intuitiva) su **array**, **matrici** o **dati volumetrici**.

Combinazioni di Griglia 1D (Esempi)

```
// 1D Grid, 1D Block
dim3 gridSize(4);
dim3 blockSize(8);
kernel_name<<<gridSize, blockSize>>>(args);
```

```
// 1D Grid, 2D Block
dim3 gridSize(4);
dim3 blockSize(8, 4);
kernel_name<<<gridSize, blockSize>>>(args);
```

```
// 1D Grid, 3D Block
dim3 gridSize(4);
dim3 blockSize(8, 4, 2);
kernel_name<<<gridSize, blockSize>>>(args);
```

Adatta per:

- Problemi con dati strutturati linearmente, come l'elaborazione di **vettori** o **stringhe**, dove ogni thread può lavorare su una porzione contigua dei dati.

Nota: L'efficienza di una configurazione dipende da vari fattori come la **dimensione dei dati**, l'**architettura della GPU** e la **natura del problema**.

Configurazioni di un Kernel CUDA

Griglie e Blocchi 1D, 2D e 3D

- La configurazione di **griglia** e **blocchi** può essere **1D, 2D o 3D** (9 combinazioni in totale), permettendo una mappatura efficiente (ed intuitiva) su **array, matrici o dati volumetrici**.

Combinazioni di Griglia 2D (Esempi)

```
// 2D Grid, 1D Block
dim3 gridSize(4, 2);
dim3 blockSize(8);
kernel_name<<<gridSize, blockSize>>>(args);
```

```
// 2D Grid, 2D Block
dim3 gridSize(4, 2);
dim3 blockSize(8, 4);
kernel_name<<<gridSize, blockSize>>>(args);
```

```
// 2D Grid, 3D Block
dim3 gridSize(4, 2);
dim3 blockSize(8, 4, 2);
kernel_name<<<gridSize, blockSize>>>(args);
```

Adatta per:

- Ideale per problemi con dati strutturati in **matrici** o **immagini**, dove ogni thread può gestire un pixel o un elemento della matrice, sfruttando la vicinanza spaziale dei dati.

Nota: L'efficienza di una configurazione dipende da vari fattori come la **dimensione dei dati**, l'**architettura della GPU** e la **natura del problema**.

Configurazioni di un Kernel CUDA

Griglie e Blocchi 1D, 2D e 3D

- La configurazione di **griglia** e **blocchi** può essere **1D, 2D o 3D** (9 combinazioni in totale), permettendo una mappatura efficiente (ed intuitiva) su **array, matrici o dati volumetrici**.

Combinazioni di Griglia 3D (Esempi)

```
// 3D Grid, 1D Block
dim3 gridSize(4, 2, 2);
dim3 blockSize(8);
kernel_name<<<gridSize, blockSize>>>(args);
```

```
// 3D Grid, 2D Block
dim3 gridSize(4, 2, 2);
dim3 blockSize(8, 4);
kernel_name<<<gridSize, blockSize>>>(args);
```

```
// 3D Grid, 3D Block
dim3 gridSize(4, 2, 2);
dim3 blockSize(8, 4, 2);
kernel_name<<<gridSize, blockSize>>>(args);
```

Adatta per:

- Ottimale per problemi con **dati volumetrici**, come simulazioni fisiche o rendering 3D, dove ogni thread può operare su un voxel o una porzione dello spazio 3D.

Nota: L'efficienza di una configurazione dipende da vari fattori come la **dimensione dei dati**, l'**architettura della GPU** e la **natura del problema**.

Numero di Thread per Blocco

- Il numero massimo totale di thread per blocco è **1024** per la maggior parte delle GPU (compute capability $\geq 2.x$).
- Un blocco può essere organizzato in 1, 2 o 3 dimensioni, ma ci sono limiti per ciascuna dimensione. Esempio:
 - x: 1024 , y: 1024, z: 64**
- Il prodotto delle dimensioni x, y e z non può superare 1024 (queste limitazioni potrebbero cambiare in futuro).

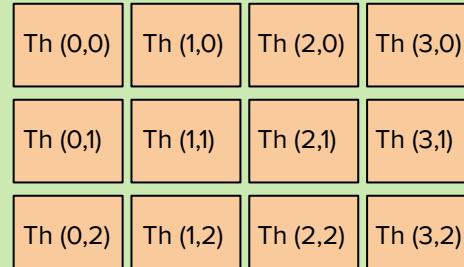
Block 1D



Esempi 1D

- (32, 1, 1)
- (96, 1, 1)
- (128, 1, 1)
- ...
- (1024, 1, 1)
- (2048, 1, 1) NO!**

Block 2D



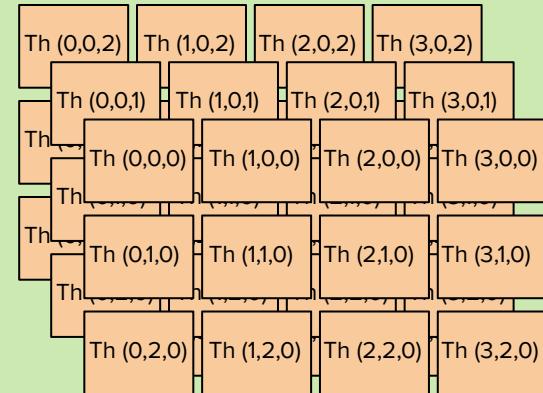
Esempi 2D

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

K = 1024

`(blockDim.x * blockDim.y * blockDim.z) <= K`

Block 3D



Esempi 3D

- (8, 8, 8)
- ...
- (64, 32, 1) NO!**

Compute Capability

- La **Compute Capability** di NVIDIA è un numero che identifica le **caratteristiche** e le **capacità** di una GPU NVIDIA in termini di funzionalità supportate e limiti hardware.
- È composta da **due numeri**: il numero principale indica la **generazione** dell'architettura, mentre il numero secondario indica **revisioni** e **miglioramenti** all'interno di quella generazione.

Compute Capability	Architettura	Max grid dimensionality	Max grid x-dimension	Max grid y/z-dimension	Max block dimensionality	Max block x/y-dimension	Max block z-dimension	Max threads per block
1.x	Tesla	2	65535	65535	3	512	64	512
2.x	Fermi	3	$2^{31}-1$	65535	3	1024	64	1024
3.x	Kepler	3	$2^{31}-1$	65535	3	1024	64	1024
5.x	Maxwell	3	$2^{31}-1$	65535	3	1024	64	1024
6.x	Pascal	3	$2^{31}-1$	65535	3	1024	64	1024
7.x	Volta/Turing	3	$2^{31}-1$	65535	3	1024	64	1024
8.x	Ampere/Ada	3	$2^{31}-1$	65535	3	1024	64	1024
9.x	Hopper	3	$2^{31}-1$	65535	3	1024	64	1024

https://en.wikipedia.org/wiki/CUDA#Version_features_and_specifications

Identificazione dei Thread in CUDA

Esempio Codice CUDA

```
#include <cuda_runtime.h>

// Kernel
__global__ void kernel_name() {
    // Accesso alle variabili built-in
    int blockDim_x = blockIdx.x, blockDim_y = blockIdx.y, blockDim_z = blockIdx.z;
    int threadIdx_x = threadIdx.x, threadIdx_y = threadIdx.y, threadIdx_z = threadIdx.z;
    int totalThreads_x = blockDim.x, totalThreads_y = blockDim.y, totalThreads_z = blockDim.z;
    int totalBlocks_x = gridDim.x, totalBlocks_y = gridDim.y, totalBlocks_z = gridDim.z;

    // Logica del kernel...
}

int main() {
    // Definizione delle dimensioni della griglia e del blocco (Caso 3D)
    dim3 gridDim(4, 4, 2); // 4x4x2 blocchi
    dim3 blockDim(8, 8, 4); // 8x8x4 thread per blocco

    // Lancio del kernel
    kernel_name<<<gridDim, blockDim>>>();
}

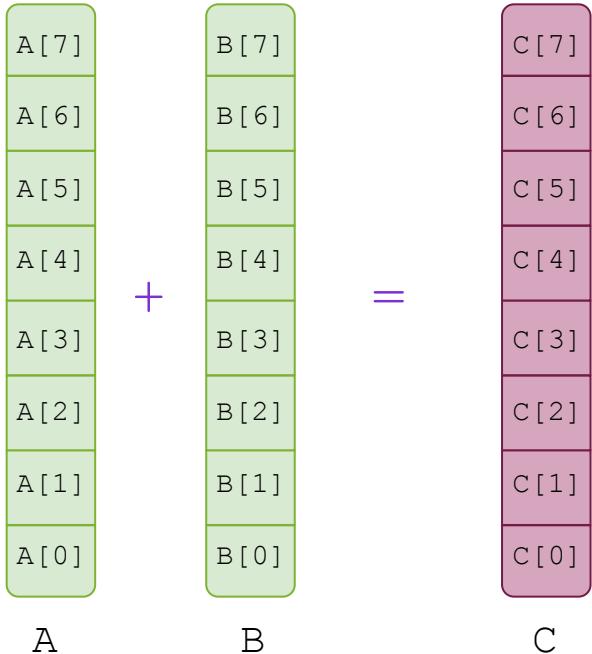
// Resto del Programma
}
```

Panoramica del CUDA Programming Model

- **Introduzione al Modello di Programmazione**
 - Concetti base e architettura CUDA
 - Ruolo di Host (CPU) e Device (GPU)
- **Gestione della Memoria in CUDA - Accenni**
 - Allocazione e trasferimento di memoria
 - Tipi di memoria: globale, condivisa (menzioni)
- **Organizzazione dei Thread**
 - Gerarchie: Grid, Block, Thread
 - Identificazione dei thread
- **Kernel CUDA**
 - Definizione e lancio dei kernel
 - Configurazione di griglia e blocchi
- **Tecniche di Mapping e Dimensionamento**
 - Esempio: Somma di array e mapping degli indici
 - Calcolo dinamico delle dimensioni della griglia
- **Analisi delle Prestazioni**
 - Correttezza dei risultati e gestione degli errori
 - Uso di strumenti di profiling (NVIDIA Nsight)
- **Applicazioni Pratiche**
 - Operazioni su matrici
 - Elaborazione di immagini (es. conversione RGB a grayscale)
 - Convoluzione 1D e 2D

Somma di Array in CUDA

Il Problema: Vogliamo **sommare due array** elemento per elemento in parallelo utilizzando CUDA.



Approccio Tradizionale (CPU)

- Gli elementi degli array vengono sommati **uno alla volta**.
- Questo approccio è **inefficiente** per array di grandi dimensioni.
- Utilizza solo **un core** della CPU, rallentando il processo.

Approccio CUDA (GPU)

- Gli elementi degli array vengono sommati **contemporaneamente**.
- La GPU è progettata per eseguire calcoli **paralleli** su larga scala.
- **Migliaia di core** della GPU lavorano insieme, accelerando enormemente il calcolo.

Confronto: Somma di Vettori in C vs CUDA C

Codice C Standard

```
void sumArraysOnHost(float *A, float *B,
float *C, int N) {
    for (int idx = 0; idx < N; idx++)
        C[idx] = A[idx] + B[idx];
}
// Chiamata della funzione
sumArraysOnHost(A, B, C, N);
```

Caratteristiche

- **Esecuzione:** Sequentiale
- **Iterazione:** Loop Esplicito
- **Indice:** Variabile di Loop (`idx`)
- **Scalabilità:** Limitata dalla CPU

Vantaggi

- Portabilità su qualsiasi sistema
- Facilità di debugging

Codice CUDA C

```
__global__ void sumArraysOnGPU(float *A, float *B,
float *C, int N) {
    int idx = ?; // Come accedere ai dati?
    if (idx < N) C[idx] = A[idx] + B[idx];
}
// Chiamata del kernel
sumArraysOnGPU<<<gridDim,blockDim>>>(A, B, C, N);
```

Caratteristiche

- **Esecuzione:** Parallela
- **Iterazione:** Implicita (un thread per elemento)
- **Indice:** ?
- **Scalabilità:** Elevata (sfrutta molti core GPU)

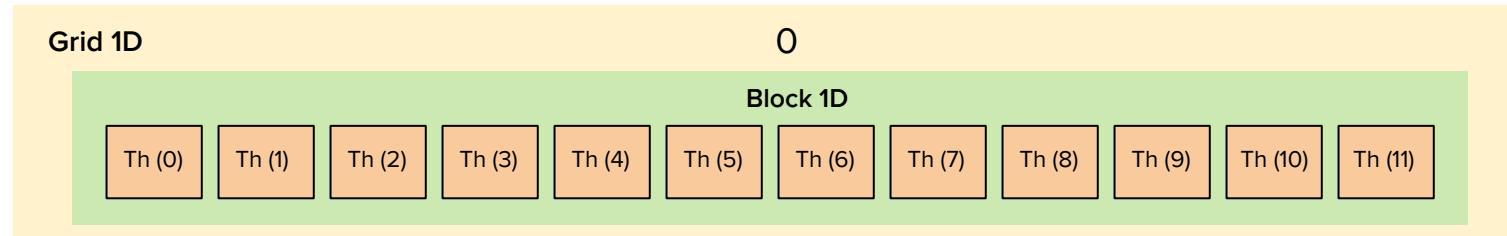
Vantaggi

- Altamente parallelo
- Eccellenti prestazioni su grandi dataset
- Sfrutta la potenza di calcolo delle GPU

Mapping degli Indici ai Dati in CUDA - Esempio 1D

- Come mappare gli indici dei thread agli elementi dell'array?

```
sumArraysOnGPU<<<1,12>>>(A, B, C)
```



Memoria GPU

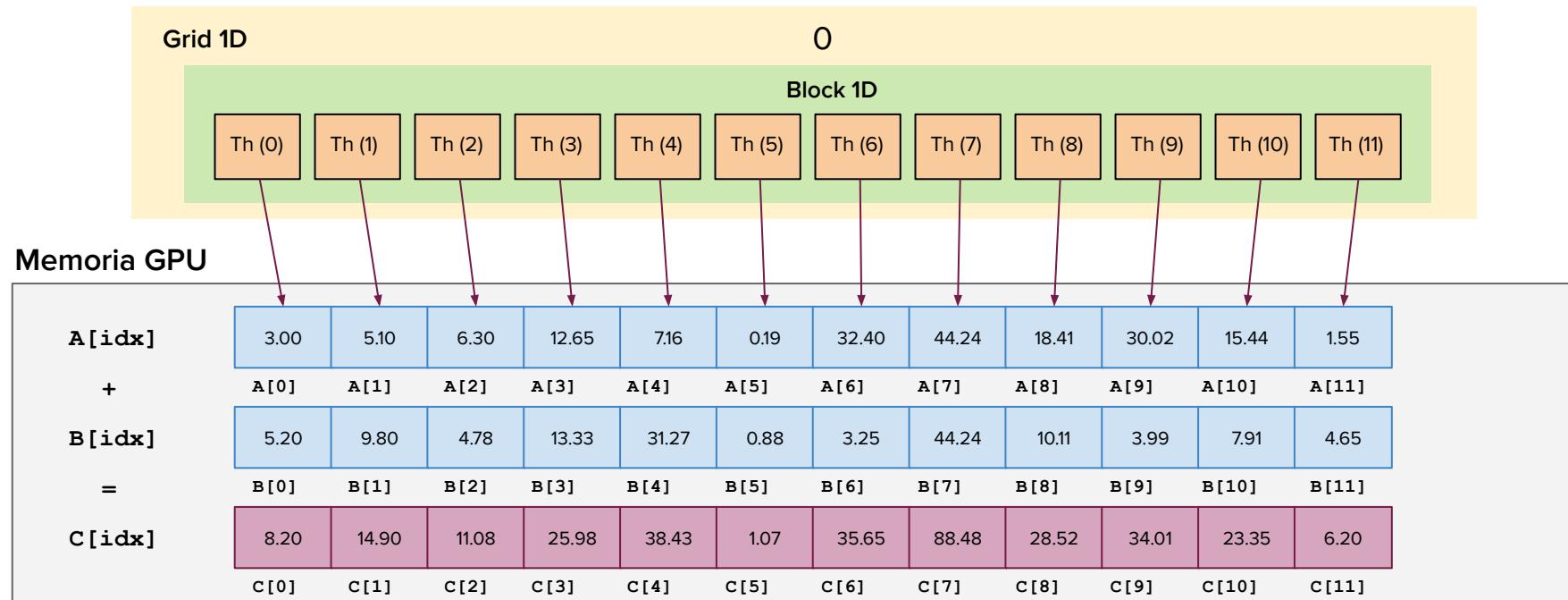
A[idx]	3.00	5.10	6.30	12.65	7.16	0.19	32.40	44.24	18.41	30.02	15.44	1.55
+	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]	A[10]	A[11]
B[idx]	5.20	9.80	4.78	13.33	31.27	0.88	3.25	44.24	10.11	3.99	7.91	4.65
=	B[0]	B[1]	B[2]	B[3]	B[4]	B[5]	B[6]	B[7]	B[8]	B[9]	B[10]	B[11]
C[idx]	?	?	?	?	?	?	?	?	?	?	?	?
	C[0]	C[1]	C[2]	C[3]	C[4]	C[5]	C[6]	C[7]	C[8]	C[9]	C[10]	C[11]

idx = ?

Mapping degli Indici ai Dati in CUDA - Esempio 1D

- Come mappare gli indici dei thread agli elementi dell'array?

```
sumArraysOnGPU<<<1,12>>>(A, B, C)
```



idx = threadIdx.x

OK! Ma..

Mapping degli Indici ai Dati in CUDA - Esempio 1D

- Come mappare gli indici dei thread agli elementi dell'array?

```
sumArraysOnGPU<<<1,12>>>(A, B, C)
```

Grid

Problemi Principali

- Scalabilità limitata:** Funziona solo per array di dimensione uguale o inferiore al numero massimo di thread per blocco (tipicamente 1024 su molte GPU).
- Mancanza di generalizzazione:** Questo approccio non si estende facilmente a problemi di dimensioni arbitrarie o a griglie multi-dimensionalni.
- Utilizzo inefficiente della GPU:** Questo approccio attiva solo uno dei molti processori disponibili sulla GPU, non sfruttando a pieno il parallelismo offerto.

Memoria GPU

```
A[idx]  
+  
B[idx]  
=  
C[idx]
```

8.20	14.90	11.08	25.98	38.43	1.07	35.65	88.48	28.52	34.01	23.35	6.20
C[0]	C[1]	C[2]	C[3]	C[4]	C[5]	C[6]	C[7]	C[8]	C[9]	C[10]	C[11]

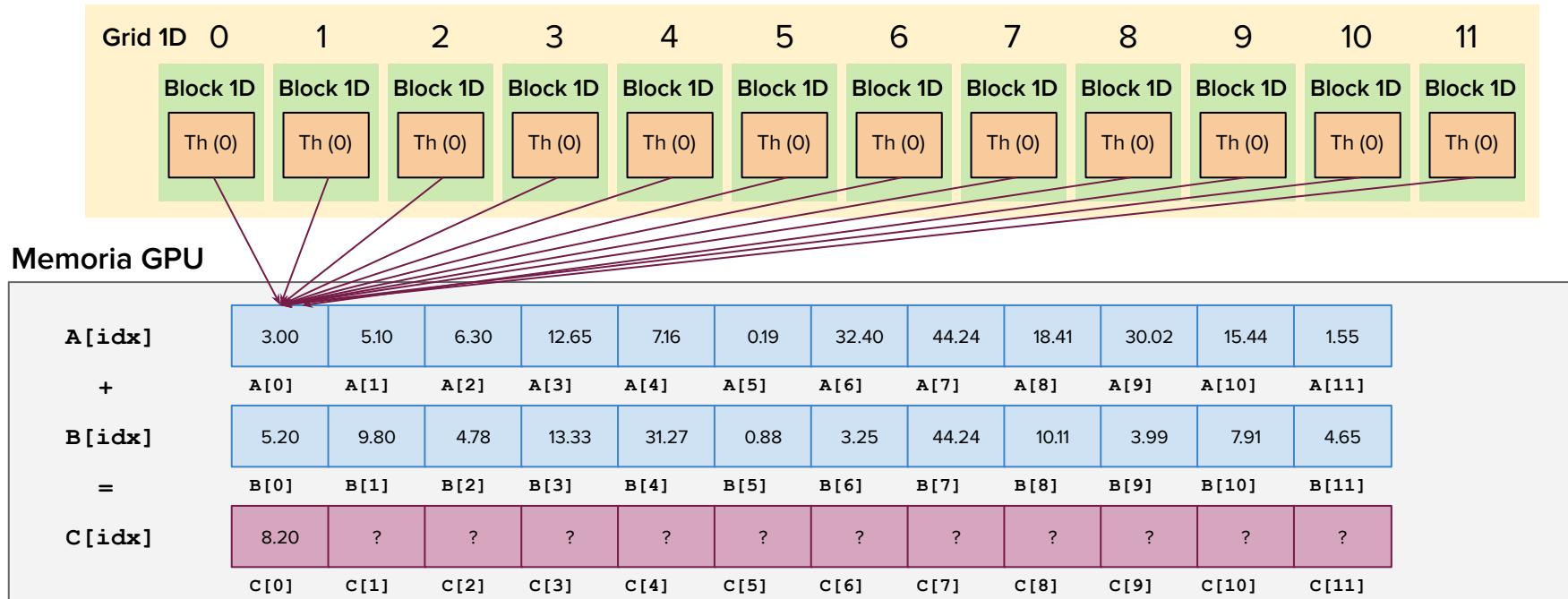
idx = threadIdx.x

OK! Ma..

Mapping degli Indici ai Dati in CUDA - Esempio 1D

- Come mappare gli indici dei thread agli elementi dell'array?

sumArraysOnGPU<<<12,1>>>(A, B, C)

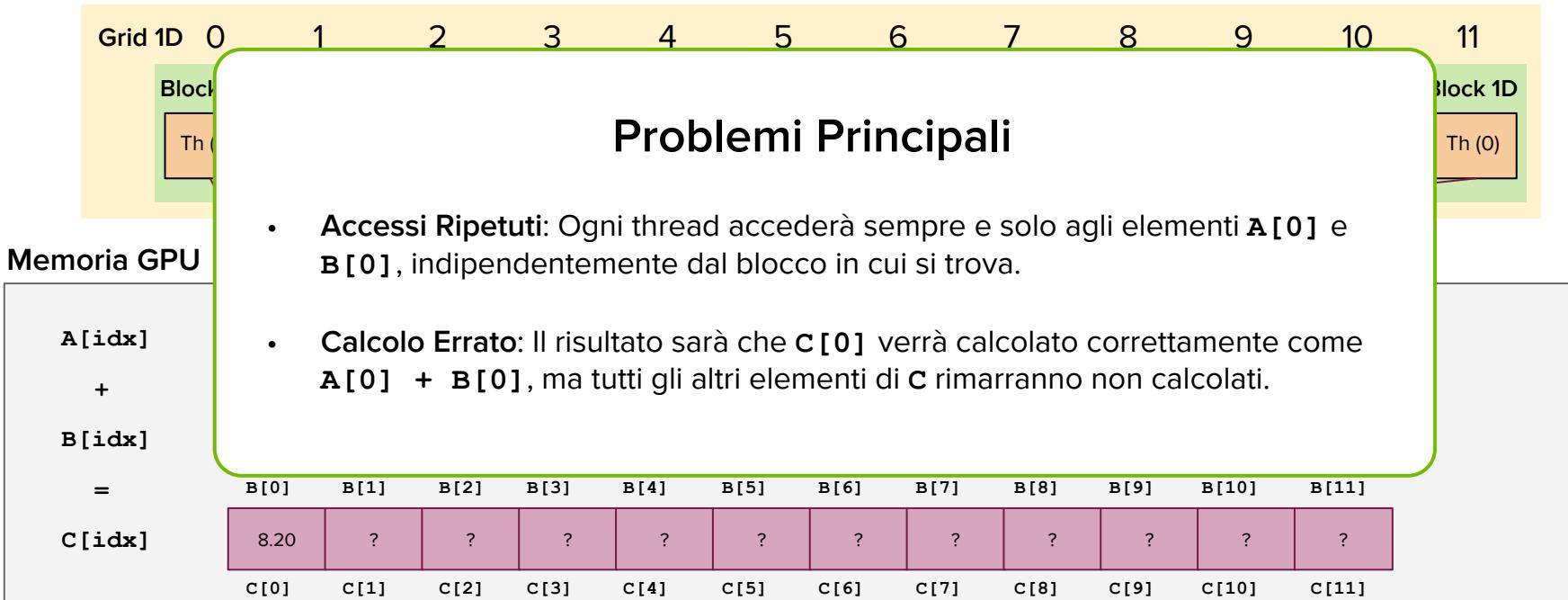


idx = threadIdx.x NO!

Mapping degli Indici ai Dati in CUDA - Esempio 1D

- Come mappare gli indici dei thread agli elementi dell'array?

sumArraysOnGPU<<<12,1>>>(A, B, C)

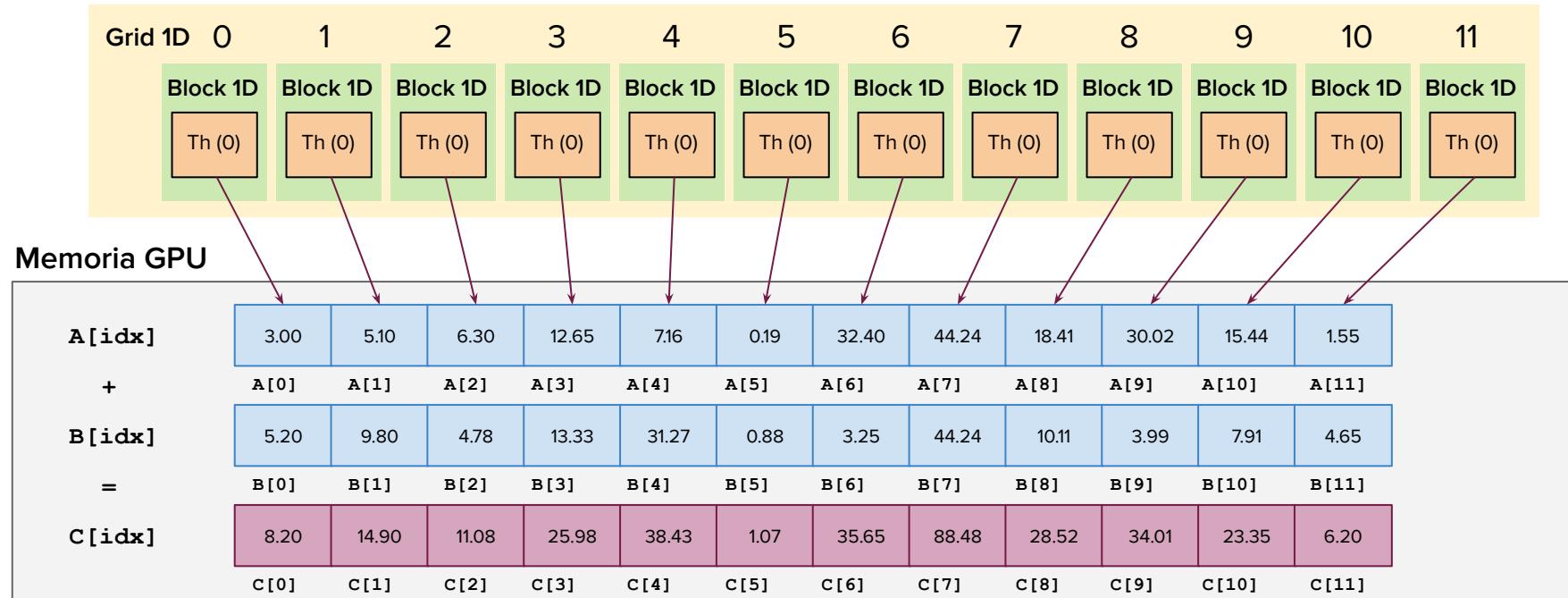


idx = threadIdx.x NO!

Mapping degli Indici ai Dati in CUDA - Esempio 1D

- Come mappare gli indici dei thread agli elementi dell'array?

sumArraysOnGPU<<<12,1>>>(A, B, C)



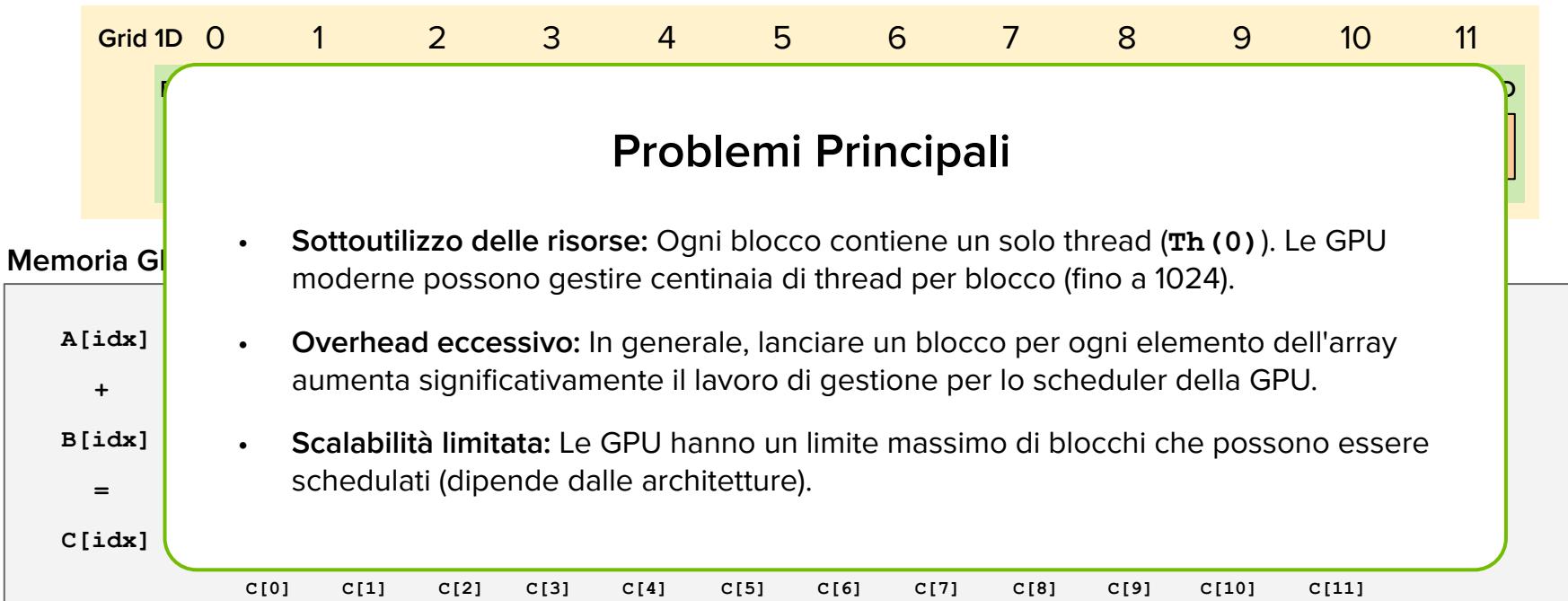
idx = blockIdx.x

OK! Ma..

Mapping degli Indici ai Dati in CUDA - Esempio 1D

- Come mappare gli indici dei thread agli elementi dell'array?

```
sumArraysOnGPU<<<12,1>>>(A, B, C)
```



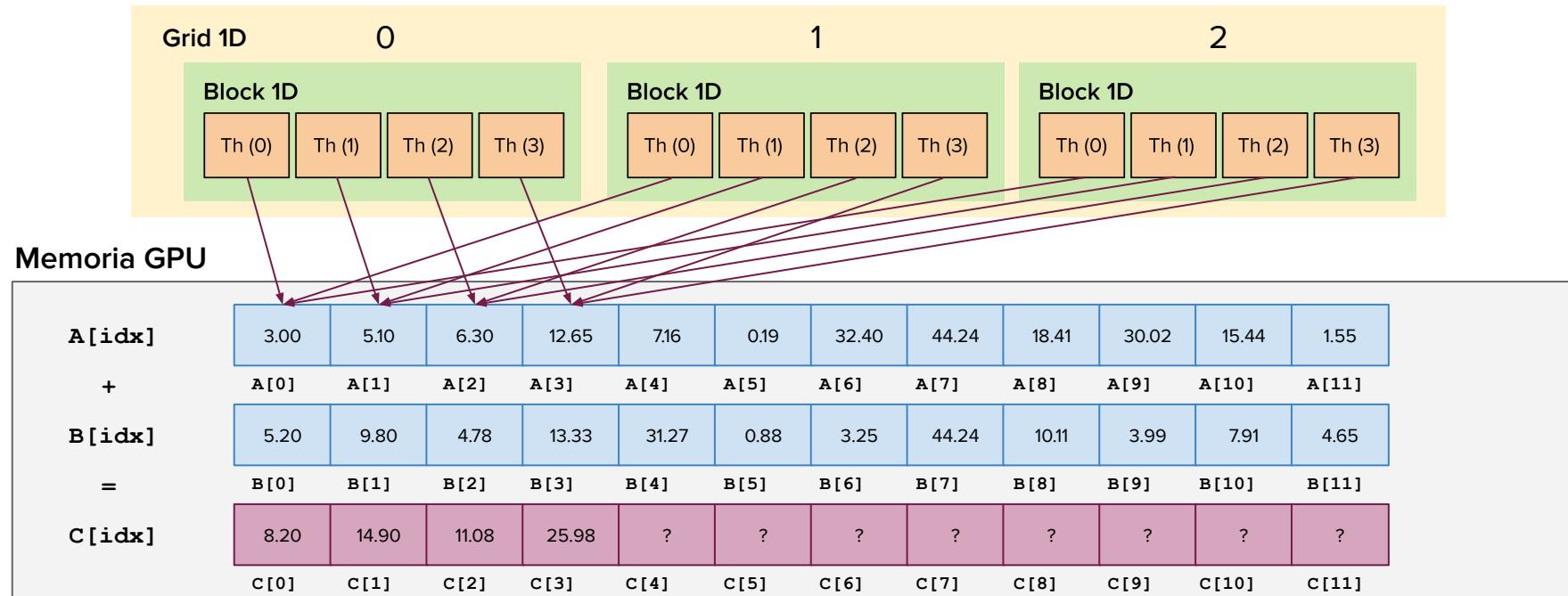
idx = blockIdx.x

OK! Ma..

Mapping degli Indici ai Dati in CUDA - Esempio 1D

- Come mappare gli indici dei thread agli elementi dell'array?

sumArraysOnGPU<<<3,4>>>(A, B, C)

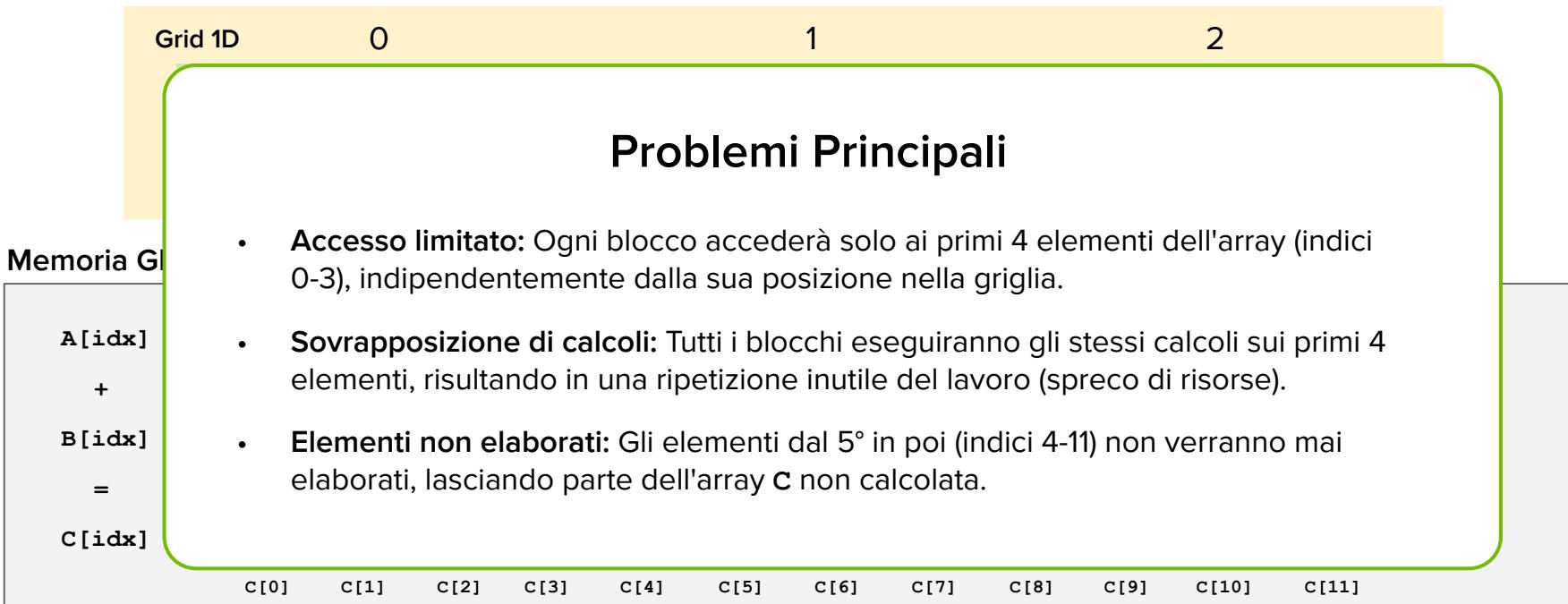


idx = threadIdx.x NO!

Mapping degli Indici ai Dati in CUDA - Esempio 1D

- Come mappare gli indici dei thread agli elementi dell'array?

```
sumArraysOnGPU<<<3,4>>>(A, B, C)
```

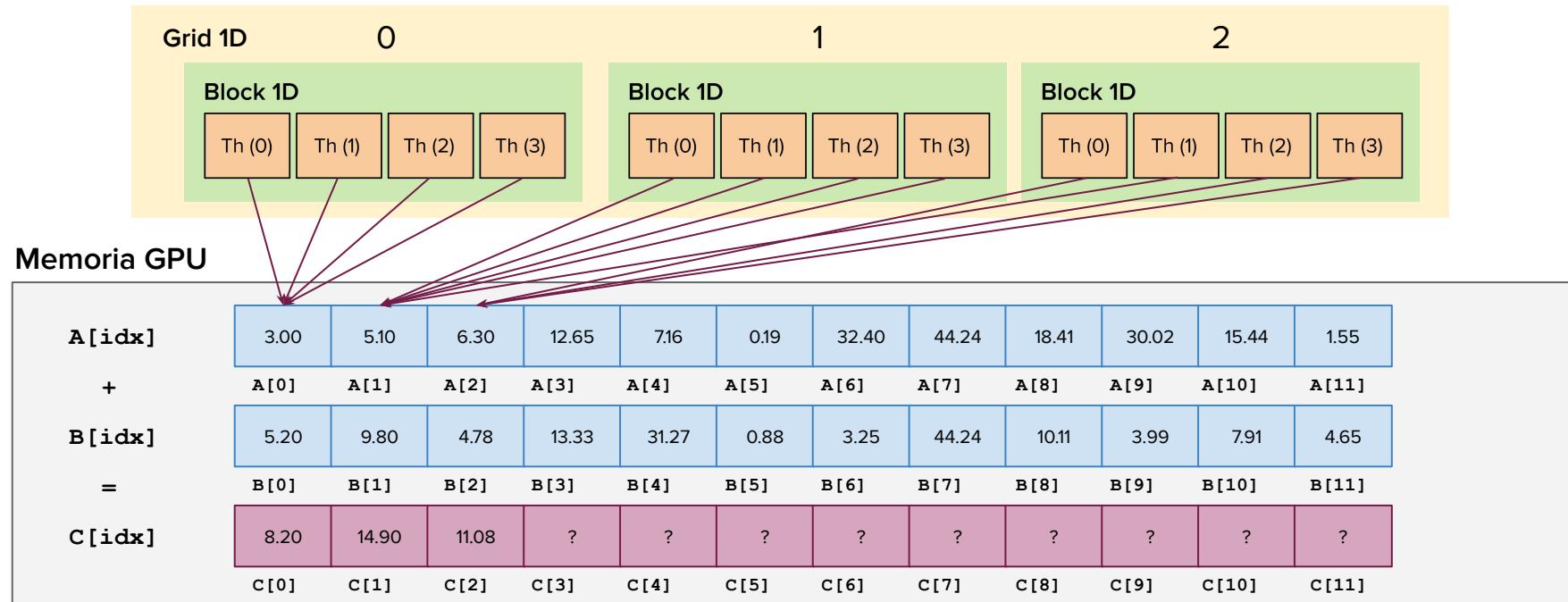


idx = threadIdx.x NO!

Mapping degli Indici ai Dati in CUDA - Esempio 1D

- Come mappare gli indici dei thread agli elementi dell'array?

sumArraysOnGPU<<<3,4>>>(A, B, C)



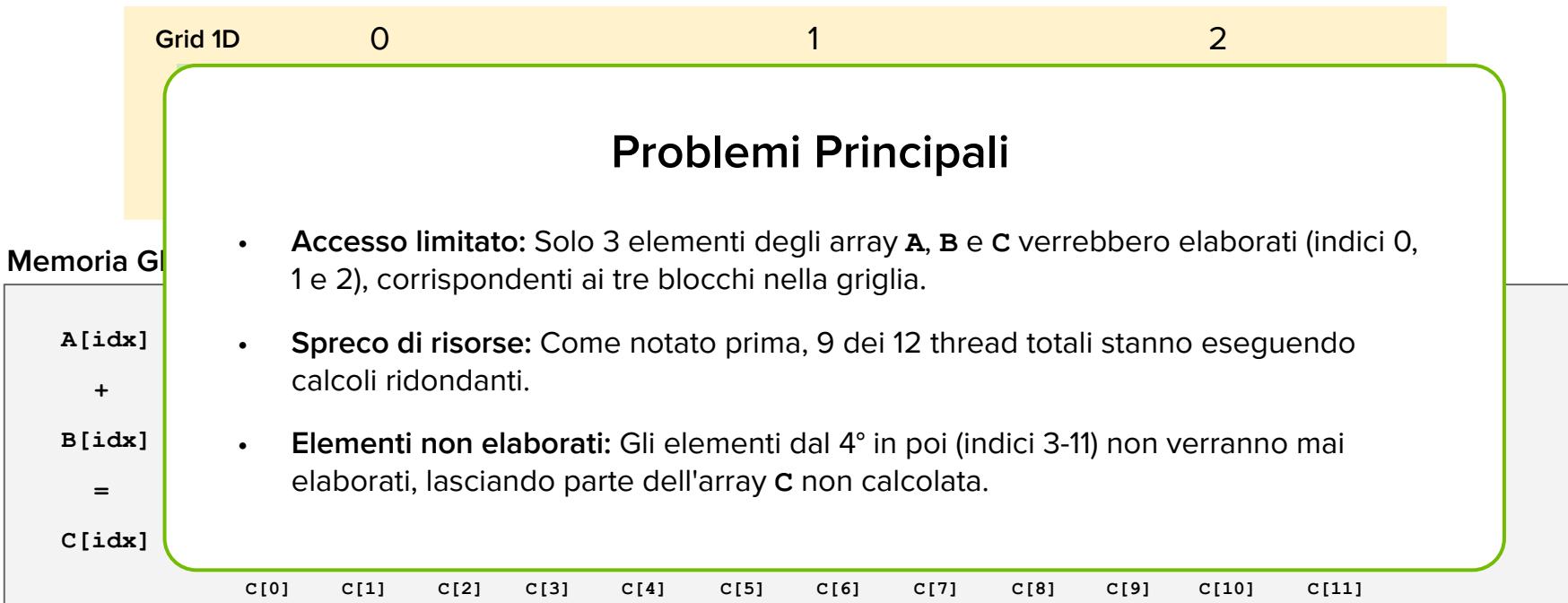
idx = blockIdx.x

NO!

