Modello di Programmazione CUDA

Sistemi di Elaborazione Accelerata, Modulo 2

A.A. 2025/2026

Fabio Tosi, Università di Bologna

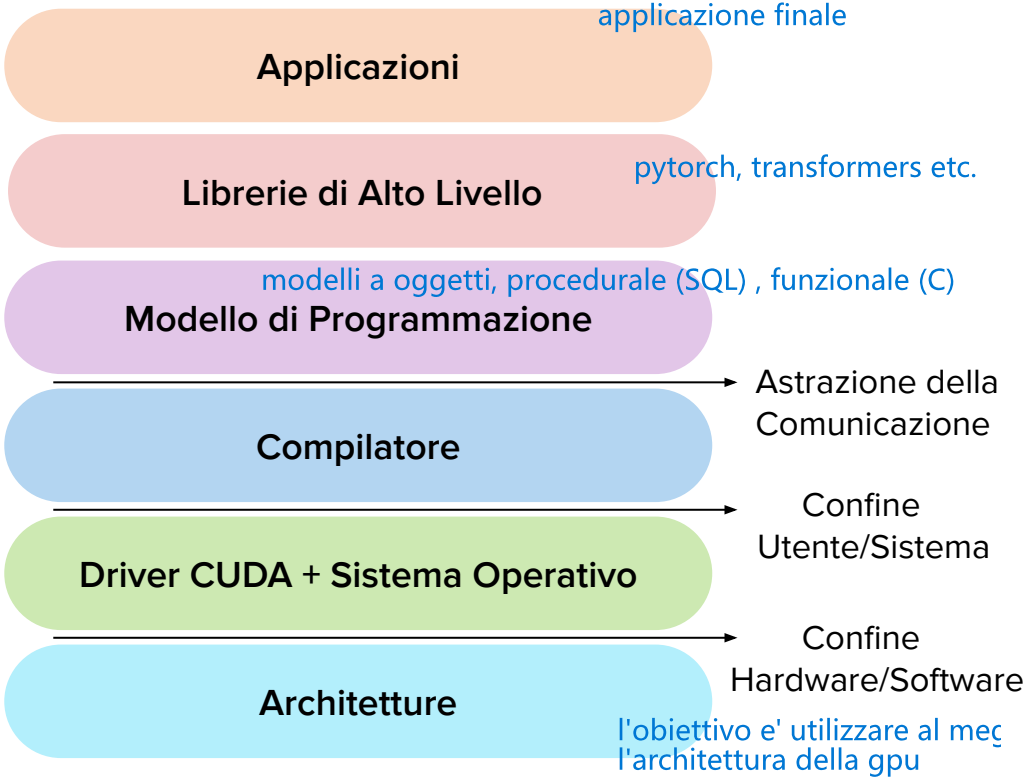
Panoramica del Modello di Programmazione CUDA

- **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

- Ecosistema stratificato per algoritmi paralleli su GPU, con semplicità e controllo hardware ottimizzati.



Applicazioni: Programmi scritti dagli sviluppatori per risolvere problemi specifici utilizzando CUDA.

Librerie: Raccolte di funzioni ottimizzate (es. cuBLAS, cuDNN) che semplificano lo sviluppo.

Modello di Programmazione: CUDA fornisce un'astrazione per la programmazione GPU, offrendo concetti come thread, blocchi e griglie.

Compilatore: Strumenti (`nvcc`) che traducono il codice in istruzioni GPU eseguibili.

Driver CUDA + Sistema Operativo: Il sistema operativo gestisce le risorse; il driver CUDA traduce le chiamate CUDA in comandi per la GPU.

Architetture: Le specifiche GPU NVIDIA su cui il codice CUDA viene eseguito, con diverse capacità e caratteristiche.

Ruolo del Modello e del Programma

Il Modello di Programmazione: insieme di strumenti messi a disposizione da cuda

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: codice

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 Parallela 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

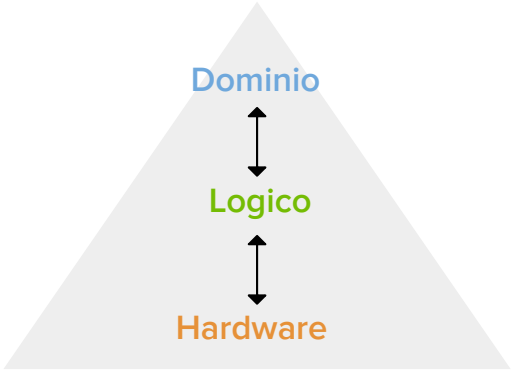
- Organizzazione e gestione dei thread.
- Implementazione della strategia di parallelizzazione.

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.