Mapping degli Indici ai Dati in CUDA - Esempio 1D

- Come mappare gli indici dei thread agli elementi dell'array?

```
sumArraysOnGPU<<<3,4>>>(A, B, C)
```

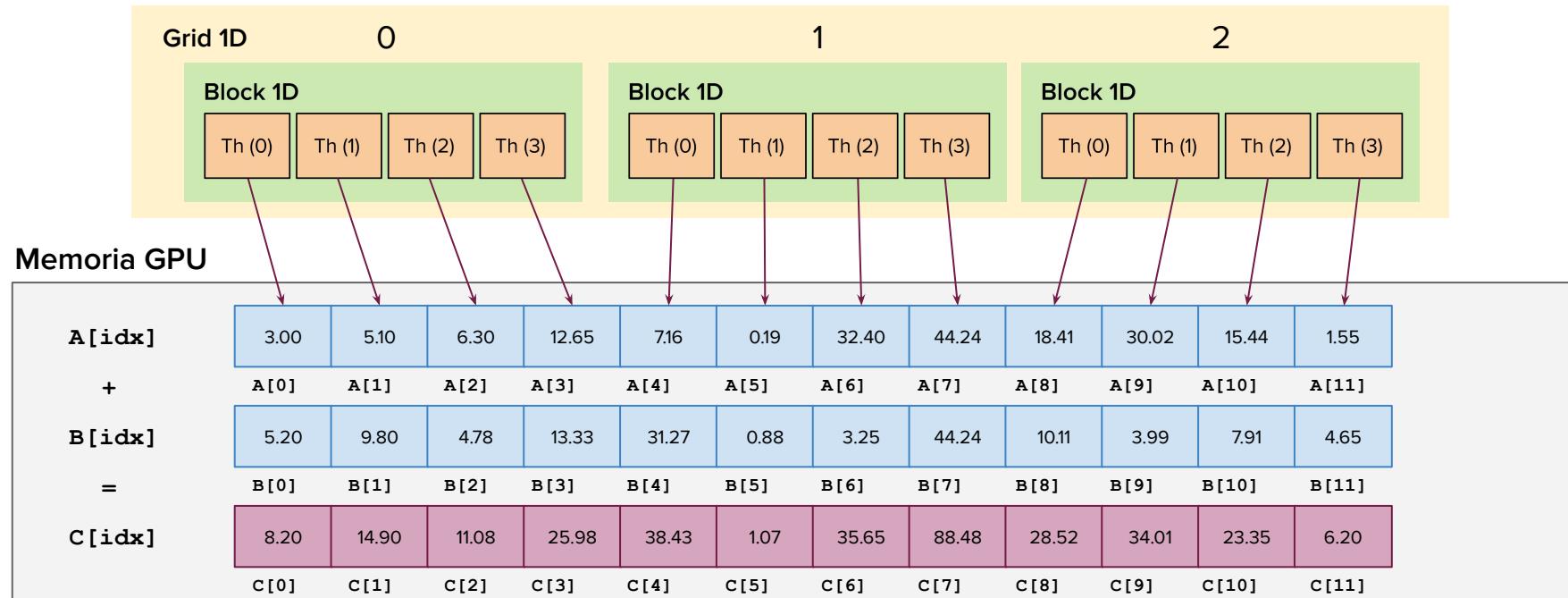


idx = blockIdx.x **NO!**

Mapping degli Indici ai Dati in CUDA - Esempio 1D

- Come mappare gli indici dei thread agli elementi dell'array?

```
sumArraysOnGPU<<<3,4>>>(A, B, C)
```



$\text{idx} = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x}$ OK!

Mapping degli Indici ai Dati in CUDA - Esempio 1D

• Com

- **Copertura completa:** Tutti i 12 thread (3 blocchi x 4 thread per blocco) sono utilizzati per elaborare i 12 elementi degli array.
- **Mapping corretto:** Ogni thread è associato a un unico elemento degli array **A**, **B** e **C**.
- **Nessuna ripetizione:** L'indice **idx**, univoco per ogni thread, assicura che ogni elemento dell'array venga elaborato esattamente una volta, evitando ridondanze.
- **Parallelismo massimizzato:** La formula **idx** permette di sfruttare appieno il parallelismo della GPU, assegnando un compito specifico ad ogni thread disponibile.
- **Scalabilità:** Questa formula si adatta bene a dimensioni di array diverse, purché si adegui il numero di blocchi.
- **Bilanciamento del carico:** Il lavoro è distribuito uniformemente tra tutti i thread, garantendo un utilizzo efficiente delle risorse.
- **Accessi coalescenti:** I thread adiacenti in un blocco accedono a elementi di memoria adiacenti, favorendo accessi coalescenti e migliorando l'efficienza della memoria.

Memoria G

```
A[idx]  
+  
B[idx]  
=  
C[idx]
```

Proprietà Chiave

idx = blockIdx.x * blockDim.x + threadIdx.x OK!

Confronto: Somma di Vettori in C vs CUDA C

Codice C Standard

```
void sumArraysOnHost(float *A, float *B,
float *C, int N) {
    for (int idx = 0; idx < N; idx++)
        C[idx] = A[idx] + B[idx];
}
// Chiamata della funzione
sumArraysOnHost(A, B, C, N);
```

Caratteristiche

- **Esecuzione:** Sequentiale
- **Iterazione:** Loop Esplicito
- **Indice:** Variabile di Loop (`idx`)
- **Scalabilità:** Limitata dalla CPU

Vantaggi

- Portabilità su qualsiasi sistema
- Facilità di debugging

Codice CUDA C

```
__global__ void sumArraysOnGPU(float *A, float
*B, float *C, int N) {
    int idx = blockDim.x*blockIdx.x + threadIdx.x;
    if (idx < N) C[idx] = A[idx] + B[idx];
}
// Chiamata del kernel (per N=12)
sumArraysOnGPU<<<gridDim,blockDim>>>(A, B, C, N);
```

Tutto ruota intorno a questa linea di codice

Per evitare accessi non consentiti in memoria

Caratteristiche

- **Esecuzione:** Parallela
- **Iterazione:** Implicita (un thread per elemento)
- **Indice:** `blockDim.x*blockIdx.x + threadIdx.x;`
- **Scalabilità:** Elevata (sfrutta molti core GPU)

Vantaggi

- Altamente parallelo
- Eccellenti prestazioni su grandi dataset
- Sfrutta la potenza di calcolo delle GPU

Identificazione dei Thread e Mapping dei Dati in CUDA

Accesso alle Variabili di Identificazione

- Le variabili di identificazione sono accessibili solo all'interno del kernel e permettono ai thread di conoscere la propria posizione all'interno della gerarchia e di adattare il proprio comportamento di conseguenza.

Perché Identificare i Thread?

- L'indice globale del thread identifica univocamente quale parte dei dati deve essere elaborata.
- Essenziale per gestire correttamente l'accesso alla memoria e coordinare l'esecuzione di algoritmi complessi.

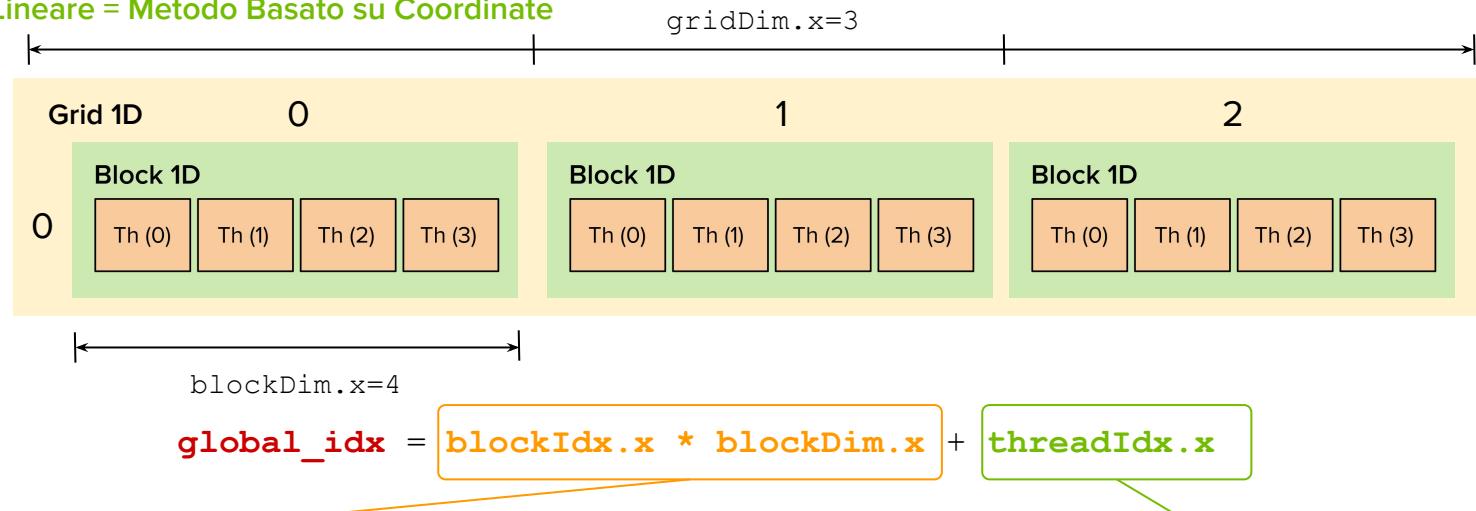
Struttura dei Dati e Calcolo dell'Indice Globale

- Anche le strutture più complesse, come matrici (2D) o array tridimensionali (3D), vengono memorizzate come una sequenza di elementi contigui in memoria nella GPU, tipicamente organizzati in array lineari.
- Ogni thread elabora uno o più elementi di questi array basandosi sul suo indice globale.
- Esistono diversi metodi per calcolare l'indice globale di un thread (es. **Metodo Lineare**, **Coordinate-based**).
- Metodi diversi possono produrre indici globali differenti per lo stesso thread (mapping diversi thread-dati), impattando la prestazione (come la coalescenza degli accessi in memoria) e la leggibilità del codice.

Calcolo dell'Indice Globale del Thread - Grid 1D, Block 1D

- In CUDA, ogni thread ha un **indice globale** (**global_idx**) che lo identifica nell'esecuzione del kernel. Il programmatore lo calcola usando l'indice del thread nel blocco e l'indice del blocco nella griglia.

Metodo Lineare = Metodo Basato su Coordinate



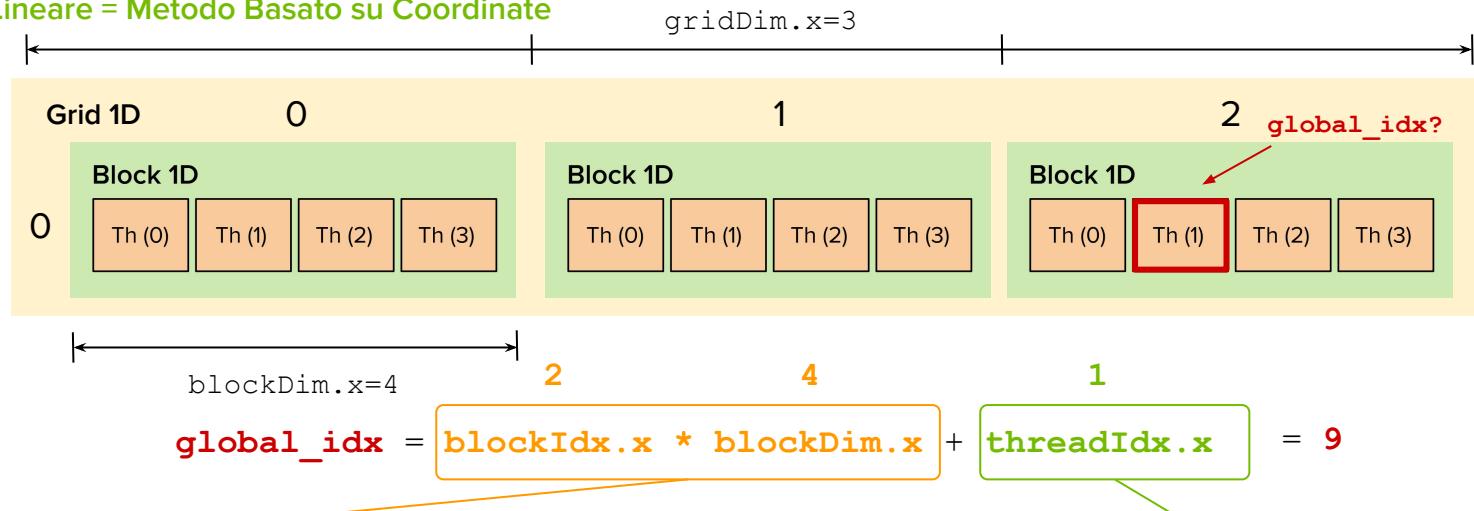
- Calcola l'offset di tutti i thread nei blocchi precedenti al blocco corrente.
- Moltiplicando **blockIdx.x** per **blockDim.x**, otteniamo il numero totale di thread che si trovano nei blocchi precedenti.

- Identifica la posizione del thread all'interno del blocco corrente.
- È l'indice del thread all'interno del blocco corrente, da 0 a `blockDim.x - 1`.

Calcolo dell'Indice Globale del Thread - Grid 1D, Block 1D

- In questo esempio viene mostrato come calcolare l'indice globale per il thread **Th (1)** appartenente al blocco unidimensionale con indice **blockIdx.x = 2**

Metodo Lineare = Metodo Basato su Coordinate

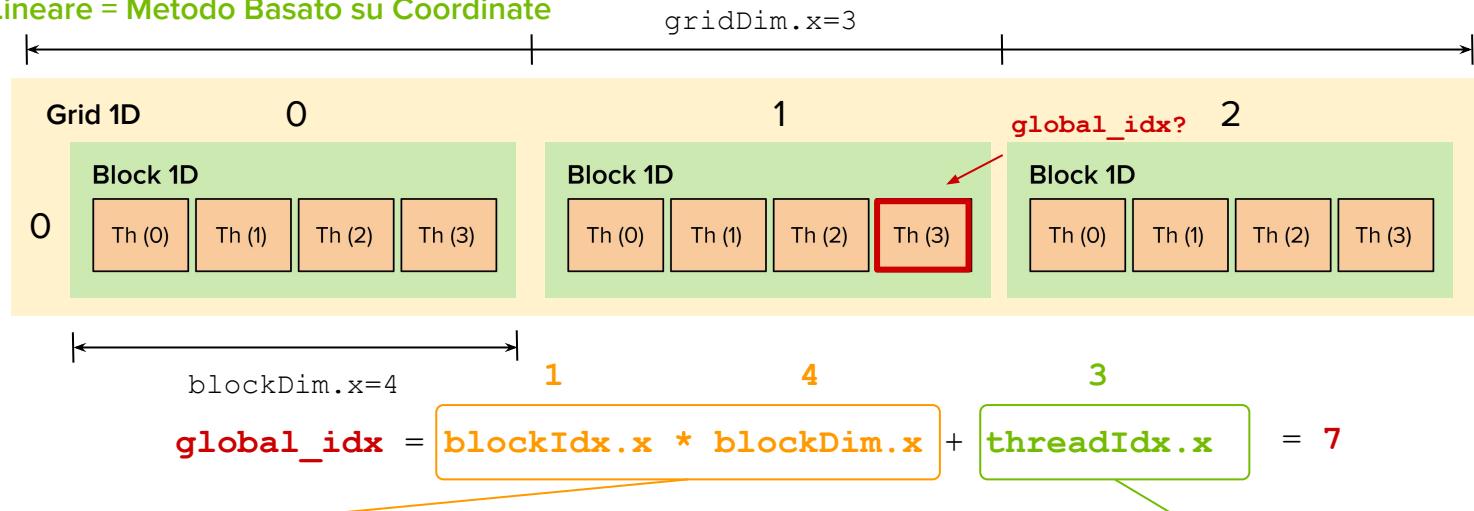


- Calcola l'offset di tutti i thread nei blocchi precedenti al blocco corrente.
- Moltiplicando **blockIdx.x** per **blockDim.x**, otteniamo il numero totale di thread che si trovano nei blocchi precedenti.
- Identifica la posizione del thread all'interno del blocco corrente.
- È l'indice del thread all'interno del blocco corrente, da 0 a `blockDim.x - 1`.

Calcolo dell'Indice Globale del Thread - Grid 1D, Block 1D

- In questo esempio viene mostrato come calcolare l'indice globale per il thread **Th (3)** appartenente al blocco unidimensionale con indice **blockIdx.x = 1**

Metodo Lineare = Metodo Basato su Coordinate



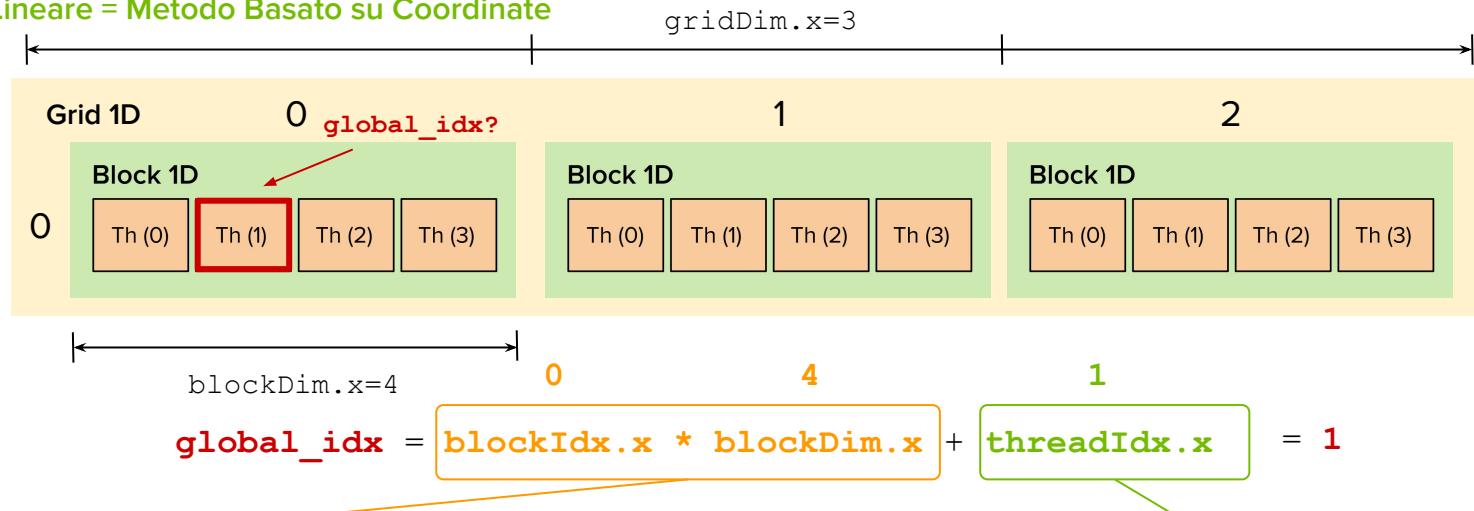
- Calcola l'offset di tutti i thread nei blocchi precedenti al blocco corrente.
- Moltiplicando **blockIdx.x** per **blockDim.x**, otteniamo il numero totale di thread che si trovano nei blocchi precedenti.

- Identifica la posizione del thread all'interno del blocco corrente.
- È l'indice del thread all'interno del blocco corrente, da 0 a **blockDim.x - 1**.

Calcolo dell'Indice Globale del Thread - Grid 1D, Block 1D

- In questo esempio viene mostrato come calcolare l'indice globale per il thread **Th (1)** appartenente al blocco unidimensionale con indice **blockIdx.x = 0**

Metodo Lineare = Metodo Basato su Coordinate

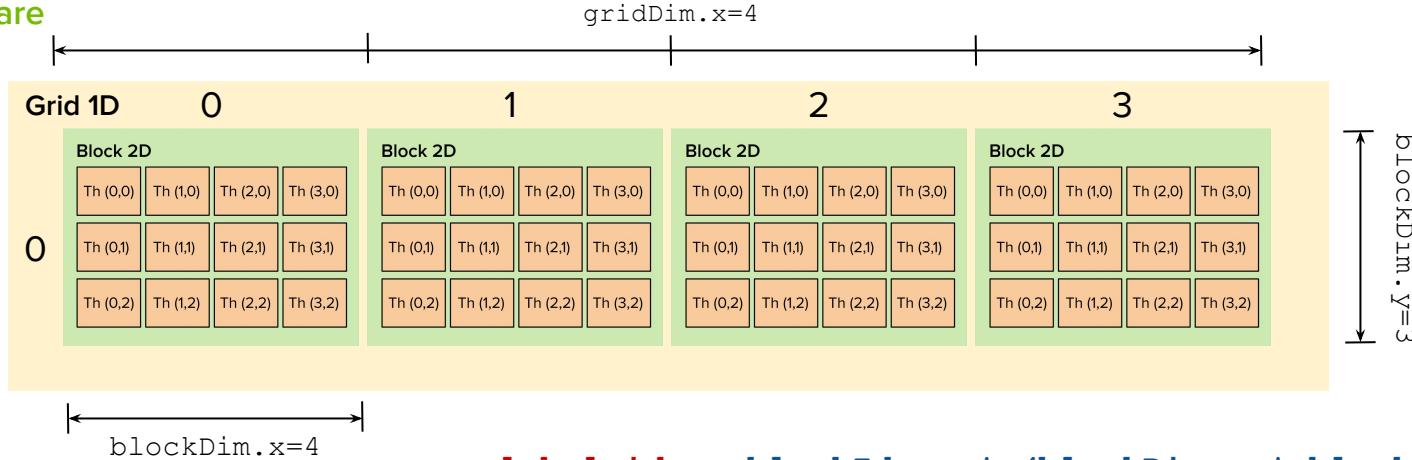


- Calcola l'offset di tutti i thread nei blocchi precedenti al blocco corrente.
- Moltiplicando **blockIdx.x** per **blockDim.x**, otteniamo il numero totale di thread che si trovano nei blocchi precedenti.

- Identifica la posizione del thread all'interno del blocco corrente.
- È l'indice del thread all'interno del blocco corrente, da 0 a **blockDim.x - 1**.

Calcolo dell'Indice Globale del Thread - Grid 1D, Block 2D

Metodo Lineare



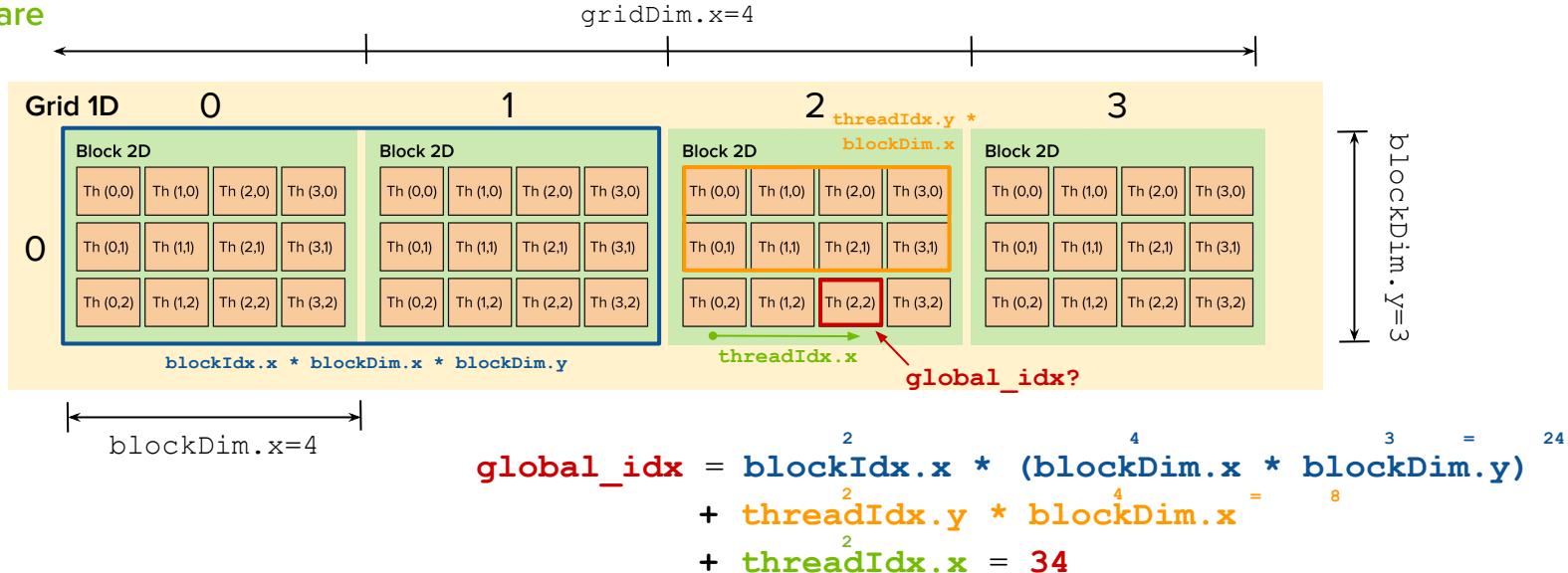
```
global_idx = blockIdx.x * (blockDim.x * blockDim.y)  
           + threadIdx.y * blockDim.x  
           + threadIdx.x
```

Metodo Lineare (Linear Indexing Method) - Derivazione

- **`blockIdx.x * blockDim.x * blockDim.y`**: Moltiplicando `blockIdx.x` per `blockDim.x * blockDim.y`, otteniamo il numero totale di thread che si trovano nei blocchi precedenti lungo x.
- **`threadIdx.y * blockDim.x`**: Moltiplichiamo `threadIdx.x` per `blockDim.x` per ottenere il numero di thread nelle righe precedenti nella matrice di thread.
- **`threadIdx.x`**: Identifica la posizione del thread all'interno della riga corrente (x).

Calcolo dell'Indice Globale del Thread - Grid 1D, Block 2D

Metodo Lineare

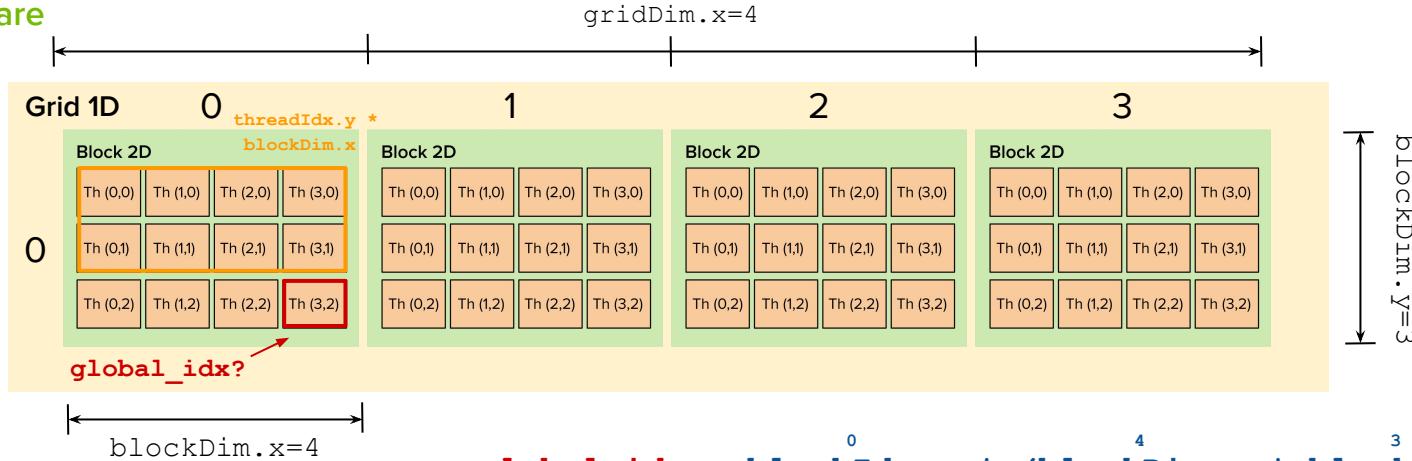


Metodo Lineare (Linear Indexing Method) - Derivazione

- **blockIdx.x * blockDim.x * blockDim.y**: Moltiplicando `blockIdx.x` per `blockDim.x * blockDim.y`, otteniamo il numero totale di thread che si trovano nei blocchi precedenti lungo x
- **threadIdx.y * blockDim.x**: Moltiplichiamo `threadIdx.y` per `blockDim.x` per ottenere il numero di thread nelle righe precedenti nella matrice di thread.
- **threadIdx.x**: Identifica la posizione del thread all'interno della riga corrente (x).

Calcolo dell'Indice Globale del Thread - Grid 1D, Block 2D

Metodo Lineare



`blockDim.x=4`

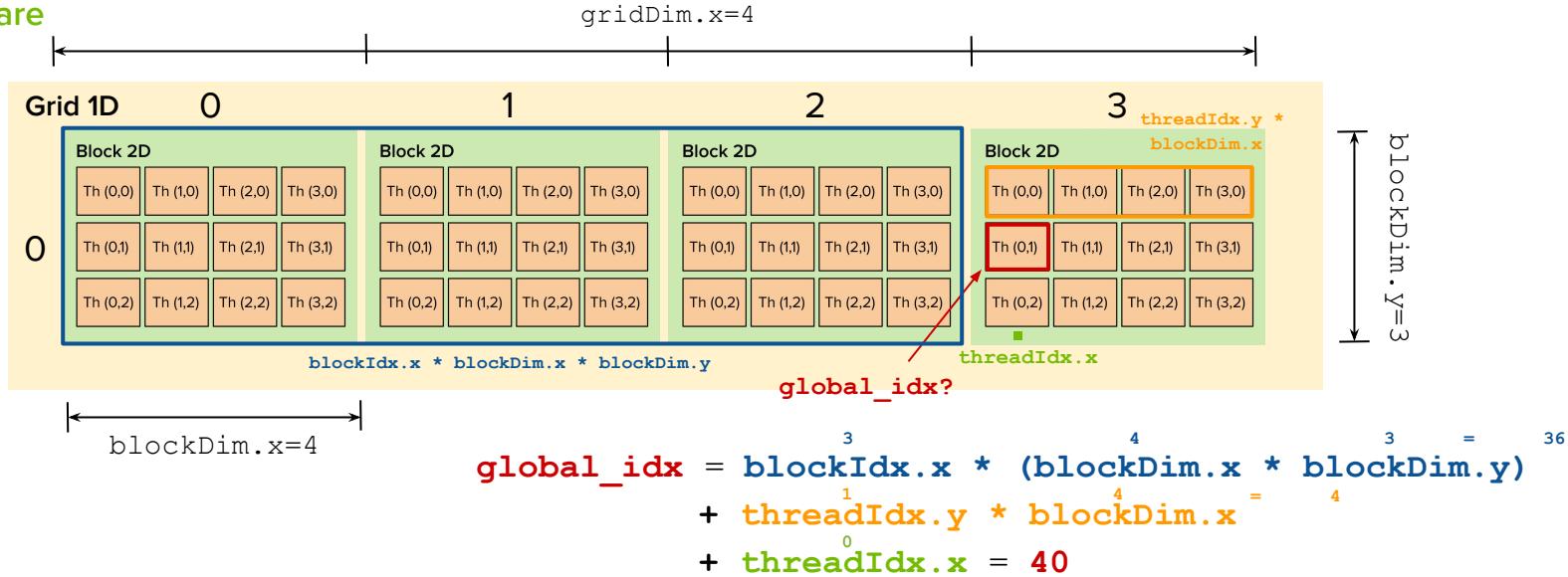
$$\begin{aligned} \text{global_idx} &= \text{blockIdx.x} * (\text{blockDim.x} * \text{blockDim.y}) \\ &\quad + \text{threadIdx.y} * \text{blockDim.x} \\ &\quad + \text{threadIdx.x} = 11 \end{aligned}$$

Metodo Lineare (Linear Indexing Method) - Derivazione

- **`blockIdx.x * blockDim.x * blockDim.y`:** Moltiplicando `blockIdx.x` per `blockDim.x * blockDim.y`, otteniamo il numero totale di thread che si trovano nei blocchi precedenti lungo x.
- **`threadIdx.y * blockDim.x`:** Moltiplichiamo `threadIdx.x` per `blockDim.x` per ottenere il numero di thread nelle righe precedenti nella matrice di thread.
- **`threadIdx.x`:** Identifica la posizione del thread all'interno della riga corrente (x).

Calcolo dell'Indice Globale del Thread - Grid 1D, Block 2D

Metodo Lineare

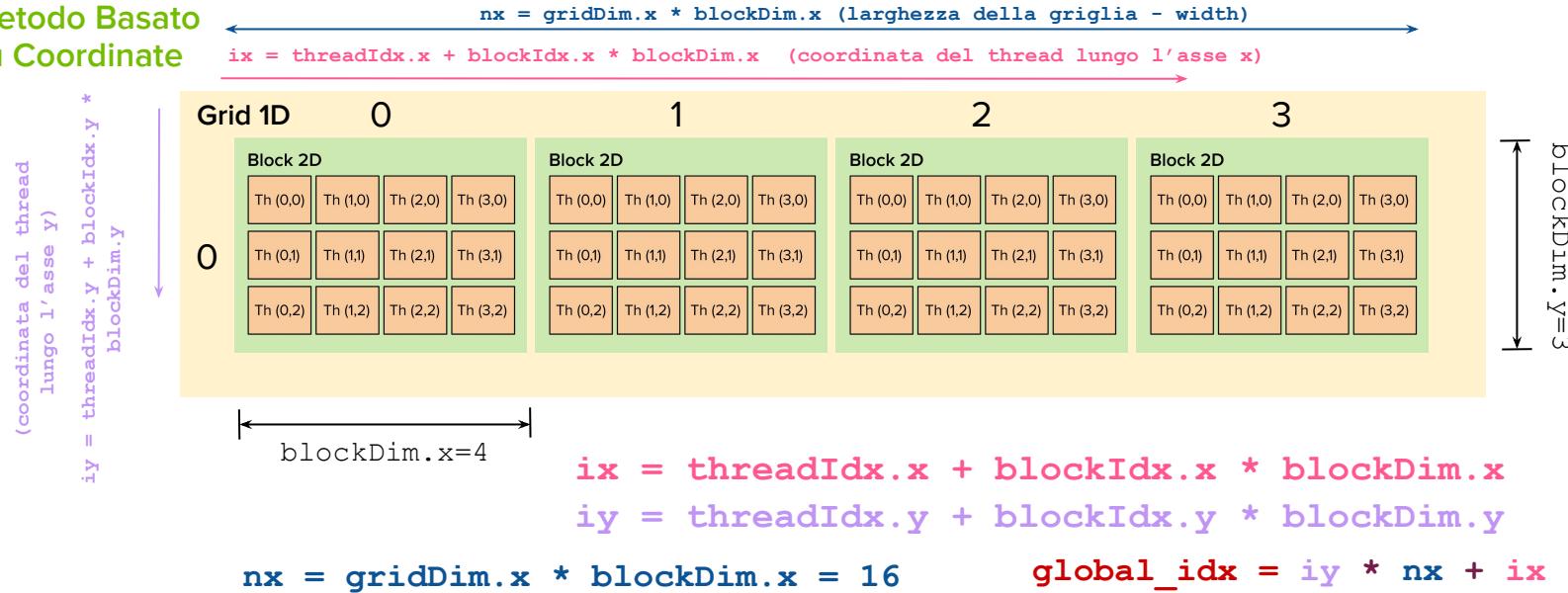


Metodo Lineare (Linear Indexing Method) - Derivazione

- **blockIdx.x * blockDim.x * blockDim.y**: Moltiplicando `blockIdx.x` per `blockDim.x * blockDim.y`, otteniamo il numero totale di thread che si trovano nei blocchi precedenti lungo x
- **threadIdx.y * blockDim.x**: Moltiplichiamo `threadIdx.y` per `blockDim.x` per ottenere il numero di thread nelle righe precedenti nella matrice di thread.
- **threadIdx.x**: Identifica la posizione del thread all'interno della riga corrente (x).

Calcolo dell'Indice Globale del Thread - Grid 1D, Block 2D

Metodo Basato
su Coordinate

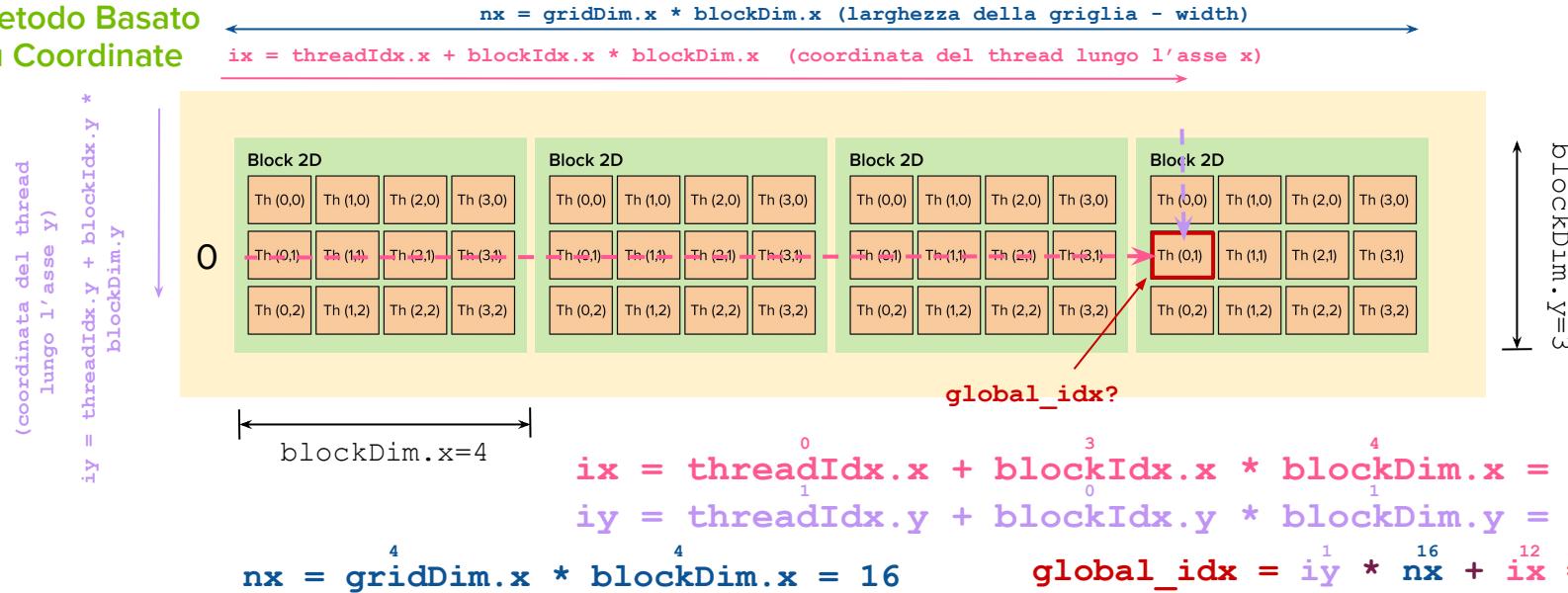


Metodo Basato su Coordinate (Coordinate-based Method) - Derivazione

- $ix = \text{threadIdx.x} + \text{blockIdx.x} * \text{blockDim.x}$: Determina l'indice del thread lungo l'asse **x**, prendendo in considerazione la posizione nel blocco (**threadIdx.x**) e il numero di blocchi precedenti (**blockIdx.x * blockDim.x**).
- $i_y = \text{threadIdx.y} + \text{blockIdx.y} * \text{blockDim.y}$: Determina l'indice del thread lungo l'asse **y**, considerando sia la posizione locale (**threadIdx.y**) che i blocchi precedenti lungo **y** (**blockIdx.y * blockDim.y**).
- $\text{global_idx} = i_y * nx + ix$: Calcola l'indice globale sommando **ix** all'indice globale lungo **y**, dove **nx** rappresenta il numero di thread per riga (in questo caso, **nx = gridDim.x * blockDim.x**).

Calcolo dell'Indice Globale del Thread - Grid 1D, Block 2D

Metodo Basato
su Coordinate



Metodo Basato su Coordinate (Coordinate-based Method) - Derivazione

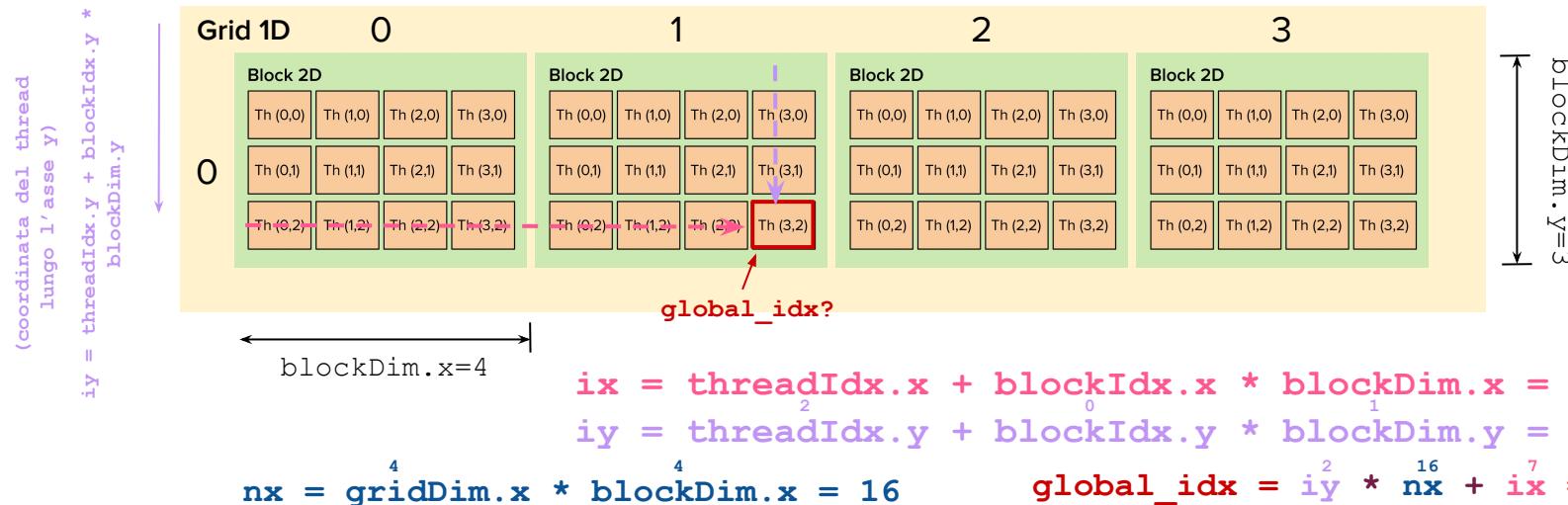
- $ix = threadIdx.x + blockIdx.x * blockDim.x$: Determina l'indice del thread lungo l'asse **x**, prendendo in considerazione la posizione nel blocco (**threadIdx.x**) e il numero di blocchi precedenti (**blockIdx.x * blockDim.x**).
- $iy = threadIdx.y + blockIdx.y * blockDim.y$: Determina l'indice del thread lungo l'asse **y**, considerando sia la posizione locale (**threadIdx.y**) che i blocchi precedenti lungo **y** (**blockIdx.y * blockDim.y**).
- $global_idx = iy * nx + ix$: Calcola l'indice globale sommando **ix** all'indice globale lungo **y**, dove **nx** rappresenta il numero di thread per riga (in questo caso, **nx = blockDim.x * blockDim.x**).

Calcolo dell'Indice Globale del Thread - Grid 1D, Block 2D

Metodo Basato
su Coordinate

$$nx = \text{gridDim.x} * \text{blockDim.x}$$
 (larghezza della griglia - width)

$$ix = \text{threadIdx.x} + \text{blockIdx.x} * \text{blockDim.x}$$
 (coordinata del thread lungo l'asse x)

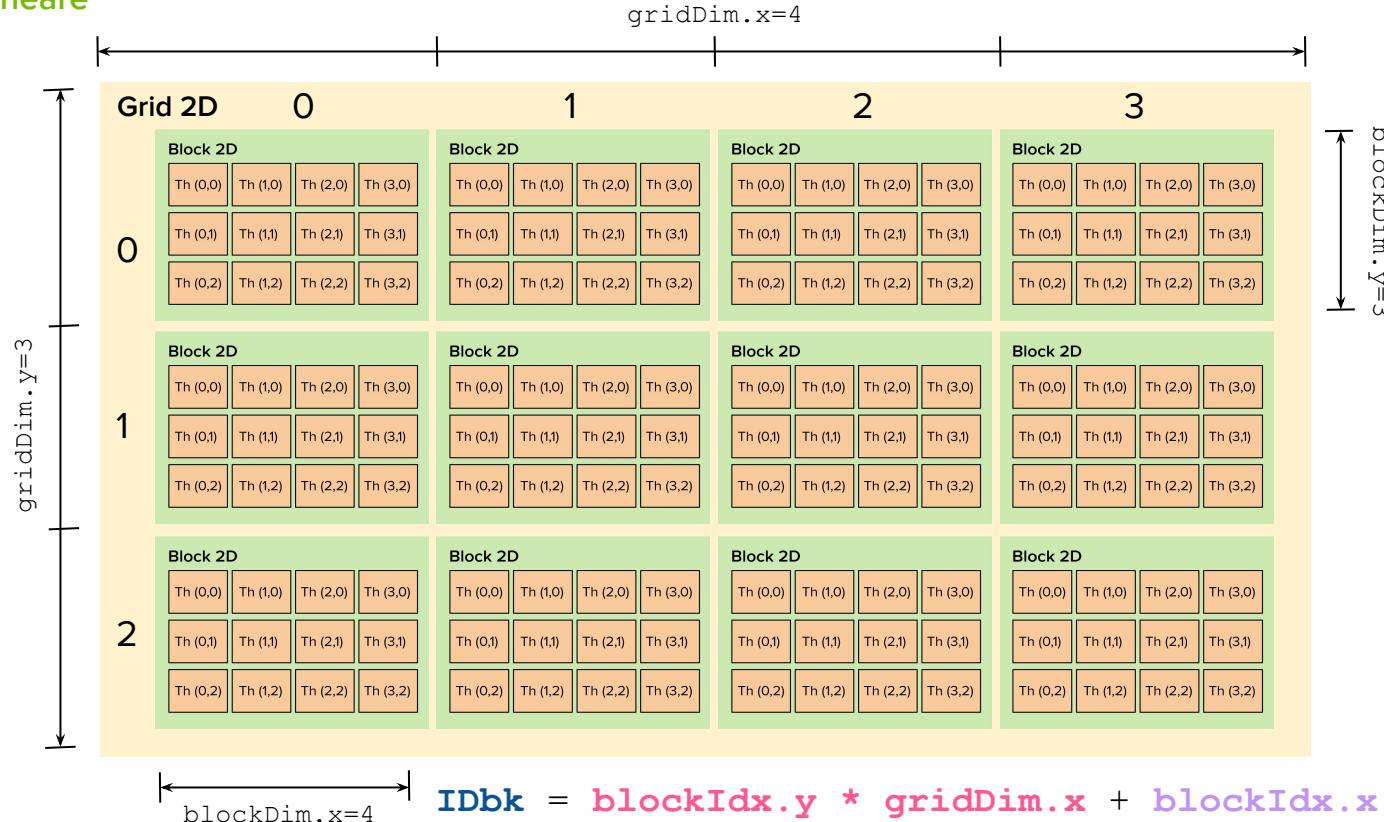


Metodo Basato su Coordinate (Coordinate-based Method) - Derivazione

- $ix = \text{threadIdx.x} + \text{blockIdx.x} * \text{blockDim.x}$: Determina l'indice del thread lungo l'asse **x**, prendendo in considerazione la posizione nel blocco (**threadIdx.x**) e il numero di blocchi precedenti (**blockIdx.x * blockDim.x**).
- $iy = \text{threadIdx.y} + \text{blockIdx.y} * \text{blockDim.y}$: Determina l'indice del thread lungo l'asse **y**, considerando sia la posizione locale (**threadIdx.y**) che i blocchi precedenti lungo **y** (**blockIdx.y * blockDim.y**).
- $global_idx = iy * nx + ix$: Calcola l'indice globale sommando **ix** all'indice globale lungo **y**, dove **nx** rappresenta il numero di thread per riga (in questo caso, **nx = gridDim.x * blockDim.x**).