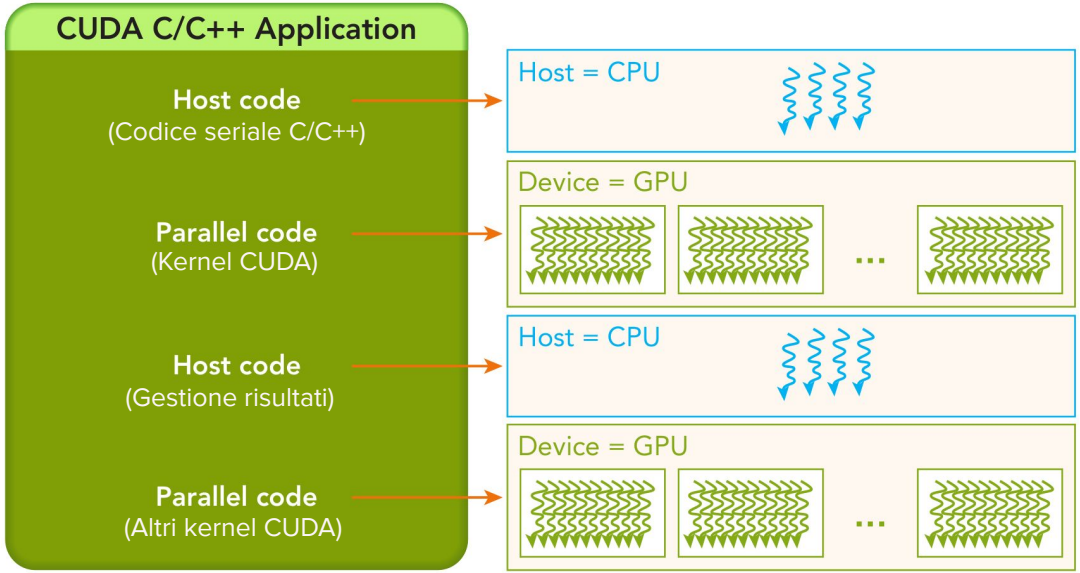
Cosa Fa un Thread CUDA?

- **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 (migliaia di core leggeri), basso overhead di gestione.
- **CPU**: parallelismo limitato (pochi core complessi), overhead più elevato.

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 (host) e sulla GPU (device).
- **Esecuzione Asincrona**: Il codice host può continuare l'esecuzione mentre i kernel GPU sono in esecuzione.
- **Kernel CUDA Multipli:** Possibilità di lanciare più kernel nella stessa applicazione, anche in overlapping temporale.
- **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)**
   - Preparazione dati e allocazione di memoria su CPU (host) e GPU (device).
2. **Trasferimento Dati (Host → Device)**
   - Copia degli input dalla memoria host alla memoria device.
3. **Esecuzione del Kernel (Device)**
   - La GPU esegue calcoli paralleli secondo la configurazione di griglia e blocchi.
4. **Recupero Risultati (Device → Host)**
   - Copia dell'output dalla memoria device alla memoria host.
5. **Post-elaborazione (Host)**
   - Analisi o elaborazione aggiuntiva dei risultati sulla CPU.
6. **Liberazione Risorse**
   - Rilascio della memoria allocata su host e device.

*__Nota__: i passi 2–5 possono essere ripetuti più volte o eseguiti in pipeline tramite stream per massimizzare l'overlap tra calcolo e trasferimento dati.

Panoramica del Modello di Programmazione CUDA

- 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

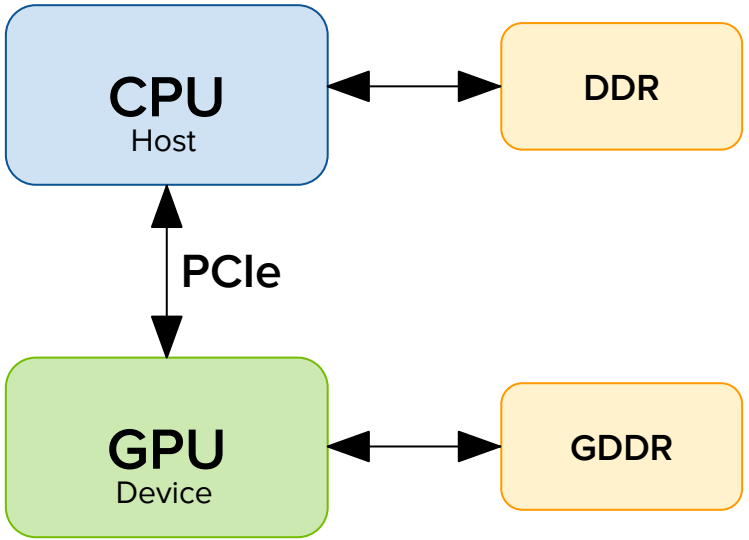
Gestione della Memoria in CUDA

Modello di Memoria CUDA

- Il modello prevede un sistema con **host** (CPU) e **device** (GPU), ciascuno con la propria memoria.
- La **comunicazione** tra memoria host e device avviene tramite **PCIe (Peripheral Component Interconnect Express)**, interfaccia seriale point-to-point che sfrutta più lane indipendenti in parallelo per aumentare la banda.

Caratteristiche PCIe

- **Lane**: Ogni lane (canale di trasmissione) è costituito da due coppie di segnali differenziali (quattro fili), una per ricevere (RX) e una per trasmettere (TX) 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 trasferimenti rapidi, adatti a scambi frequenti.
- **Collo di Bottiglia**: Può diventare un collo di bottiglia in trasferimenti di grandi volumi tra CPU e GPU.



Gestione della Memoria in CUDA

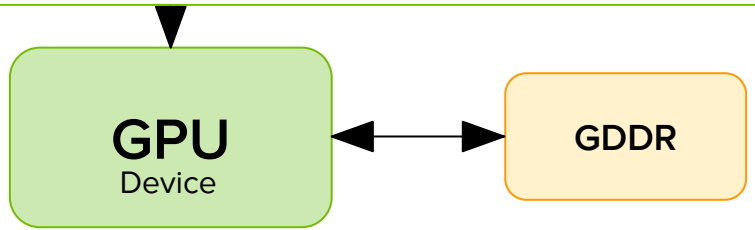
Modello di Memoria CUDA

- Il modello prevede un sistema con **host** (CPU) e **device** (GPU), ciascuno con la propria memoria.
- La **comunicazione** tra memoria host e device avviene tramite **PCIe (Peripheral Component Interconnect**

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)	PCIe 7.0 (2025)
x1	250 MB/s	500 MB/s	1 GB/s	2 GB/s	4 GB/s	8 GB/s	16 GB/s
x2	500 MB/s	1 GB/s	2 GB/s	4 GB/s	8 GB/s	16 GB/s	32 GB/s
x4	1 GB/s	2 GB/s	4 GB/s	8 GB/s	16 GB/s	32 GB/s	64 GB/s
x8	2 GB/s	4 GB/s	8 GB/s	16 GB/s	32 GB/s	64 GB/s	128 GB/s
x16	**4 GB/s**	**8 GB/s**	**16 GB/s**	**32 GB/s**	**64 GB/s**	**128 GB/s**	**256 GB/s**

- **Bassa Latenza:** Garantisce trasferimenti rapidi, adatti a scambi frequenti.
- **Collo di Bottiglia**: Può diventare un collo di bottiglia in trasferimenti di grandi volumi tra CPU e GPU.

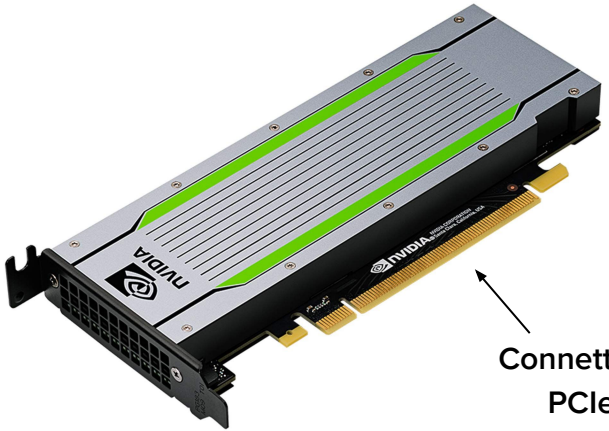


Collegamento Fisico della GPU tramite PCIe

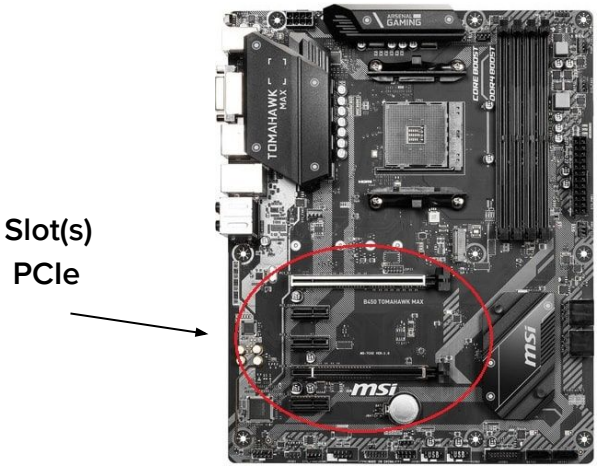
Connessione Fisica GPU

- La GPU si collega alla scheda madre attraverso uno **slot PCI Express** (**PCIe**).
- Il connettore, costituito da contatti metallici dorati sul bordo della scheda, si inserisce nello slot PCIe corrispondente.
- La maggior parte delle schede madri moderne ha **uno o più slot PCIe**, generalmente 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`	**Alloca** 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