Calcolo dell'Indice Globale del Thread - Grid 2D, Block 2D

Metodo Lineare

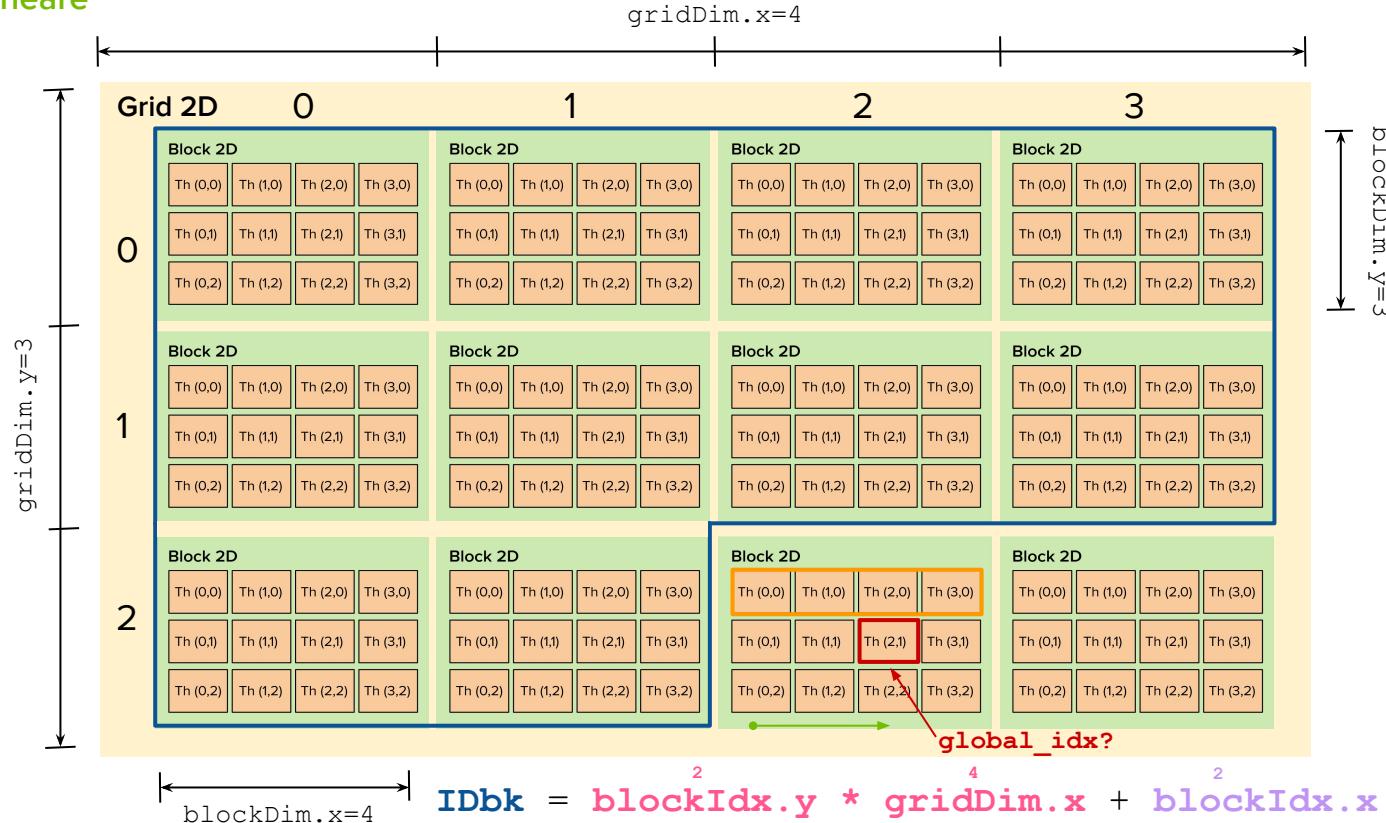


$$\text{IDbk} = \text{blockIdx.y} * \text{gridDim.x} + \text{blockIdx.x}$$

$$\text{global_idx} = \text{IDbk} * (\text{blockDim.x} * \text{blockDim.y}) + \text{threadIdx.y} * \text{blockDim.x} + \text{threadIdx.x}$$

Calcolo dell'Indice Globale del Thread - Grid 2D, Block 2D

Metodo Lineare

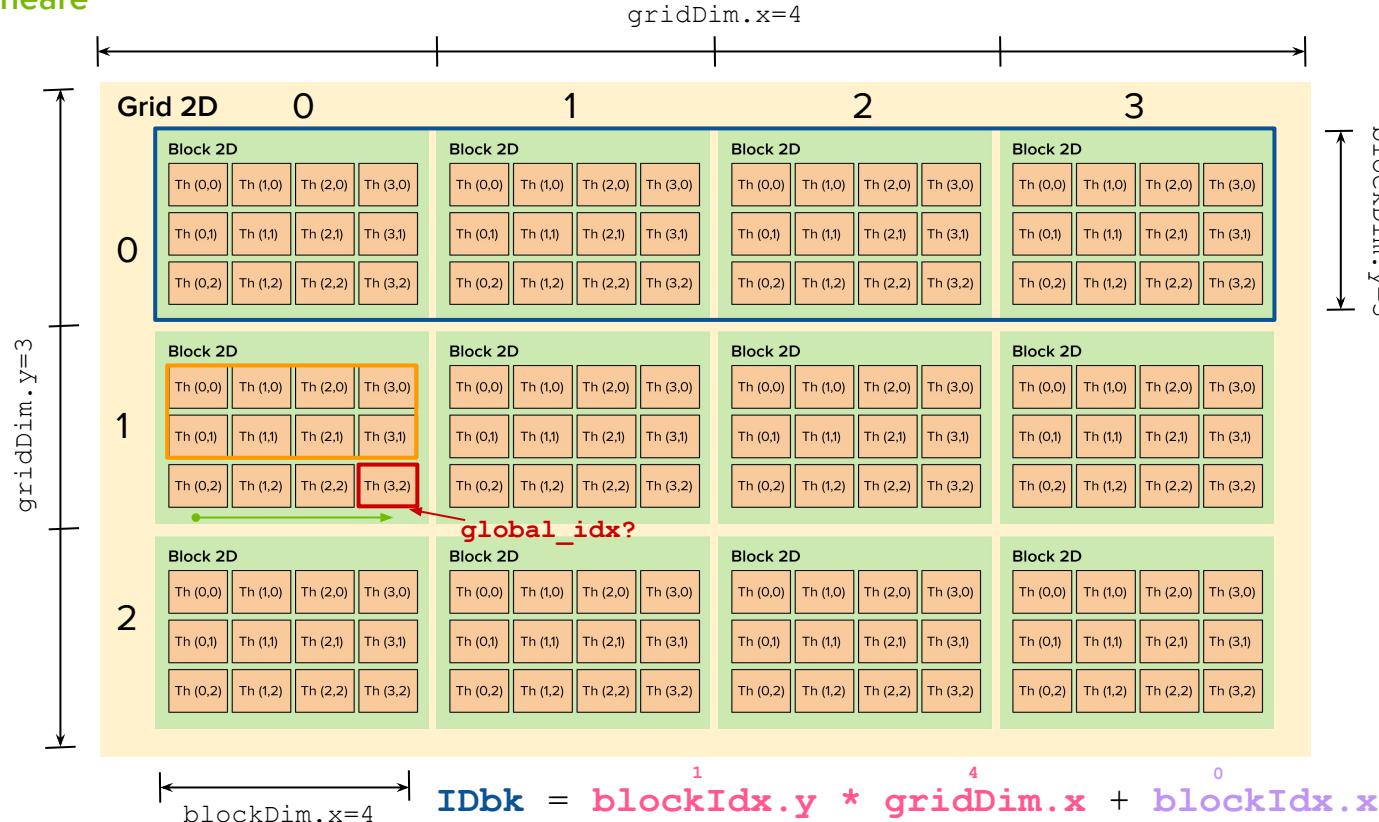


$$IDbk = \text{blockIdx.y} * \overset{2}{\text{gridDim.x}} + \overset{4}{\text{blockIdx.x}} = 10$$

$$\begin{aligned} \text{global_idx} = & IDbk * (\overset{10}{blockDim.x} * \overset{4}{blockDim.y}) \\ & + \overset{4}{threadIdx.y} * \overset{2}{blockDim.x} + \overset{3}{threadIdx.x} = 126 \end{aligned}$$

Calcolo dell'Indice Globale del Thread - Grid 2D, Block 2D

Metodo Lineare

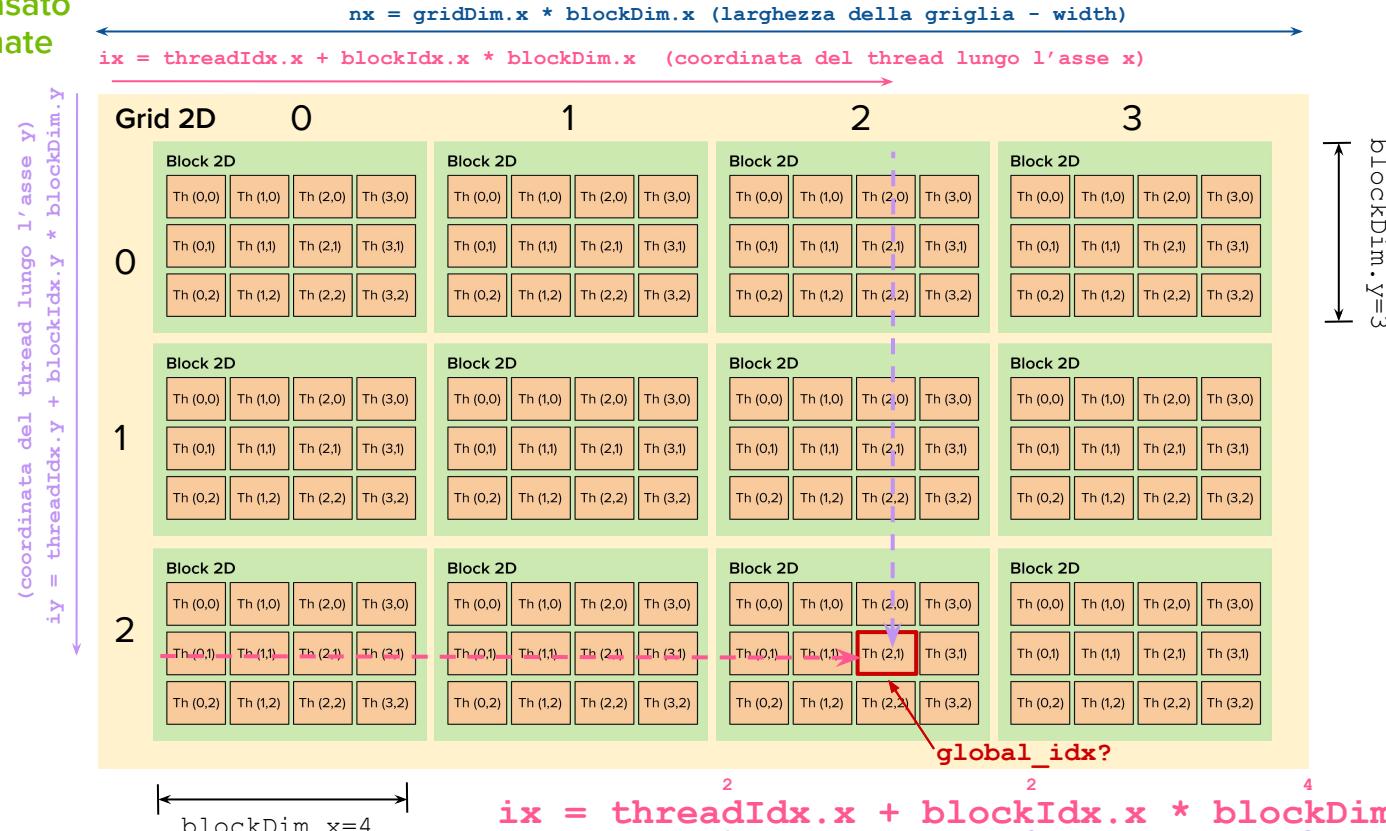


$$\text{IDbk} = \text{blockIdx.y} * \text{gridDim.x} + \text{blockIdx.x} = 4$$

$$\begin{aligned}\text{global_idx} = & \text{IDbk} * (\text{blockDim.x} * \text{blockDim.y}) \\ & + \text{threadIdx.y} * \text{blockDim.x} + \text{threadIdx.x} = 59\end{aligned}$$

Calcolo dell'Indice Globale del Thread - Grid 2D, Block 2D

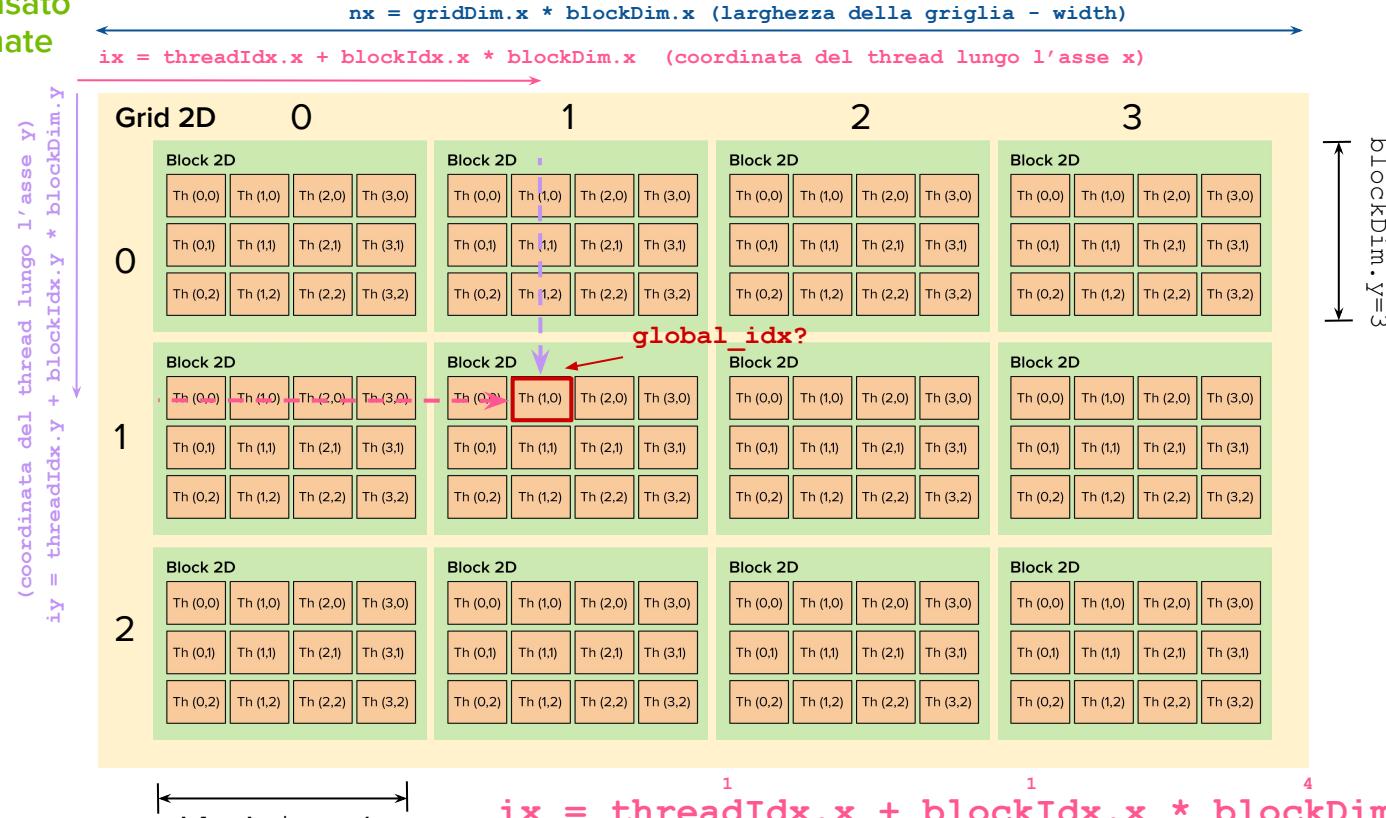
Metodo Basato
su Coordinate



$$ix = \text{threadIdx.x} + \text{blockIdx.x} * \text{blockDim.x} = 10$$
$$iy = \text{threadIdx.y} + \text{blockIdx.y} * \text{blockDim.y} + \text{blockDim.y} * \text{blockIdx.y} * \text{blockDim.y} * \text{blockIdx.y} * \text{blockDim.y} = 7$$
$$nx = \text{gridDim.x} * \text{blockDim.x}$$
$$\text{global_idx} = iy * nx + ix = 114$$

Calcolo dell'Indice Globale del Thread - Grid 2D, Block 2D

Metodo Basato
su Coordinate



$$\begin{aligned}
 \text{nx} &= \text{gridDim.x} * \text{blockDim.x} & \text{ix} &= \text{threadIdx.x} + \text{blockIdx.x} * \text{blockDim.x} = 5 \\
 \text{iy} &= \text{threadIdx.y} + \text{blockIdx.y} * \text{blockDim.y} & \text{global_idx} &= \text{iy} * \text{nx} + \text{ix} = 53 \\
 \text{nx} &= 4 * 4 & \text{iy} &= 3 * 4 + 1 = 13 \\
 \end{aligned}$$

Metodo Lineare per Indici Globali in CUDA

Caratteristiche del Metodo Lineare

- Calcola un **unico indice scalare** per la posizione del thread in un **array lineare**, indipendentemente dalla sua struttura multidimensionale (mappa **direttamente** a memoria lineare).
- Utilizza una formula diretta che combina gli **indici dei blocchi e dei thread**.
- Efficiente per l'**accesso sequenziale** a dati memorizzati in array lineari.
- **Meno intuitivo** per strutture dati complesse (matrici, array 3D).

Formule per Calcolo Indice Lineare

Caso 1D) `idx = blockIdx.x * blockDim.x + threadIdx.x`

Caso 2D) `idx = (blockIdx.y * gridDim.x + blockIdx.x) * (blockDim.y * blockDim.x)`
`+ (threadIdx.y * blockDim.x + threadIdx.x)`

Caso 3D) `idx = (blockIdx.z * gridDim.y * gridDim.x + blockIdx.y * gridDim.x + blockIdx.x)`
`* (blockDim.z * blockDim.y * blockDim.x)`
`+ (threadIdx.z * blockDim.y * blockDim.x + threadIdx.y * blockDim.x + threadIdx.x)`

Esempio di Utilizzo (Caso 2D)

```
__global__ void kernel2D(float* data, int width, int height) {
    int idx = (blockIdx.y * gridDim.x + blockIdx.x) * (blockDim.y * blockDim.x) +
              (threadIdx.y * blockDim.x + threadIdx.x);
    if (idx < width * height) { // width e height si riferiscono alle dimensioni dell'array dati
        // Operazioni su data[idx]
    }
}
```

Metodo Basato su Coordinate per Indici Globali in CUDA

Caratteristiche del Metodo Basato su Coordinate

- Calcola indici **separati** per ogni dimensione della griglia e dei blocchi.
- Riflette **naturalmente** la disposizione multidimensionale dei dati.
- Facilita la **comprensione** della posizione del thread nello spazio
- Richiede un **passaggio aggiuntivo** per combinare gli indici in un indice globale.

Calcolo degli Indici Coordinati

Caso 1D) `x = blockIdx.x * blockDim.x + threadIdx.x` → `idx = x` (equivalente al caso lineare)

Caso 2D) `x = blockIdx.x * blockDim.x + threadIdx.x` → `idx = y * width + x`
`y = blockIdx.y * blockDim.y + threadIdx.y`

Caso 3D) `x = blockIdx.x * blockDim.x + threadIdx.x` → `idx = z * (height * width)`
`y = blockIdx.y * blockDim.y + threadIdx.y` → `+ y * width`
`z = blockIdx.z * blockDim.z + threadIdx.z` → `+ x`

Calcolo dell'Indice Globale

Esempio di Utilizzo (Caso 2D)

```
__global__ void kernel2D(float* data, int width, int height) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    if (x < width && y < height) { // width e height si riferiscono alle dimensioni dell'array dati
        int idx = y * width + x;
        // Operazioni su data[global_idx]
    }
}
```

Come Calcolare la Dimensione della Griglia e del Blocco?

Approccio Generale

- Definire **manualmente** prima la dimensione del blocco (cioè quanti thread per blocco).
- Poi, calcolare **automaticamente** la dimensione della griglia in base ai **dati** e alla **dimensione del blocco**.

Motivazioni

- La **dimensione del blocco** è legata alle **caratteristiche hardware** della GPU e la natura del problema.
- La **dimensione della griglia** si adatta alla **dimensione del blocco** e al **volume dei dati** da processare.

Calcolo delle Dimensioni (Caso 1D)

```
int blockSize = 256;  int dataSize = 1024;    // Dimensione del blocco e dei dati
dim3 blockDim(blockSize); dim3 gridDim((dataSize + blockSize - 1) / blockSize);
kernel_name<<<gridDim, blockDim>>>(args);    // Lancio del kernel
```

Spiegazione del Calcolo

- La formula **(dataSize + blockSize - 1) / blockSize** garantisce abbastanza blocchi per coprire tutti i dati, anche se **dataSize** non è un multiplo esatto di **blockSize**.
 - Divisione semplice: **dataSize / blockSize** fornisce il numero di blocchi completamente pieni.
 - Se ci sono **dati residui** che non riempiono un intero blocco, la divisione semplice li **ignorerebbe**.
 - Aggiungere **blockSize - 1** a **dataSize** "sposta" questi dati residui, assicurando che la divisione includa anche l'ultimo blocco parziale. Equivalente a calcolare la **ceil** della divisione.

Come Calcolare la Dimensione della Griglia e del Blocco?

Approccio Generale

- Definire **manualmente** prima la dimensione del blocco (cioè quanti thread per blocco).
- Poi, calcolare **automaticamente** la dimensione della griglia in base ai **dati** e alla **dimensione del blocco**.

Motivazioni

- La **dimensione del blocco** è legata alle **caratteristiche hardware** della GPU e la natura del problema.
- La **dimensione della griglia** si adatta alla **dimensione del blocco** e al **volume dei dati** da processare.

Calcolo delle Dimensioni (Caso 1D)

```
int blockSize = 256;  int dataSize = 1024;    // Dimensione del blocco e dei dati
dim3 blockDim(blockSize); dim3 gridDim((dataSize + blockSize - 1) / blockSize);
kernel_name<<<gridDim, blockDim>>>(args);    // Lancio del kernel
```

Esempio: **dataSize = 1030, blockSize = 256**

- Divisione semplice: $(1030 / 256 = 4)$ (ignorerebbe l'ultimo blocco parziale - 6 elementi residui).
- Risultato della formula: $((1030 + 256 - 1) / 256 = 1285 / 256 = 5)$
- In questo caso, la divisione semplice avrebbe dato 4 blocchi, ma c'è un **residuo di 6 elementi** ($1030 \text{ mod } 256 = 6$). La formula include anche il blocco parziale, quindi otteniamo **5 blocchi**.

Come Calcolare la Dimensione della Griglia e del Blocco?

Approccio Generale

- Definire **manualmente** prima la dimensione del blocco (cioè quanti thread per blocco).
- Poi, calcolare **automaticamente** la dimensione della griglia in base ai **dati** e alla **dimensione del blocco**.

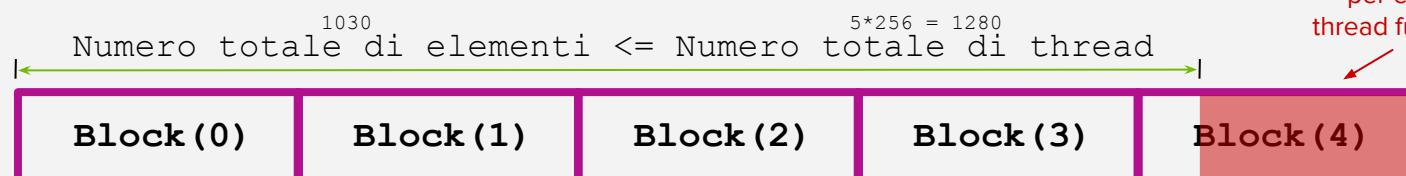
Motivazioni

- La **dimensione del blocco** è legata alle **caratteristiche hardware** della GPU e la natura del problema.
- La **dimensione della griglia** si adatta alla **dimensione del blocco** e al **volume dei dati** da processare.

Calcolo delle Dimensioni (Caso 1D)

```
int blockSize = 256; int dataSize = 1024; // Dimensione del blocco e dei dati  
dim3 blockDim(blockSize); dim3 gridDim((dataSize + blockSize - 1) / blockSize);  
kernel_name<<<gridDim, blockDim>>>(args); // Lancio del kernel
```

Esempio: **dataSize = 1030, blockSize = 256**



Verificare gli indici
per escludere i
thread fuori dai limiti.

Come Calcolare la Dimensione della Griglia e del Blocco?

Approccio Generale

- Definire **manualmente** prima la dimensione del blocco (cioè quanti thread per blocco).
- Poi, calcolare **automaticamente** la dimensione della griglia in base ai **dati** e alla **dimensione del blocco**.

Motivazioni

- La **dimensione del blocco** è legata alle **caratteristiche hardware** della GPU e la natura del problema.
- La **dimensione della griglia** si adatta alla **dimensione del blocco** e al **volume dei dati** da processare.

Calcolo delle Dimensioni (Caso 2D)

```
int blockSizeX = 16, blockSizeY = 16;                                // Dimensione del blocco
int dataSizeX = 1024, dataSizeY = 512;                                // Dimensione dei dati

dim3 blockDim(blockSizeX, blockSizeY);                                // Definizione del blocco 2D
dim3 gridDim(                                                       // Calcolo della griglia 2D
    (dataSizeX + blockSizeX - 1) / blockSizeX,                         // Numero di blocchi in X
    (dataSizeY + blockSizeY - 1) / blockSizeY                          // Numero di blocchi in Y
);

kernel_name<<<gridDim, blockDim>>>(args);                           // Lancio del kernel
```

Come Calcolare la Dimensione della Griglia e del Blocco?

Approccio Generale

- Definire **manualmente** prima la dimensione del blocco (cioè quanti thread per blocco).
- Poi, calcolare **automaticamente** la dimensione della griglia in base ai **dati** e alla **dimensione del blocco**.

Motivazioni

- La **dimensione del blocco** è legata alle **caratteristiche hardware** della GPU e la natura del problema.
- La **dimensione della griglia** si adatta alla **dimensione del blocco** e al **volume dei dati** da processare.

Calcolo delle Dimensioni (Caso Generale 3D)

```
int blockSizeX = 16, blockSizeY = 16, blockSizeZ = 16;      // Dimensione del blocco
int dataSizeX = 1024, dataSizeY = 512, dataSizeZ = 256;    // Dimensione dei dati

dim3 blockDim(blockSizeX, blockSizeY, blockSizeZ);        // Definizione del blocco 3D
dim3 gridDim(                                                       // Calcolo della griglia 3D
    (dataSizeX + blockSizeX - 1) / blockSizeX,                // Numero di blocchi in X
    (dataSizeY + blockSizeY - 1) / blockSizeY,                // Numero di blocchi in Y
    (dataSizeZ + blockSizeZ - 1) / blockSizeZ                 // Numero di blocchi in Z
);

kernel_name<<<gridDim, blockDim>>>(args);           // Lancio del kernel
```

Esempio di Dimensionamento Grid e Block in CUDA

- Questo esempio mostra come configurare e lanciare un kernel CUDA, e come stampare le informazioni sugli indici e le dimensioni dei thread e dei blocchi.

Esempio di Dimensionamento Grid e Block in CUDA

- Questo esempio mostra come configurare e lanciare un kernel CUDA, e come **stampare le informazioni sugli indici e le dimensioni dei thread e dei blocchi**.

```
#include <cuda_runtime.h>
#include <stdio.h>

__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 // Indici del thread
        blockIdx.x, blockIdx.y, blockIdx.z, blockDim.x, blockDim.y, blockDim.z,
        gridDim.x, gridDim.y, gridDim.z);
}

int main(int argc, char **argv) {
    int nElem = 6;
    dim3 block(3); // Numero // Definiamo // Calcolo
    dim3 grid((nElem+block.x-1)/block.x); // Esecuzione

    printf("grid.x %d grid.y %d grid.z %d\n", grid.x,
    printf("block.x %d block.y %d block.z %d\n", block.x,
    checkIndex<<<grid, block>>>(); // Esecuzione

    cudaDeviceReset(); // Reset del device
    return(0);
}
```

Dimensione dei Dati

dim3 block (3)

Definiamo un block 1D con 3 thread

- Semplice per l'esempio
- In pratica, si scelgono dimensioni multiple di 32 per efficienza delle operazioni sulla GPU, dato che i **warp** (unità di esecuzione parallela della GPU) sono composti da 32 thread (lo analizzeremo in seguito nel dettaglio).

Esempio di Dimensionamento Grid e Block in CUDA

- Questo esempio mostra come configurare e lanciare un kernel CUDA, e come **stampare le informazioni sugli indici e le dimensioni dei thread e dei blocchi**.

```
#include <cuda_runtime.h>
#include <stdio.h>

__global__ void checkIndex(void) {
    printf("threadIdx: (%d, %d, %d) blockIdx: (%d, %d, %d)\n",
           threadIdx.x, threadIdx.y, threadIdx.z,
           blockIdx.x, blockIdx.y, blockIdx.z);
    printf("gridDim: (%d, %d, %d)\n", gridDim.x, gridDim.y, gridDim.z);

int main(int argc, char **argv) {
    int nElem = 6;
    dim3 block(3);
    dim3 grid((nElem+block.x-1)/block.x); // Numero di blocchi necessari per coprire tutti gli elementi.

    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);

    checkIndex<<<grid, block>>>(); // Esecuzione del kernel

    cudaDeviceReset(); // Reset del device
    return(0);
}
```

Calcolo della Dimensione del Grid

```
dim3 grid((nElem+block.x-1)/block.x);
```

Calcoliamo il numero di blocchi necessari per coprire tutti gli elementi.

- Arrotondamento per eccesso per coprire tutti gli elementi
- Formula:** $(nElem + block.x - 1) / block.x$
- Esempio:** $(6 + 3 - 1) / 3 = 2$ blocchi

Motivazioni:

- Garantisce la copertura di tutti gli elementi, anche se non perfettamente divisibili
- Evita l'accesso a memoria fuori dai limiti dell'array
- Prepara per la gestione di dataset di dimensioni arbitrarie

Esempio di Dimensionamento Grid e Block in CUDA

- Questo esempio mostra come configurare e lanciare un kernel CUDA, e come **stampare le informazioni sugli indici e le dimensioni dei thread e dei blocchi**.

```
#include <cuda_runtime.h>
#include <stdio.h>

__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, // Indici del thread
        blockIdx.x, blockIdx.y, blockIdx.z, // Indici del blocco
        blockDim.x, blockDim.y, blockDim.z, // Dimensioni del blocco
        gridDim.x, gridDim.y, gridDim.z); // Dimensioni della griglia

int main(int argc, char **argv) {
    int nElem = 6;                                // Numero
    dim3 block(3);                                // Definizione
    dim3 grid((nElem+block.x-1)/block.x);          // Calcolo

    printf("grid.x %d grid.y %d grid.z %d\n", grid.x,
    printf("block.x %d block.y %d block.z %d\n", block.x,
    printf("block.x %d block.y %d block.z %d\n", block.y,
    printf("block.x %d block.y %d block.z %d\n", block.z);

    checkIndex<<<grid, block>>>();                // Esecuzione del kernel

    cudaDeviceReset();                             // Reset del device
    return(0);}
```

Comportamento Asincrono dei Kernel CUDA

- Asincronicità:** Le chiamate ai kernel CUDA sono asincrone; il controllo ritorna subito all'host dopo l'invocazione.
- Sincronizzazione Esplicita:** Usare `cudaDeviceSynchronize()` per attendere il completamento di tutti i kernel.

Esempio di Dimensionamento Grid e Block in CUDA

- Questo esempio mostra come configurare e lanciare un kernel CUDA, e come **stampare le informazioni sugli indici e le dimensioni dei thread e dei blocchi**.

Output

```
grid.x 2 grid.y 1 grid.z 1
block.x 3 block.y 1 block.z 1
threadIdx:(0, 0, 0) blockIdx:(1, 0, 0) blockDim:(3, 1, 1) gridDim:(2, 1, 1)
threadIdx:(1, 0, 0) blockIdx:(1, 0, 0) blockDim:(3, 1, 1) gridDim:(2, 1, 1)
threadIdx:(2, 0, 0) blockIdx:(1, 0, 0) blockDim:(3, 1, 1) gridDim:(2, 1, 1)
threadIdx:(0, 0, 0) blockIdx:(0, 0, 0) blockDim:(3, 1, 1) gridDim:(2, 1, 1)
threadIdx:(1, 0, 0) blockIdx:(0, 0, 0) blockDim:(3, 1, 1) gridDim:(2, 1, 1)
threadIdx:(2, 0, 0) blockIdx:(0, 0, 0) blockDim:(3, 1, 1) gridDim:(2, 1, 1)
```

Panoramica del CUDA Programming Model

- **Introduzione al Modello di Programmazione**
 - Concetti base e architettura CUDA
 - Ruolo di Host (CPU) e Device (GPU)
- **Gestione della Memoria in CUDA - Accenni**
 - Allocazione e trasferimento di memoria
 - Tipi di memoria: globale, condivisa (menzioni)
- **Organizzazione dei Thread**
 - Gerarchie: Grid, Block, Thread
 - Identificazione dei thread
- **Kernel CUDA**
 - Definizione e lancio dei kernel
 - Configurazione di griglia e blocchi
- **Tecniche di Mapping e Dimensionamento**
 - Esempio: Somma di array e mapping degli indici
 - Calcolo dinamico delle dimensioni della griglia
- **Analisi delle Prestazioni**
 - Correttezza dei risultati e gestione degli errori
 - Uso di strumenti di profiling (NVIDIA Nsight)
- **Applicazioni Pratiche**
 - Operazioni su matrici
 - Elaborazione di immagini (es. conversione RGB a grayscale)
 - Convoluzione 1D e 2D

Verifica del Kernel CUDA (Somma di Array)

- Il controllo dei kernel CUDA mira a confermare l'affidabilità dei calcoli eseguiti sulla GPU.

```
void checkResult(float *hostRef, ← Risultati attesi
                  float *gpuRef, ← della somma
                  const int N) {   Risultati calcolati
                    dal kernel

double epsilon = 1.0E-8;
int match = 1;
for (int i = 0; i < N; i++) {
    if (abs(hostRef[i] - gpuRef[i]) > epsilon)
    {
        match = 0;
        printf("Arrays do not match!\n");
        printf("host %5.2f gpu %5.2f
              at current %d\n",
              hostRef[i], gpuRef[i], i);
        break; }}
```

if (match) **printf**("Arrays match.\n\n");

Suggerimenti per la Verifica (basic)

- Confronto sistematico:** Verifica ogni elemento degli array per assicurarsi che i risultati del kernel corrispondano ai valori attesi.
- Tolleranza:** Usa una piccola tolleranza (epsilon) per confronti in virgola mobile. Possibilità di errori di arrotondamento legate alla natura delle rappresentazioni numeriche nei computer.
- (Alternativa) Configurazione <<< 1, 1>>>:**
 - Forza l'esecuzione del kernel con un solo blocco e un thread.
 - Emula un'implementazione sequenziale.

Gestione degli Errori in CUDA

Il Problema

- **Asincronicità:** Molte chiamate CUDA sono asincrone, rendendo difficile associare un errore alla specifica chiamata che lo ha causato.
- **Complessità di Debugging:** Gli errori possono manifestarsi in punti del codice distanti da dove sono stati generati.
- **Gestione Manuale:** Controllare ogni chiamata CUDA manualmente è tedioso e soggetto a errori.

Macro CHECK

```
// Fornisce file, riga, codice e descrizione dell'errore.

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

Gestione degli Errori in CUDA

Il Problema

- **Asincronicità:** Molte chiamate CUDA sono asincrone, rendendo difficile associare un errore alla specifica chiamata che lo ha causato.
- **Complessità di Debugging:** Gli errori possono manifestarsi in punti del codice distanti da dove sono stati generati.
- **Gestione Manuale:** Controllare ogni chiamata CUDA manualmente è tedioso e soggetto a errori.

Esempi di Utilizzo

```
// e.g. Per operazioni di memoria
CHECK(cudaMemcpy(d_C, gpuRef, nBytes, cudaMemcpyHostToDevice));

// e.g. Dopo il lancio di un kernel
kernel_function<<<numBlocks, blockSize>>>(argument list);
CHECK(cudaDeviceSynchronize());
```

Profiling delle Prestazioni dei Kernel CUDA

Introduzione al Profiling

- Misurare e ottimizzare le prestazioni dei kernel CUDA è cruciale per garantire l'efficienza del codice.
- Il profiling permette di analizzare l'uso delle risorse e identificare le aree di miglioramento.

Importanza della Misurazione del Tempo

- Identificazione dei Colli di Bottiglia: Individuare le sezioni di codice che limitano le prestazioni. Generalmente una implementazione *naive* del kernel non garantisce prestazioni ottimali.
- Analisi degli Effetti delle Modifiche: Valutare come le modifiche al codice influenzano le prestazioni.
- Confronto tra Implementazioni: Valutare le prestazioni tra diverse strategie di implementazione.
- Analisi del Bilanciamento Carico/Calcolo: Verificare se il carico di lavoro è distribuito in modo efficiente tra i thread e i blocchi CUDA.

Metodi Principali

- 1. Timer CPU: Semplice e diretto, utilizza funzioni di sistema per ottenere il tempo di esecuzione.
- 2. NVIDIA Profiler (deprecato): Strumento da riga di comando per analizzare attività di CPU e GPU.
- 3. NVIDIA Nsight Systems e Nsight Compute: Strumenti avanzati per analisi approfondita e ottimizzazione a livello di sistema e kernel.

Metodo 1: Timer CPU

- Il CPU Timer si distingue come una soluzione **pratica** ed **efficace** per la misurazione temporale dei kernel CUDA, bilanciando la semplicità di implementazione con la capacità di fornire dati temporali dal punto di vista dell'host.

Funzione del Timer della CPU

```
#include <time.h>

double cpuSecond() {
    struct timespec ts;
    timespec_get(&ts, TIME_UTC);
    return ((double)ts.tv_sec + (double)ts.tv_nsec * 1.e-9);
}
```

- La funzione utilizza **timespec_get()** per ottenere il **tempo corrente** del sistema.
- Restituisce il tempo in secondi, combinando secondi e nanosecondi.
- La precisione è nell'ordine dei **nanosecondi**.

Metodo 1: Timer CPU

Utilizzo Per Misurare un Kernel CUDA

```
double iStart = cpuSecond(); // Registra il tempo di inizio  
kernel_name<<<grid, block>>>(argument list); // Lancia il kernel CUDA  
cudaDeviceSynchronize(); // Attende il completamento del kernel  
double iElaps = cpuSecond() - iStart; // Calcola il tempo trascorso
```

- La chiamata a `cudaDeviceSynchronize()` è cruciale per assicurare che tutto il lavoro sulla GPU sia completato prima di misurare il tempo finale. Questo è necessario poiché le chiamate ai kernel CUDA sono **asincrone** rispetto all'host (senza rifletterebbe solo il tempo di lancio del kernel).
- Il tempo misurato include l'overhead di lancio del kernel e la sincronizzazione.

Pro

- **Facile** da implementare e utilizzare.
- Non richiede librerie CUDA **specifiche** per il timing.
- Funziona su **qualsiasi sistema** con supporto CUDA.
- Efficace per kernel lunghi e misure approssimative.

Contro

- **Impreciso** per kernel molto brevi (< 1 ms).
- Include **overhead** non relativo all'esecuzione del kernel (es., sistema operativo, utilizzo CPU, etc.).
- Non fornisce dettagli sulle **fasi interne** del kernel.
- Precisione influenzata dal **carico dell'host**.

Metodo 1: Timer CPU

Somma di due array

1/3

```
int main(int argc, char **argv)
{
    printf("%s Starting...\n", argv[0]);

    // Configurazione del device
    int dev = 0;
    cudaDeviceProp deviceProp;
    CHECK(cudaGetDeviceProperties(&deviceProp, dev)); // Ottiene le proprietà del device
    printf("Using Device %d: %s\n", dev, deviceProp.name);
    CHECK(cudaSetDevice(dev)); // Seleziona il device CUDA da utilizzare

    // Dimensione dei vettori
    int nElem = 1 << 24; // 2^24 elementi (16M) - bit shifting
    printf("Vector size %d\n", nElem);

    // Allocazione della memoria host
    size_t nBytes = nElem * sizeof(float);
    float *h_A, *h_B, *hostRef, *gpuRef;
    h_A = (float *)malloc(nBytes); // Alloca memoria per il vettore A su host
    h_B = (float *)malloc(nBytes); // Alloca memoria per il vettore B su host
    hostRef = (float *)malloc(nBytes); // Alloca memoria per il risultato calcolato su host
    gpuRef = (float *)malloc(nBytes); // Alloca memoria per il risultato calcolato su GPU
```

Metodo 1: Timer CPU

Somma di due array

1/3

```
int main(int argc, char **argv)
{
    printf("%s Starting...\n", argv[0]);

    // Configurazione del device
    int dev = 0;
    cudaDeviceProp deviceProp;
    CHECK(cudaGetDeviceProperties(&deviceProp, dev));
    printf("Using Device %d: %s\n", dev, deviceProp.name);
    CHECK(cudaSetDevice(dev)); // Selezione del dispositivo

    // Dimensione dei vettori
    int nElem = 1 << 24; // 2^24 elementi
    printf("Vector size %d\n", nElem);

    // Allocazione della memoria host
    size_t nBytes = nElem * sizeof(float);
    float *h_A, *h_B, *hostRef, *gpuRef;
    h_A = (float *)malloc(nBytes); // Alloca memoria per il vettore A su host
    h_B = (float *)malloc(nBytes); // Alloca memoria per il vettore B su host
    hostRef = (float *)malloc(nBytes); // Alloca memoria per il risultato calcolato su host
    gpuRef = (float *)malloc(nBytes); // Alloca memoria per il risultato calcolato su GPU
```

```
int dev = 0;
```

- Rappresenta l'**indice del dispositivo CUDA** che si intende utilizzare.
- 0 solitamente si riferisce al primo dispositivo CUDA disponibile nel sistema.

Alternative e Pratiche Comuni

1. Selezione del dispositivo tramite argomenti:

```
if (argc > 1) dev = atoi(argv[1]);
```

2. Utilizzo di Variabili d'Ambiente:

```
char* deviceIndex =
getenv("CUDA_VISIBLE_DEVICES");
if (deviceIndex) dev=atoi(deviceIndex);
```

e

Metodo 1: Timer CPU

1/3

Somma di due array

```
int main(int argc, char **argv)
{
    printf("%s Starting...\n", argv[0]);

    // Configurazione del device
    int dev = 0;
    cudaDeviceProp deviceProp;
    CHECK(cudaGetDeviceProperties(&deviceProp, dev));
    printf("Using Device %d: %s\n", dev, deviceProp.name);
    CHECK(cudaSetDevice(dev)); // Selezione dispositivo

    // Dimensione dei vettori
    int nElem = 1 << 24; // 2^24 elementi
    printf("Vector size %d\n", nElem);

    // Allocazione della memoria host
    size_t nBytes = nElem * sizeof(float);
    float *h_A, *h_B, *hostRef, *gpuRef;
    h_A = (float *)malloc(nBytes); // Alloca memoria per il vettore A su host
    h_B = (float *)malloc(nBytes); // Alloca memoria per il vettore B su host
    hostRef = (float *)malloc(nBytes); // Alloca memoria per il risultato calcolato su host
    gpuRef = (float *)malloc(nBytes); // Alloca memoria per il risultato calcolato su GPU
```

cudaDeviceProp

- **Struttura** definita nell'API CUDA.
- Contiene campi che descrivono in dettaglio le **caratteristiche** di un dispositivo CUDA (GPU).
- Utilizzata per interrogare e memorizzare informazioni complete su un dispositivo CUDA specifico.

Campi Principali

- **name**: Nome del dispositivo (es. "GeForce RTX 3080").
- **totalGlobalMem**: Memoria globale totale disponibile.
- **clockRate**: Frequenza di clock della GPU.
- **maxThreadsPerBlock**: Numero massimo di thread per blocco.
- **maxThreadsDim**: Dimensioni massime della griglia di thread.
- **maxGridSize**: Dimensioni massime della griglia di blocchi.
- etc..

Metodo 1: Timer CPU

Somma di due array

1/3