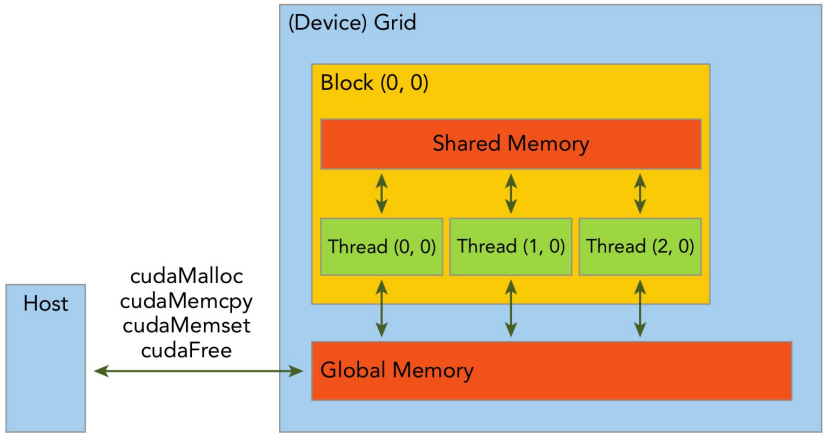
Shared Memory

- 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 (`cudaMemcpyKind`).

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

- **cud**
  all'ir

Firma della

```
cudaEr
```

Parametri

- **dst**
- **src**
- **cou**
- **kin**

Valore di R

- **cud**

o

vice

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**: Accesso a indirizzi non validi ➜ possibile blocco o crash dell'applicazione.
- **Soluzione:** La Unified Memory, introdotta in CUDA 6 e oggi ottimizzata, consente di usare un unico puntatore valido per CPU e GPU, con gestione automatica della migrazione dati (vedremo in seguito).

Note importa

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

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 è sincrona e può avere overhead significativo; 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 Modello di Programmazione CUDA

- ➤ 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

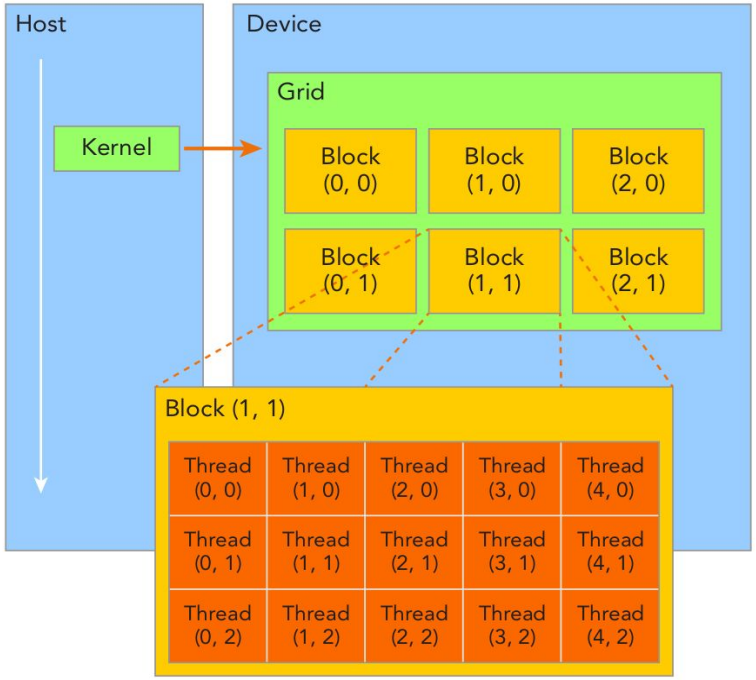
- CUDA adotta una **gerarchia a due livelli** per organizzare i thread basata su **blocchi di thread** e **griglie di blocchi**.

Struttura Gerarchica

1. **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.
2. **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 sincronizzarsi** direttamente (solo tramite memoria globale o kernel successivi)

Thread

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



(Thread ∈ Block ∈ Grid)

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 **definire le dimensioni** 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 possono condividere 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

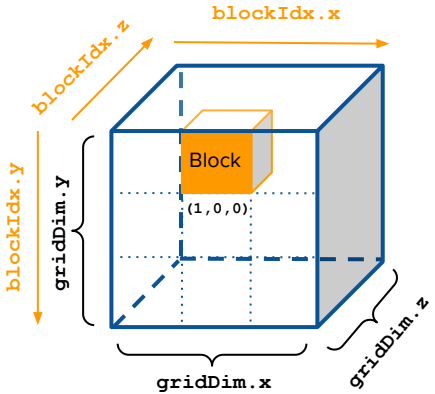
- La gerarchia è **scalabile** e permette di **adattare l'esecuzione** a GPU con diverse capacità e numero di core. Il codice CUDA, quindi, risulta più **portabile** e può essere eseguito su diverse architetture GPU.

➤ Sincronizzazione Granulare

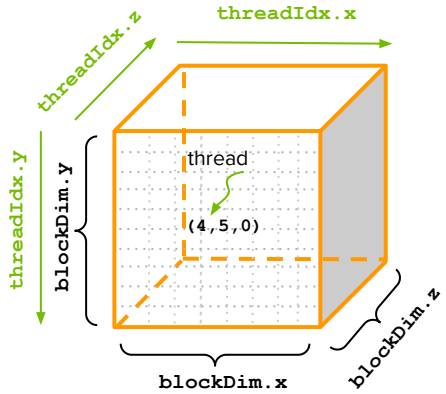
- I thread possono essere sincronizzati solo **all'interno del proprio blocco**, evitando costose sincronizzazioni globali che possono creare colli di bottiglia.

Identificazione dei Thread in CUDA

- **Ogni thread** ha un'identità unica definita da **coordinate** specifiche nella gerarchia grid-block. Tali coordinate, **diverse per ogni thread**, sono essenziali per calcolare indici di lavoro e accedere correttamente ai dati.



Grid



Block

Un singolo thread di calcolo che opera in maniera indipendente

Thread

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

Variabili di Identificazione (Coordinate)

1. **blockIdx** (indice del blocco all'interno della griglia)
   - 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**.

Variabili di Dimensioni

1. **blockDim** (dimensione del blocco in termini di thread)
   - Tipo: `dim3` (lato host), `uint3` (lato device, built-in)
   - 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`**

Identificazione dei Thread in CUDA

- **Ogni thread** ha un'identità unica definita da **coordinate** specifiche nella gerarchia grid-block. Tali coordinate, **diverse pe** …dati.

Dimensione delle Griglie e dei Blocchi

- La scelta delle dimensioni ottimali dipende dalla struttura dati del problema e dalle capacità hardware/risorse della GPU.
- Le dimensioni di griglia e blocchi vengono definite nel codice host **prima del lancio del 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** (1D, 2D, 3D per griglia e blocco) in tutto anche se in genere si usa la stessa per entrambi.

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

Variabili di Identi…

1. **blockIdx** (indice …read)
   - Compo… …e, built-in)
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**.

   - 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:

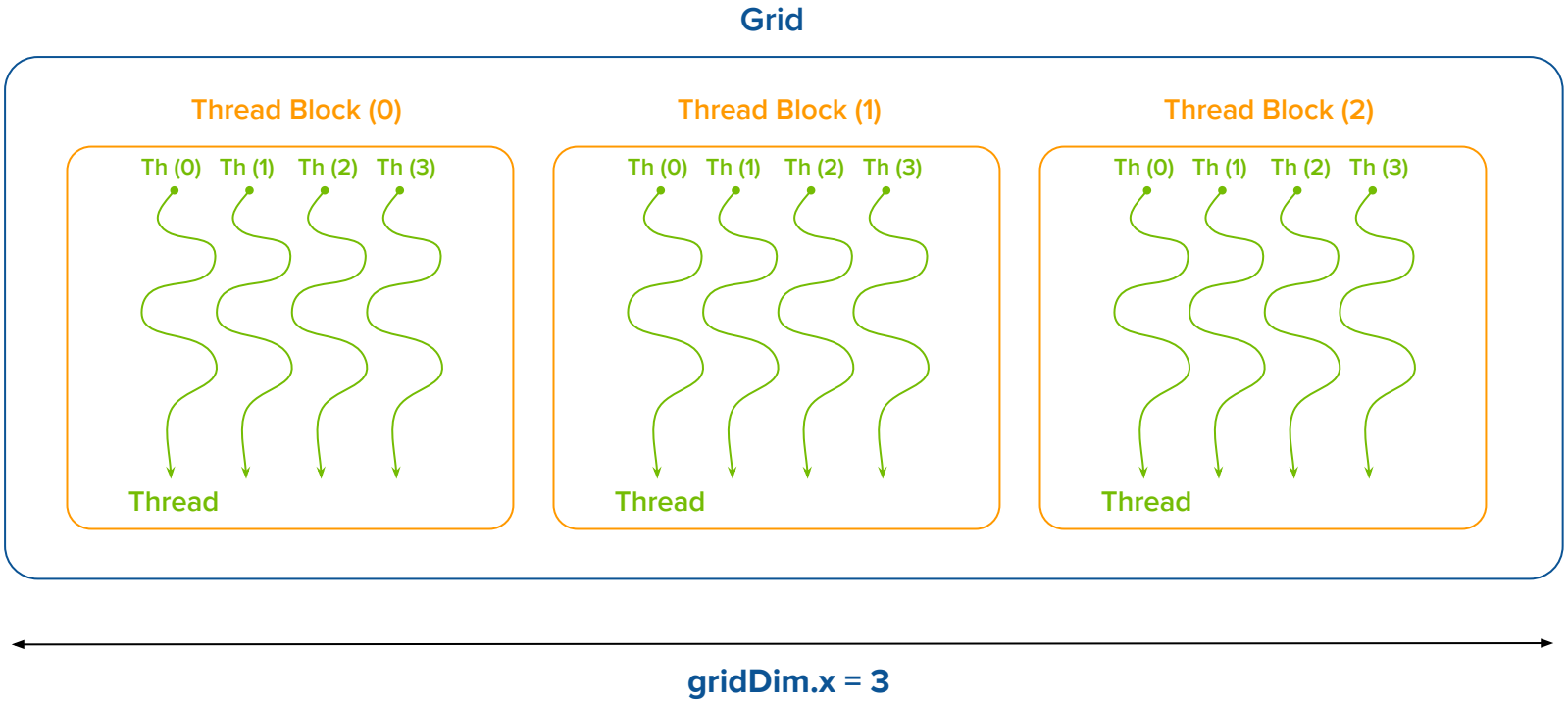
```
kernel_name<<<gridDim, blockDim>>>(...);
```