```
int main(int argc, char **argv)
{
    printf("%s Starting...\n", argv[0]);

    // Configurazione del device
    int dev = 0;
    cudaDeviceProp deviceProp;
    CHECK(cudaGetDeviceProperties(&deviceProp, dev));
    printf("Using Device %d: %s\n", dev, deviceProp.name);
    CHECK(cudaSetDevice(dev)); // Selezione dispositivo CUDA

    // Dimensione dei vettori
    int nElem = 1 << 24; // 2^24 elementi (16M) - bit shifting
    printf("Vector size %d\n", nElem);

    // Allocazione della memoria host
    size_t nBytes = nElem * sizeof(float);
    float *h_A, *h_B, *hostRef, *gpuRef;
    h_A = (float *)malloc(nBytes); // Alloca memoria per il vettore A su host
    h_B = (float *)malloc(nBytes); // Alloca memoria per il vettore B su host
    hostRef = (float *)malloc(nBytes); // Alloca memoria per il risultato calcolato su host
    gpuRef = (float *)malloc(nBytes); // Alloca memoria per il risultato calcolato su GPU
```

cudaSetDevice(dev)

- **Imposta** il dispositivo CUDA attivo per le operazioni successive
- Assicura che tutte le **allocazioni e le operazioni CUDA** successive utilizzino questo dispositivo specifico.

Metodo 1: Timer CPU

2/3

```
// Inizializzazione dei dati su host
double iStart, iElaps;
iStart = cpuSecond();
initialData(h_A, nElem); // Inizializza il vettore A
initialData(h_B, nElem); // Inizializza il vettore B
iElaps = cpuSecond() - iStart;
printf("Data initialization time: %f sec\n", iElaps);
memset(hostRef, 0, nBytes); // Inizializza a zero il vettore risultato su host
memset(gpuRef, 0, nBytes); // Inizializza a zero il vettore risultato della GPU

// Somma dei vettori su host per verifica
iStart = cpuSecond();
sumArraysOnHost(h_A, h_B, hostRef, nElem); // Calcola la somma su CPU per confronto
iElaps = cpuSecond() - iStart;
printf("sumArraysOnHost elapsed %f sec\n", iElaps);

// Allocazione della memoria su device
float *d_A, *d_B, *d_C;
CHECK(cudaMalloc((float**)&d_A, nBytes)); // Alloca memoria per A su GPU
CHECK(cudaMalloc((float**)&d_B, nBytes)); // Alloca memoria per B su GPU
CHECK(cudaMalloc((float**)&d_C, nBytes)); // Alloca memoria per il risultato su GPU

// Copia dei dati dall'host al device
CHECK(cudaMemcpy(d_A, h_A, nBytes, cudaMemcpyHostToDevice)); // Copia A su GPU
CHECK(cudaMemcpy(d_B, h_B, nBytes, cudaMemcpyHostToDevice)); // Copia B su GPU
```

Metodo 1: Timer CPU

2/3

```
// Inizializzazione dei dati su host
double iStart, iElaps;
iStart = cpuSecond();
initialData(h_A, nElem); // Inizializza un array di float con valori casuali (compresi fra 0 e 25.5)
initialData(h_B, nElem); // Inizializza un array di float con valori casuali (compresi fra 0 e 25.5)
iElaps = cpuSecond() - iStart;
printf("Data initialization time: %f sec\n", iElaps);
memset(hostRef, 0, nBytes); // Inizializza un array di float con valori casuali (compresi fra 0 e 25.5)
memset(gpuRef, 0, nBytes); // Inizializza un array di float con valori casuali (compresi fra 0 e 25.5)

// Somma dei vettori su host per verificare la correttezza della somma
iStart = cpuSecond();
sumArraysOnHost(h_A, h_B, hostRef, nElem);
iElaps = cpuSecond() - iStart;
printf("sumArraysOnHost elapsed %f sec\n", iElaps);

// Allocazione della memoria su device
float *d_A, *d_B, *d_C;
CHECK(cudaMalloc((float**)&d_A, nBytes)); // Alloca memoria per A su GPU
CHECK(cudaMalloc((float**)&d_B, nBytes)); // Alloca memoria per B su GPU
CHECK(cudaMalloc((float**)&d_C, nBytes)); // Alloca memoria per il risultato su GPU

// Copia dei dati dall'host al device
CHECK(cudaMemcpy(d_A, h_A, nBytes, cudaMemcpyHostToDevice)); // Copia A su GPU
CHECK(cudaMemcpy(d_B, h_B, nBytes, cudaMemcpyHostToDevice)); // Copia B su GPU
```

Metodo 1: Timer CPU

3/3

```
// Configurazione del kernel
dim3 block(1024); // Dimensione del blocco: 1024 threads
dim3 grid((nElem + block.x - 1) / block.x); // Calcola il numero di blocchi necessari

// Esecuzione del kernel su device
iStart = cpuSecond();
sumArraysOnGPU<<<grid, block>>>(d_A, d_B, d_C, nElem); // Lancia il kernel CUDA
CHECK(cudaDeviceSynchronize()); // Attende il completamento del kernel
iElaps = cpuSecond() - iStart;
printf("sumArraysOnGPU <<%d, %d>>> elapsed %f sec\n", grid.x, block.x, iElaps);

// Copia dei risultati dal device all'host
CHECK(cudaMemcpy(hostRef, d_C, nBytes, cudaMemcpyDeviceToHost)); // Copia dalla GPU

// Verifica dei risultati
checkResult(hostRef, gpuRef, nElem); // Confronta i risultati di CPU e GPU

// Liberazione della memoria su device
CHECK(cudaFree(d_A)); // Libera la memoria di A su GPU
CHECK(cudaFree(d_B)); // Libera la memoria di B su GPU
CHECK(cudaFree(d_C)); // Libera la memoria del risultato su GPU

// Liberazione della memoria su host
free(h_A); // Libera la memoria di A su host
free(h_B); // Libera la memoria di B su host
free(hostRef); // Libera la memoria del risultato CPU su host
free(gpuRef); // Libera la memoria del risultato GPU su host
return 0;}
```

Metodo 1: Timer CPU - Compilazione e Esecuzione

Compilazione con nvcc

```
nvcc array_sum.cu -o array_sum
```

Esecuzione e Risultato

```
./array_sum Starting...
Using Device 0: NVIDIA GeForce RTX 3090
Vector size 16777216
Data initialization time: 0.425670 sec
sumArraysOnHost elapsed 0.033285 sec
sumArraysOnGPU <<<16384, 1024>>> elapsed 0.000329 sec
Arrays match.
```

WorkStation

Intel Core i9-10920X (CPU)

- Cores: 12 fisici (24 Threads)
- Base Clock: 3.50 GHz

NVIDIA GeForce RTX 3090 (GPU)

- CUDA Cores: 10,496
- Base Clock: 1.40 GHz

Accesso ai Dati

- **Efficienza della GPU:** La GPU esegue l'operazione circa 101 volte più velocemente della CPU ($0.033285 / 0.000329 \approx 101$)
- **Overhead di Inizializzazione:** L'inizializzazione dei dati (0.425670 s) richiede circa 13 volte più tempo dell'elaborazione CPU.
- **Latenza vs Throughput:** Nonostante la CPU abbia una frequenza di clock più alta, la GPU supera significativamente le prestazioni grazie al massiccio parallelismo.

Metodo 1: Timer CPU - Compilazione e Esecuzione

Compilazione con nvcc

```
nvcc array_sum.cu -o array_sum
```

Esecuzione e Risultato

```
./array_sum Starting...
Using Device 0: NVIDIA GeForce RTX 3070 Laptop GPU
Vector size 16777216
Data initialization time: 0.439789 sec
sumArraysOnHost elapsed 0.039411 sec
sumArraysOnGPU <<<16384, 1024>>> elapsed 0.000650 sec
Arrays match.
```

Laptop

Intel(R) Core(TM) i7-11800H

- Cores: 8 fisici (16 Threads)
- Base Clock: 2.30 GHz

NVIDIA GeForce RTX 3070 (GPU)

- CUDA Cores: 5,120
- Base Clock: 1.50 GHz

Accesso ai Dati

- **Efficienza della GPU:** La GPU esegue l'operazione circa 60 volte più velocemente della CPU ($0.039411 / 0.000650 \approx 60$)
- **Overhead di Inizializzazione:** L'inizializzazione dei dati (0.439789 s) richiede circa 11 volte più tempo dell'elaborazione CPU.
- **Latenza vs Throughput:** Nonostante la CPU abbia una frequenza di clock più alta, la GPU supera significativamente le prestazioni grazie al massiccio parallelismo.
- **Confronto fra GPU:** Per questa operazione, la NVIDIA GeForce RTX 3090 è circa 1.97 volte più veloce della RTX 3070, con un tempo di esecuzione di 0.000329 secondi rispetto a 0.000650 secondi.

Metodo 2: NVIDIA Profiler [5.0 <= Compute Capability < 8.0]

Dalla CUDA 5.0 è disponibile **nvprof**, uno strumento da riga di comando per raccogliere informazioni sull'attività di CPU e GPU dell'applicazione, inclusi kernel, trasferimenti di memoria e chiamate all'API CUDA.

Come si usa? ([Documentazione Online](#))

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

Ulteriori informazioni sulle opzioni di **nvprof** possono essere trovate utilizzando il seguente comando:

```
$ nvprof --help
```

Nel nostro esempio:

```
$ nvprof ./array_sum
```

Nota

- **nvprof** non è supportato su dispositivi con compute capability **8.0 e superiori**. Per questi dispositivi, si consiglia di utilizzare **NVIDIA Nsight Systems** per il tracing della GPU e il campionamento della CPU, e **NVIDIA Nsight Compute** per il profiling della GPU.
- **nvprof** è disponibile su **Google Colab** (GPU NVIDIA Tesla T4 - Compute Capability: 7.5).

Metodo 2: NVIDIA Profiler [5.0 <= Compute Capability < 8.0]

Esempio di Profilazione su Google Colab

```
[3] !nvcc array_sum.cu -o array_sum  
  
[4] !nvprof ./array_sum 16777216 256  
  
./array_sum Starting...  
==1380== NVPROF is profiling process 1380, command: ./array_sum 16777216 256  
Using Device 0: Tesla T4  
Vector size: 16777216  
Block size: 256  
sumArrayOnHost Time elapsed 0.051397 sec  
sumArrayOnGPU <<<65536, 256>>> Time elapsed 0.001025 sec  
Arrays match.  
  
==1380== Profiling application: ./array_sum 16777216 256  
==1380== Profiling result:  


| Type            | Time(%) | Time     | Calls | Avg      | Min      | Max             | Name                                       |
|-----------------|---------|----------|-------|----------|----------|-----------------|--------------------------------------------|
| GPU activities: | 64.94%  | 29.658ms | 2     | 14.829ms | 14.756ms | 14.903ms        | [CUDA memcpy HtoD]                         |
|                 | 33.36%  | 15.238ms | 1     | 15.238ms | 15.238ms | 15.238ms        | [CUDA memcpy DtoH]                         |
|                 | 1.70%   | 776.75us | 1     | 776.75us | 776.75us | 776.75us        | sumArrayOnGPU(float*, float*, float*, int) |
| API calls:      | 65.05%  | 95.410ms | 1     | 95.410ms | 95.410ms | 95.410ms        | cudaSetDevice                              |
|                 | 31.16%  | 45.708ms | 3     | 15.236ms | 14.934ms | 15.655ms        | cudaMemcpy                                 |
|                 | 2.43%   | 3.5698ms | 3     | 1.1899ms | 266.77us | 2.1709ms        | cudaFree                                   |
|                 | 0.53%   | 779.98us | 1     | 779.98us | 779.98us | 779.98us        | cudaDeviceSynchronize                      |
|                 | 0.47%   | 695.53us | 3     | 231.84us | 146.46us | 381.48us        | cudaMalloc                                 |
|                 | 0.16%   | 239.79us | 1     | 239.79us | 239.79us | 239.79us        | cudaLaunchKernel                           |
|                 | 0.10%   | 142.14us | 114   | 1.2460us | 141ns    | 57.528us        | cuDeviceGetAttribute                       |
|                 | 0.07%   | 102.36us | 1     | 102.36us | 102.36us | 102.36us        | cudaGetDeviceProperties                    |
|                 | 0.01%   | 13.085us | 1     | 13.085us | 13.085us | 13.085us        | cuDeviceGetName                            |
|                 | 0.00%   | 6.7260us | 1     | 6.7260us | 6.7260us | 6.7260us        | cuDeviceGetPCIBusId                        |
|                 | 0.00%   | 4.7470us | 1     | 4.7470us | 4.7470us | 4.7470us        | cuDeviceTotalMem                           |
|                 | 0.00%   | 2.5600us | 3     | 853ns    | 249ns    | 2.0180us        | cuDeviceGetCount                           |
|                 | 0.00%   | 1.0920us | 2     | 546ns    | 164ns    | 928ns           | cuDeviceGet                                |
|                 | 0.00%   | 835ns    | 1     | 835ns    | 835ns    | 835ns           | cuModuleGetLoadingMode                     |
| 0.00%           | 236ns   | 1        | 236ns | 236ns    | 236ns    | cuDeviceGetUuid |                                            |


```

NVIDIA Profiler

Include informazioni su:

- Attività GPU:** Tempi per trasferimenti di memoria (Host a Device e Device a Host) e l'esecuzione del kernel.
- Chiamate API:** Tempi per funzioni come `cudaSetDevice`, `cudaMemcpy`, e gestione della memoria.

Metodo 3.1 - NVIDIA Nsight Systems

Cos'è? ([Documentazione Online](#))

- Strumento avanzato di **profilazione** e **analisi** delle prestazioni a livello di sistema.
- **Visione d'insieme** delle prestazioni dell'applicazione, inclusi CPU, GPU e interazioni di sistema.
- Permette di:
 - Identificare **colli di bottiglia** nelle prestazioni.
 - Analizzare l'**overhead** delle chiamate API.
 - Esaminare le operazioni di **input/output**.
 - **Ottimizzare** il flusso di lavoro dell'applicazione.



Caratteristiche Chiave

- **Visualizzazione grafica** delle timeline di esecuzione.
- **Analisi** dei kernel CUDA.
- **Monitoraggio** dell'utilizzo di memoria e cache.
- **Supporto** per sistemi multi-GPU.

Output e Analisi

- Genera report dettagliati in vari formati (HTML, SQLite).
- Fornisce grafici interattivi per visualizzare l'esecuzione nel tempo.
- Permette di zoomare e navigare attraverso diverse sezioni dell'esecuzione.
- Evidenzia automaticamente aree di potenziale ottimizzazione.

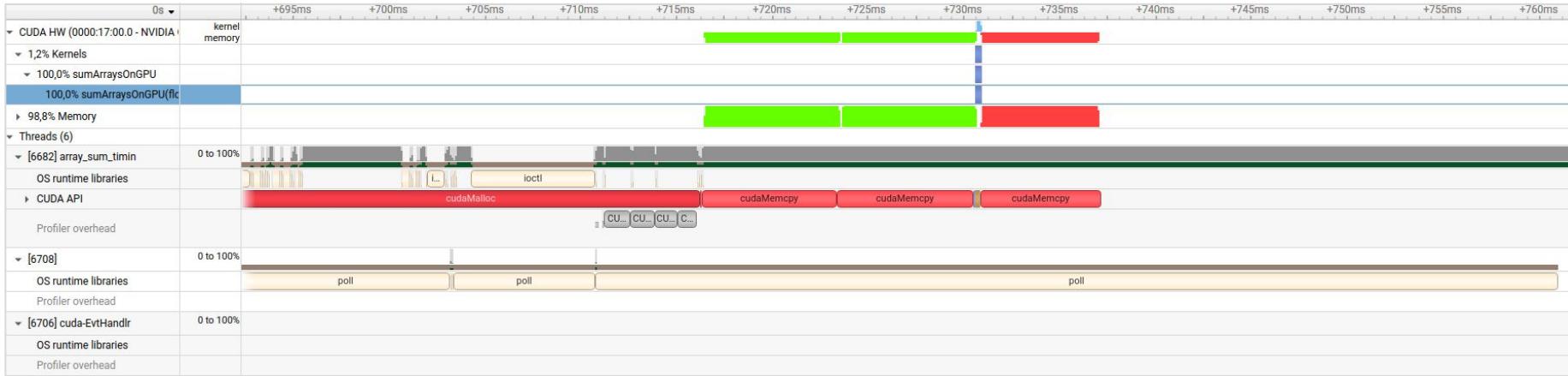
Come si usa?

```
$ nsys profile --stats=true ./array_sum
```

- Questo comando avvia il profiler e fornisce un'analisi dettagliata delle prestazioni (non disponibile su Colab).

Metodo 3.1 - NVIDIA Nsight Systems

Timeline View



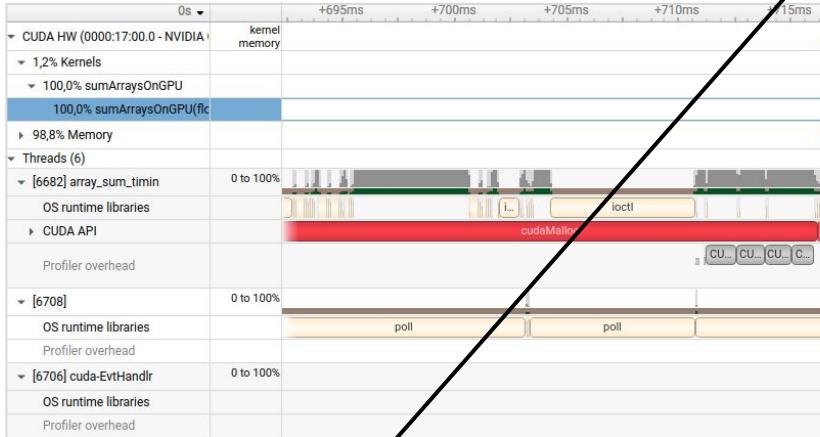
CUDA Summary (API/Kernels/MemOps)

Dim. Array (2^{24})

Time	Total Time	Instances	Avg	Med	Min	Max	StdDev	Category	Operation
55.0%	56,034 ms	3	18,678 ms	44,942 µs	39,524 µs	55,950 ms	32,278 ms	CUDA_API	cudaMalloc
20.0%	20,465 ms	3	6,822 ms	7,101 ms	6,235 ms	7,129 ms	508,250 µs	CUDA_API	cudaMemcpy
14.0%	14,086 ms	2	7,043 ms	7,043 ms	7,019 ms	7,066 ms	33,012 µs	MEMORY_OPER	[CUDA memcpy Host-to-Device]
6.0%	6,117 ms	1	6,117 ms	6,117 ms	6,117 ms	6,117 ms	0 ns	MEMORY_OPER	[CUDA memcpy Device-to-Host]
3.0%	3,350 ms	3	1,117 ms	1,079 ms	152,841 µs	2,118 ms	983,174 µs	CUDA_API	cudaFree
0.0%	305,355 µs	1	305,355 µs	305,355 µs	305,355 µs	305,355 µs	0 ns	CUDA_API	cudaDeviceSynchronize
0.0%	244,222 µs	1	244,222 µs	244,222 µs	244,222 µs	244,222 µs	0 ns	CUDA_KERNEL	sumArraysOnGPU(float *, float *, float *, int)
0.0%	25,741 µs	1	25,741 µs	25,741 µs	25,741 µs	25,741 µs	0 ns	CUDA_API	cudaLaunchKernel

Metodo 3.1 - NVIDIA Nsight Systems

Timeline View



Analisi del Profiling

- **Gestione memoria domina:**
 - Allocazione (cudaMalloc): 55%
 - Trasferimenti (cudaMemcpy - operazioni di memoria): 20%
- **Esecuzione kernel GPU trascurabile:** 0.0% (244,222µs)
- **Operazioni ausiliarie minime:**
 - cudaFree: 3%
 - cudaDeviceSynchronize: ~0%
- **Conclusione:** Prestazioni limitate dalla gestione memoria, non dal calcolo GPU. Ottimizzazione dovrebbe concentrarsi su riduzione allocazioni e trasferimenti dati.

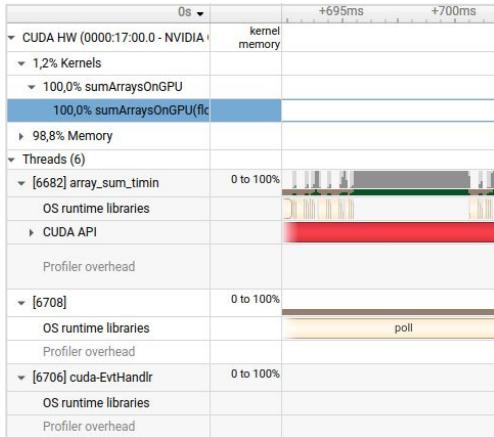
CUDA Summary (API/Kernels/MemOps)

Dim. Array (2^{24})

Time	Total Time	Instances	Avg	Med	Min	Max	StdDev	Category	Operation
55.0%	56,034 ms	3	18,678 ms	44,942 µs	39,524 µs	55,950 ms	32,278 ms	CUDA_API	cudaMalloc
20.0%	20,465 ms	3	6,822 ms	7,101 ms	6,235 ms	7,129 ms	508,250 µs	CUDA_API	cudaMemcpy
14.0%	14,080 ms	2	7,043 ms	7,043 ms	7,019 ms	7,066 ms	33,012 µs	MEMORY_OPER	[CUDA memcpy Host-to-Device]
6.0%	6,117 ms	1	6,117 ms	6,117 ms	6,117 ms	6,117 ms	0 ns	MEMORY_OPER	[CUDA memcpy Device-to-Host]
3.0%	3,350 ms	3	1,117 ms	1,079 ms	152,841 µs	2,118 ms	983,174 µs	CUDA_API	cudaFree
0.0%	305,355 µs	1	305,355 µs	305,355 µs	305,355 µs	305,355 µs	0 ns	CUDA_API	cudaDeviceSynchronize
0.0%	244,222 µs	1	244,222 µs	244,222 µs	244,222 µs	244,222 µs	0 ns	CUDA_KERNEL	sumArraysOnGPU(float *, float *, float *, int)
0.0%	25,741 µs	1	25,741 µs	25,741 µs	25,741 µs	25,741 µs	0 ns	CUDA_API	cudaLaunchKernel

Metodo 3.1 - NVIDIA Nsight Systems

Timeline View



Timer CPU (~329 µs) vs Nsight Systems (244 µs)

- Precisione**
 - Nsight Systems**: Misurazioni precise, direttamente dalla GPU.
 - Timer CPU**: Meno preciso, include overhead extra (lancio, sincronizzazione).
- Contesto**
 - Nsight Systems**: Visione isolata del tempo di esecuzione del kernel GPU.
 - Timer CPU**: Include attività di sistema e altri processi, meno preciso.
- Affidabilità**
 - Nsight Systems**: Misurazioni stabili, meno influenzate da fattori esterni.
 - Timer CPU**: Vulnerabile alle fluttuazioni del sistema, meno affidabile.
- Implicazioni per lo sviluppo**
 - Nsight Systems**: Ottimizzazioni critiche, analisi approfondite, profiling accurato.
 - Timer CPU**: Stime approssimative nelle fasi iniziali, non per analisi dettagliate.

CUDA Summary (API/Kernels)

Time	Total Time	Instances	Avg	Med	Min	Max	StdDev	Category	Operation
55.0%	56,034 ms	3	18,678 ms	44,942 µs	39,524 µs	55,950 ms	32,278 ms	CUDA_API	cudaMalloc
20.0%	20,465 ms	3	6,822 ms	7,101 ms	6,235 ms	7,129 ms	508,250 µs	CUDA_API	cudaMemcpy
14.0%	14,086 ms	2	7,043 ms	7,043 ms	7,019 ms	7,066 ms	33,012 µs	MEMORY_OPER	[CUDA memcpy Host-to-Device]
6.0%	6,117 ms	1	6,117 ms	6,117 ms	6,117 ms	6,117 ms	0 ns	MEMORY_OPER	[CUDA memcpy Device-to-Host]
3.0%	3,350 ms	3	1,117 ms	1,079 ms	152,841 µs	2,119 ms	983,174 µs	CUDA_API	cudaFree
0.0%	305,355 µs	1	305,355 µs	305,355 µs	305,355 µs	305,355 µs	0 ns	CUDA_API	cudaDeviceSynchronize
0.0%	244,222 µs	1	244,222 µs	244,222 µs	244,222 µs	244,222 µs	0 ns	CUDA_KERNEL	sumArraysOnGPU(float *, float *, float *, int)
0.0%	25,741 µs	1	25,741 µs	25,741 µs	25,741 µs	25,741 µs	0 ns	CUDA_API	cudaLaunchKernel

Ottimizzazione della Gestione della Memoria in CUDA

Sfide

- **Trasferimenti lenti:** I trasferimenti di dati tra host e device attraverso il bus PCIe rappresentano un collo di bottiglia.
- **Allocazione sulla GPU:** L'allocazione di memoria sulla GPU è un'operazione relativamente lenta.

Best Practices

Minimizzare i Trasferimenti di Memoria

- I trasferimenti di dati tra host e device hanno un'alta latenza.
- Raggruppare i dati in buffer più grandi per ridurre i trasferimenti e sfruttare la larghezza di banda.

Allocazione e Deallocazione Efficiente

- L'allocazione di memoria sulla GPU tramite `cudaMalloc` è un'operazione relativamente lenta.
- Allocare la memoria una volta all'inizio dell'applicazione e riutilizzarla quando possibile.
- Liberare la memoria con `cudaFree` quando non serve più, per evitare perdite e sprechi di risorse.

Sfruttare la Shared Memory (lo vedremo)

- La shared memory è una memoria on-chip a bassa latenza accessibile a tutti i thread di un blocco.
- Utilizzare la shared memory per i dati frequentemente acceduti e condivisi tra i thread di un blocco per ridurre l'accesso alla memoria globale più lenta.

Metodo 2.2 - NVIDIA Nsight Compute

Cos'è? ([Documentazione Online](#))

- Strumento di **profilazione e analisi approfondita** per singoli kernel CUDA.
- Fornisce **metriche dettagliate** sulle prestazioni a livello di kernel.
- Permette di:
 - **Analizzare** l'utilizzo delle risorse GPU.
 - Identificare **colli di bottiglia** nelle prestazioni dei kernel.
 - Offre **report dettagliati** che possono essere utilizzati per ottimizzare il codice a livello di kernel.



Output e Analisi

- Genera report dettagliati in formato GUI o CLI.
- Fornisce grafici e tabelle per visualizzare l'utilizzo delle risorse.
- Permette l'analisi riga per riga del codice sorgente in relazione alle metriche.
- Offre raccomandazioni specifiche per l'ottimizzazione basate sui dati raccolti.

Caratteristiche chiave

- **Analisi** dettagliata delle metriche hardware per ogni kernel.
- **Visualizzazione grafica** dell'utilizzo della memoria.
- **Confronto** side-by-side di diverse esecuzioni dei kernel.
- **Suggerimenti automatici** per l'ottimizzazione.

Come si usa?

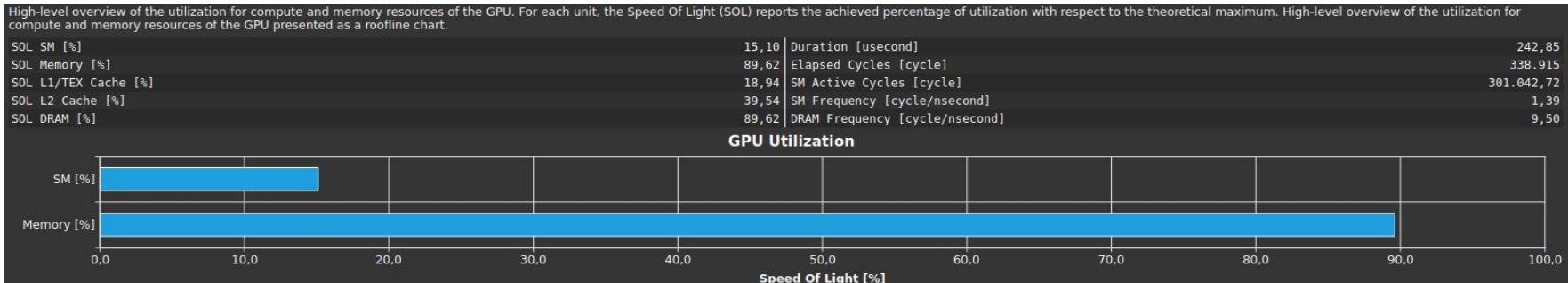
Necessario per generare file per la visualizzazione grafica.
Rimuovere per visualizzazione a terminale