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 */
};
```

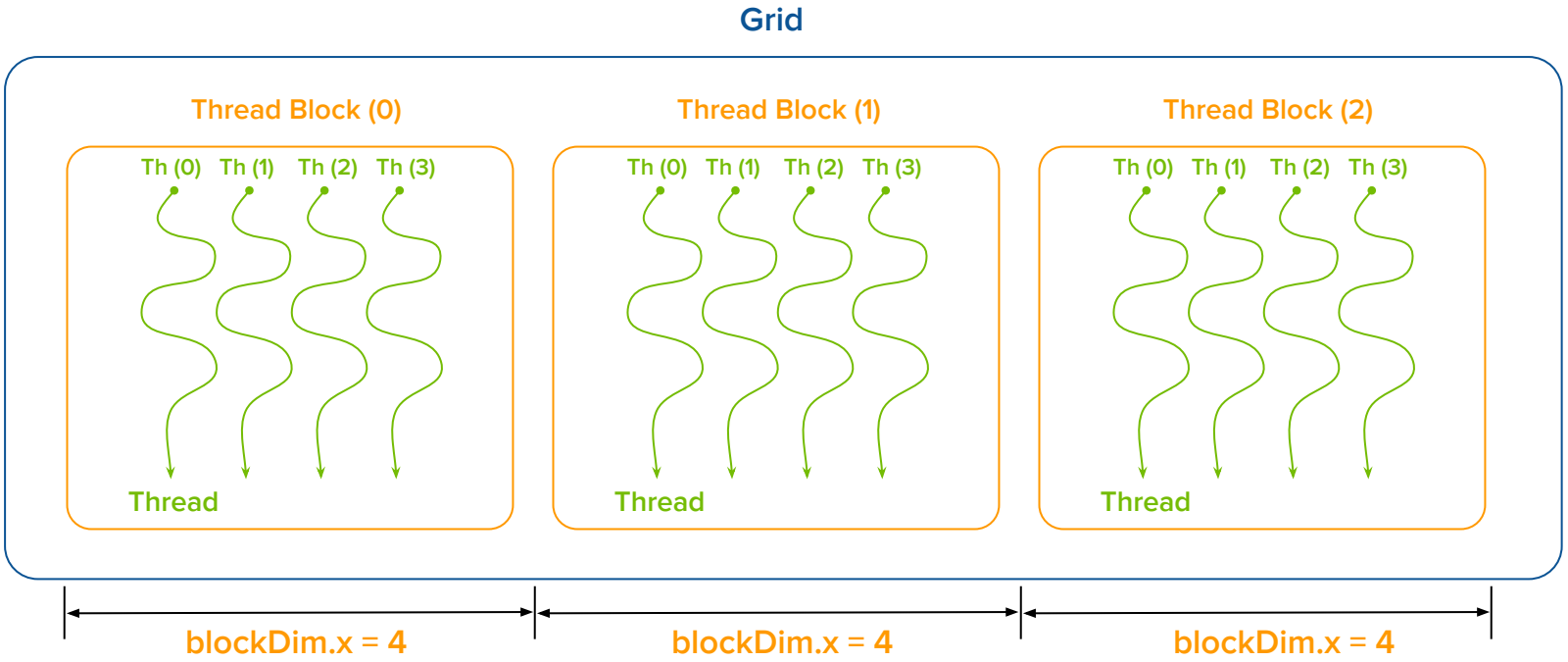
Identificazione dei Thread: Esempio Grid 1D, Block 1D

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



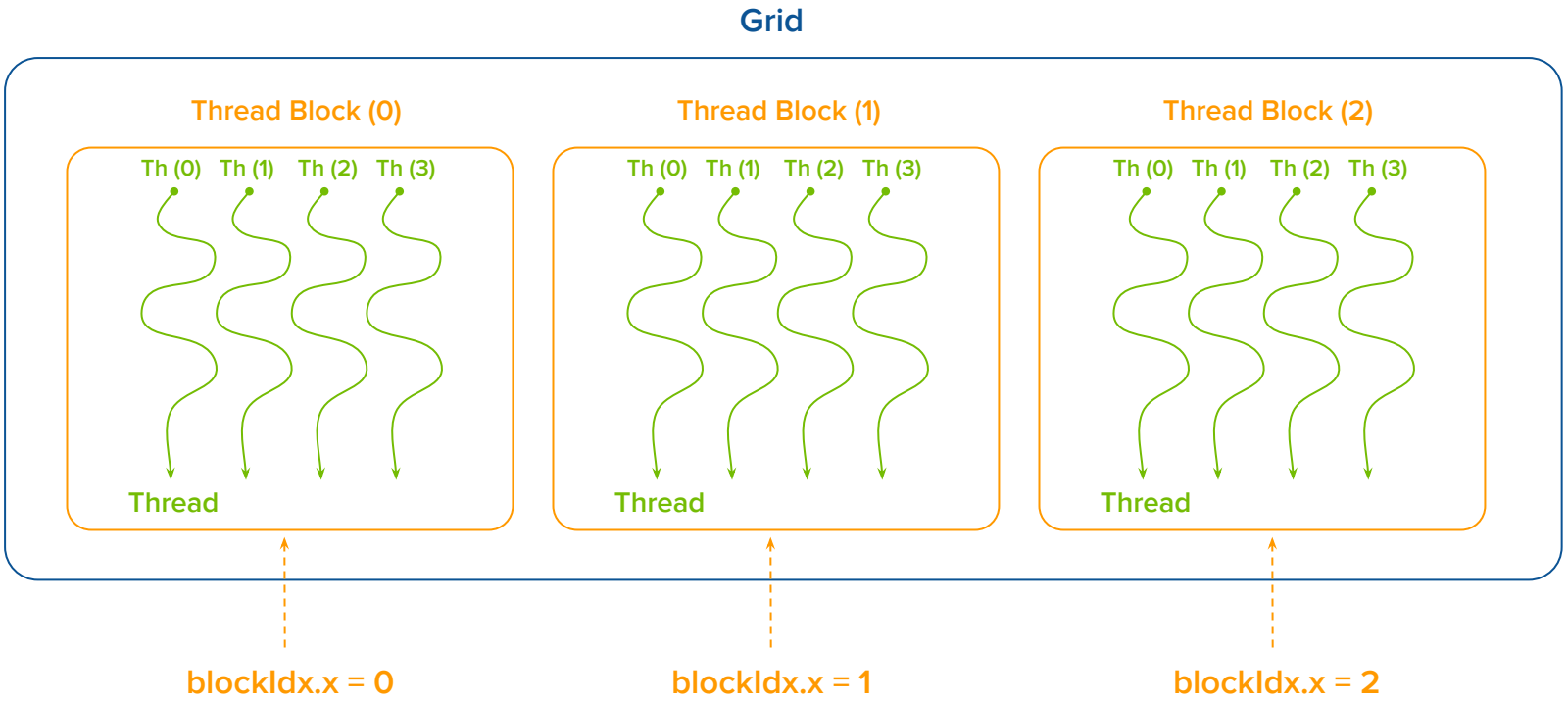
Identificazione dei Thread: Esempio Grid 1D, Block 1D

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



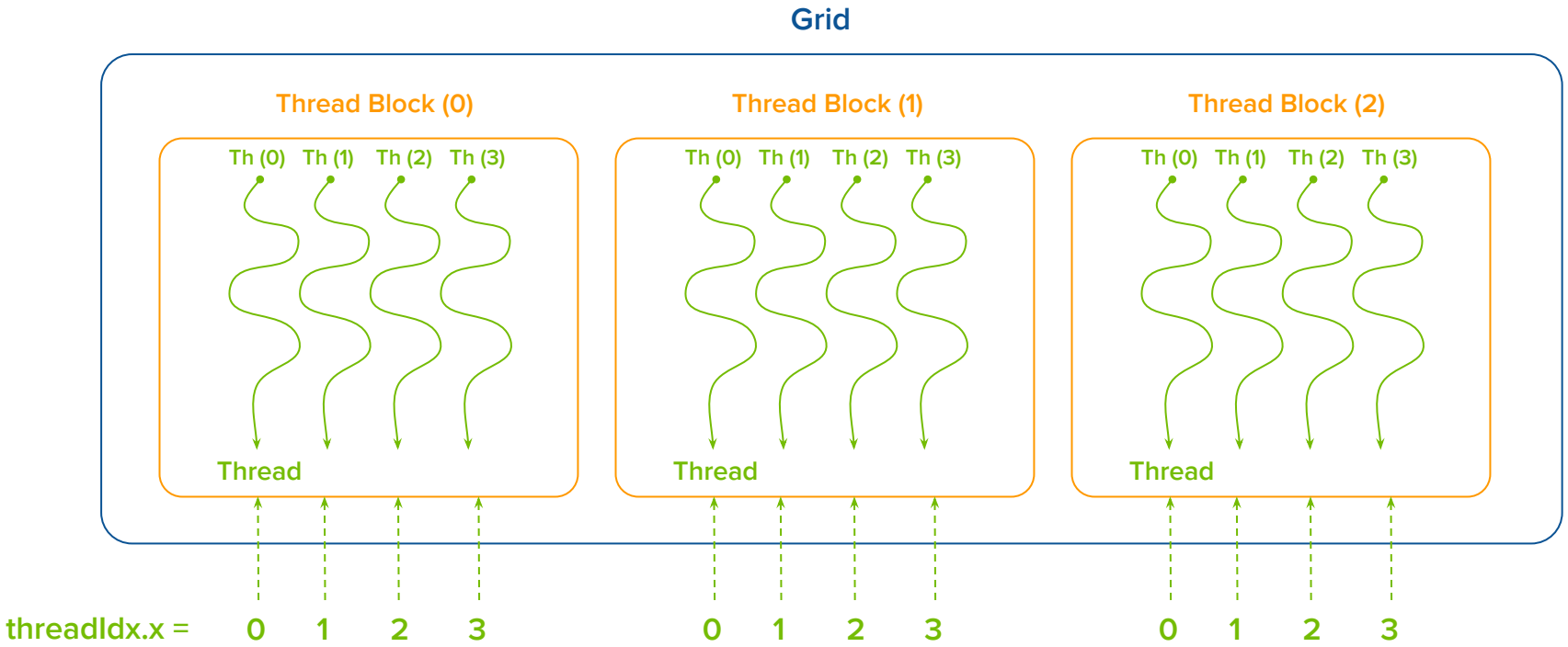
Identificazione dei Thread: Esempio Grid 1D, Block 1D