```
$ ncu --set full [-o test_report] ./array_sum
```

- Avvia il profiler Nsight Compute e fornisce un'analisi dettagliata delle prestazioni dei kernel CUDA.

Metodo 2.2 - NVIDIA Nsight Compute

- Utilizzando NVIDIA Nsight Compute, si può esaminare il **tempo di esecuzione del kernel**, evidenziando dettagli cruciali sull'uso della memoria e delle unità di calcolo.



Tempo di esecuzione del kernel

- 242,85 μ s

Throughput (specifico per l'esecuzione del kernel)

- Compute (SM):** 15,10% - Basso utilizzo delle unità di calcolo
- Memoria:** 89,62% - Alto utilizzo della banda di memoria
- Nota:** Questi valori si riferiscono all'efficienza interna del kernel, non alle operazioni `cudaMalloc/cudaMemcpy` viste in Nsight Systems.

Considerazioni

- Il kernel stesso è **memory-bound**, un aspetto non evidente dall'analisi di Nsight Systems.
- Nsight Compute rivela che anche all'interno del kernel l'accesso alla memoria è il **collo di bottiglia**.
- L'ottimizzazione dovrebbe considerare sia le **operazioni di memoria** a livello API (viste in Nsight Systems) che il **pattern di accesso alla memoria** all'interno del kernel (evidenziato da Nsight Compute).

Nvidia Nsight Systems vs. Compute

In Sintesi

- **Nsight Systems** è uno strumento di analisi delle prestazioni a livello di sistema per identificare i colli di bottiglia delle prestazioni in tutto il sistema, inclusa la CPU, la GPU e altri componenti hardware.
- **Nsight Compute** è uno strumento di analisi e debug delle prestazioni a livello di kernel per ottimizzare le prestazioni e l'efficienza di singoli kernel CUDA.

Scegliere lo Strumento Giusto:

- **Nsight Systems**: Perfetto per ottenere una panoramica delle prestazioni dell'applicazione nel suo complesso, identificare aree di interesse (CPU bound vs. GPU bound) e analizzare le interazioni tra CPU e GPU.
- **Nsight Compute**: Ideale per analisi approfondite di kernel specifici, ottimizzazione di codice CUDA e identificazione di colli di bottiglia a basso livello.

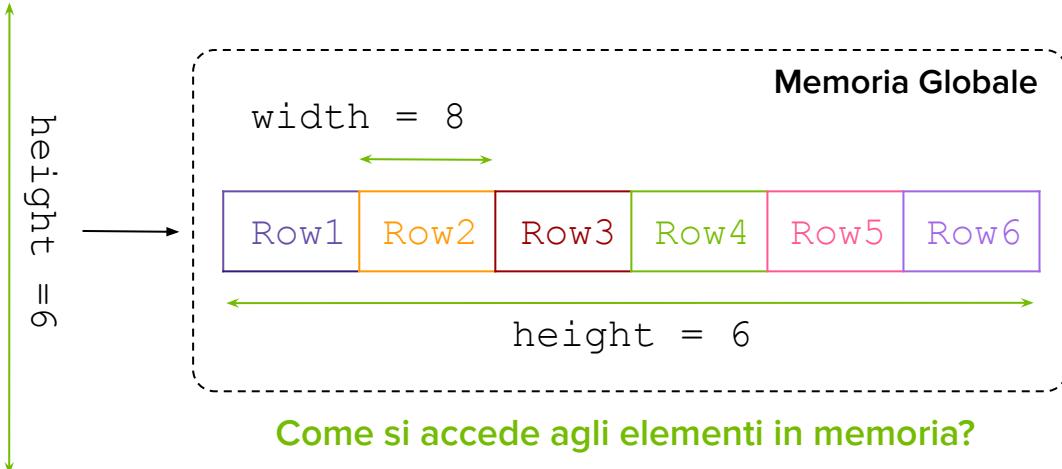
Panoramica del CUDA Programming Model

- **Introduzione al Modello di Programmazione**
 - Concetti base e architettura CUDA
 - Ruolo di Host (CPU) e Device (GPU)
- **Gestione della Memoria in CUDA - Accenni**
 - Allocazione e trasferimento di memoria
 - Tipi di memoria: globale, condivisa (menzioni)
- **Organizzazione dei Thread**
 - Gerarchie: Grid, Block, Thread
 - Identificazione dei thread
- **Kernel CUDA**
 - Definizione e lancio dei kernel
 - Configurazione di griglia e blocchi
- **Tecniche di Mapping e Dimensionamento**
 - Esempio: Somma di array e mapping degli indici
 - Calcolo dinamico delle dimensioni della griglia
- **Analisi delle Prestazioni**
 - Correttezza dei risultati e gestione degli errori
 - Uso di strumenti di profiling (NVIDIA Nsight)
- **Applicazioni Pratiche**
 - Operazioni su matrici
 - Elaborazione di immagini (es. conversione RGB a grayscale)
 - Convoluzione 1D e 2D

Operazioni su Matrici in CUDA

- Dalla grafica 3D all'intelligenza artificiale, le **operazioni su matrici** sono il cuore di molti algoritmi. CUDA ci permette di eseguire queste operazioni in modo incredibilmente veloce, sfruttando la potenza delle GPU.
- In CUDA, come in molti altri contesti di programmazione, le matrici sono tipicamente memorizzate in **modo lineare** nella memoria globale utilizzando un approccio "row-major" (riga per riga).

$A(i, j)$ (i : Indice di riga, j : Indice di colonna)



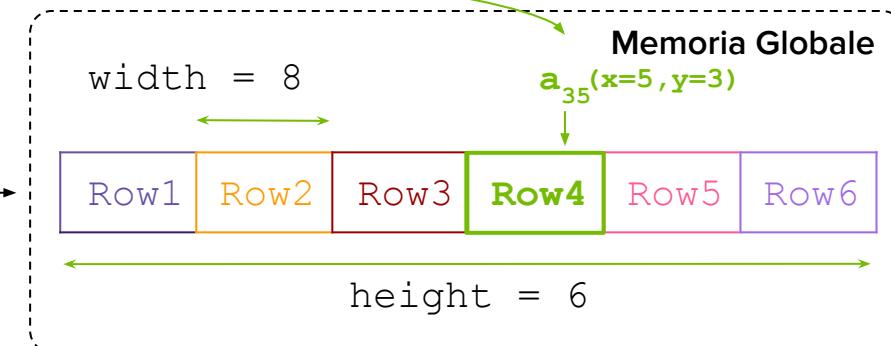
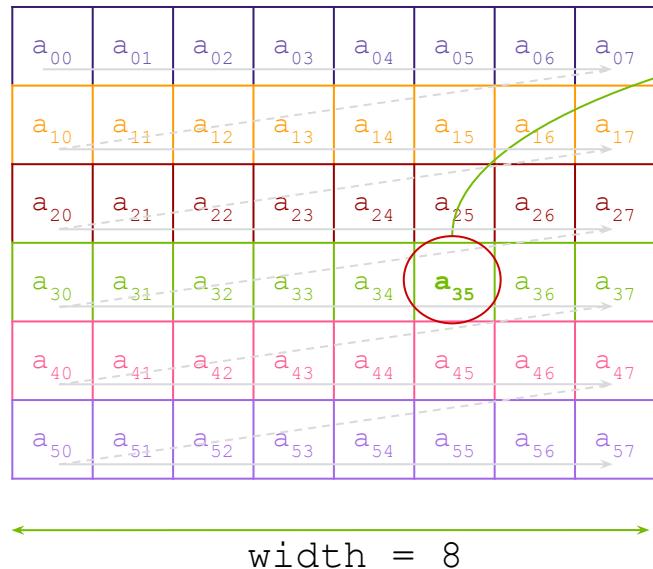
Come si accede agli elementi in memoria?

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

Operazioni su Matrici in CUDA

- Dalla grafica 3D all'intelligenza artificiale, le **operazioni su matrici** sono il cuore di molti algoritmi. CUDA ci permette di eseguire queste operazioni in modo incredibilmente veloce, sfruttando la potenza delle GPU.
- In CUDA, come in molti altri contesti di programmazione, le matrici sono tipicamente memorizzate in **modo lineare** nella memoria globale utilizzando un approccio "row-major" (riga per riga).

$A(i, j)$ (i : Indice di riga, j : Indice di colonna)

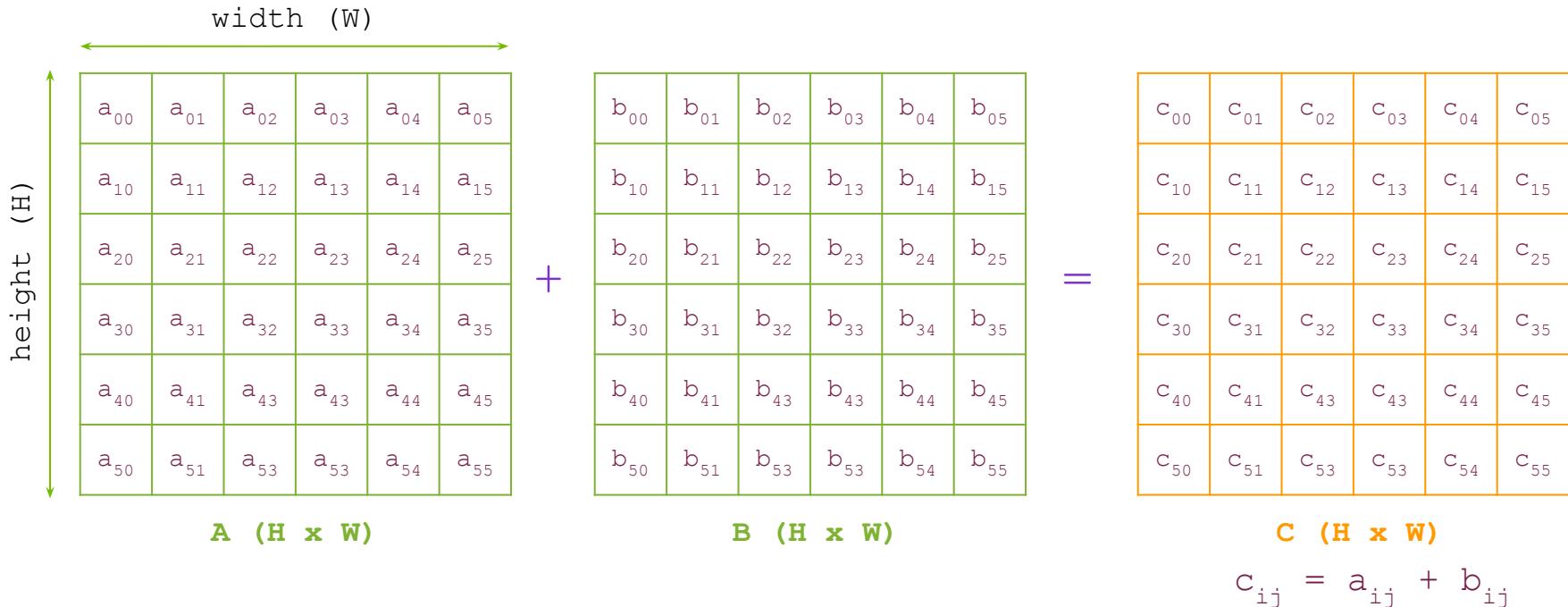


Come si accede agli elementi in memoria?

$$idx = i * \text{width} + j = 29$$

Somma di Matrici in CUDA

- **Obiettivo:** Realizzare in CUDA la somma parallela di due matrici **A** e **B**, salvando il risultato in una matrice **C**.



Mapping degli Indici

- Nell'elaborazione di matrici con CUDA, è fondamentale definire come i **thread vengono mappati agli elementi** della matrice. Questo processo di mapping incide direttamente sulle prestazioni dell'algoritmo.

Problema Generale

- Le matrici vengono linearizzate in memoria, quindi ogni elemento della matrice 2D deve essere mappato a un **indice lineare**: $idx = i * width + j$, dove `width` è il numero di colonne della matrice e (i, j) sono le coordinate dell'elemento.

Impatto della Configurazione

- La configurazione scelta per la griglia e i blocchi (1D o 2D) influenza **come i thread sono associati agli elementi della matrice**.
 - Una configurazione adeguata permette a ogni thread di gestire **porzioni ben definite** dei dati.
 - Una configurazione non ottimale può portare a inefficienze, come thread che gestiscono **intere colonne o righe** della matrice, oppure che elaborano dati in modo non bilanciato.

Suddivisione della Matrice

- Come possiamo suddividere questa matrice per eseguire il calcolo in parallelo? Cosa bisogna garantire?

a_{00}	a_{01}	a_{02}	a_{03}	a_{04}	a_{05}	a_{06}	a_{07}
a_{10}	a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	a_{16}	a_{17}
a_{20}	a_{21}	a_{22}	a_{23}	a_{24}	a_{25}	a_{26}	a_{27}
a_{30}	a_{31}	a_{32}	a_{33}	a_{34}	a_{35}	a_{36}	a_{37}
a_{40}	a_{41}	a_{42}	a_{43}	a_{44}	a_{45}	a_{46}	a_{47}
a_{50}	a_{51}	a_{52}	a_{53}	a_{54}	a_{55}	a_{56}	a_{57}

\longleftrightarrow
 $W = 8$

$\uparrow H = 8$

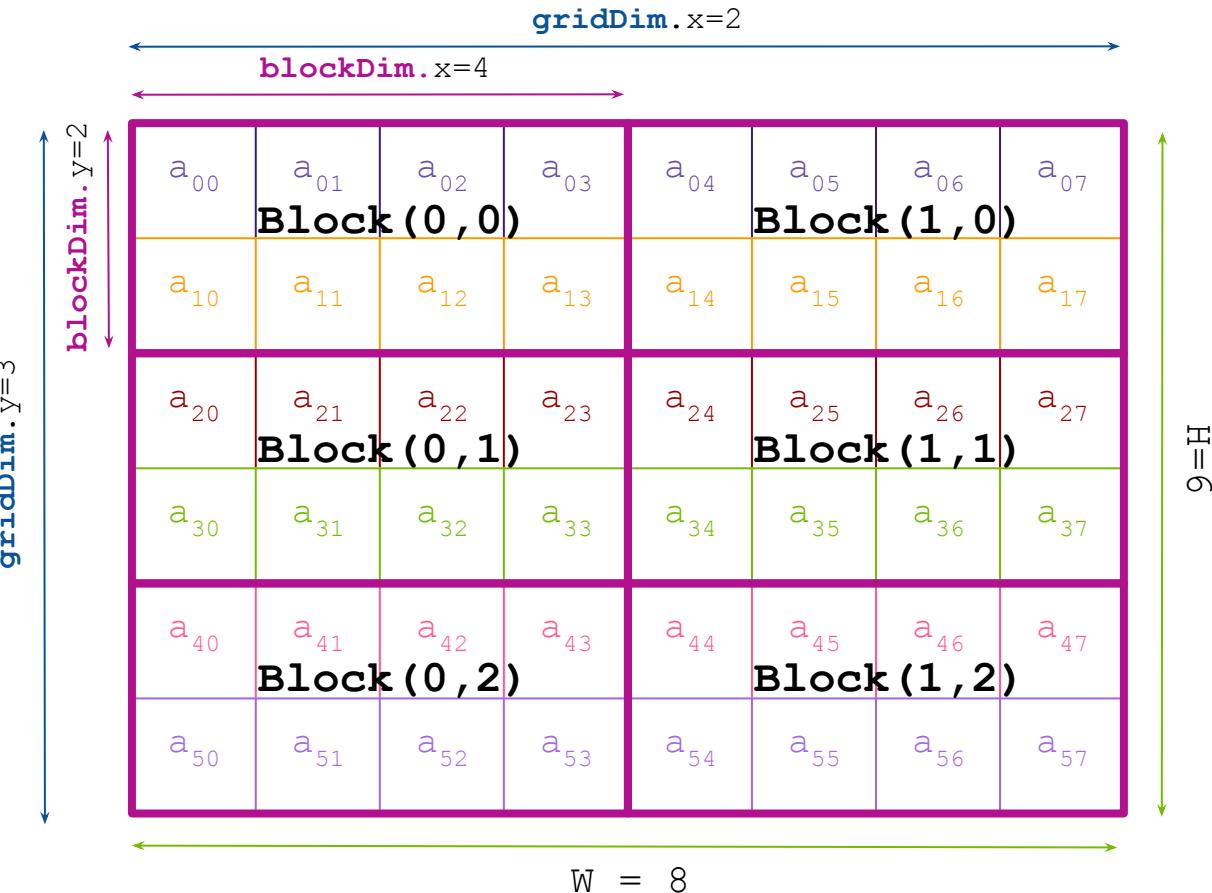
Suddivisione

- La matrice può essere suddivisa in sottoblocki di **dimensioni arbitrarie**.
- La scelta delle dimensioni dei blocchi influenza le **prestazioni**.

Cosa Garantire

- Copertura completa** della matrice.
- Scalabilità** per diverse dimensioni di matrice.
- Coerenza dei risultati** con l'elaborazione sequenziale.
- Accesso efficiente** alla memoria (lo vedremo in seguito).

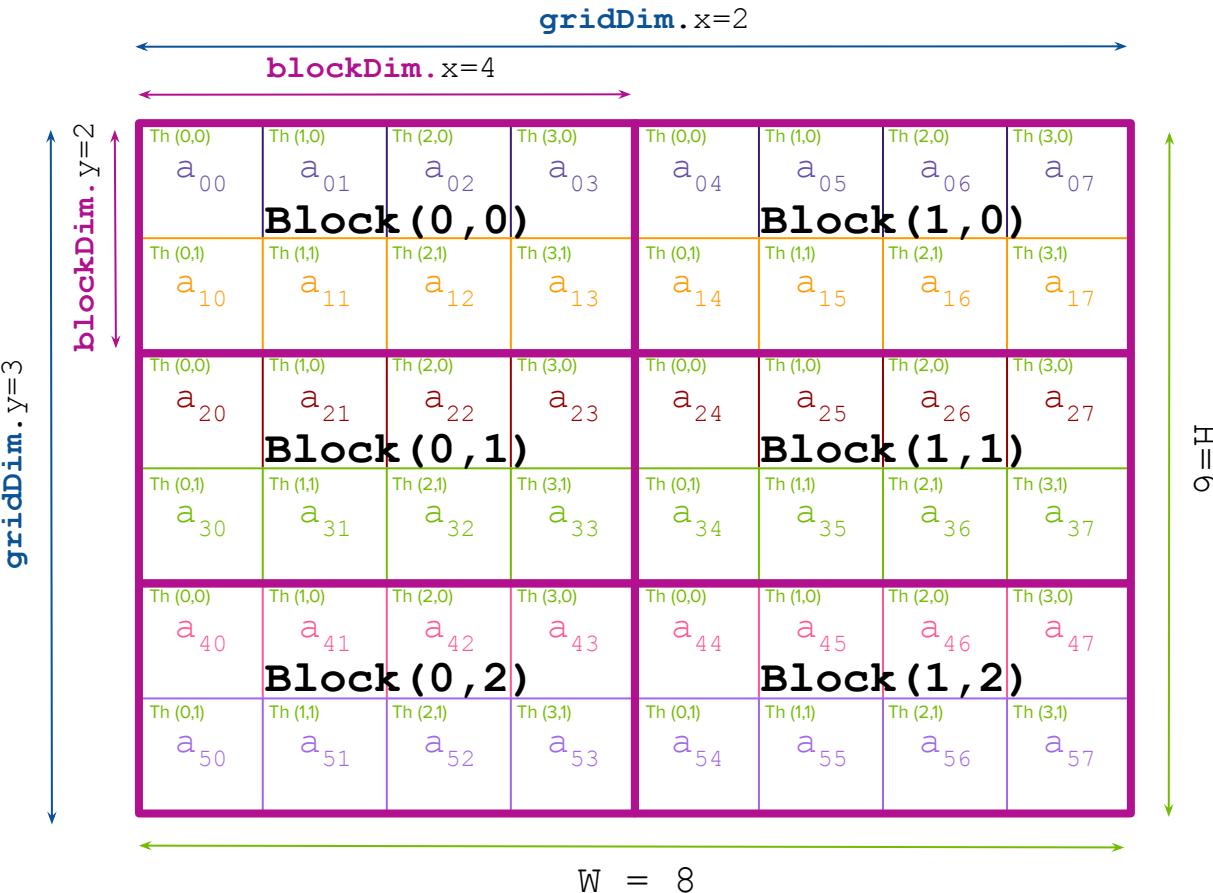
Suddivisione della Matrice - 1) Griglia 2D e Blocchi 2D



Organizzazione della Griglia

- La matrice è divisa, in questo caso specifico, in **6 blocchi**, in una configurazione 2×3 ($\text{gridDim.x} = 2$, $\text{gridDim.y} = 3$)
- Ogni blocco è di dimensione 4×2 , ovvero **8 thread** ($\text{blockDim.x} = 4$, $\text{blockDim.y} = 2$)

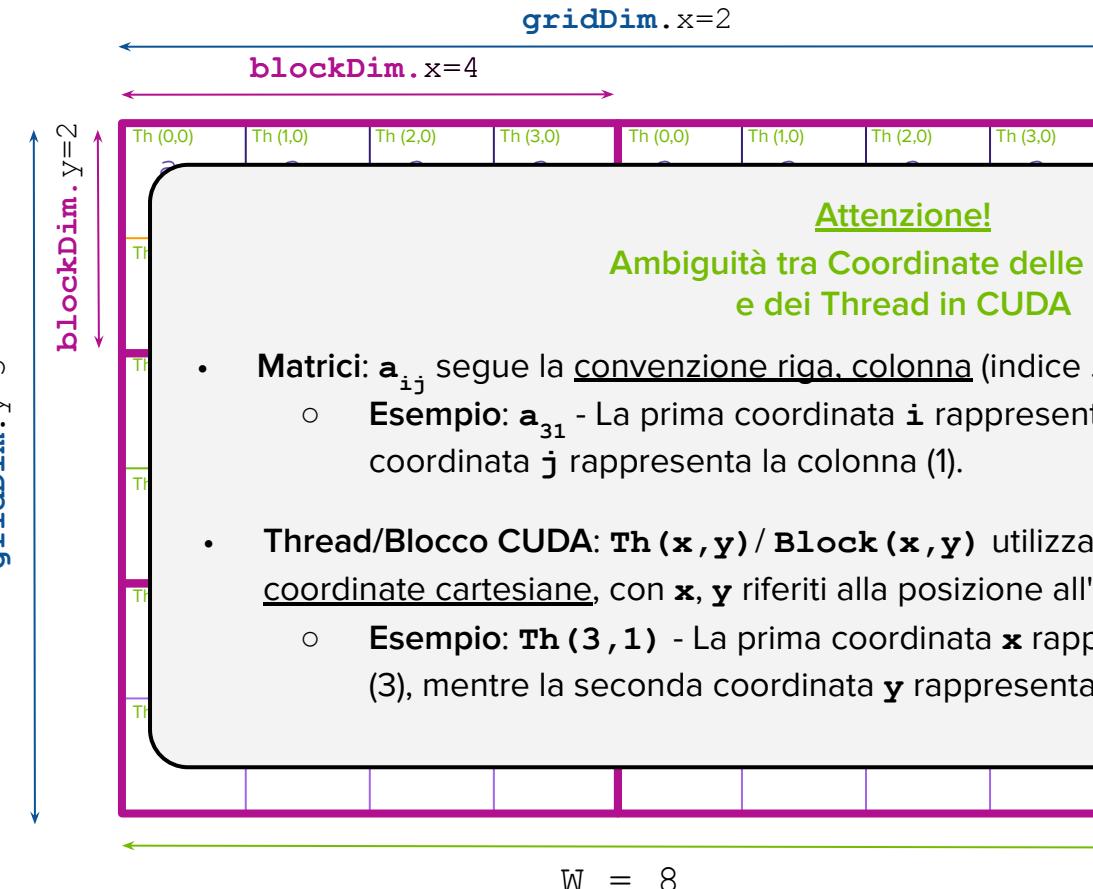
Indici Locali dei Thread - 1) Griglia 2D e Blocchi 2D



Organizzazione della Griglia

- La matrice è divisa, in questo caso specifico, in **6 blocchi**, in una configurazione 2×3 ($gridDim.x = 2$, $gridDim.y = 3$)
- Ogni blocco è di dimensione 4×2 , ovvero **8 thread** ($blockDim.x = 4$, $blockDim.y = 2$)
- Ogni thread ha un **indice locale** (x , y) all'interno del blocco.
- Ogni thread elabora un elemento della matrice.

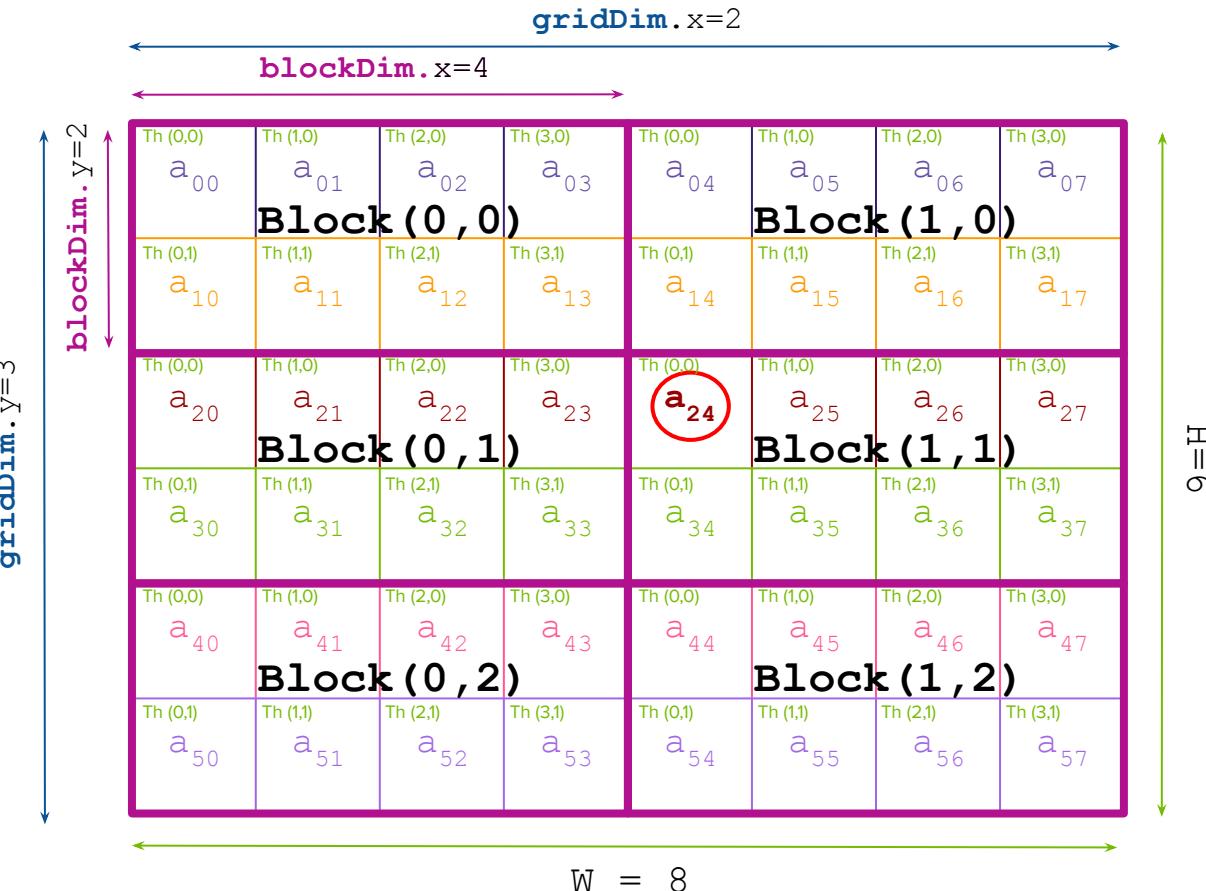
Indici Locali dei Thread - 1) Griglia 2D e Blocchi 2D



Organizzazione della Griglia

- La matrice è divisa, in questo caso specifico, in **6 blocchi**, in una

Mapping degli Indici - 1) Griglia 2D e Blocchi 2D



Organizzazione della Griglia

- La matrice è divisa, in questo caso specifico, in **6 blocchi**, in una configurazione 2×3 ($\text{gridDim.x} = 2$, $\text{gridDim.y} = 3$)
- Ogni blocco è di dimensione 4×2 , ovvero **8 thread** ($\text{blockDim.x} = 4$, $\text{blockDim.y} = 2$)
- Ogni thread ha un **indice locale** (x , y) all'interno del blocco.
- Ogni thread elabora un elemento della matrice.

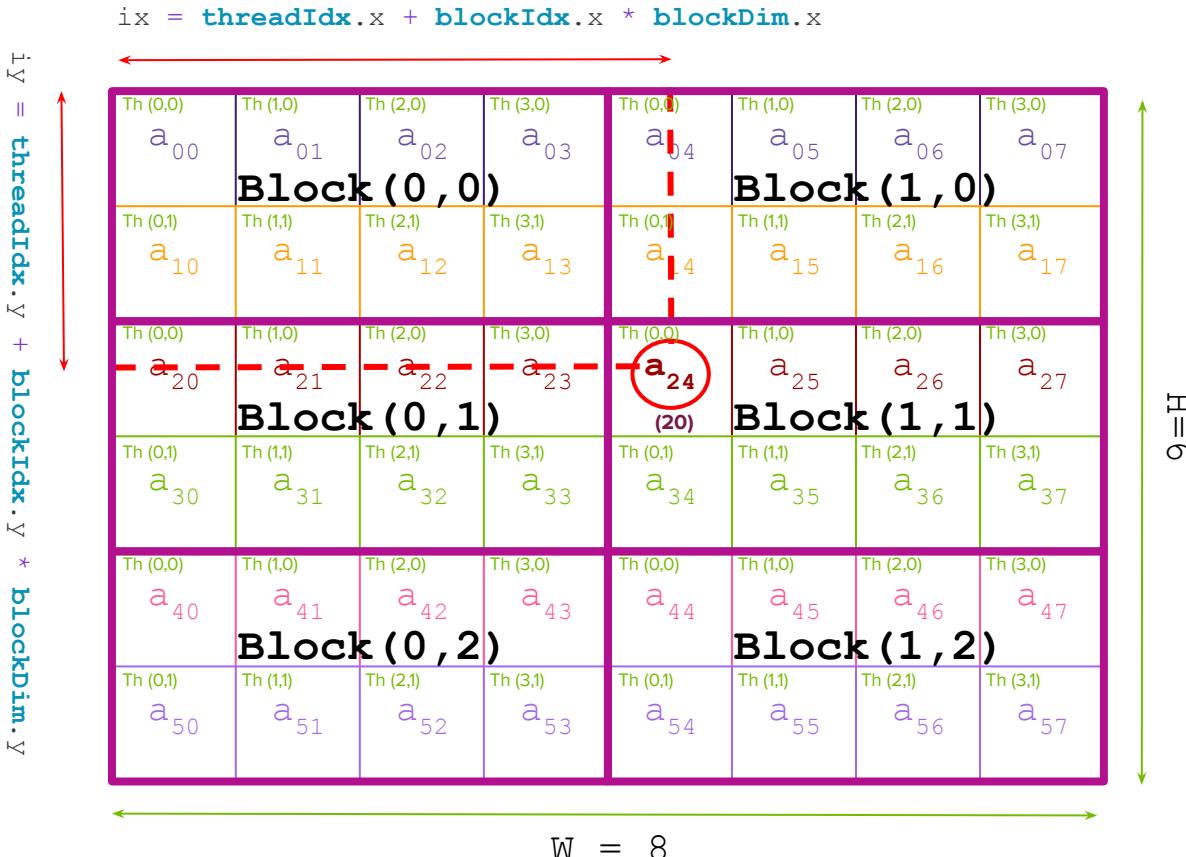
Come calcolo l'indice globale?

- Scelgo un metodo di mapping (es. lineare oppure **basato su coordinate**)

↓
Ci concentriamo su quest'ultimo per la sua maggiore **intuitività**

Calcolo dell'Indice Globale - 1) Griglia 2D e Blocchi 2D

Metodo Basato su Coordinate



Organizzazione della Griglia

- La matrice è divisa, in questo caso specifico, in **6 blocchi**, in una configurazione 2×3 (`gridDim.x = 2`, `gridDim.y = 3`)
- Ogni blocco è di dimensione 4×2 , ovvero **8 thread** (`blockDim.x = 4`, `blockDim.y = 2`)
- Ogni thread ha un **indice locale** (x , y) all'interno del blocco.
- Ogni thread elabora un elemento della matrice.

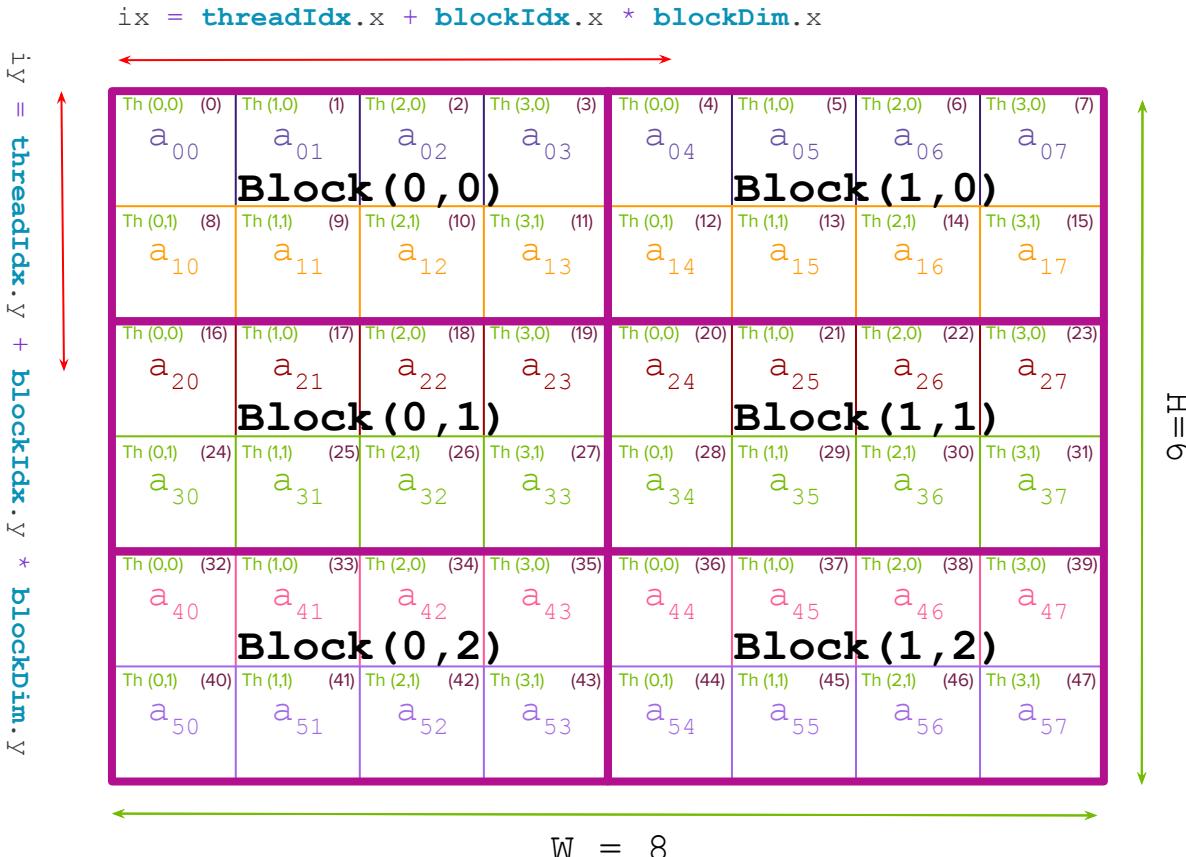
Esempio di Mapping (in rosso)

- Indice x nella matrice
 - $ix = 0 + 1 * 4 = 4$
- Indice y nella matrice
 - $iy = 0 + 1 * 2 = 2$
- Indice lineare
 - $idx = iy * W + ix = 20$

L'indice 20 corrisponde all'elemento a_{24}

Calcolo dell'Indice Globale - 1) Griglia 2D e Blocchi 2D

Metodo Basato su Coordinate



Organizzazione della Griglia

- La matrice è divisa, in questo caso specifico, in **6 blocchi**, in una configurazione 2×3 ($gridDim.x = 2$, $gridDim.y = 3$)
- Ogni blocco è di dimensione 4×2 , ovvero **8 thread** ($blockDim.x = 4$, $blockDim.y = 2$)
- Ogni thread ha un **indice locale** (x , y) all'interno del blocco.
- Ogni thread elabora un elemento della matrice.
- Il mapping si calcola per ogni thread combinando gli indici del blocco e quelli locali.

$$idx = iy * W + ix$$

Somma di Matrici con Griglia 2D e Blocchi 2D

- **Obiettivo:** Realizzare in CUDA la somma parallela di due matrici **A** e **B**, salvando il risultato in una matrice **C**.

Passi Chiave

1. **Validazione su Host:** Implementazione di una funzione di validazione `sumMatrixOnHost` in C.
2. **Kernel CUDA:** Definizione del kernel `sumMatrixOnGPU2D` che eseguirà la somma sulla GPU.
 - Viene configurata una griglia 2D di blocchi 2D per sfruttare il parallelismo massivo della GPU.
 - Ogni thread del kernel calcola il proprio indice **globale** dalle coordinate (`ix, iy`).
 - Ogni thread esegue l'operazione su un elemento delle matrici **A** e **B** e memorizza il risultato in **C**.
3. **Configurazione:**
 - Si definiscono le matrici su cui operare.
 - Si scelgono le dimensioni dei blocchi (`blockDim.x, blockDim.y`) per ottimizzare l'esecuzione sulle unità di calcolo della GPU.
 - Dimensioni della griglia in base alle dimensioni delle matrici e dei blocchi per coprire l'intera matrice:
$$(\text{dataSize} + \text{blockSize} - 1) / \text{blockSize}$$
 (per ogni asse)
4. **Esecuzione:** Lanciare il kernel `sumMatrixOnGPU2D` sulla GPU con la configurazione definita.

Confronto: Somma di Matrici in C vs CUDA C

Codice C Standard

```
// Funzione host per la somma di matrici
void sumMatrixOnHost(float *MatA, float *MatB, float *MatC, int W, int H) {
    for (int i = 0; i < H; i++) { // Cicla su ogni riga
        for (int j = 0; j < W; j++) { // Cicla su ogni colonna
            int idx = i * W + j; // Calcola indice lineare
            MatC[idx] = MatA[idx] + MatB[idx]; // Somma elementi corrispondenti
        }
    }
}
```

Codice CUDA C

```
// Kernel CUDA per la somma di matrici
__global__ void sumMatrixOnGPU2D(float *MatA, float *MatB, float *MatC, int W, int H) {
    unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x; // Calcola indice x globale
    unsigned int iy = threadIdx.y + blockIdx.y * blockDim.y; // Calcola indice y globale
    if (ix < W && iy < H){ // Controlla limiti matrice
        unsigned int idx = iy * W + ix; // Calcola indice lineare
        MatC[idx] = MatA[idx] + MatB[idx]; // Somma elementi corrispondenti
    }
}
```

Somma di Matrici con Griglia 2D e Blocchi 2D

Somma di Due Matrici

1/3

```
int main(int argc, char **argv) {
    int dev = 0;
    cudaDeviceProp deviceProp;
    CHECK(cudaGetDeviceProperties(&deviceProp, dev)); // Ottiene le proprietà del device
    printf("Using Device %d: %s\n", dev, deviceProp.name);
    CHECK(cudaSetDevice(dev)); // Seleziona il device CUDA

    // Imposta le dimensioni della matrice (16384 x 16384)
    int W = 1 << 14; // Soluzione migliore: passare le dimensioni tramite argomenti
    int H = 1 << 14;
    int size = W * H;
    int nBytes = size * sizeof(float);
    printf("Matrix size: W %d H %d\n", W, H);

    // Alloca la memoria host
    float *h_A, *h_B, *hostRef, *gpuRef;
    h_A = (float *)malloc(nBytes);           // Matrice A
    h_B = (float *)malloc(nBytes);           // Matrice B
    hostRef = (float *)malloc(nBytes);        // Risultato CPU
    gpuRef = (float *)malloc(nBytes);         // Risultato GPU
```

Somma di Matrici con Griglia 2D e Blocchi 2D

Somma di Due Matrici

2/3

```
// Inizializza i dati delle matrici (randomicamente)
initialData(h_A, size);
initialData(h_B, size);

memset(hostRef, 0, nBytes);
memset(gpuRef, 0, nBytes);

// Somma la matrice sulla CPU
iStart = cpuSecond();
sumMatrixOnHost(h_A, h_B, hostRef, W, H);
iElaps = cpuSecond() - iStart;

// Alloca la memoria del device
float *d_MatA, *d_MatB, *d_MatC;
CHECK(cudaMalloc((void **) &d_MatA, nBytes));
CHECK(cudaMalloc((void **) &d_MatB, nBytes));
CHECK(cudaMalloc((void **) &d_MatC, nBytes));

// Trasferisce i dati dall'host al device
CHECK(cudaMemcpy(d_MatA, h_A, nBytes, cudaMemcpyHostToDevice));
CHECK(cudaMemcpy(d_MatB, h_B, nBytes, cudaMemcpyHostToDevice));
```

Somma di Matrici con Griglia 2D e Blocchi 2D

Somma di Due Matrici

3/3

```
// Configura e invoca il kernel CUDA
int block_dimx = 32; // Potrebbe assumere valori diversi (es. 16, 64, 128..)
int block_dimy = 32; // Potrebbe assumere valori diversi (es. 16, 64, 128..)
dim3 block(block_dimx, block_dimy);
dim3 grid((W + block.x - 1) / block.x,
            (H + block.y - 1) / block.y);

iStart = cpuSecond();
sumMatrixOnGPU2D<<<grid, block>>>(d_MatA, d_MatB, d_MatC, W, H);
CHECK(cudaDeviceSynchronize()); // Sincronizza per misurare il tempo correttamente
iElaps = cpuSecond() - iStart;

// Copia il risultato del kernel dal device all'host
CHECK(cudaMemcpy(gpuRef, d_MatC, nBytes, cudaMemcpyDeviceToHost));

// Verifica il risultato
checkResult(hostRef, gpuRef, size);

// continue..
```

Griglia 2D e Blocchi 2D - Confronto fra Diverse Configurazioni

NVIDIA Nsight Compute*

Dim. Matrice (16384,16384)

Dim. Griglia	Dim. Blocco	Runtime (ms)	Speedup vs CPU	Device
-	-	516,08 (TimerCPU)		i9-10920X (CPU)
(16384,16384)	(1,1)	223,70*	2,31x	RTX 3090 (GPU)
(4096,4096)	(4,4)	13,99*	36,89x	RTX 3090 (GPU)
(1024,1024)	(16,16)	3,75*	137,62x	RTX 3090 (GPU)
(512,512)	(32,32)	3,91*	131,98x	RTX 3090 (GPU)

Osservazioni

- Tutte le configurazioni GPU offrono un **miglioramento** rispetto alla CPU.
- Miglioramento drastico passando da **(1,1)** a dimensioni di blocco maggiori.
- Le configurazioni con **più blocchi e thread** mostrano miglioramenti drammatici, con speedup superiori a **131x**.
- Le differenze tra le configurazioni **(16,16)** e **(32,32)** sono relativamente piccole, suggerendo una **saturazione** dell'utilizzo delle risorse GPU.
- Esiste un punto di ottimizzazione oltre il quale ulteriori aumenti nella dimensione o nel numero dei blocchi non producono miglioramenti significativi.

* vedremo successivamente i motivi dell'impatto di differenti configurazioni sulle performance dei kernel

Griglia 2D e Blocchi 2D - Confronto fra Diverse Configurazioni

NVIDIA Nsight Compute*

Dim. Griglia	Dim. Blocco	Dim. M	Perchè Inefficiente?
-	-		
(16384, 16384)	(1, 1)		
(4096, 4096)	(4, 4)		
(1024, 1024)	(16, 16)		
(512, 512)	(32, 32)		

Osservazioni

- Tutte le configurazioni GPU offrono un **miglioramento**.
- Miglioramento drastico passando da **(1, 1)** a dimensioni di blocco maggiori.
- Le configurazioni con **più blocchi e thread** mostrano miglioramenti drammatici, con speedup superiori a **131x**.
- Le differenze tra le configurazioni **(16, 16)** e **(32, 32)** sono relativamente piccole, suggerendo una **saturazione** dell'utilizzo delle risorse GPU.
- Esiste un punto di ottimizzazione oltre il quale ulteriori aumenti nella dimensione o nel numero dei blocchi non producono miglioramenti significativi.

* vedremo successivamente i motivi dell'impatto di differenti configurazioni sulle performance dei kernel

Griglia 2D e Blocchi 2D - Confronto fra Diverse Configurazioni

NVIDIA Nsight Compute*

Dim. Matrice (16384, 16384)

Dim.

(16384)

(4096)

(1024)

(512)

Osservazi

- Tutti
- Mig
- Le
- Le
- del
- Esi

produttivo miglioramenti significativi.

Analisi Dettagliata da Nsight Compute - (1,1) vs (16,16)

- Utilizzo della memoria (Memory [%])
 - Blocco 2D (1,1): 10,49%
 - Blocco 2D (32,16): 94,28%
- Throughput di memoria:
 - Blocco 2D (1,1): 14,42 GB/s
 - Blocco 2D (32,16): 860,79 GB/s
- SM Busy:
 - Blocco 2D (1,1): 5,64%
 - Blocco 2D (32,16): 10,3%
- Occupancy Risorse:
 - Blocco 2D (1,1): 12,94
 - Blocco 2D (32,16): 66,41

- **SM Busy:** La configurazione (16,16) raddoppia l'utilizzo degli SM, migliorando l'efficienza di calcolo.
- **Occupancy Risorse:** Aumento di 5,1 volte nell'occupancy, indicando un uso molto più efficiente dei thread disponibili.
- **Utilizzo della memoria:** Miglioramento drammatico nell'utilizzo della larghezza di banda nel caso (16,16), ottimizzando gli accessi alla memoria.
- **Throughput di memoria:** Aumento di circa 60 volte, principale fattore del boost di performance complessivo.

vice

X (CPU)

0 (GPU)

0 (GPU)

0 (GPU)

0 (GPU)

ri a 131x.

saturazione

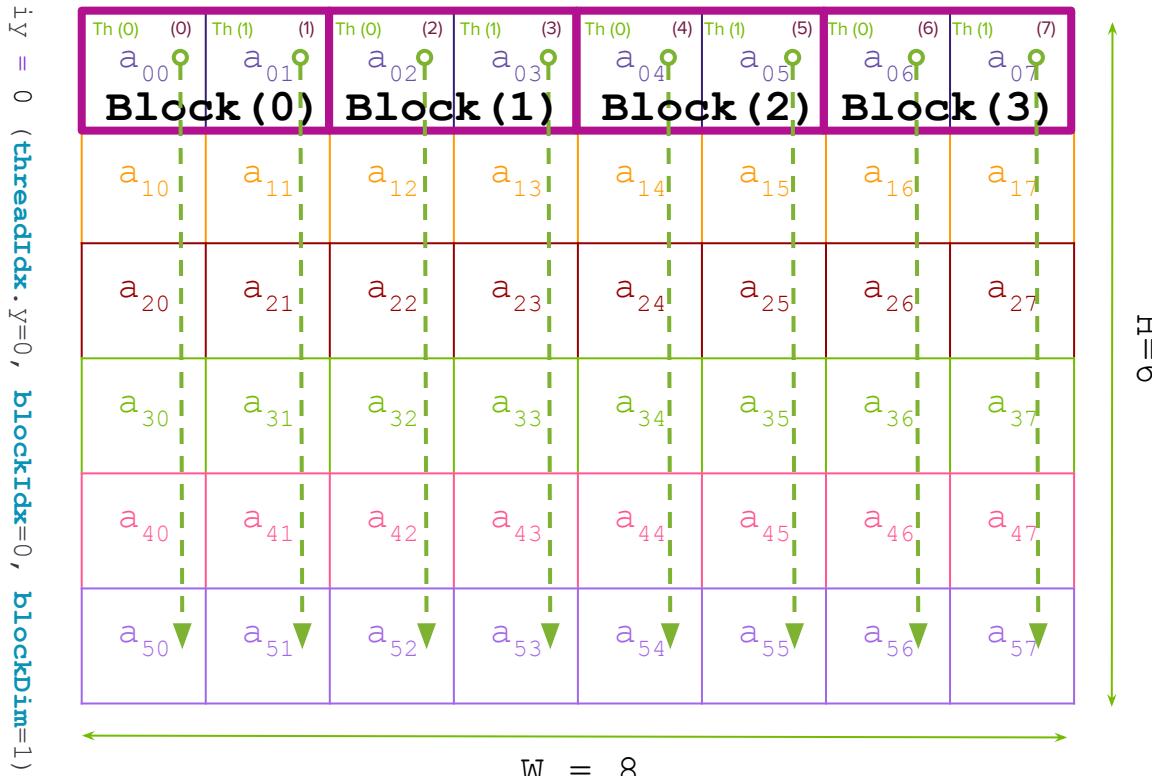
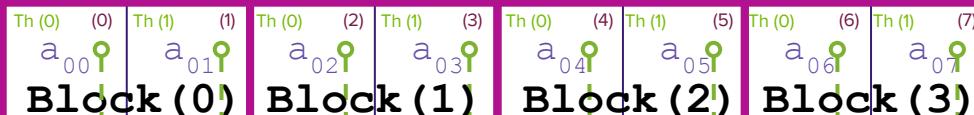
occhi non

* vedremo successivamente i motivi dell'impatto di differenti configurazioni sulle performance dei kernel

Suddivisione della Matrice - 2) Griglia 1D e Blocchi 1D

Metodo Basato su Coordinate

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



Organizzazione della Griglia

- La matrice è divisa, in questo caso specifico, in **4 blocchi**, in una configurazione 1D (`gridDim.x = 4`).
- Ogni blocco ha configurazione 1D e contiene 2 thread (`blockDim.x = 2`)
- Ogni thread ha un **indice locale** (`x`) all'interno del blocco.
- L'indice di mapping si calcola per ogni thread **combinando gli indici del blocco e quelli locali** lungo l'asse x
 $\text{idx} = \text{ix}$
- Ogni thread elabora **una colonna** della matrice (parallelismo limitato)

Confronto Kernel CUDA per la Somma fra Matrici

Griglia 2D e Blocchi 2D (Esempio Precedente)

```
// Kernel CUDA per la somma di matrici
__global__ void sumMatrixOnGPU2D(float *MatA, float *MatB, float *MatC, int W, int H)
{
    unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x; // Calcola indice x globale
    unsigned int iy = threadIdx.y + blockIdx.y * blockDim.y; // Calcola indice y globale
    if (ix < W && iy < H){ // Controlla limiti matrice
        unsigned int idx = iy * W + ix; // Calcola indice lineare
        MatC[idx] = MatA[idx] + MatB[idx]; // Somma elementi corrispondenti
    }
}
```

Griglia 1D e Blocchi 1D

```
__global__ void sumMatrixOnGPU1D(float *MatA, float *MatB, float *MatC, int W, int H) {
    unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x; // Calcola indice x globale
    if (ix < W ) { // Controlla limiti matrice lungo l'asse x
        for (int iy = 0; iy < H; iy++) { // Scorre lungo l'asse y
            unsigned int idx = iy * W + ix; // Calcola indice lineare
            MatC[idx] = MatA[idx] + MatB[idx]; // Somma elementi corrispondenti
        } } }
```

Confronto Kernel CUDA per la Somma fra Matrici

```
// Kernel CUDA per la somma di matrici
__global__ void sumMatrixOnGPU2D(float *Ma
{
    unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x;
    unsigned int iy = threadIdx.y + blockIdx.y * blockDim.y;
    if (ix < W && iy < H){ // Controlla limiti matrice
        unsigned int idx = iy * W + ix; // calcola indice lineare
        MatC[idx] = MatA[idx] + MatB[idx]; // Somma elementi corrispondenti
    }
}
```

Griglia 2D e Blocchi 2D

- Mappatura diretta:** Ogni thread gestisce un solo elemento della matrice, sfruttando la natura bidimensionale del problema.
- Maggiore parallelismo:** Permette di sfruttare al massimo il parallelismo offerto dalla GPU, con un thread per ogni elemento.

```
__global__ void sumMatrixOnGPU1D(float *Ma
{
    unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x;
    if (ix < W ) { // Controlla limiti matrice
        for (int iy = 0; iy < H; iy++) { // Itera su riga
            unsigned int idx = iy * W + ix;
            MatC[idx] = MatA[idx] + MatB[idx]; // Somma elementi corrispondenti
        }
    }
}
```

Griglia 1D & Blocchi 1D

- Minore parallelismo:** Ogni thread gestisce una colonna intera, limitando il parallelismo a livello di riga.
- Loop interno:** Il ciclo for introduce un'inefficienza, poiché ogni thread deve iterare su tutti gli elementi della sua colonna.

Griglia 1D e Blocchi 1D - Confronto fra Diverse Configurazioni

NVIDIA Nsight Compute*

Dim. Matrice (16384,16384)

Dim. Griglia	Dim. Blocco	Runtime (ms)	Speedup vs CPU	Device
-	-	516,08 (TimerCPU)		i9-10920X (CPU)
4096	4	24,49*	21,07x	RTX 3090 (GPU)
1024	16	7,69*	67,11x	RTX 3090 (GPU)
512	32	7,22*	71,48x	RTX 3090 (GPU)
256	64	7,22*	71,48x	RTX 3090 (GPU)
128	128	7,20*	71,68x	RTX 3090 (GPU)
64	256	7,22*	71,48x	RTX 3090 (GPU)

Osservazioni

- Prestazioni relativamente **uniformi** con **Dim.Blocco > 16**, con tempi di esecuzione tra **7,20** e **7,69** ms.
- Lo speedup rispetto alla CPU varia da **67,11x** a **71,68x**, inferiore all'approccio Grid 2D e Blocchi 1D ma comunque significativo.
- Mentre abbiamo **parallelismo lungo l'asse x** (ogni thread gestisce una colonna), l'elaborazione **lungo l'asse y è sequenziale**. Questo riduce significativamente il parallelismo effettivo rispetto agli approcci 2D.

Confronto fra Griglia 1D, Blocchi 1D e Griglia 2D, Blocchi 2D

NVIDIA Nsight Compute*

Se aumentassimo il numero di righe?

Dim. Matrice	Dim. Griglia	Dim. Blocco	Runtime (ms)	Speedup (vs CPU)	Device
(16384, 16384)	-	-	516,08 (TimerCPU)		i9-10920X (CPU)
(32768, 8192)	-	-	516,20 (TimerCPU)		i9-10920X (CPU)
(65536, 4096)	-	-	516,96 (TimerCPU)		i9-10920X (CPU)
(1048576, 256)	-	-	524,37 (TimerCPU)		i9-10920X (CPU)
<hr/>					
Griglia 1D, Blocchi 1D (un thread elabora una colonna)					
(16384, 16384)	64	256	7,22*	71,48x	RTX 3090 (GPU)
(32768, 8192)	32	256	13,02*	39,65x	RTX 3090 (GPU)
(65536, 4096)	16	256	25,13*	20,57x	RTX 3090 (GPU)
(1048576, 256)	1	256	375,85*	1,40x	RTX 3090 (GPU)
<hr/>					
Griglia 2D, Blocchi 2D (un thread elabora un singolo elemento)					
(16384, 16384)	(1024, 1024)	(16, 16)	3,74*	137,99x	RTX 3090 (GPU)
(32768, 8192)	(512, 2048)	(16, 16)	3,73*	138,39x	RTX 3090 (GPU)
(65536, 4096)	(256, 4096)	(16, 16)	3,75*	137,86x	RTX 3090 (GPU)
(1048576, 256)	(16, <u>65536</u>)	(16, 16)	X	X	RTX 3090 (GPU)

Confronto fra Griglia 1D, Blocchi 1D e Griglia 2D, Blocchi 2D

NVIDIA Nsight Compute*

Se aumentassimo il numero di righe?

Dim. Matrice	Dim. Griglia	Dim. Blocco	Runtime (ms)	Speedup (vs CPU)	Device
(16384, 16384)	-	-	516,08 (TimerCPU)		i9-10920X (CPU)
(32768, 8192)	-	-	516,20 (TimerCPU)		i9-10920X (CPU)
(65536, 4096)	-	-	516,96 (TimerCPU)		i9-10920X (CPU)
(1048576, 256)	-	-	524,37 (TimerCPU)		i9-10920X (CPU)
<hr/>					
Griglia 1D, Blocchi 1D (un thread elabora una colonna)					
(16384, 16384)	64	256	7,22*	71,48x	RTX 3090 (GPU)
(32768, 8192)	32	256	13,02*	39,65x	RTX 3090 (GPU)
(65536, 4096)	16	256	25,13*	20,57x	RTX 3090 (GPU)
(1048576, 256)					
<ul style="list-style-type: none">Limite dei blocchi: Limite massimo dei 65535 thread per blocco sull'asse y (vedi compute capability GPU)Necessità di adattamento: Per gestire matrici con un numero di righe superiori è necessario modificare la configurazione per suddividere il lavoro in modo diverso.					
<hr/>					
Griglia 2D, Blocchi 2D (un thread elabora una cella)					
(16384, 16384)	(256, 256)	(16, 16)	3,75*	137,86x	RTX 3090 (GPU)
(32768, 8192)	(256, 256)	(16, 16)	X	X	RTX 3090 (GPU)
(65536, 4096)	(256, 256)	(16, 16)			
(1048576, 256)	(16, <u>65536</u>)	(16, 16)			

Confronto fra Griglia 1D, Blocchi 1D e Griglia 2D, Blocchi 2D

NVIDIA Nsight Compute*

Se aumentassimo il numero di righe?

Dim. Matrice	Dim. Griglia	Dim. Blocco	Runtime (ms)	Speedup (vs CPU)	Device
(16384, 16384)	-	-	516,08 (TimerCPU)		i9-10920X (CPU)
(32768, 8192)	-	-	516,20 (TimerCPU)		i9-10920X (CPU)
(65536, 4096)	-	-	516,96 (TimerCPU)		i9-10920X (CPU)
(1048576, 256)	-	-	524,37 (TimerCPU)		i9-10920X (CPU)
<hr/>					
Griglia 1D, Blocchi 1D (un thread elabora una colonna)					
(16384, 16384)	64	256	7,22*	71,48x	RTX 3090 (GPU)
(32768, 8192)	32	256	13,02*	39,65x	RTX 3090 (GPU)
(65536, 4096)	16	256	25,13*	20,57x	RTX 3090 (GPU)
(1048576, 256)	1	256	375,85*	1,40x	RTX 3090 (GPU)
<hr/>					
Griglia 2D, Blocchi 2D (un thread elabora un singolo elemento)					
(16384, 16384)	(1024, 1024)	(16, 16)	3,74*	137,99x	RTX 3090 (GPU)
(32768, 8192)	(512, 2048)	(16, 16)	3,73*	138,39x	RTX 3090 (GPU)
(65536, 4096)	(256, 4096)	(16, 16)	3,75*	137,86x	RTX 3090 (GPU)
(1048576, 256)	(8, <u>32768</u>)	<u>(32, 32)</u>	4,00*	131,09x	RTX 3090 (GPU)

OK!

Confronto fra Griglia 1D, Blocchi 1D e Griglia 2D, Blocchi 2D

NV

Tendenza nelle Prestazioni

- **Griglia 1D, Blocchi 1D:** Le prestazioni peggiorano significativamente all'aumentare del numero di righe delle matrici (da **71,48x** a **1,40x** di speedup rispetto alla CPU).
- **Griglia 2D, Blocchi 2D:** Mantiene prestazioni costanti e elevate (speedup tra **131,09x** e **137,99x** rispetto alla CPU) per tutte le dimensioni di matrice.
- **Caso estremo:** Per la matrice **(1048576, 256)**, l'approccio 1D1D diventa di poco superiore ad un approccio sequenziale (**1,40x**), mentre il 2D2D mantiene un alto speedup (**131,09x**).

Griglia 1D, Blocchi 1D (un thread elabora una colonna)

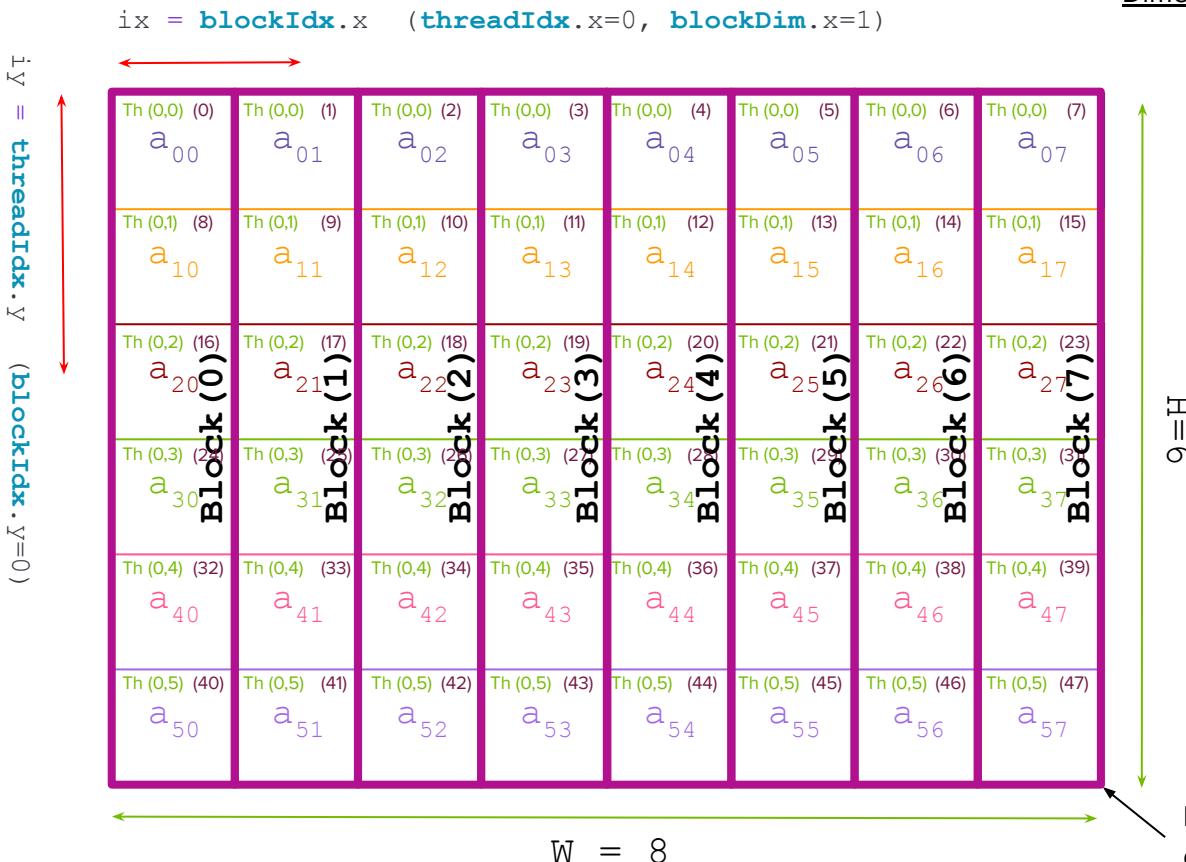
(16384,16384)	64	256	7,22*	71,48x	RTX 3090 (GPU)
(32768,8192)	32	256	13,02*	39,65x	RTX 3090 (GPU)
(65536,4096)	16	256	25,13*	20,57x	RTX 3090 (GPU)
(1048576,256)	1	256	375,85*	1,40x	RTX 3090 (GPU)

Griglia 2D, Blocchi 2D (un thread elabora un singolo elemento)

(16384,16384)	(1024,1024)	(16,16)	3,74*	137,99x	RTX 3090 (GPU)
(32768,8192)	(512,2048)	(16,16)	3,73*	138,39x	RTX 3090 (GPU)
(65536,4096)	(256,4096)	(16,16)	3,75*	137,86x	RTX 3090 (GPU)
(1048576,256)	(8,32768)	(32,32)	4,00*	131,09x	RTX 3090 (GPU)

Suddivisione della Matrice - 3) Griglia 1D e Blocchi 2D

Metodo Basato su Coordinate



Dimensione sull'asse x pari a 1
(caso degenere)

Organizzazione della Griglia

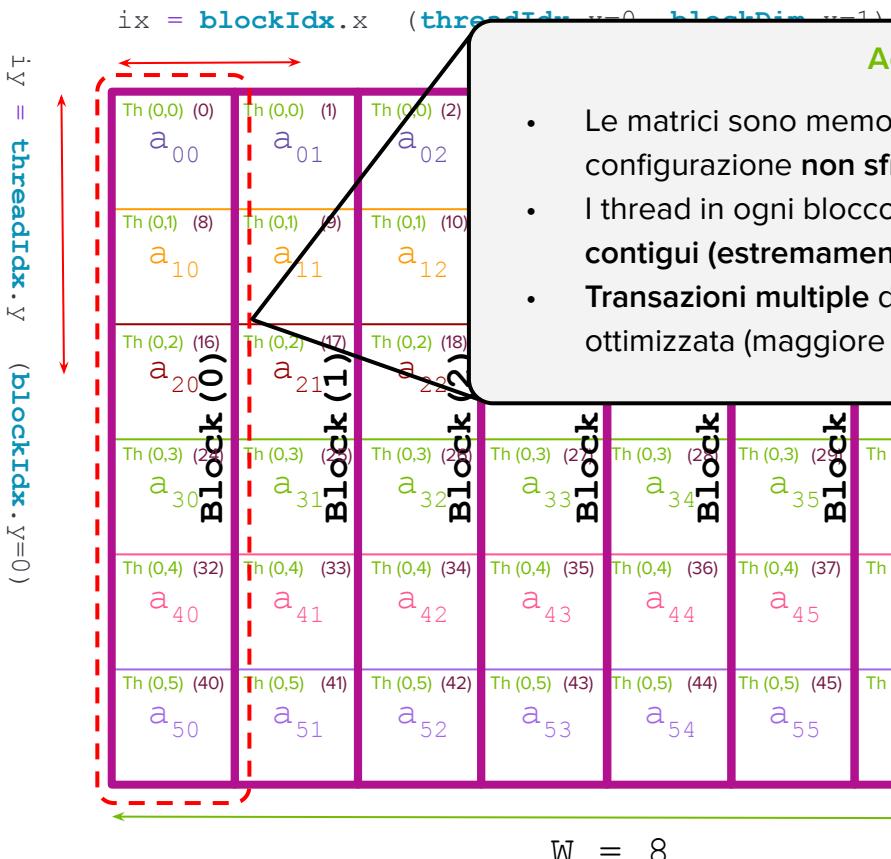
- La matrice è divisa, in questo caso specifico, in **8 blocchi**, in una configurazione 1D (`gridDim.x = 8`).
- Ogni blocco ha configurazione 2D e contiene 6 thread (`blockDim.x = 1, blockDim.y = 6`) - degenere
- Ogni thread ha un **indice locale** ($0, y$) all'interno del blocco.
- Ogni thread elabora un **elemento** della matrice (sempre?)
- L'indice di mapping si calcola per ogni thread **combinando gli indici del blocco e quelli locali**.

$$\text{idx} = iy * W + ix$$

Dimensione del blocco (da definire manualmente)
deve essere almeno uguale al numero delle righe

Suddivisione della Matrice - 3) Griglia 1D e Blocchi 2D

Metodo Basato su Coordinate



Dimensione sull'asse x pari a 1

Accesso in Memoria

- Le matrici sono memorizzate in ordine "row-major". Questa configurazione **non sfrutta la località spaziale** dei dati in memoria.
- I thread in ogni blocco accedono ad elementi di memoria **non contigui** (**estremamente inefficiente** - lo vedremo)
- Transazioni multiple** di memoria invece di una singola transazione ottimizzata (maggiore latenza e ridotto throughput)

della Griglia

in questo caso chi, in una gridDim.x = 8). configurazione 2D e blockDim.x = 6) - degener

- Ogni thread ha un **indice locale** ($0, y$) all'interno del blocco.
- Ogni thread elabora un **elemento** della matrice (sempre?)
- L'indice di mapping si calcola per ogni thread **combinando gli indici del blocco e quelli locali**.

$$\text{idx} = iy * W + ix$$

Dimensione del blocco (da definire manualmente)
deve essere almeno uguale al numero delle righe

Confronto Kernel CUDA per la Somma fra Matrici

Griglia 2D e Blocchi 2D (Esempio Precedente)

```
// Kernel CUDA per la somma di matrici
__global__ void sumMatrixOnGPU2D(float *MatA, float *MatB, float *MatC, int W, int H) {
    unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x; // Calcola indice x globale
    unsigned int iy = threadIdx.y + blockIdx.y * blockDim.y; // Calcola indice y globale
    if (ix < W && iy < H){ // Controlla limiti matrice
        unsigned int idx = iy * W + ix; // Calcola indice lineare
        MatC[idx] = MatA[idx] + MatB[idx]; // Somma elementi corrispondenti
    }
}
```

Griglia 1D e Blocchi 2D (con una dimensione degenera)

```
__global__ void sumMatrixOnGPU1D2D(float *MatA, float *MatB, float *MatC, int W, int H) {
    unsigned int ix = blockIdx.x; // Calcola indice x globale
    unsigned int iy = threadIdx.y; // Calcola indice y globale
    if (ix < W && iy < H) { // Controlla limiti matrice
        unsigned int idx = iy * W + ix; // Calcola indice lineare
        MatC[idx] = MatA[idx] + MatB[idx]; // Somma elementi corrispondenti
    }
}
```

Confronto Kernel CUDA per la Somma fra Matrici

Griglia 2D e Blocchi

```
// Kernel CUDA per la somma di matrici
__global__ void sumMatrixOnGPU2D(float *MatA,
                                  unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x;
                                  unsigned int iy = threadIdx.y + blockIdx.y * blockDim.y;
                                  if (ix < W && iy < H){ // Controlla limiti
                                      unsigned int idx = iy * W + ix; // Calcola indice
                                      MatC[idx] = MatA[idx] + MatB[idx]; // Somma elementi corrispondenti
                                  }
}
```

Griglia 2D & Blocchi 2D

- Limiti Thread:** Max 1024 thread/colonna, vincolati dalla compute capability.
- Distribuzione:** Può distribuire i thread su entrambe le dimensioni. Divide sia righe che colonne in blocchi.
- Scalabilità:** Buona per matrici grandi, suddivide il lavoro uniformemente.

Griglia 1D e Blocchi 2D

```
__global__ void sumMatrixOnGPU1D2D(float *MatA,
                                    unsigned int ix = blockIdx.x; // Calcola righe
                                    unsigned int iy = threadIdx.y; // Calcola colonne
                                    if (ix < W && iy < H){ // Controlla limiti
                                        unsigned int idx = iy * W + ix; // Calcola indice
                                        MatC[idx] = MatA[idx] + MatB[idx];
                                    }
}
```

Griglia 1D & Blocchi 2D (degenero)

- Limiti Thread:** Max 1024 thread/colonna, vincolati dalla compute capability.
- Distribuzione:** Thread limitati a una singola dimensione (y) del blocco. Divide la matrice solo per colonne. Dimensione del blocco almeno pari al numero di righe (a meno di adattamenti al codice).
- Scalabilità:** Potenziali difficoltà con matrici con molte righe (>1024). Richiede adattamenti del codice (es. loop).

Confronto Kernel CUDA per la Somma fra Matrici

Griglia 2D e Blocchi 2D (Esempio Precedente)

```
// Kernel CUDA per la somma di matrici
__global__ void sumMatrixOnGPU2D(float *MatA, float *MatB, float *MatC, int W, int H) {
    unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x; // Calcola indice x globale
    unsigned int iy = threadIdx.y + blockIdx.y * blockDim.y; // Calcola indice y globale
    if (ix < W && iy < H){ // Controlla limiti matrice
        unsigned int idx = iy * W + ix; // Calcola indice lineare
        MatC[idx] = MatA[idx] + MatB[idx]; // Somma elementi corrispondenti
    }
}
```

```
__global__ void sum(
    unsigned int ix,
    unsigned int iy,
    if (ix < W && iy < H)
        unsigned int idx =
            MatC[idx] =
}
int H) {
```

Nota: Lo stesso kernel funziona per entrambe le configurazioni. Il calcolo degli indici `ix` e `iy` si adatterebbe automaticamente in base alla configurazione di griglia e blocchi passata durante l'invocazione.

- Per 2D2D, sia `blockIdx` che `threadIdx` sono utilizzati per entrambe le dimensioni.
- Per 1D2D, `ix = blockIdx.x` (poiché `threadIdx.x=0`, `blockDim.x=1`), e `iy = threadIdx.y` (poiché `blockIdx.y = 0`).

Griglia 1D e Blocchi 2D - Confronto fra Diverse Configurazioni

NVIDIA Nsight Compute*

Dim. Matrice	Dim. Griglia	Dim. Blocco	Runtime (ms)	Speedup (vs CPU)	Device
(512, 512)	-	-	0,505 (TimerCPU)		i9-10920X (CPU)
(512, 4096)	-	-	4,065 (TimerCPU)		i9-10920X (CPU)
(512, 16384)	-	-	16,12 (TimerCPU)		i9-10920X (CPU)
(1024, 16384)	-	-	33,92 (TimerCPU)		i9-10920X (CPU)
(2048, 16384)	-	-	64,41 (TimerCPU)		i9-10920X (CPU)
<hr/>					
Griglia 1D, Blocchi 2D (degenero)					
(512, 512)	512	(1, 512)	0,021*	24,05x	RTX 3090 (GPU)
(512, 4096)	4096	(1, 512)	0,153*	26,57x	RTX 3090 (GPU)
(512, 16384)	16384	(1, 512)	0,607*	26,56x	RTX 3090 (GPU)
(1024, 16384)	16384	(1, 512)	x	x	RTX 3090 (GPU)
(1024, 16384)	16384	(1, 1024)	1,21*	28,03x	RTX 3090 (GPU)
(<u>2048</u> , 16384)	16384	(1, x)	x	x	RTX 3090 (GPU)
<hr/>					
Griglia 2D, Blocchi 2D					
(1024, 16384)	(512, 32)	(32, 32)	0,245*	138,45x	RTX 3090 (GPU)

Griglia 1D e Blocchi 2D - Confronto fra Diverse Configurazioni

NVIDIA Nsight Compute*

Dim. Matrice	Dim. Griglia	Dim. Blocco	Runtime (ms)	Speedup (vs CPU)	Device
(512, 512)	-	-	0,505 (TimerCPU)		i9-10920X (CPU)
(512, 4096)	-	-	4,065 (TimerCPU)		i9-10920X (CPU)
(512, 16384)	-	-	16,12 (TimerCPU)		i9-10920X (CPU)
(1024, 16384)	-	-	33,92 (TimerCPU)		i9-10920X (CPU)
(2048, 16384)	-	-	64,41 (TimerCPU)		i9-10920X (CPU)

Griglia 1D, Blocchi 2D (degenero)

(512, 512)	• Limite dei thread: Limite massimo dei 1024 thread per blocco (vedi compute capability GPU)
(512, 4096)	• Necessità di adattamento: Per gestire matrici così grandi con questa configurazione, sarebbe necessario modificare il codice per suddividere il lavoro in modo diverso. Come?
(512, 16384)	
(1024, 16384)	
(1024, 16384)	16384
(2048, 16384)	16384
	(1, 1024)
	(1, ×)
	1,21*
	28,03x
	×
	×
	RTX 3090 (GPU)
	RTX 3090 (GPU)

Griglia 2D, Blocchi 2D

(1024, 16384)	(512, 32)	(32, 32)	0,245*	138,45x	RTX 3090 (GPU)
---------------	-----------	----------	--------	---------	----------------

Griglia 1D e Blocchi 2D - Confronto fra Diverse Configurazioni

NVIDIA Nsight

Dim. Mat.

(512, 51)

(512, 40)

(512, 163)

(1024, 16)

(2048, 16)

Griglia 1D, Blocchi 1D

(512, 51)

(512, 40)

(512, 163)

(1024, 16)

(1024, 16)

(2048, 16)

Analisi Dettagliata da Nsight Compute (2D2D vs 1D2D)

- Utilizzo della memoria (Memory [%]):
 - **1D2D:** 83,95%
 - **2D2D:** 89,40%
- Throughput di memoria:
 - **1D2D:** 164,20 GB/s
 - **2D2D:** 810,80 GB/s
- Occupancy dei multiprocessori (SM Busy):
 - **1D2D:** 1,66%
 - **2D2D:** 11,28%
- Cicli di stallo per istruzione:
 - **1D2D:** 361,92 cicli
 - **2D2D:** 50,85 cicli

- **Accesso alla Memoria:** La versione 2D ottimizza gli accessi coalescenti, migliorando drasticamente il throughput di memoria.
- **Parallelismo:** Migliore distribuzione del carico di lavoro nella 2D, con maggiore utilizzo dei multiprocessori e efficienza delle istruzioni.
- **Riduzione stalli:** La 2D minimizza i cicli di attesa per thread, migliorando l'efficienza complessiva.
- **Granularità:** La suddivisione del lavoro nella 2D permette una migliore sovrapposizione di calcolo e accessi memoria.

Griglia 2D, Blocchi 2D

(1024, 16384)

(512, 32)

(32, 32)

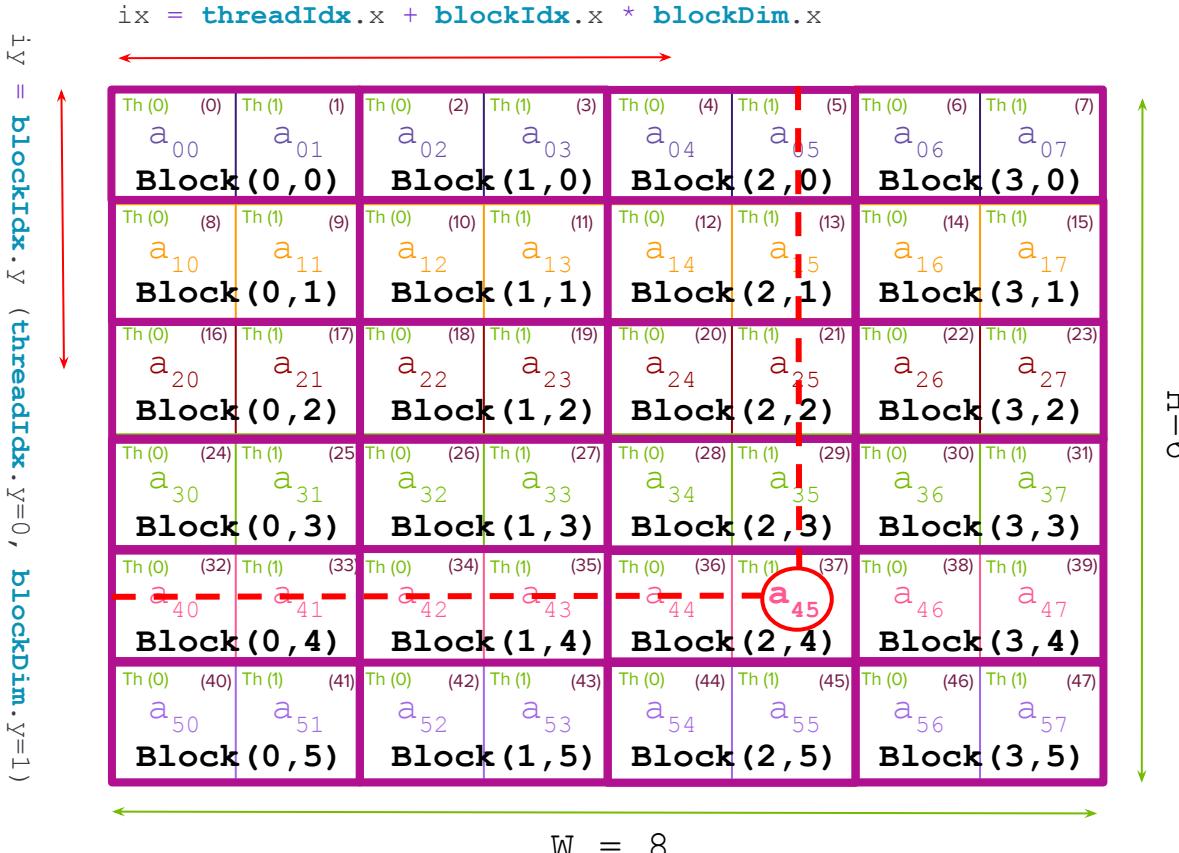
0,245*

138,45x

RTX 3090 (GPU)

Suddivisione della Matrice - 4) Griglia 2D e Blocchi 1D

Metodo Basato su Coordinate



Organizzazione della Griglia

- La matrice è divisa, in questo caso specifico, in **24 blocchi**, in una configurazione 4×6 (`gridDim.x = 4`, `gridDim.y = 6`)
- Ogni blocco è 1D di dimensione 2, ovvero **2 thread** (`blockDim.x = 2`)
- Ogni thread ha un **indice locale** (`x`) all'interno del blocco.
- Ogni thread elabora un elemento della matrice.

Esempio di Mapping (in rosso)

- Indice `x` nella matrice
 - $ix = 1 + 2 * 2 = 5$
- Indice `y` nella matrice
 - $i_y = 0 + 4 * 1 = 4$
- Indice lineare
 - $idx = i_y * W + ix = 37$

L'indice **37** corrisponde all'elemento **a₂₄**

Confronto Kernel CUDA per la Somma fra Matrici

Griglia 2D e Blocchi 2D (Esempio Precedente)

```
// Kernel CUDA per la somma di matrici
__global__ void sumMatrixOnGPU2D(float *MatA, float *MatB, float *MatC, int W, int H) {
    unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x; // Calcola indice x globale
    unsigned int iy = threadIdx.y + blockIdx.y * blockDim.y; // Calcola indice y globale
    if (ix < W && iy < H){ // Controlla limiti matrice
        unsigned int idx = iy * W + ix; // Calcola indice lineare
        MatC[idx] = MatA[idx] + MatB[idx]; // Somma elementi corrispondenti
    }
}
```

Griglia 2D e Blocchi 1D

```
__global__ void sumMatrixOnGPU2D1D(float *MatA, float *MatB, float *MatC, int W, int H) {
    unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x; // Calcola indice x globale
    unsigned int iy = blockIdx.y; // Calcola indice y globale
    if (ix < W && iy < H){ // Controlla limiti matrice
        unsigned int idx = iy * W + ix; // Calcola indice lineare
        MatC[idx] = MatA[idx] + MatB[idx]; // Somma elementi corrispondenti
    }
}
```

Griglia 2D e Blocchi 1D - Confronto fra Diverse Configurazioni

NVIDIA Nsight Compute*

Dim. Matrice (16384, 16384)

Dim. Griglia	Dim. Blocco	Runtime (ms)	Speedup vs CPU	Device
-	-	516,08 (TimerCPU)		i9-10920X (CPU)
(1024, 16384)	16	13,98*	36,92x	RTX 3090 (GPU)
(512, 16384)	32	6,99*	73,83x	RTX 3090 (GPU)
(256, 16384)	64	3,75*	137,62x	RTX 3090 (GPU)
(128, 16384)	128	3,75*	137,62x	RTX 3090 (GPU)
(64, 16384)	256	3,75*	137,62x	RTX 3090 (GPU)
(32, 16384)	512	3,76*	137,25x	RTX 3090 (GPU)

Osservazioni

- Le migliori prestazioni si raggiungono con configurazioni a > 64 thread per blocco, tutte con un tempo di esecuzione di **3,75 ms**.
- Miglioramento significativo passando da 16 thread (**13,98 ms**) a 32 thread (**6,99 ms**), e ulteriore miglioramento fino a 64.
- La configurazione con più thread per blocco permette un migliore utilizzo delle risorse hardware, risultando in prestazioni superiori (**Suggerimento:** osservare analisi completa con Nsight Compute)

Confronto fra le Migliori Configurazioni di Blocchi e Griglie

NVIDIA Nsight Compute*

Dim. Matrice (16384, 16384)

Dim. Griglia	Dim. Blocco	Runtime (ms)	Speedup vs CPU	Device
-	-	516,08 (TimerCPU)		i9-10920X (CPU)
1D1D → 128	128	7,20*	71,68x	RTX 3090 (GPU)
1D2D → 16384	(1, <u>16384</u>) (NO!)	-	-	RTX 3090 (GPU)
2D1D → (256, 16384)	64	3,75*	137,62x	RTX 3090 (GPU)
2D2D → (1024, 1024)	(16, 16)	3,75*	137,62x	RTX 3090 (GPU)

Osservazioni

- L'approccio **Grid 1D e Blocchi 1D** mostra prestazioni generalmente inferiori, con uno speedup massimo di **71,68x** rispetto alla CPU (il loop per thread limita le prestazioni).
- L'approccio **Grid 1D e Blocchi 2D** (degenero) non è in grado di gestire queste dimensioni della matrice (righe > 1024) senza modifiche al codice. Ogni thread dovrebbe processare più elementi della matrice.
- L'approccio **Grid 2D e Blocchi 1D** raggiunge prestazioni identiche al 2D con configurazioni ottimali, ma richiede una regolazione più attenta della dimensione dei blocchi (vedi slide precedente).
- L'approccio **Grid 2D e Blocchi 2D** offre le migliori prestazioni complessive, con uno speedup di **137,62x**.
- La scelta dell'approccio ottimale dipende dalle caratteristiche specifiche del problema, come le **dimensioni** della matrice, la **struttura dei dati** e le **capacità dell'hardware**.

Immagini come Matrici Multidimensionali

Struttura di Base

- Un'immagine digitale è una griglia di pixel.
- Ogni pixel rappresenta il **colore** o l'**intensità** di un punto specifico nell'immagine.
- Questa griglia può essere rappresentata matematicamente come una **matrice**.

Immagine a Colore (RGB)



Immagine Grayscale



- **Dimensioni:** Larghezza x Altezza x 3 (canali)
- Ogni pixel è rappresentato da tre valori:
Rosso, Verde, Blu (RGB).

- **Dimensioni:** Larghezza x Altezza
- Ogni elemento della matrice è un singolo valore di intensità [0 .. 255]

Immagini come Matrici Multidimensionali

Struttura di Base

- Un'immagine digitale è una griglia di pixel.
- Ogni pixel rappresenta il **colore** o l'**intensità** di un punto specifico nell'immagine.
- Questa griglia può essere rappresentata matematicamente come una **matrice**.

Immagine a Colore (RGB)

B_{00}	B	B	B	B	B	B	
B_{10}	G_{00}	G	G	G	G	G	
B_{20}	R_{00}	R_{01}	R_{02}	R_{03}	R_{04}	R_{05}	R_{06}
B_{30}	R_{10}	R_{11}	R_{12}	R_{13}	R_{14}	R_{15}	R_{16}
B_{40}	R_{20}	R_{21}	R_{22}	R_{23}	R_{24}	R_{25}	R_{26}
B_{50}	R_{30}	R_{31}	R_{32}	R_{33}	R_{34}	R_{35}	R_{36}
	R_{40}	R_{41}	R_{43}	R_{43}	R_{44}	R_{45}	R_{46}
	R_{50}	R_{51}	R_{53}	R_{53}	R_{54}	R_{55}	R_{56}

- **Dimensioni:** Larghezza x Altezza x 3 (canali)
- Ogni pixel è rappresentato da tre valori:
Rosso, Verde, Blu (RGB).

Immagine Grayscale

I_{00}	I_{01}	I_{02}	I_{03}	I_{04}	I_{05}	I_{06}
I_{10}	I_{11}	I_{12}	I_{13}	I_{14}	I_{15}	I_{16}
I_{20}	I_{21}	I_{22}	I_{23}	I_{24}	I_{25}	I_{26}
I_{30}	I_{31}	I_{32}	I_{33}	I_{34}	I_{35}	I_{36}
I_{40}	I_{41}	I_{43}	I_{43}	I_{44}	I_{45}	I_{46}
I_{50}	I_{51}	I_{53}	I_{53}	I_{54}	I_{55}	I_{56}

- **Dimensioni:** Larghezza x Altezza
- Ogni elemento della matrice è un singolo valore di intensità [0 .. 255]

Memorizzazione Lineare di Immagini RGB in CUDA

- Per le immagini in **scala di grigi**, la memorizzazione in memoria globale è diretta e segue esattamente il principio **row-major** delle matrici classiche viste in precedenza.
- Per le immagini **RGB**, il principio di base rimane lo stesso, ma con una **complessità aggiuntiva** dovuta ai tre canali di colore.

Approccio di Memorizzazione (Caso RGB)

Ci sono due approcci principali per memorizzare un'immagine RGB in modo lineare:

1. Planar:

- Tutti i valori R, poi tutti i G, poi tutti i B

R ₀₀	R ₀₁	R ₀₂	R ₀₃	...	G ₀₀	G ₀₁	G ₀₂	G ₀₃	...	B ₀₀	B ₀₁	B ₀₂	B ₀₃	...
-----------------	-----------------	-----------------	-----------------	-----	-----------------	-----------------	-----------------	-----------------	-----	-----------------	-----------------	-----------------	-----------------	-----

2. Interleaved (più comune):

- I valori R, G, B per ogni pixel sono memorizzati consecutivamente

R ₀₀	G ₀₀	B ₀₀	R ₀₁	G ₀₁	B ₀₁	R ₀₂	G ₀₂	B ₀₂	R ₀₃	G ₀₃	B ₀₃
-----------------	-----------------	-----------------	-----------------	-----------------	-----------------	-----------------	-----------------	-----------------	-----------------	-----------------	-----------------	-----	-----	-----

Accesso agli Elementi dell'Immagine

RGB



3

width (W)

height (H)

Grayscale



Per accedere a un pixel specifico (i, j):

- **Calcola l'indice di base:**

$$\text{baseIndex} = (i * \text{width} + j) * 3$$

- **Accesso ai canali:**

- **R:** baseIndex
- **G:** $\text{baseIndex} + 1$
- **B:** $\text{baseIndex} + 2$

Per accedere a un pixel specifico (i, j):

- **Calcola l'indice di base:**

$$\text{baseIndex} = i * \text{width} + j$$

Parallelismo GPU nella Conversione RGB a Grayscale

Perché le GPU sono Ideali per l'Elaborazione delle Immagini

- **Struttura delle Immagini**
 - Le immagini sono composte da molti **pixel indipendenti**.
 - Ogni pixel può essere elaborato **separatamente**.
- **Operazioni Uniformi**
 - La **stessa operazione** viene spesso applicata a tutti i pixel.
 - Perfetto per il paradigma **SIMD** (Single Instruction, Multiple Data).

Esempio: Conversione RGB a Grayscale



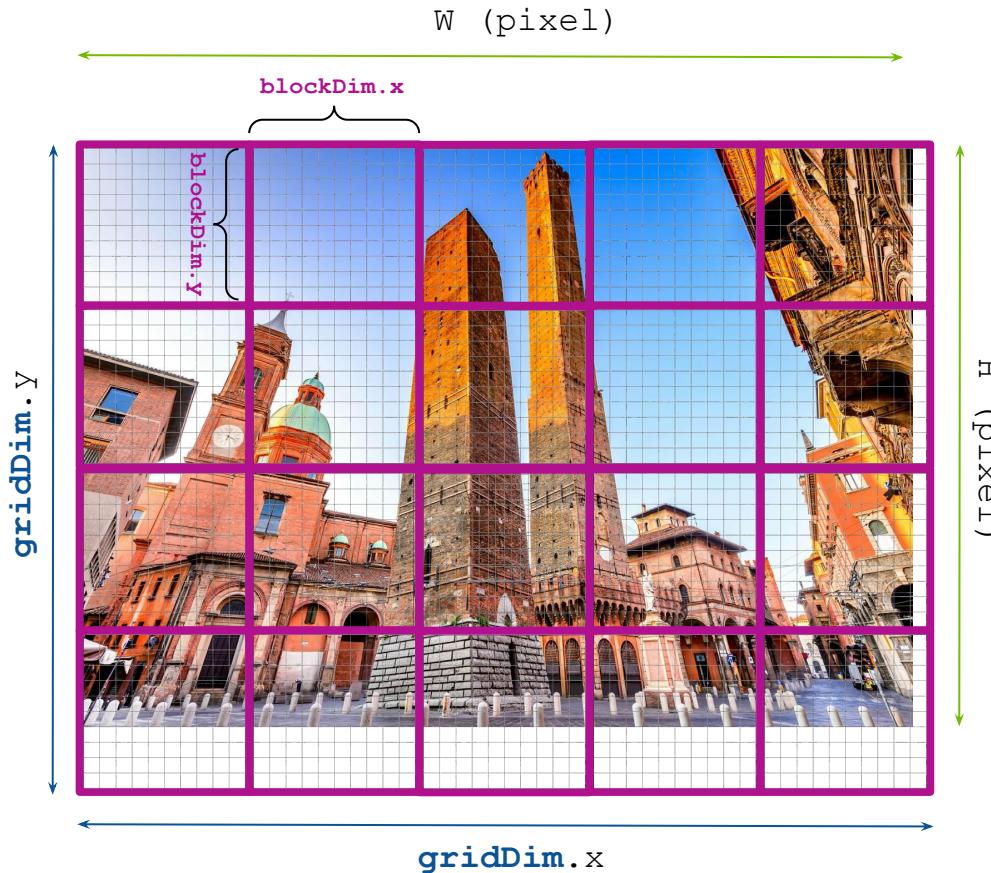
Formula: Gray = 0.299R + 0.587G + 0.114B (per pixel)

Suddivisione dell'Immagine in Blocchi per l'Elaborazione GPU

- L'elaborazione di immagini su GPU richiede la **suddivisione del lavoro in unità parallele**.
- L'immagine viene divisa in una **griglia di blocchi**, ciascuno elaborato da un gruppo di thread.
 - **gridDim**: Numero di blocchi nella griglia.
 - **blockDim**: Numero di thread in ciascun blocco.

Calcolo degli indici nel buffer RGB

```
ix = threadIdx.x + blockDim.x * blockDim.x  
iy = threadIdx.y + blockDim.y * blockDim.y  
base_index = (iy * width + ix) * 3  
index_R = base_index  
index_G = base_index + 1  
index_B = base_index + 2
```



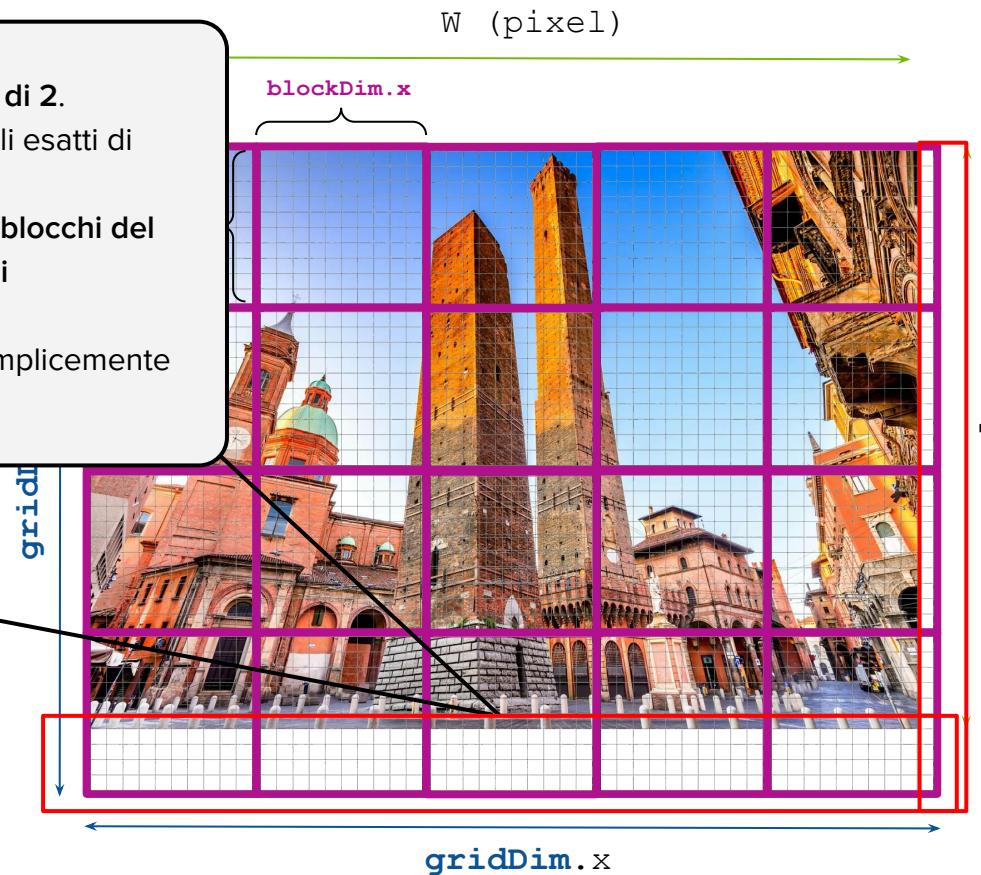
Suddivisione dell'Immagine in Blocchi per l'Elaborazione GPU

Threads Oltre i Limiti dell'Immagine

- Le dimensioni dei blocchi sono tipicamente **potenze di 2**.
- Le dimensioni delle immagini **raramente** sono multipli esatti di queste dimensioni dei blocchi.
- Per coprire l'intera immagine, si lanciano spesso **più blocchi del necessario**, alcuni dei quali si estendono **oltre i bordi** dell'immagine.
- I thread **che cadono fuori** dai limiti dell'immagine semplicemente **non eseguono** alcuna operazione.

Calcolo degli indici nel buffer RGB

```
ix = threadIdx.x + blockIdx.x * blockDim.x  
iy = threadIdx.y + blockIdx.y * blockDim.y  
base_index = (iy * width + ix) * 3  
index_R = base_index  
index_G = base_index + 1  
index_B = base_index + 2
```



Confronto: Conversione RGB a Grayscale in C vs CUDA C

Codice C Standard

```
// Funzione host per la conversione RGB->Gray
void rgbToGrayCPU(unsigned char *rgb, unsigned char *gray, int width, int height) {
    for (int y = 0; y < height; y++) { // Ciclo su tutte le righe dell'immagine
        for (int x = 0; x < width; x++) { // Ciclo su tutti i pixel di una riga
            int rgbOffset = (y * width + x) * 3; // Calcola l'offset per il pixel RGB
            int grayOffset = y * width + x; // Calcola l'offset per il pixel in scala di grigi

            unsigned char r = rgb[rgbOffset]; // Legge il valore rosso
            unsigned char g = rgb[rgbOffset + 1]; // Legge il valore verde
            unsigned char b = rgb[rgbOffset + 2]; // Legge il valore blu

            gray[grayOffset] = (unsigned char) (0.299f * r + 0.587f * g + 0.114f * b); // RGB->Gray
        }
    }
}
```

Confronto: Conversione RGB a Grayscale in C vs CUDA C

Codice CUDA C

```
// Funzione kernel per la conversione RGB->Gray
__global__ void rgbToGrayGPU(unsigned char *d_rgb, unsigned char *d_gray, int width, int height) {
    int ix = blockIdx.x * blockDim.x + threadIdx.x; // Calcola la coordinata x del pixel
    int iy = blockIdx.y * blockDim.y + threadIdx.y; // Calcola la coordinata y del pixel

    if (ix < width && iy < height) { // Controllo dei bordi: assicura che il thread sia dentro l'immagine
        int rgbOffset = (iy * width + ix) * 3; // Calcola l'offset per il pixel RGB
        int grayOffset = iy * width + ix; // Calcola l'offset per il pixel in scala di grigi

        unsigned char r = d_rgb[rgbOffset]; // Legge il valore rosso
        unsigned char g = d_rgb[rgbOffset + 1]; // Legge il valore verde
        unsigned char b = d_rgb[rgbOffset + 2]; // Legge il valore blu

        d_gray[grayOffset] = (unsigned char) (0.299f * r + 0.587f * g + 0.114f * b); // RGB->Gray
    }
}
```

Conversione RGB a Grayscale in CUDA

Conversione RGB -> Grayscale

1/3

```
int main(int argc, char **argv) {
    if (argc != 2) {
        printf("Usage: %s <image_file>\n", argv[0]);
        return 1;
    }
    printf("%s Starting...\n", argv[0]);

    // Imposta il device CUDA
    int dev = 0;
    cudaDeviceProp deviceProp;
    CHECK(cudaGetDeviceProperties(&deviceProp, dev)); // Ottiene le proprietà del dispositivo CUDA
    CHECK(cudaSetDevice(0)); // Seleziona il dispositivo CUDA

    // Carica l'immagine usando "stb_image.h" e "stb_image_write.h"
    int width, height, channels;
    unsigned char *rgb = stbi_load(argv[1], &width, &height, &channels, 3);
    if (!rgb) {
        printf("Error loading image %s\n", argv[1]);
        return 1;
    }
    printf("Image loaded: %dx%d with %d channels\n", width, height, channels);

    // Alloca la memoria host per l'immagine in scala di grigi
    int imageSize = width * height;
    int rgbSize = imageSize * 3;
    unsigned char *h_gray = (unsigned char *)malloc(imageSize); // Alloca memoria per l'output GPU
    unsigned char *cpu_gray = (unsigned char *)malloc(imageSize); // Alloca memoria per l'output CPU
```

Conversione RGB a Grayscale in CUDA

Conversione RGB -> Grayscale

2/3

```
// Converti l'immagine in scala di grigi sulla CPU
rgbToGrayscaleCPU(rgb, cpu_gray, width, height);

// Alloca la memoria del device
unsigned char *d_rgb, *d_gray;
CHECK(cudaMalloc((void **) &d_rgb, rgbSize)); // Alloca memoria GPU per l'immagine RGB
CHECK(cudaMalloc((void **) &d_gray, imageSize)); // Alloca memoria GPU per l'output
// Trasferisce i dati dall'host al device
CHECK(cudaMemcpy(d_rgb, rgb, rgbSize, cudaMemcpyHostToDevice));

// Configura e invoca il kernel CUDA
dim3 block(32, 32); // Dimensione del blocco: 32x32 thread
dim3 grid((width + block.x - 1) / block.x, (height + block.y - 1) / block.y);

rgbToGrayscaleGPU<<<grid, block>>>(d_rgb, d_gray, width, height); // Lancia il kernel
CHECK(cudaDeviceSynchronize()); // Aspetta il completamento del kernel

// Copia il risultato del kernel dal device all'host
CHECK(cudaMemcpy(h_gray, d_gray, imageSize, cudaMemcpyDeviceToHost));
```

Conversione RGB a Grayscale in CUDA

Conversione RGB -> Grayscale

3/3

```
// Verifica il risultato
bool match = true;
for (int i = 0; i < imageSize; i++) {
    if (abs(cpu_gray[i] - h_gray[i]) > 1) { // Tollerà piccole differenze di arrotondamento.
        match = false;
        printf("Mismatch at pixel %d: CPU %d, GPU %d\n", i, cpu_gray[i], h_gray[i]);
        break;
    }
}
if (match) printf("CPU and GPU results match.\n");

// Salva l'immagine in scala di grigi
stbi_write_png("output_gray.png", width, height, 1, h_gray, width);

// Libera la memoria
stbi_image_free(rgb);
free(h_gray);
free(cpu_gray);
CHECK(cudaFree(d_rgb));
CHECK(cudaFree(d_gray));

// Resetta il device CUDA
CHECK(cudaDeviceReset());
return 0;
}
```

Image Flipping con CUDA

Introduzione all'Image Flipping

- L'image flipping è una tecnica di elaborazione delle immagini che **inverte l'ordine dei pixel lungo un asse specifico per ciascun canale di colore**, creando un **effetto specchio**. Il flipping può essere:
 - Orizzontale:** Invertendo l'ordine dei pixel da sinistra a destra.
 - Verticale:** Invertendo l'ordine dei pixel dall'alto verso il basso.



Image Flipping con CUDA

Processo di Flipping in CUDA

- In CUDA, ogni thread è responsabile del calcolo e della gestione di un singolo pixel dell'immagine.
 - Per un **flip orizzontale**, il thread calcola la nuova posizione speculare del pixel. Per un pixel inizialmente in posizione (i, j) , il thread calcola la nuova posizione come $(\text{width} - i - 1, j)$.
 - Per un **flip verticale**, la nuova posizione è calcolata come $(i, \text{height} - j - 1)$.
- Il thread **copia i valori** dei canali RGB del pixel originale nella nuova posizione calcolata.

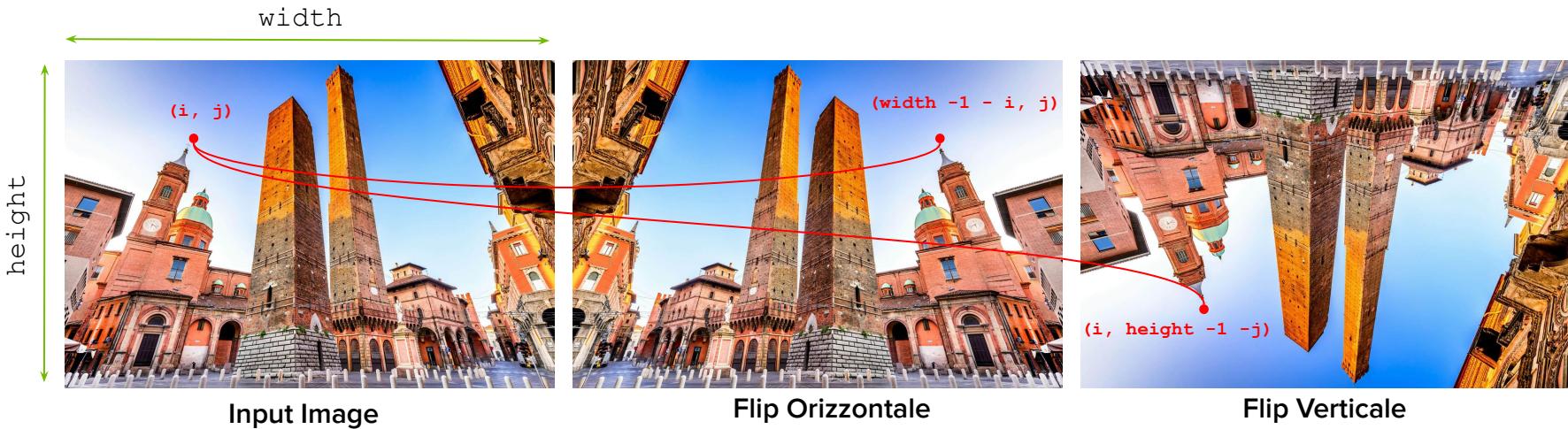


Image Flipping con CUDA

Flipping di un'Immagine

```
__global__ void cudaImageFlip(unsigned char* input, unsigned char* output,
                             int width, int height, int channels, bool horizontal) {
    int ix = blockIdx.x * blockDim.x + threadIdx.x; // Calcola la coordinata x del pixel
    int iy = blockIdx.y * blockDim.y + threadIdx.y; // Calcola la coordinata y del pixel

    if (ix < width && iy < height) { // Verifica se il pixel è all'interno dell'immagine
        int outputIdx;
        int inputIdx = (iy * width + ix) * channels;

        if (horizontal) {
            outputIdx = (iy * width + (width - 1 - ix)) * channels; // Indice flip orizzontale
        } else {
            outputIdx = ((height - 1 - iy) * width + ix) * channels; // Indice flip verticale
        }

        for (int c = 0; c < channels; ++c) {
            output[outputIdx + c] = input[inputIdx + c]; // Copia i valori nella nuova posizione
        }
    }
}
```

Image Blur con CUDA: Un Kernel più Complesso

Introduzione all'Image Blurring

L'image blurring è una tecnica di elaborazione delle immagini che **riduce i dettagli** e le variazioni di intensità, creando un **effetto di sfocatura**. Viene utilizzata per:

- **Riduzione del rumore:** Attenuando le fluttuazioni casuali dei pixel.
- **Enfasi degli oggetti:** Sfumando i dettagli irrilevanti e mettendo in risalto gli elementi principali.
- **Preprocessing per la Computer Vision:** Semplificando l'immagine per facilitarne l'analisi da parte degli algoritmi.



Input Image



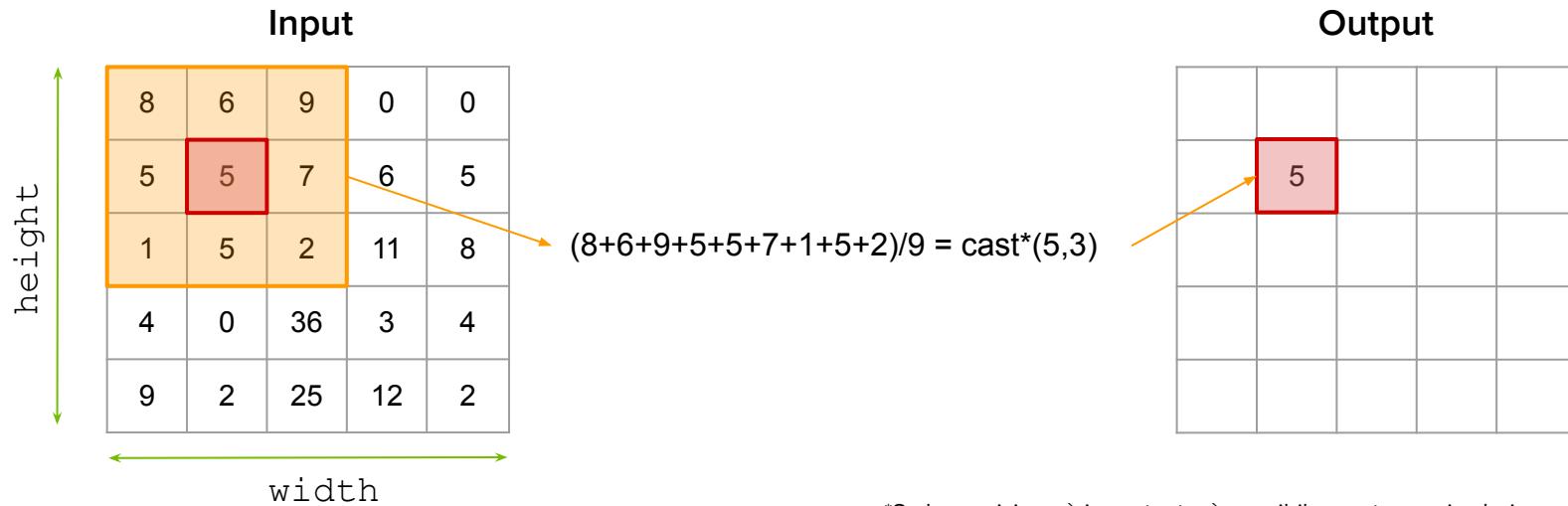
Blurred Image (`window_size=25`)

Image Blur con CUDA: Un Kernel più Complesso

Concetto di Base

Il blurring si ottiene calcolando la **media dei valori di intensità** dei pixel vicini di ogni pixel dell'immagine originale. L'operazione può essere riassunta come segue:

- **Patch di dimensioni N×N:** Una patch (o finestra) di dimensioni fisse scorre su ciascun pixel dell'immagine.
- **Pixel centrale:** Ogni pixel di output è la media dei pixel nella patch che lo circondano.
- **Esempio con patch 3×3:** Include il pixel centrale più gli 8 pixel che lo circondano, formando una matrice di 3 righe e 3 colonne.



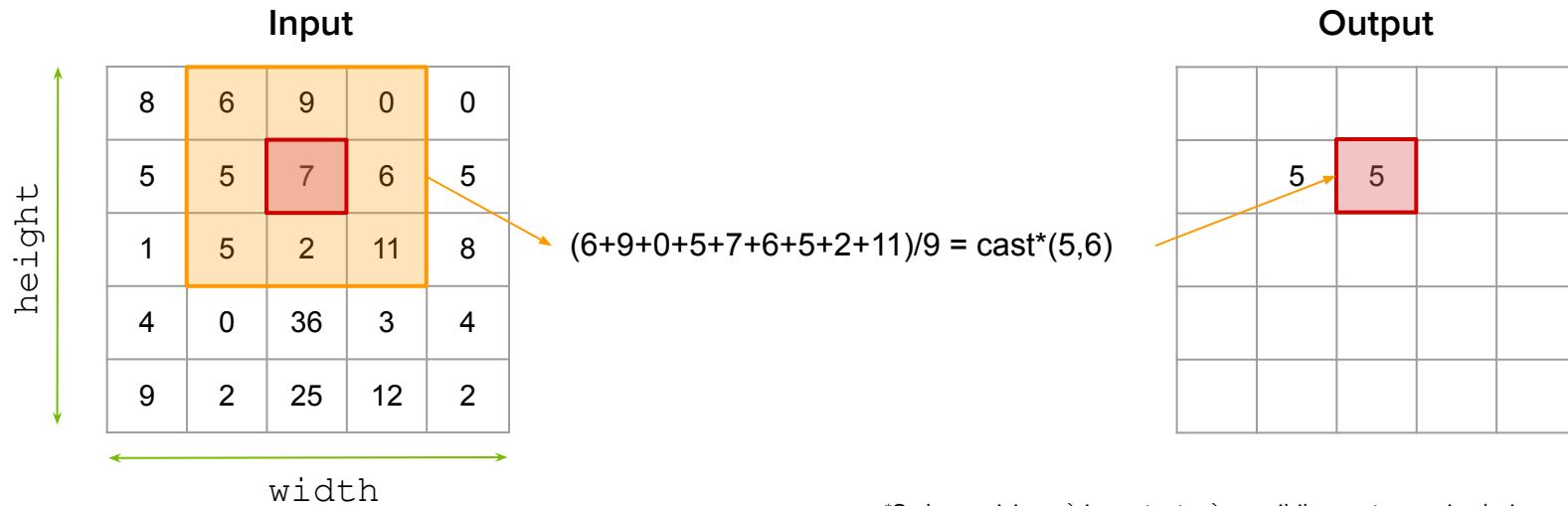
*Se la precisione è importante, è possibile mantenere i valori come float o double.

Image Blur con CUDA: Un Kernel più Complesso

Concetto di Base

Il blurring si ottiene calcolando la **media dei valori di intensità** dei pixel vicini di ogni pixel dell'immagine originale. L'operazione può essere riassunta come segue:

- **Patch di dimensioni N×N:** Una patch (o finestra) di dimensioni fisse scorre su ciascun pixel dell'immagine.
- **Pixel centrale:** Ogni pixel di output è la media dei pixel nella patch che lo circondano.
- **Esempio con patch 3×3:** Include il pixel centrale più gli 8 pixel che lo circondano, formando una matrice di 3 righe e 3 colonne.



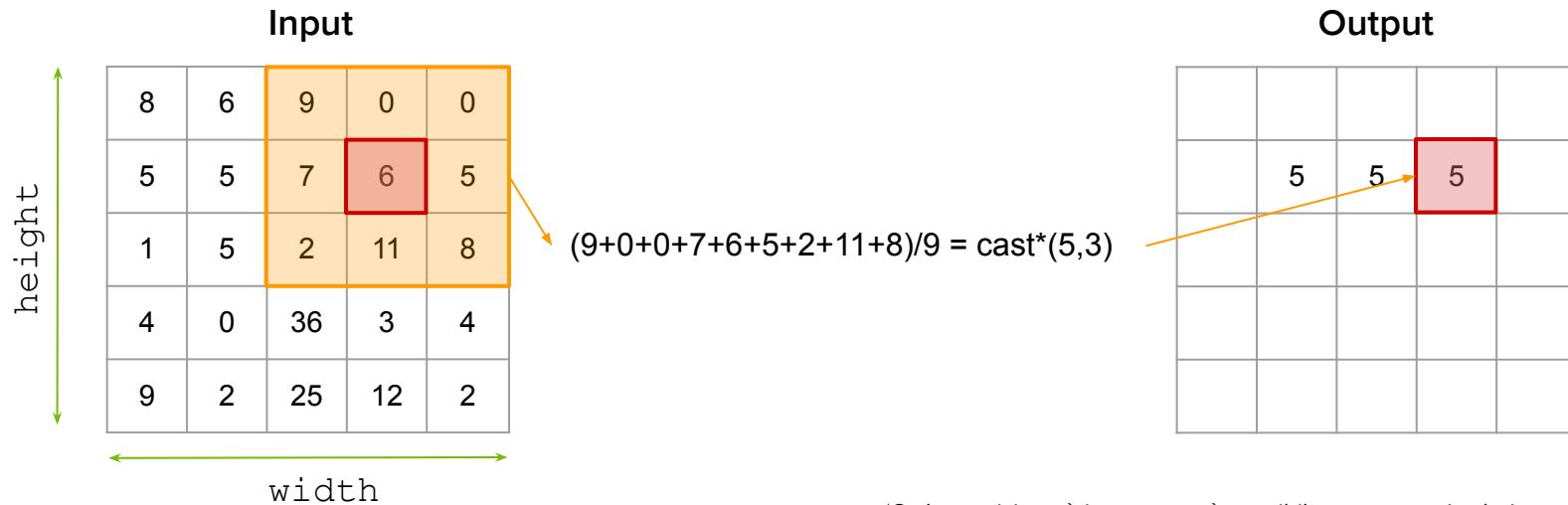
*Se la precisione è importante, è possibile mantenere i valori come float o double.

Image Blur con CUDA: Un Kernel più Complesso

Concetto di Base

Il blurring si ottiene calcolando la **media dei valori di intensità** dei pixel vicini di ogni pixel dell'immagine originale. L'operazione può essere riassunta come segue:

- **Patch di dimensioni N×N:** Una patch (o finestra) di dimensioni fisse scorre su ciascun pixel dell'immagine.
- **Pixel centrale:** Ogni pixel di output è la media dei pixel nella patch che lo circondano.
- **Esempio con patch 3×3:** Include il pixel centrale più gli 8 pixel che lo circondano, formando una matrice di 3 righe e 3 colonne.



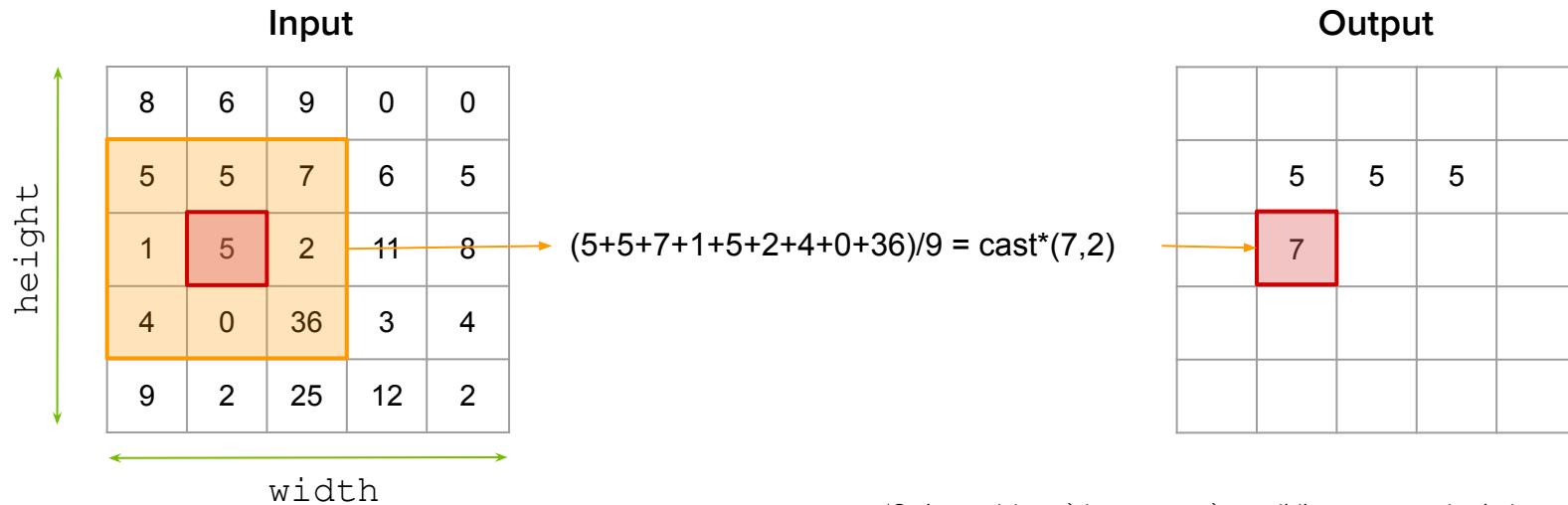
*Se la precisione è importante, è possibile mantenere i valori come float o double.

Image Blur con CUDA: Un Kernel più Complesso

Concetto di Base

Il blurring si ottiene calcolando la **media dei valori di intensità** dei pixel vicini di ogni pixel dell'immagine originale. L'operazione può essere riassunta come segue:

- **Patch di dimensioni N×N:** Una patch (o finestra) di dimensioni fisse scorre su ciascun pixel dell'immagine.
- **Pixel centrale:** Ogni pixel di output è la media dei pixel nella patch che lo circondano.
- **Esempio con patch 3×3:** Include il pixel centrale più gli 8 pixel che lo circondano, formando una matrice di 3 righe e 3 colonne.



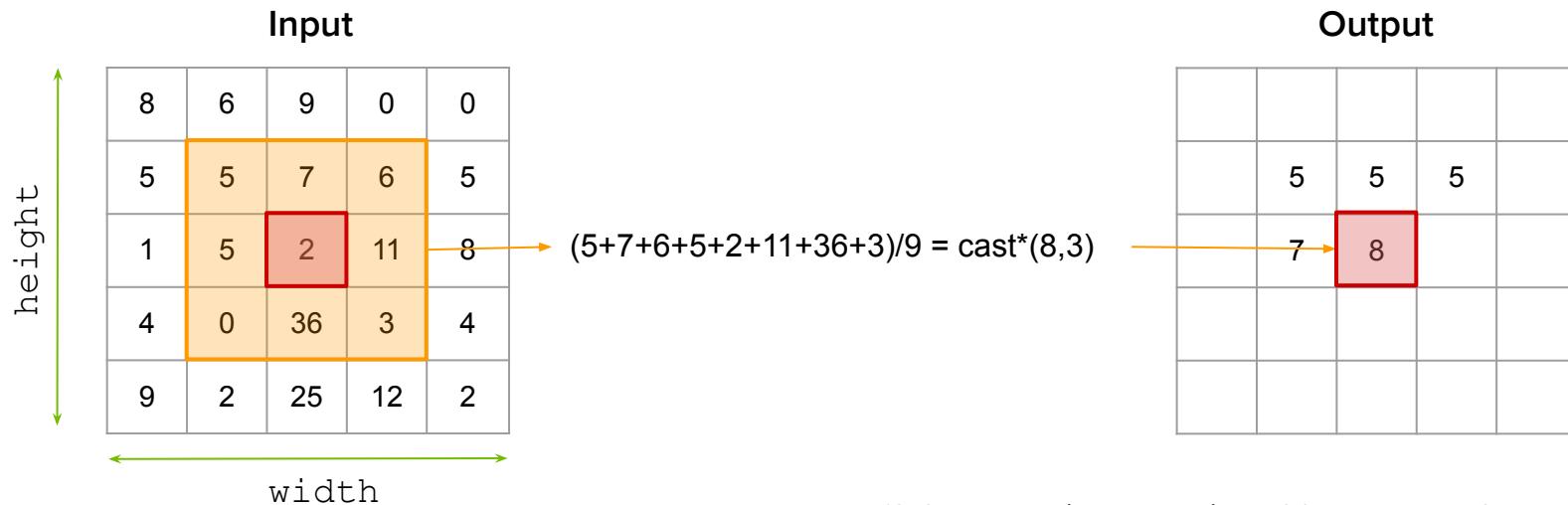
*Se la precisione è importante, è possibile mantenere i valori come float o double.

Image Blur con CUDA: Un Kernel più Complesso

Concetto di Base

Il blurring si ottiene calcolando la **media dei valori di intensità** dei pixel vicini di ogni pixel dell'immagine originale. L'operazione può essere riassunta come segue:

- **Patch di dimensioni N×N:** Una patch (o finestra) di dimensioni fisse scorre su ciascun pixel dell'immagine.
- **Pixel centrale:** Ogni pixel di output è la media dei pixel nella patch che lo circondano.
- **Esempio con patch 3×3:** Include il pixel centrale più gli 8 pixel che lo circondano, formando una matrice di 3 righe e 3 colonne.



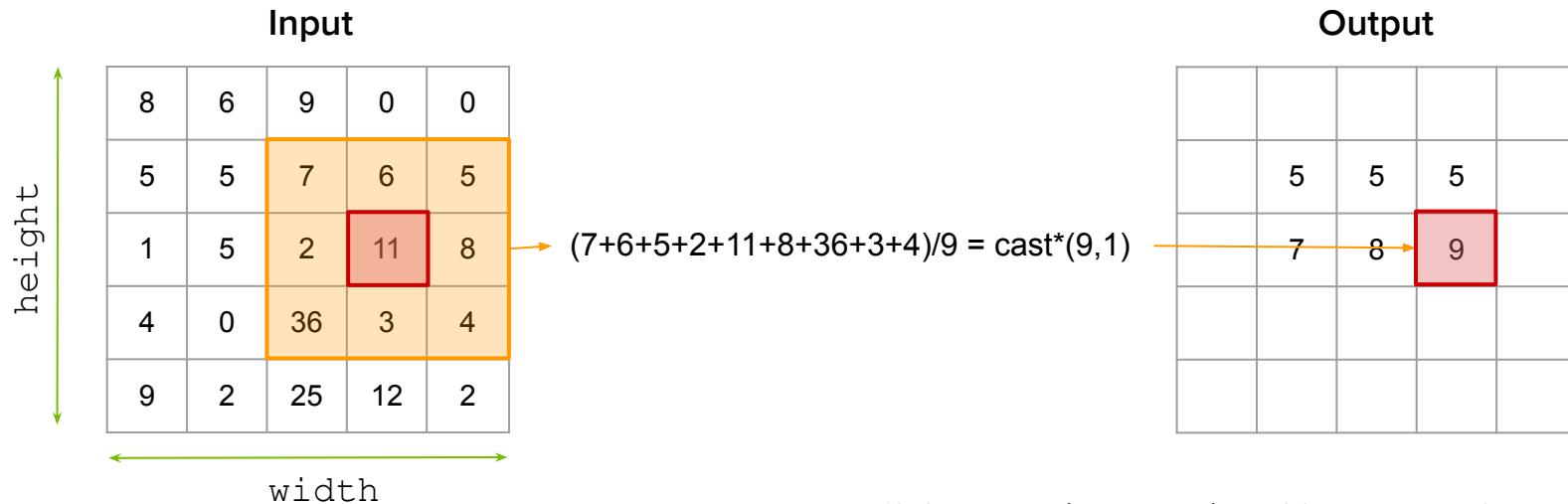
*Se la precisione è importante, è possibile mantenere i valori come float o double.

Image Blur con CUDA: Un Kernel più Complesso

Concetto di Base

Il blurring si ottiene calcolando la **media dei valori di intensità** dei pixel vicini di ogni pixel dell'immagine originale. L'operazione può essere riassunta come segue:

- **Patch di dimensioni N×N:** Una patch (o finestra) di dimensioni fisse scorre su ciascun pixel dell'immagine.
- **Pixel centrale:** Ogni pixel di output è la media dei pixel nella patch che lo circondano.
- **Esempio con patch 3×3:** Include il pixel centrale più gli 8 pixel che lo circondano, formando una matrice di 3 righe e 3 colonne.



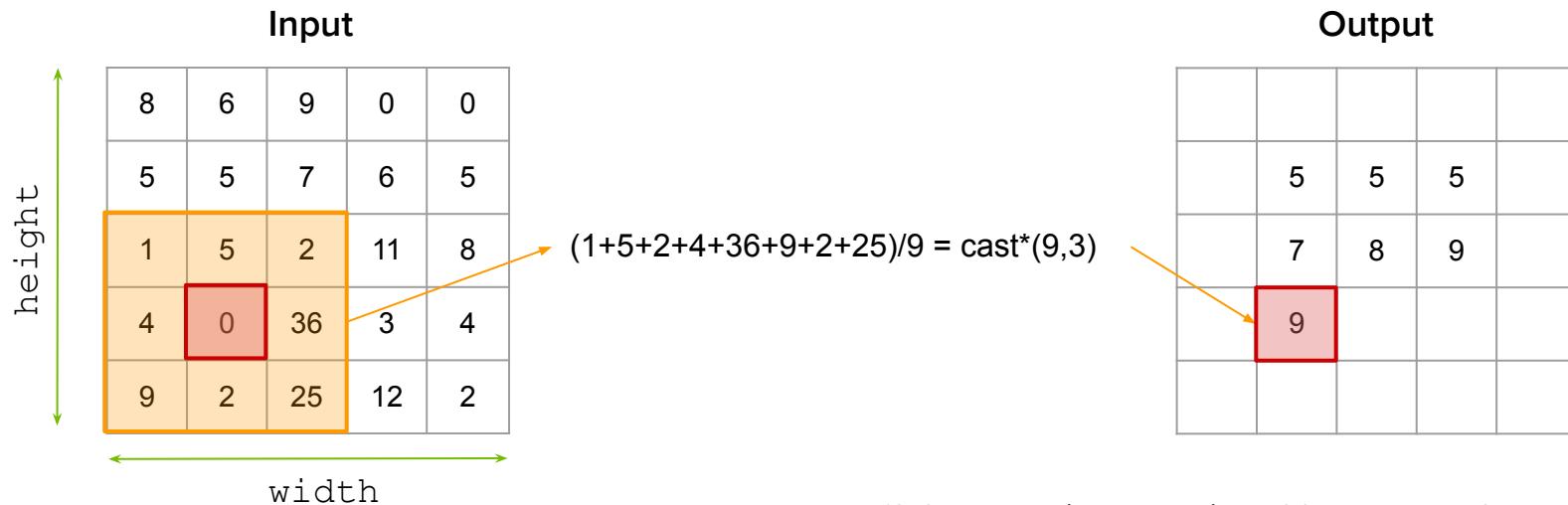
*Se la precisione è importante, è possibile mantenere i valori come float o double.

Image Blur con CUDA: Un Kernel più Complesso

Concetto di Base

Il blurring si ottiene calcolando la **media dei valori di intensità** dei pixel vicini di ogni pixel dell'immagine originale. L'operazione può essere riassunta come segue:

- **Patch di dimensioni N×N:** Una patch (o finestra) di dimensioni fisse scorre su ciascun pixel dell'immagine.
- **Pixel centrale:** Ogni pixel di output è la media dei pixel nella patch che lo circondano.
- **Esempio con patch 3×3:** Include il pixel centrale più gli 8 pixel che lo circondano, formando una matrice di 3 righe e 3 colonne.



*Se la precisione è importante, è possibile mantenere i valori come float o double.

Image Blur con CUDA: Un Kernel più Complesso

Concetto di Base

Il blurring si ottiene calcolando la **media dei valori di intensità** dei pixel vicini di ogni pixel dell'immagine originale. L'operazione può essere riassunta come segue:

- **Patch di dimensioni N×N:** Una patch (o finestra) di dimensioni fisse scorre su ciascun pixel dell'immagine.
- **Pixel centrale:** Ogni pixel di output è la media dei pixel nella patch che lo circondano.
- **Esempio con patch 3×3:** Include il pixel centrale più gli 8 pixel che lo circondano, formando una matrice di 3 righe e 3 colonne.

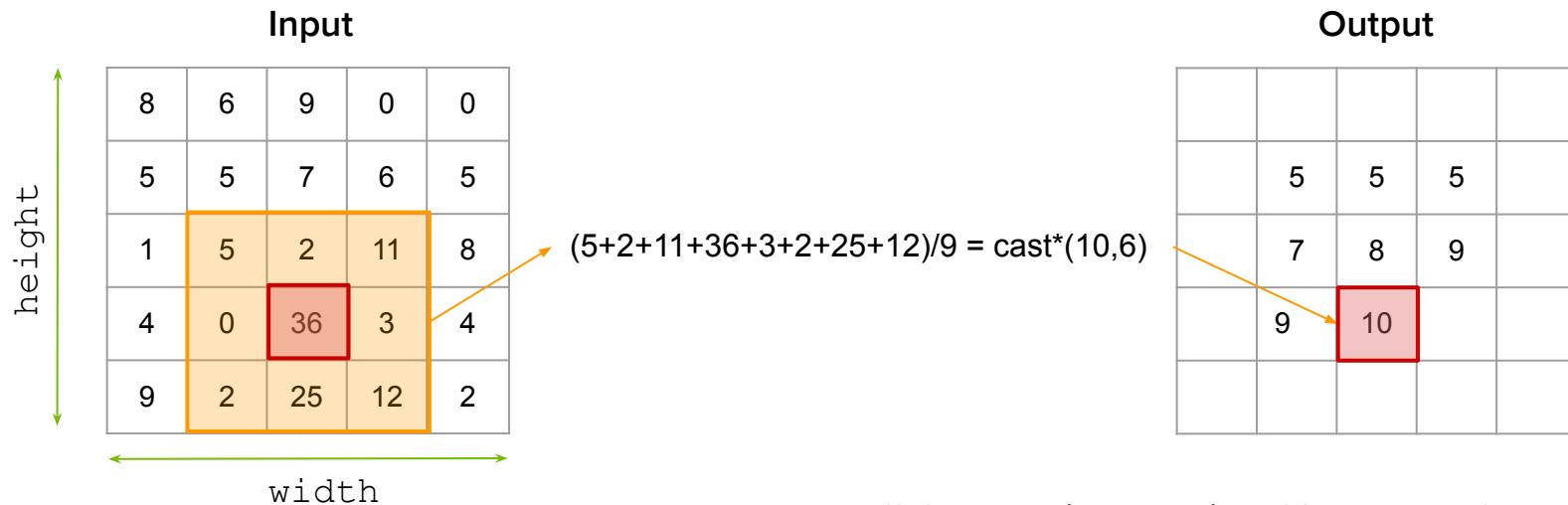
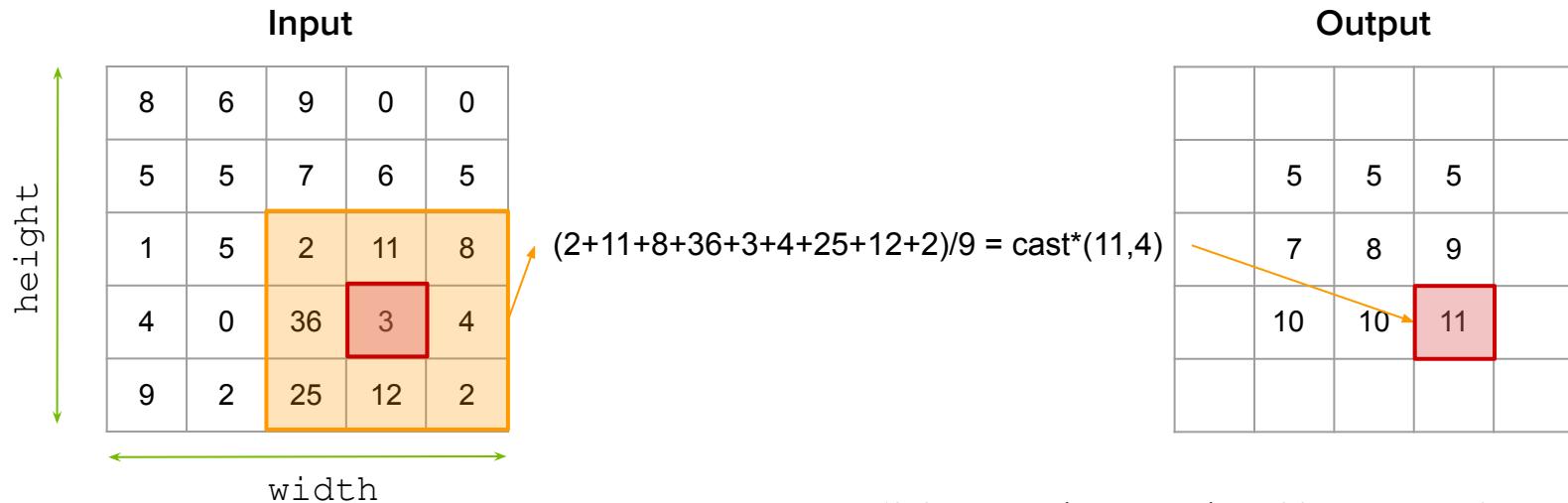


Image Blur con CUDA: Un Kernel più Complesso

Concetto di Base

Il blurring si ottiene calcolando la **media dei valori di intensità** dei pixel vicini di ogni pixel dell'immagine originale. L'operazione può essere riassunta come segue:

- **Patch di dimensioni N×N:** Una patch (o finestra) di dimensioni fisse scorre su ciascun pixel dell'immagine.
- **Pixel centrale:** Ogni pixel di output è la media dei pixel nella patch che lo circondano.
- **Esempio con patch 3×3:** Include il pixel centrale più gli 8 pixel che lo circondano, formando una matrice di 3 righe e 3 colonne.



*Se la precisione è importante, è possibile mantenere i valori come float o double.

Image Blur con CUDA: Un Kernel più Complesso

Concetto di Base

Il blurring si ottiene calcolando la **media dei valori di intensità** dei pixel vicini di ogni pixel dell'immagine originale. L'operazione può essere riassunta come segue:

- **Patch di dimensioni N×N:** Una patch (o finestra) di dimensioni fisse scorre su ciascun pixel dell'immagine.
- **Pixel centrale:** Ogni pixel di output è la media dei pixel nella patch che lo circondano.
- **Esempio con patch 3×3:** Include il pixel centrale più gli 8 pixel che lo circondano, formando una matrice di 3 righe e 3 colonne.

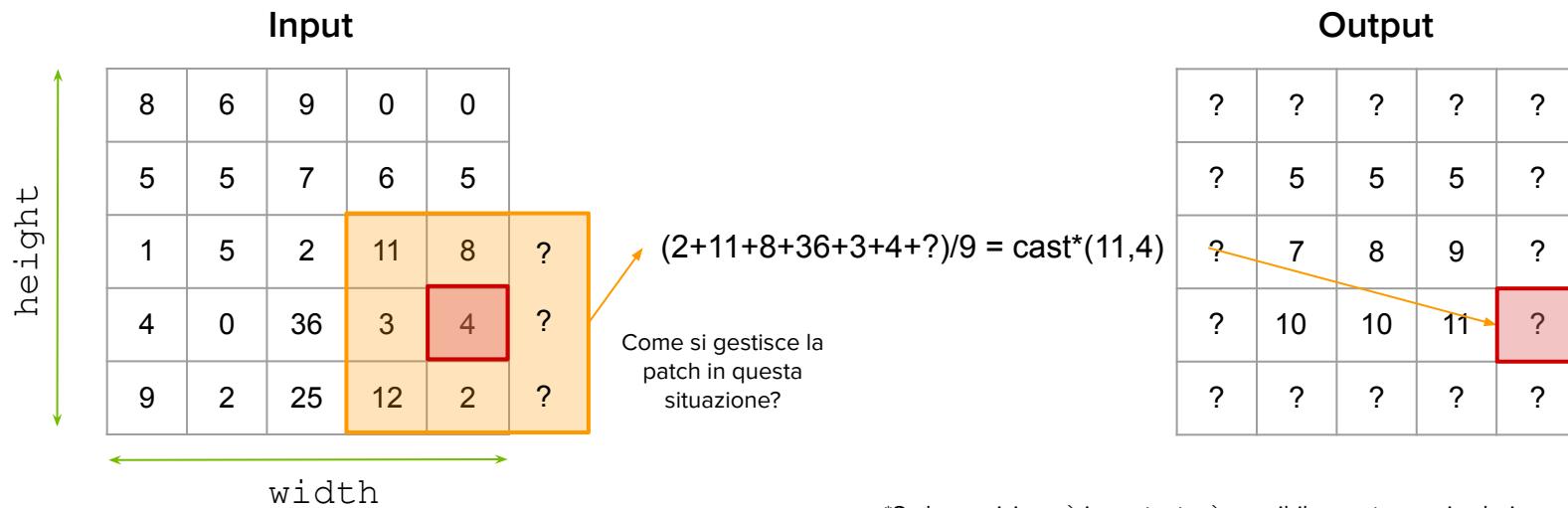


Image Blur con CUDA: Un Kernel più Complesso

Caratteristiche Chiave del Kernel Blur

- **Mappatura Thread-Pixel:** Ogni thread è responsabile del calcolo di un singolo pixel nell'immagine di output.
- **Gestione dei Bordi:** Controlli specifici assicurano che la finestra di blur rimanga entro i confini dell'immagine, evitando letture di memoria non valide ai margini.
- **Parallelismo:** Il kernel sfrutta il parallelismo massiccio delle GPU, dato che il calcolo per ciascun pixel è indipendente dagli altri.
- **Pattern di Accesso alla Memoria:** Ogni thread accede a un vicinato di pixel (la patch) che, a seconda della disposizione dei dati in memoria, può comportare accessi **non sempre sequenziali**.

Confronto con Kernel Precedenti

- **Complessità:** Rispetto a semplici kernel come **vecAdd** (addizione vettoriale) o **rgbToGray** (conversione in scala di grigi), questo kernel è più complesso a causa della necessità di gestire più pixel e calcoli per ogni thread.
- **Accessi alla Memoria:** Ogni thread accede a più pixel rispetto a kernel semplici, aumentando la frequenza di accessi alla memoria globale.
- **Scalabilità:** La dimensione della patch di blur (**BLUR_SIZE**) impatta direttamente la quantità di calcolo e gli accessi alla memoria. Patch più grandi producono sfocature più intense ma richiedono più risorse.

Image Blur con CUDA: Un Kernel più Complesso

Esercizio

- Implementare in CUDA un kernel che applichi un filtro di blurring su un'immagine, sfruttando il parallelismo offerto dalle GPU per accelerare l'elaborazione rispetto a una soluzione sequenziale.

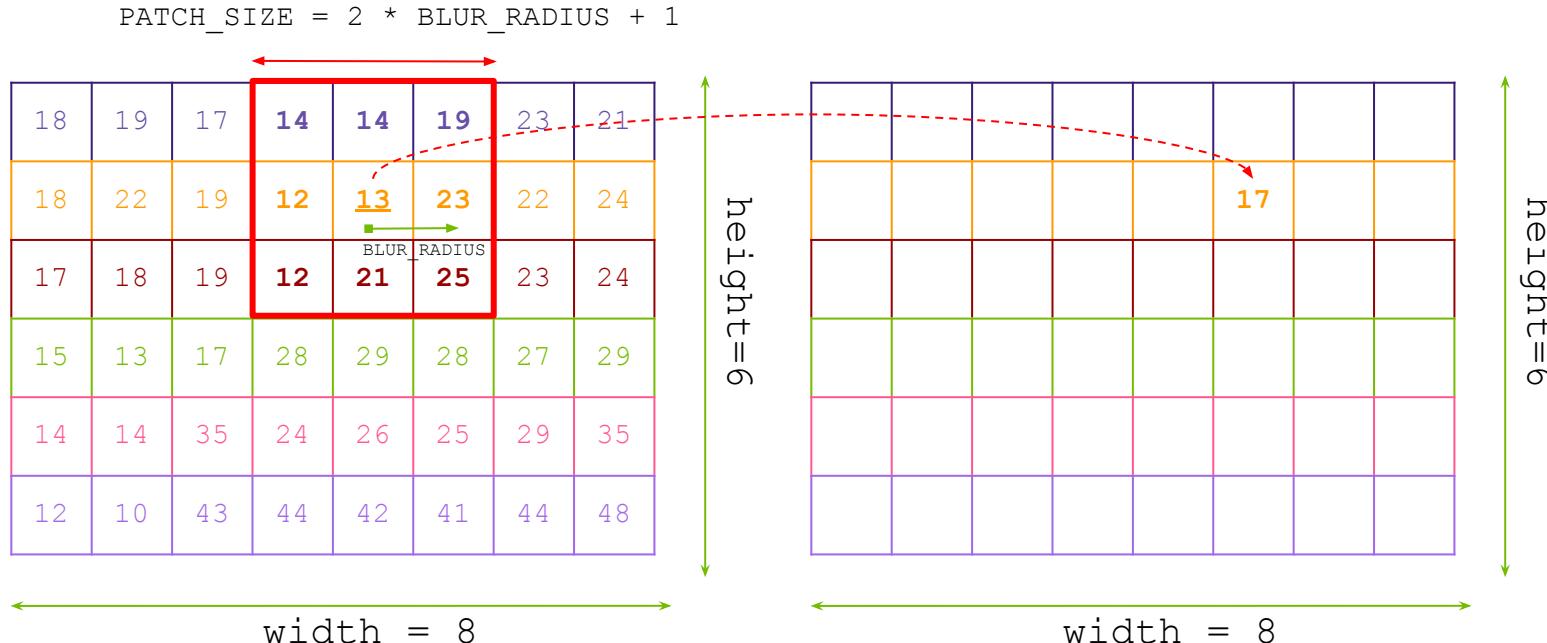


Image Blur con CUDA: Soluzione

```
#define BLUR_RADIUS 1 // Raggio del blur (1 significa una finestra 3x3)

__global__ void cudaImageBlur(unsigned char* input, unsigned char* output, int width, int height) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    if (x < width && y < height) {
        int pixelSum = 0, pixelCount = 0;
        // Itera sulla finestra di blur
        for (int dy = -BLUR_RADIUS; dy <= BLUR_RADIUS; ++dy) {
            for (int dx = -BLUR_RADIUS; dx <= BLUR_RADIUS; ++dx) {
                int currentY = y + dy, currentX = x + dx;
                // Verifica se il pixel è all'interno dell'immagine
                if (currentY >= 0 && currentY < height && currentX >= 0 && currentX < width) {
                    pixelSum += input[currentY * width + currentX];
                    pixelCount++;
                }
            }
        }
        // Calcola e scrive il valore medio del pixel
        output[y * width + x] = (unsigned char)(pixelSum / pixelCount);
    }
}
```

Introduzione alla Convoluzione 1D e 2D

Che cos'è la Convoluzione?

- Operazione matematica lineare tra **due funzioni**, segnale e kernel (fuorviante - spesso indicato come **filtro**).
- Misura la **sovraposizione** del filtro con il segnale mentre scorre su di esso.
- Produce una nuova funzione (segnale di output) che rappresenta le **caratteristiche estratte** dal segnale di input.

Convoluzione 1D

- Applicata a **dati unidimensionali** (segnali audio, serie temporali, sequenze di testo).
- Il filtro è un vettore che **scorre** sul segnale.
- L'output ad ogni punto è la **somma dei prodotti elemento per elemento (prodotto scalare)** tra il filtro e la porzione di segnale sottostante.
- **Esempio:** Applicazione di un filtro di media mobile su un segnale audio per ridurre il rumore.

Convoluzione 2D

- Applicata a **dati bidimensionali** (es. immagini).
- Il filtro è una matrice che **scorre** sull'immagine.
- L'output ad ogni pixel è la **somma dei prodotti elemento per elemento (prodotto scalare)** tra il filtro e la regione dell'immagine sottostante.
- **Esempio:** (Image Blur caso particolare di convoluzione 2D. Perchè?)
 - Applicazione di un filtro di **rilevamento dei bordi** a un'immagine per estrarre i contorni degli oggetti.
 - Fondamentale nelle **reti neurali convoluzionali (CNN)** per l'elaborazione di immagini.

Introduzione alla Convoluzione 1D e 2D

Che cos'è la Convoluzione?

- Operazione matematica lineare tra due funzioni, segnale e kernel (fuorviante - spesso indicato come **filtro**).
- Misura la **sovraposizione** del filtro con il segnale mentre scorre su di esso.
- Pro

Convoluzio

- App
- Il filt
- L'out
- por
- Ese

Convoluzio

- App
- Il filt
- L'out

dell'immagine sottostante.

- **Esempio:** (Image Blur caso particolare di convoluzione 2D. Perchè?)
 - Applicazione di un filtro di **rilevamento dei bordi** a un'immagine per estrarre i contorni degli oggetti.
 - Fondamentale nelle **reti neurali convoluzionali (CNN)** per l'elaborazione di immagini.

Concetti Aggiuntivi

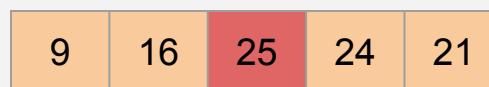
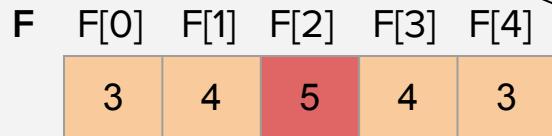
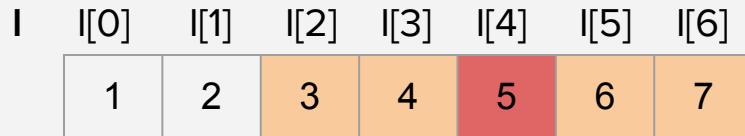
di input.

regione

Esempio di Convoluzione 1D

Descrizione

- **Input (I):** Array di 7 elementi ($I[0] \dots I[6]$).
- **Filtro (F):** Array di 5 elementi ($F[0] \dots F[4]$).
- **Output (O):** Array risultante dalla convoluzione di I con F.



raggio

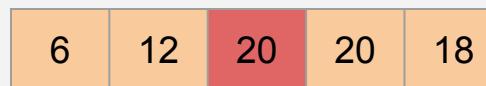
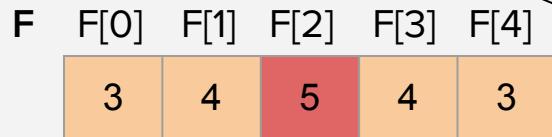
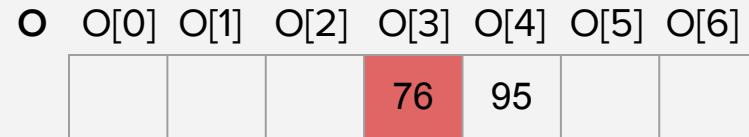
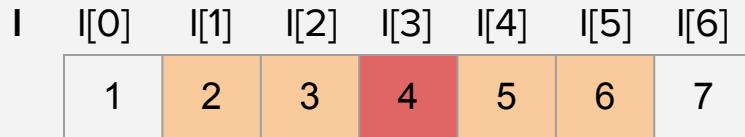
$$\begin{aligned} O[4] &= I[2]*F[0] + I[3]*F[1] + I[4]*F[2] + I[5]*F[3] + I[6]*F[4] \\ &= 3*3 + 4*4 + 5*5 + 6*4 + 7*3 = \\ &= 9 + 16 + 25 + 24 + 21 = \\ &= 95 \end{aligned}$$

Calcolo di $O[4]$

Esempio di Convoluzione 1D

Descrizione

- **Input (I):** Array di 7 elementi ($I[0] \dots I[6]$).
- **Filtro (F):** Array di 5 elementi ($F[0] \dots F[4]$).
- **Output (O):** Array risultante dalla convoluzione di I con F.



raggio

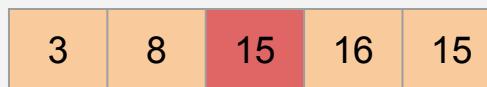
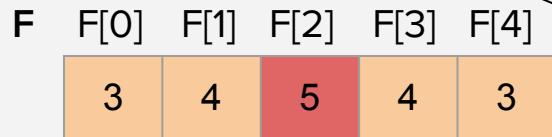
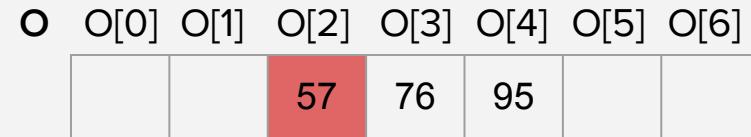
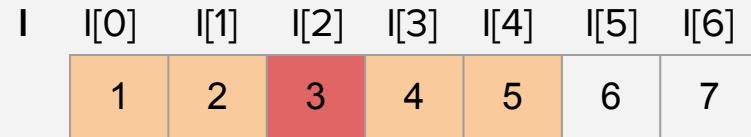
$$\begin{aligned} O[3] &= I[1]*F[0] + I[2]*F[1] + I[3]*F[2] + I[4]*F[3] + I[5]*F[4] \\ &= 2*3 + 3*4 + 4*5 + 5*4 + 6*3 = \\ &= 6 + 12 + 20 + 20 + 18 = \\ &= 76 \end{aligned}$$

Calcolo di $O[3]$

Esempio di Convoluzione 1D

Descrizione

- **Input (I):** Array di 7 elementi ($I[0] \dots I[6]$).
- **Filtro (F):** Array di 5 elementi ($F[0] \dots F[4]$).
- **Output (O):** Array risultante dalla convoluzione di I con F.



raggio

$$\begin{aligned} O[2] &= I[0]*F[0] + I[1]*F[1] + I[2]*F[2] + I[3]*F[3] + I[4]*F[4] \\ &= 1*3 + 2*4 + 3*5 + 4*4 + 5*3 = \\ &= 3 + 8 + 15 + 16 + 15 = \\ &= 57 \end{aligned}$$

Calcolo di $O[2]$

Esempio di Convoluzione 1D

Descrizione

- Input (I):** Array di 7 elementi ($I[0] \dots I[6]$).
- Filtro (F):** Array di 5 elementi ($F[0] \dots F[4]$).
- Output (O):** Array risultante dalla convoluzione di I con F.

I	$I[0]$	$I[1]$	$I[2]$	$I[3]$	$I[4]$	$I[5]$	$I[6]$
0	1	2	3	4	5	6	7

Padding

F	$F[0]$	$F[1]$	$F[2]$	$F[3]$	$F[4]$
3	4	5	4	3	

raggio

O	$O[0]$	$O[1]$	$O[2]$	$O[3]$	$O[4]$	$O[5]$	$O[6]$
		38	57	76	95		

0	4	10	12	12
---	---	----	----	----

$$\begin{aligned} O[1] &= 0 * F[0] + I[0] * F[1] + I[1] * F[2] + I[2] * F[3] + I[3] * F[4] \\ &= 0 * 3 + 1 * 4 + 2 * 5 + 3 * 4 + 4 * 3 = \\ &= 0 + 4 + 10 + 12 + 12 = \\ &= 38 \end{aligned}$$

Calcolo di $O[1]$

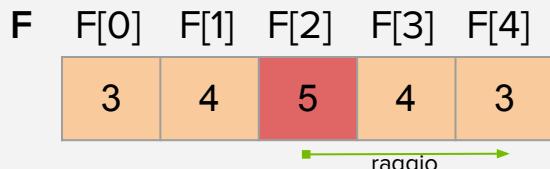
Esempio di Convoluzione 1D

Descrizione

- **Input (I):** Array di 7 elementi ($I[0] \dots I[6]$).
- **Filtro (F):** Array di 5 elementi ($F[0] \dots F[4]$).
- **Output (O):** Array risultante dalla convoluzione di I con F.

I	$I[0]$	$I[1]$	$I[2]$	$I[3]$	$I[4]$	$I[5]$	$I[6]$
	1	2	3	4	5	6	7

O	$O[0]$	$O[1]$	$O[2]$	$O[3]$	$O[4]$	$O[5]$	$O[6]$
	?	38	57	76	95		



$$O[0] = ?$$

Come calcolare $O[0]$?

Calcolo di $O[0]$

Perché la Convoluzione si Adatta al Calcolo Parallello

Indipendenza dei Calcoli

- Ogni elemento di output è calcolato indipendentemente.
- Permette l'elaborazione parallela.

Operazioni Uniformi

- Stesse operazioni ripetute su diverse porzioni dei dati.
- Si allinea con l'architettura SIMD.

Mapping Diretto Thread-Output

- Ogni thread può calcolare un elemento di output.
- Semplifica la parallelizzazione del problema.

Implementazione Generica: Passi

- Un thread GPU per ogni elemento di output.
- Ogni thread:
 - Identifica regione input corrispondente.
 - Applica il filtro e calcola risultato.
 - Scrive output.

Nota: Questa è un'implementazione "naive". Ottimizzazioni avanzate saranno trattate successivamente.

CUDA Convoluzione 1D: Soluzione (non ottimale)

1/2

```
int main() {    // Parametri fittizi
    const int W = 1000, filterSize = 3;    // Dimensioni input e filtro (dispari)
    float h_input[W];    // Input
    for (int i = 0; i < W; ++i) h_input[i] = static_cast<float>(i + 1);    // Inizializzazione
    float h_filter[filterSize] = {0.2f, 0.5f, 0.2f};    // Filtro fittizio
    float h_output[W];    // Output per i risultati
    float *d_input, *d_output, *d_filter;    // Puntatori su device

    // Allocazione memoria su GPU
    cudaMalloc(&d_input, W * sizeof(float));
    cudaMalloc(&d_output, W * sizeof(float));
    cudaMalloc(&d_filter, filterSize * sizeof(float));

    // Copia input e filtro dalla CPU alla GPU
    cudaMemcpy(d_input, h_input, W * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_filter, h_filter, filterSize * sizeof(float), cudaMemcpyHostToDevice);

    int blockSize = 256;    // Numero di thread per blocco
    int numBlocks = (W + blockSize - 1) / blockSize;    // Numero di blocchi

    // Lancio del kernel
    cudaConvolution1D <<< numBlocks, blockSize >>> (d_input, d_output, d_filter, W, filterSize);
}

// continue..
```

CUDA Convoluzione 1D: Soluzione (non ottimale)

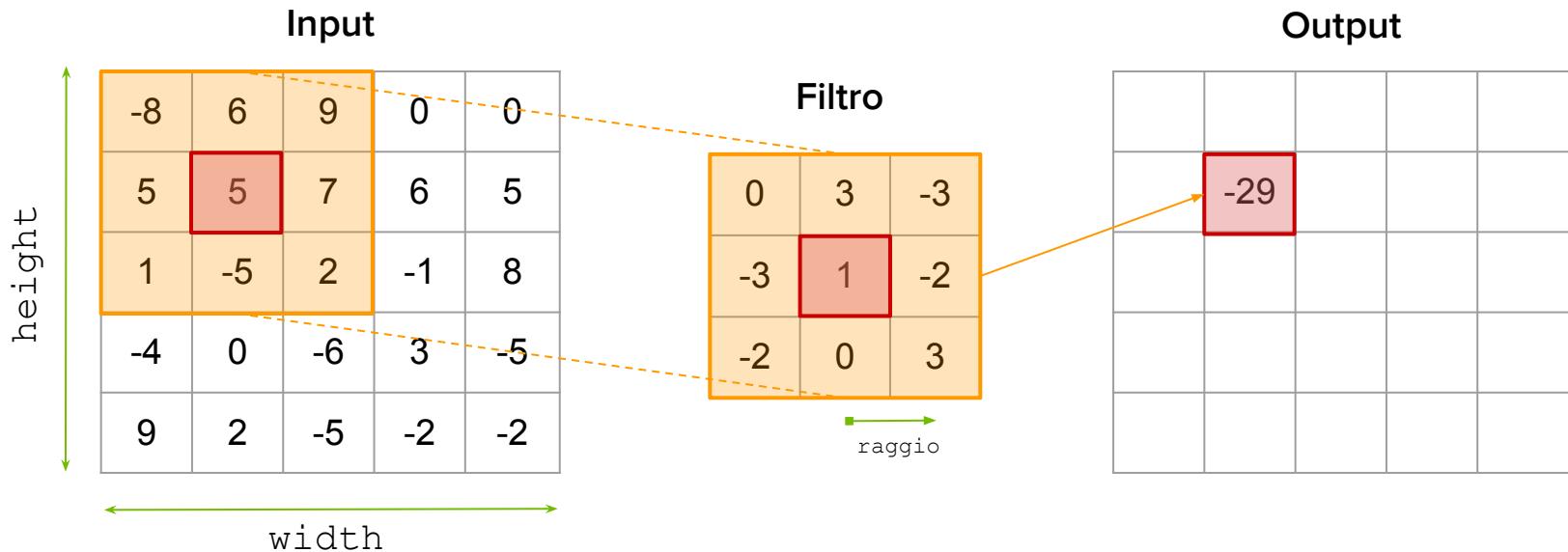
2/2

```
__global__ void cudaConvolution1D(float* input, float* output, float* filter, int W, int  
filterSize)  
{  
    int x = blockIdx.x * blockDim.x + threadIdx.x; // Indice globale del thread  
    int radius = filterSize / 2; // Raggio del filtro (supponiamo filterSize dispari)  
  
    if (x < W) // Verifica che il thread sia all'interno dei limiti dell'input  
    {  
        float result = 0.0f;  
        for (int i = -radius; i <= radius; i++)  
        {  
            int currentPos = x + i; // Posizione corrente nell'input  
            if (currentPos >= 0 && currentPos < W)  
            {  
                result += input[currentPos] * filter[i + radius]; // Applica il filtro  
            }  
        }  
        output[x] = result; // Salva il risultato  
    }  
}
```

Esempio di Convoluzione 2D

Descrizione

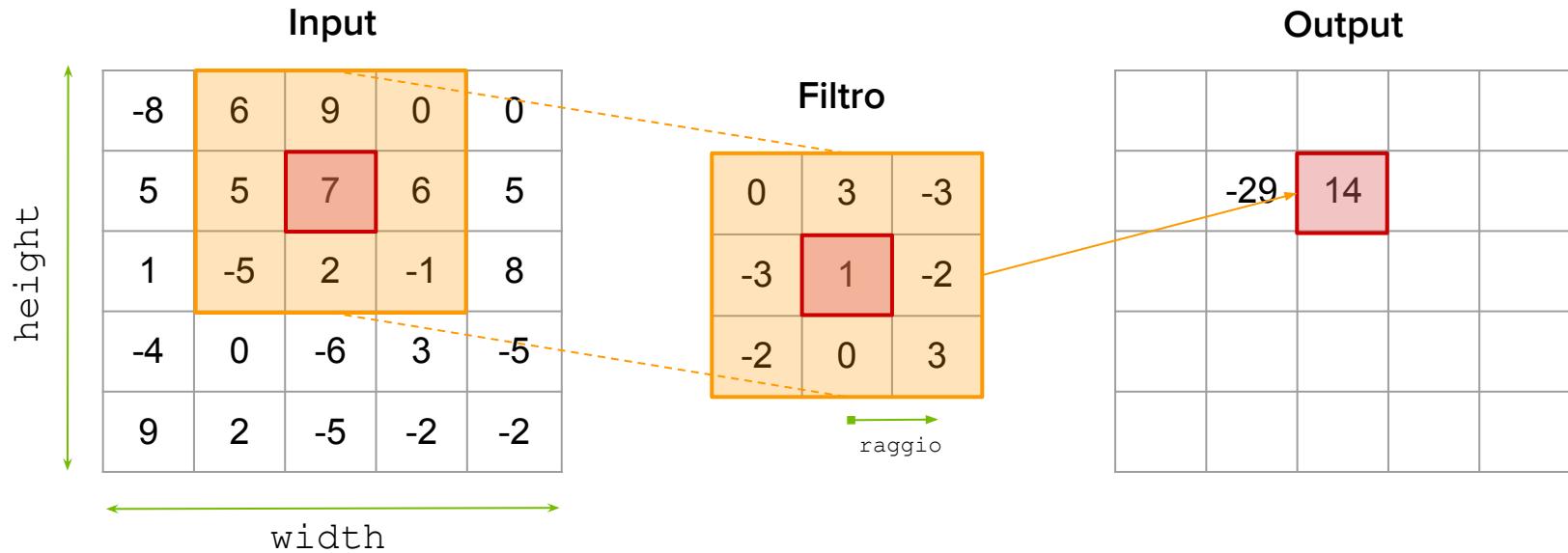
- **Input (I)**: Matrice di 25 elementi ($I[0,0] \dots I[4,4]$).
- **Filtro (F)**: Matrice di 9 elementi ($F[0,0] \dots F[2,2]$).
- **Output (O)**: Matrice risultante dalla convoluzione di I con F.



Esempio di Convoluzione 2D

Descrizione

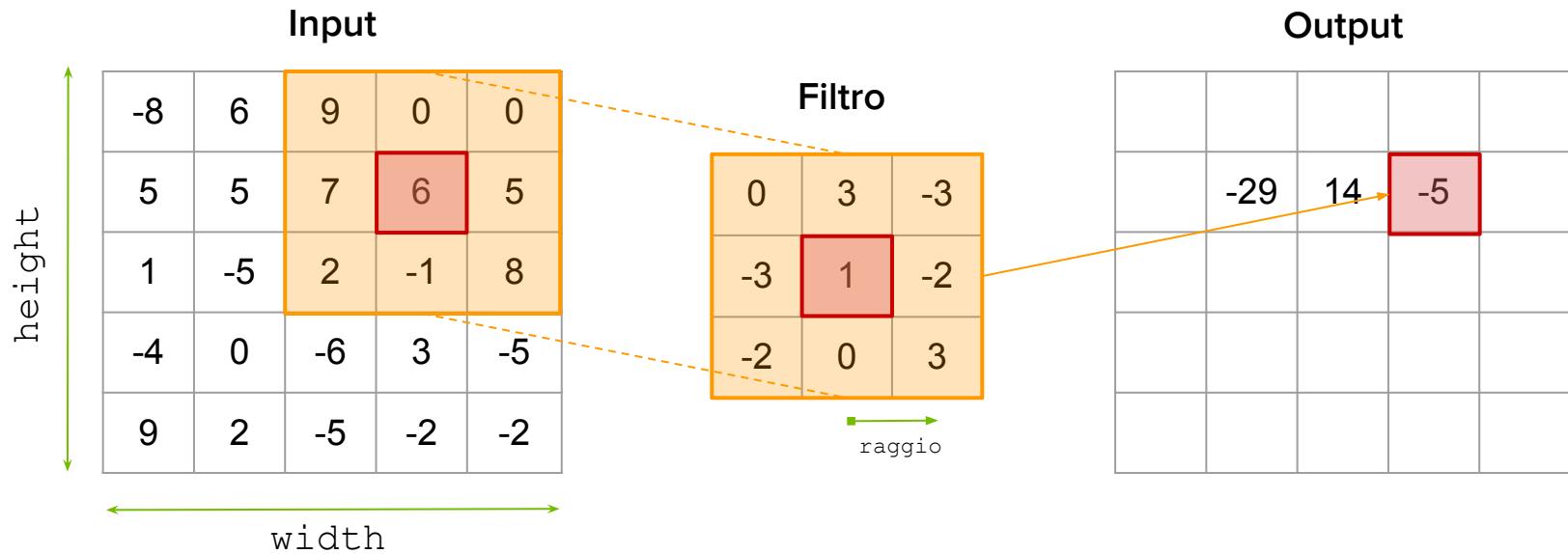
- **Input (I):** Matrice di 25 elementi ($I[0,0] \dots I[4,4]$).
- **Filtro (F):** Matrice di 9 elementi ($F[0,0] \dots F[2,2]$).
- **Output (O):** Matrice risultante dalla convoluzione di I con F.



Esempio di Convoluzione 2D

Descrizione

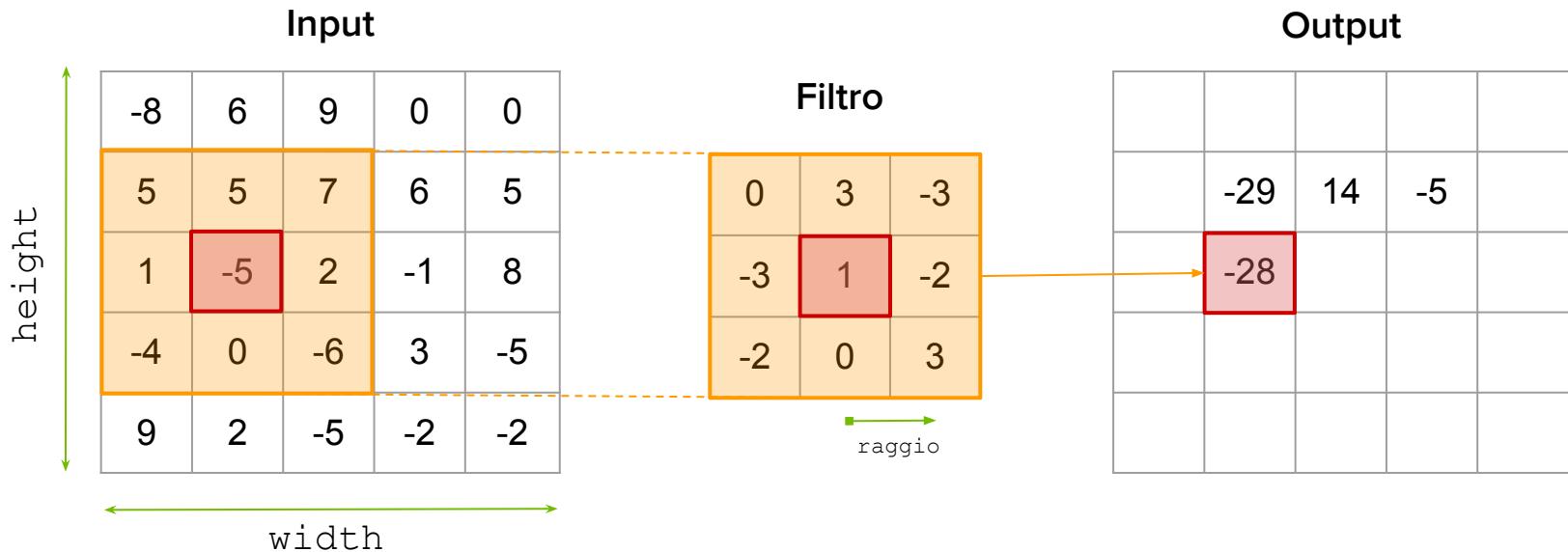
- **Input (I):** Matrice di 25 elementi ($I[0,0] \dots I[4,4]$).
- **Filtro (F):** Matrice di 9 elementi ($F[0,0] \dots F[2,2]$).
- **Output (O):** Matrice risultante dalla convoluzione di I con F.



Esempio di Convoluzione 2D

Descrizione

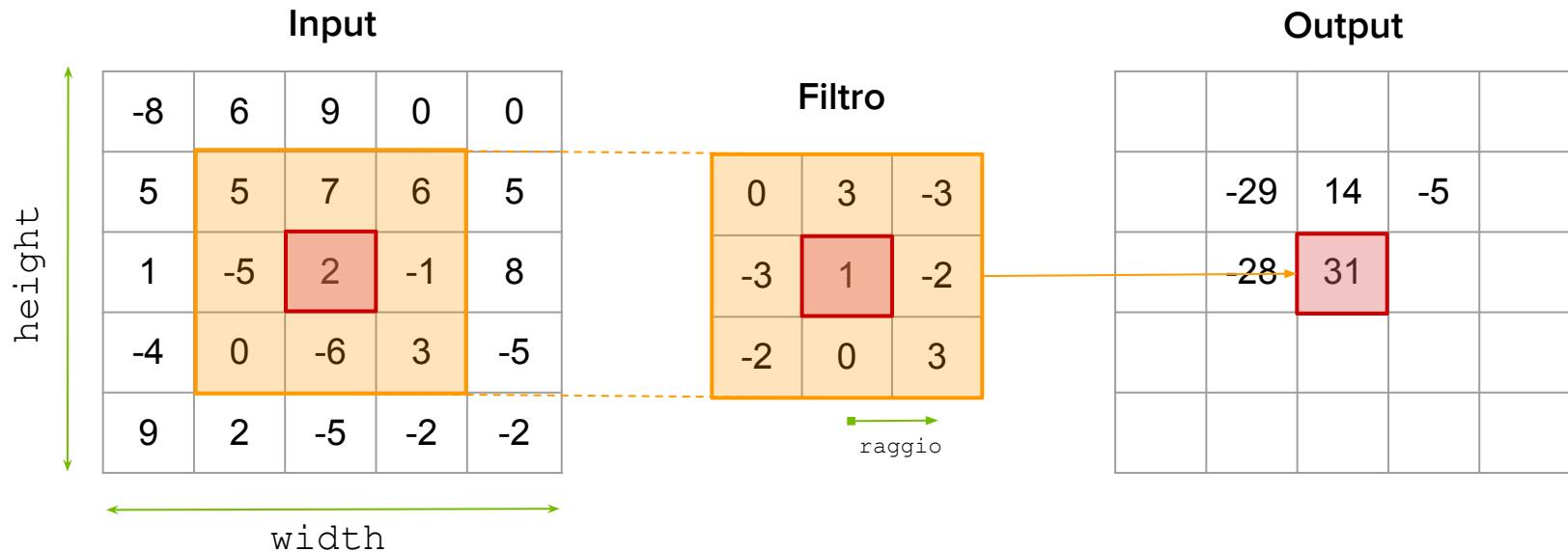
- **Input (I):** Matrice di 25 elementi ($I[0,0] \dots I[4,4]$).
- **Filtro (F):** Matrice di 9 elementi ($F[0,0] \dots F[2,2]$).
- **Output (O):** Matrice risultante dalla convoluzione di I con F.



Esempio di Convoluzione 2D

Descrizione

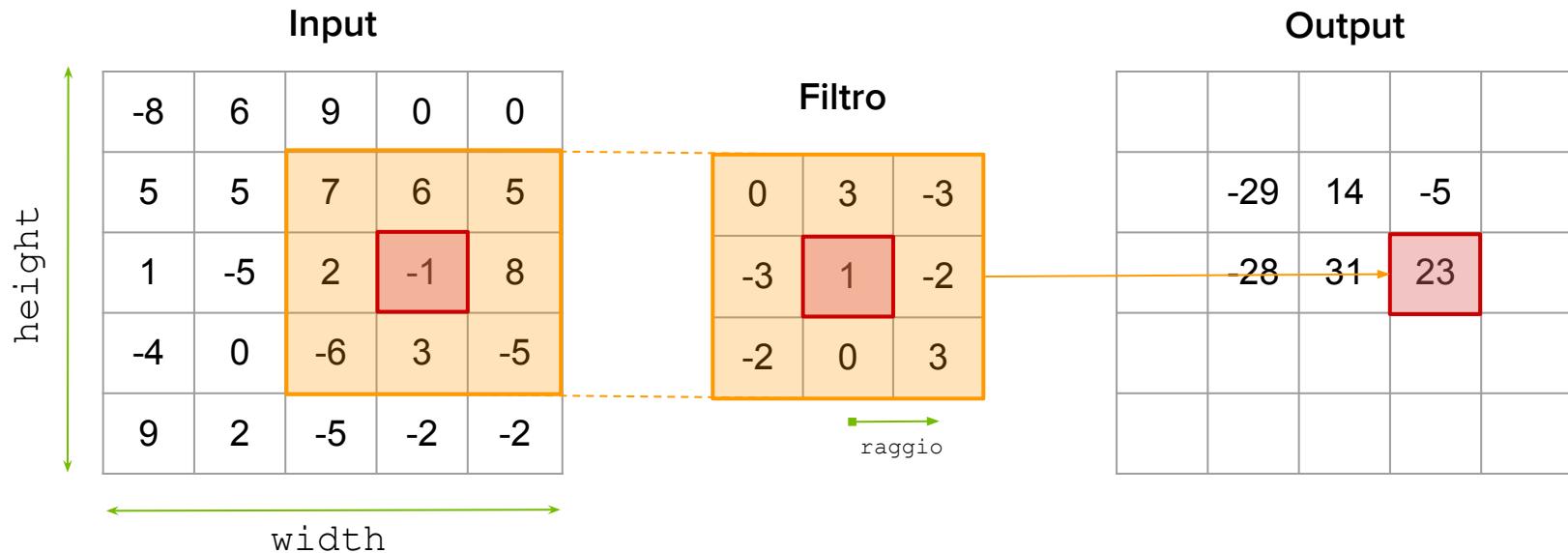
- **Input (I):** Matrice di 25 elementi ($I[0,0] \dots I[4,4]$).
- **Filtro (F):** Matrice di 9 elementi ($F[0,0] \dots F[2,2]$).
- **Output (O):** Matrice risultante dalla convoluzione di I con F.



Esempio di Convoluzione 2D

Descrizione

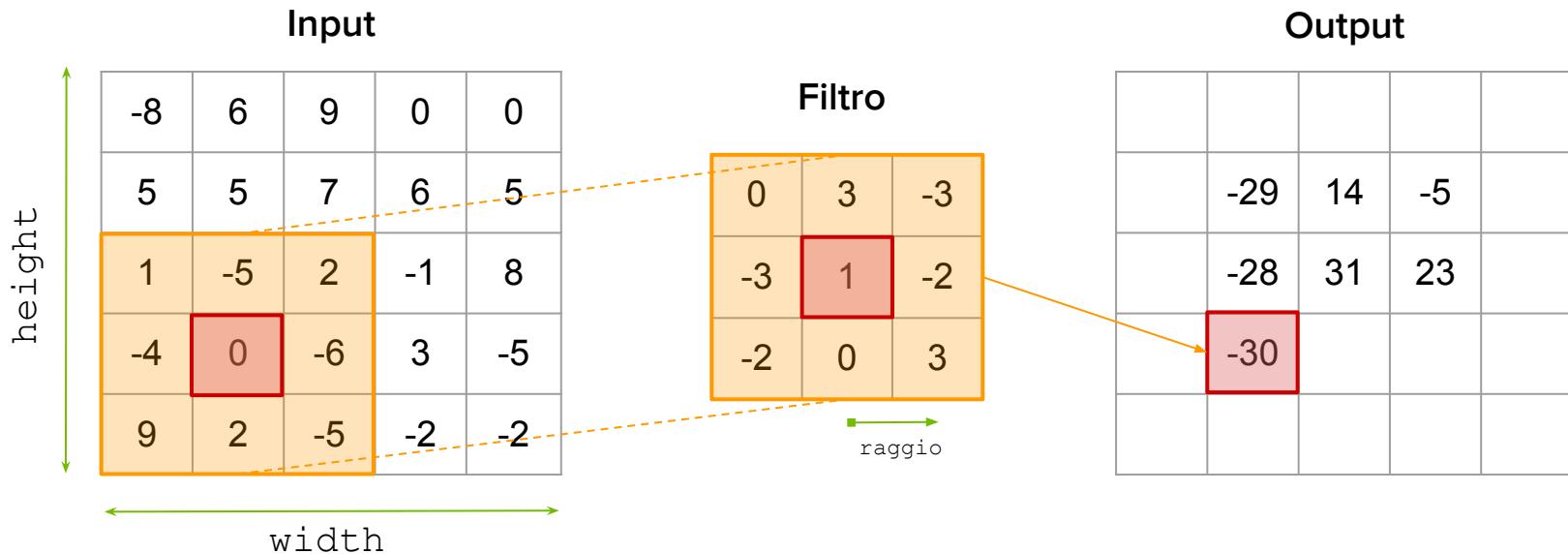
- **Input (I)**: Matrice di 25 elementi ($I[0,0] \dots I[4,4]$).
- **Filtro (F)**: Matrice di 9 elementi ($F[0,0] \dots F[2,2]$).
- **Output (O)**: Matrice risultante dalla convoluzione di I con F.



Esempio di Convoluzione 2D

Descrizione

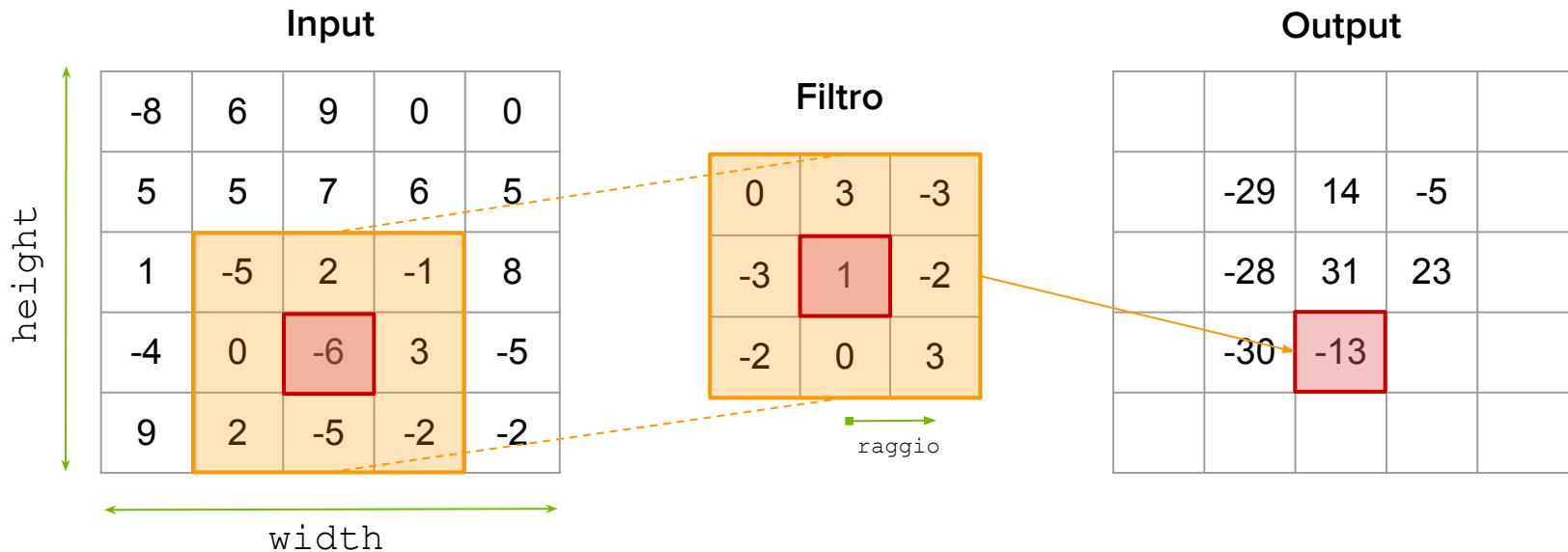
- **Input (I):** Matrice di 25 elementi ($I[0,0] \dots I[4,4]$).
- **Filtro (F):** Matrice di 9 elementi ($F[0,0] \dots F[2,2]$).
- **Output (O):** Matrice risultante dalla convoluzione di I con F.



Esempio di Convoluzione 2D

Descrizione

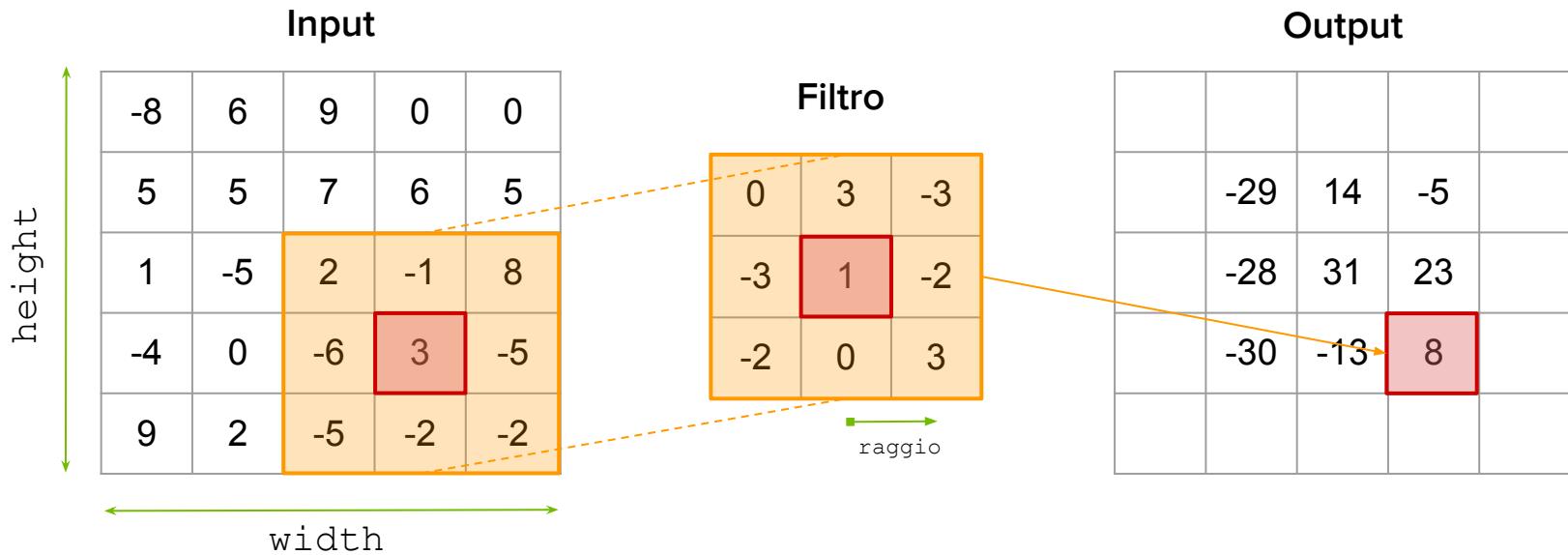
- **Input (I):** Matrice di 25 elementi ($I[0,0] \dots I[4,4]$).
- **Filtro (F):** Matrice di 9 elementi ($F[0,0] \dots F[2,2]$).
- **Output (O):** Matrice risultante dalla convoluzione di I con F.



Esempio di Convoluzione 2D

Descrizione

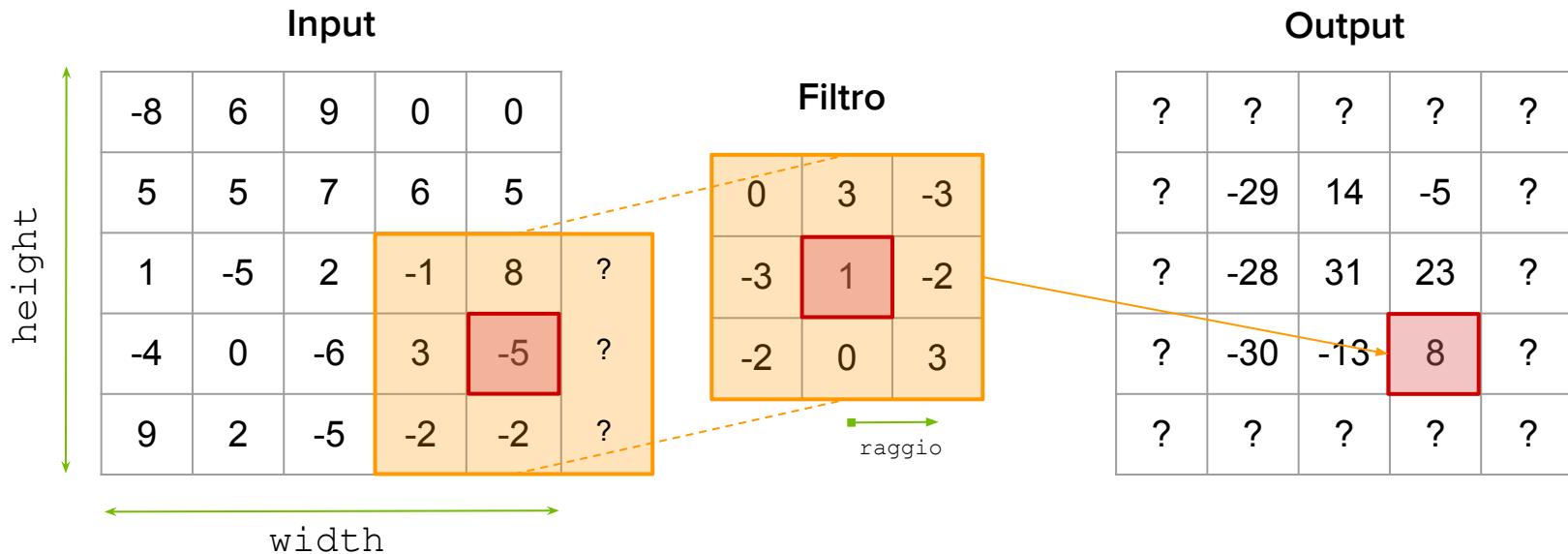
- **Input (I):** Matrice di 25 elementi ($I[0,0] \dots I[4,4]$).
- **Filtro (F):** Matrice di 9 elementi ($F[0,0] \dots F[2,2]$).
- **Output (O):** Matrice risultante dalla convoluzione di I con F.



Esempio di Convoluzione 2D

Descrizione

- **Input (I):** Matrice di 25 elementi ($I[0,0] \dots I[4,4]$).
- **Filtro (F):** Matrice di 9 elementi ($F[0,0] \dots F[2,2]$).
- **Output (O):** Matrice risultante dalla convoluzione di I con F.



Convoluzione 2D: Rilevazione dei Contorni con Sobel

Processo di Convoluzione 2D

- Il filtro di **Sobel** scorre sull'immagine pixel per pixel.
- Per ogni posizione:
 - Si **moltiplica** il filtro per la regione corrispondente dell'immagine.
 - Si **somma** i risultati per ottenere il valore del pixel di output.
- Evidenzia le **transizioni verticali** di intensità nell'immagine.

Filtro di Sobel
Verticale

-1	0	1
-2	0	2
-1	0	1



Input Image



Contorni Verticali

Convoluzione 2D: Rilevazione dei Contorni con Sobel

Processo di Convoluzione 2D

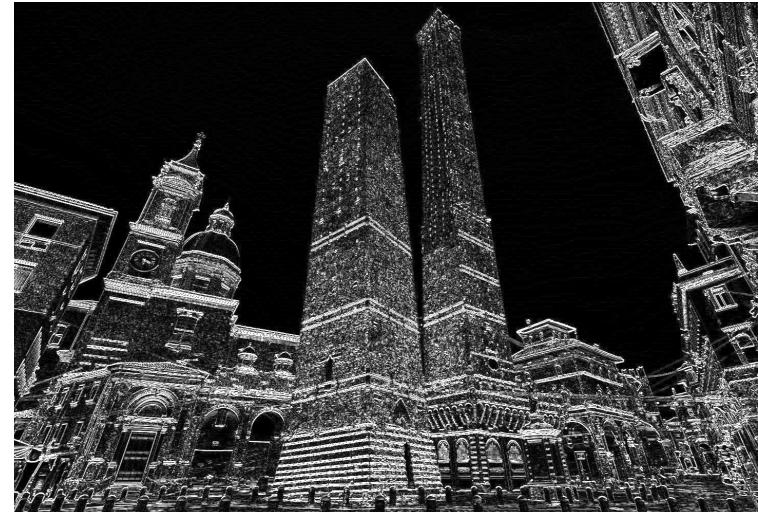
- Il filtro di **Sobel** scorre sull'immagine pixel per pixel
- Per ogni posizione:
 - Si **moltiplica** il filtro per la regione corrispondente dell'immagine.
 - Si **somma** i risultati per ottenere il valore del pixel di output.
- Evidenzia le **transizioni orizzontali** di intensità nell'immagine.

Filtro di Sobel
Orizzontale

-1	-2	-1
0	0	0
-1	-2	-1



Input Image



Contorni Orizzontali

CUDA Convoluzione 2D: Soluzione (non ottimale)

1/3

```
int main()
{
    // Parametri
    const int W, H, filterSize = 3;

    // Carica immagine usando stb_image.h
    // ..
    const int size = W * H * sizeof(float);
    const int filterBytes = filterSize * filterSize * sizeof(float);

    // Allocazione memoria host
    float* h_input = (float*)malloc(size);
    float* h_output = (float*)malloc(size);
    float* h_filter = (float*)malloc(filterBytes);

    // Inizializzazione filtro Sobel verticale (come esempio)
    float sobelY[9] = {-1.0f, -2.0f, -1.0f,
                        0.0f, 0.0f, 0.0f,
                        1.0f, 2.0f, 1.0f};
    memcpy(h_filter, sobelY, filterBytes);
```

CUDA Convoluzione 2D: Soluzione (non ottimale)

2/3

```
// Allocazione memoria device
float *d_input, *d_output, *d_filter;
cudaMalloc(&d_input, size);
cudaMalloc(&d_output, size);
cudaMalloc(&d_filter, filterBytes);

// Copia dati dall'host al device
cudaMemcpy(d_input, h_input, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_filter, h_filter, filterBytes, cudaMemcpyHostToDevice);

// Configurazione lancio kernel
dim3 blockSize(BLOCK_SIZE_X, BLOCK_SIZE_Y);
dim3 gridSize((W + BLOCK_SIZE_X - 1) / BLOCK_SIZE_X,
                (H + BLOCK_SIZE_Y - 1) / BLOCK_SIZE_Y);

// Lancio del kernel
cudaConvolution2D<<<gridSize, blockSize>>>(d_input, d_output, d_filter, W, H, filterSize);

// Copia risultato dal device all'host
cudaMemcpy(h_output, d_output, size, cudaMemcpyDeviceToHost);

// Continue..
```

CUDA Convoluzione 2D: Soluzione (non ottimale)

3/3

```
__global__ void cudaConvolution2D(float* input, float* output, float* filter,
                                  int W, int H, int filterSize){
    int x = blockIdx.x * blockDim.x + threadIdx.x; // Coordinata x globale del thread
    int y = blockIdx.y * blockDim.y + threadIdx.y; // Coordinata y globale del thread
    int radius = filterSize / 2; // Raggio del filtro

    if (x < W && y < H) {
        float result = 0.0f;
        for (int i = -radius; i <= radius; i++) {
            for (int j = -radius; j <= radius; j++) {
                int currentPosX = x + j; // Posizione x corrente nell'input
                int currentPosY = y + i; // Posizione y corrente nell'input

                if (currentPosX >= 0 && currentPosX < W &&
                    currentPosY >= 0 && currentPosY < H) {
                    int inputIdx = currentPosY * W + currentPosX; // Indice dell'input
                    int filterIdx = (i + radius) * filterSize + (j + radius); // Indice del filtro
                    result += input[inputIdx] * filter[filterIdx]; // Applica il filtro
                }
            }
        }
        output[y * W + x] = result; // Salva il risultato
    }
}
```

Effetti della Convoluzione 2D con Kernel Differenti

- La convoluzione 2D, applicata con kernel diversi, **estrae informazioni specifiche** dall'immagine, generando effetti differenti come il rilevamento dei bordi, la sfocatura o l'evidenziazione dei dettagli.



Input Image



Canny Edge Detection



Sharpening



Filtro Bilaterale



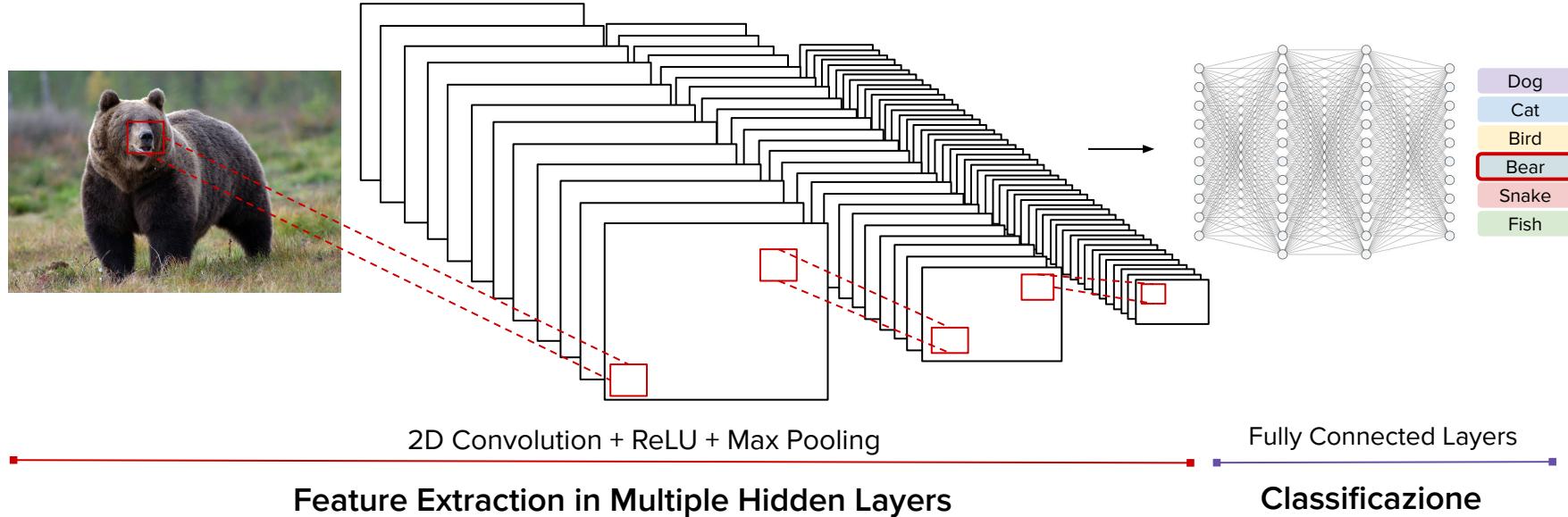
Image Blur



Prewitt

Convoluzioni 2D nelle Deep Neural Networks

- Le reti neurali convoluzionali (CNN) eseguono milioni di operazioni su matrici (convoluzioni 2D) per estrarre **feature** (apprese durante l'addestramento delle reti) dalle immagini.
- Ogni operazione su pixel e feature può essere eseguita in **parallelo**, accelerando il calcolo.
- Le GPU processano queste operazioni simultaneamente, permettendo applicazioni come il riconoscimento di immagini, ricostruzione 3D, segmentazione semantica e molto altro.



Riferimenti Bibliografici

Testi Generali

- Cheng, J., Grossman, M., McKercher, T. (2014). **Professional CUDA C Programming**. Wrox Pr Inc. (1^a edizione)
- Kirk, D. B., Hwu, W. W. (2013). **Programming Massively Parallel Processors**. Morgan Kaufmann (3^a edizione)

NVIDIA Docs

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

Risorse Online

- Corso GPU Computing (Prof. G. Grossi): Dipartimento di Informatica, Università degli Studi di Milano
 - <http://gpu.di.unimi.it/lezioni.html>