blockIdx.x: Indice di un blocco nella griglia.



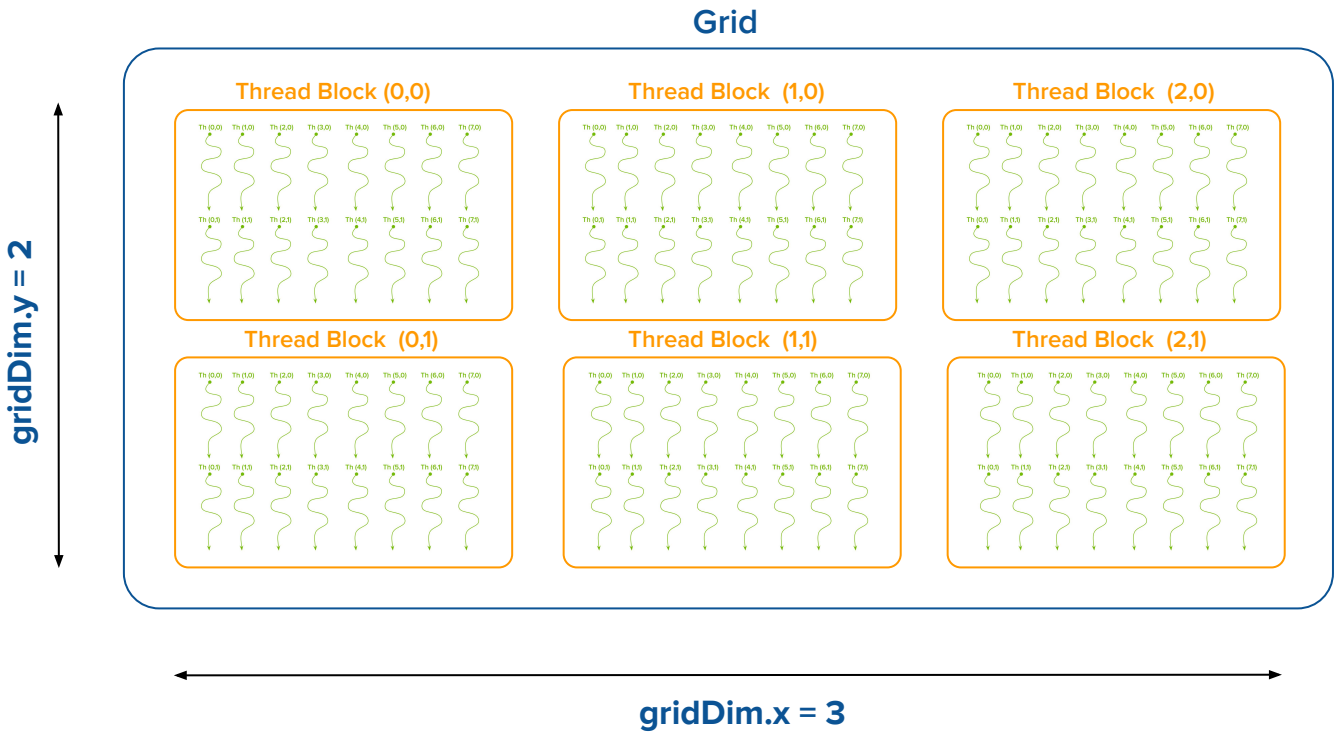
Identificazione dei Thread: Esempio Grid 1D, Block 1D

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



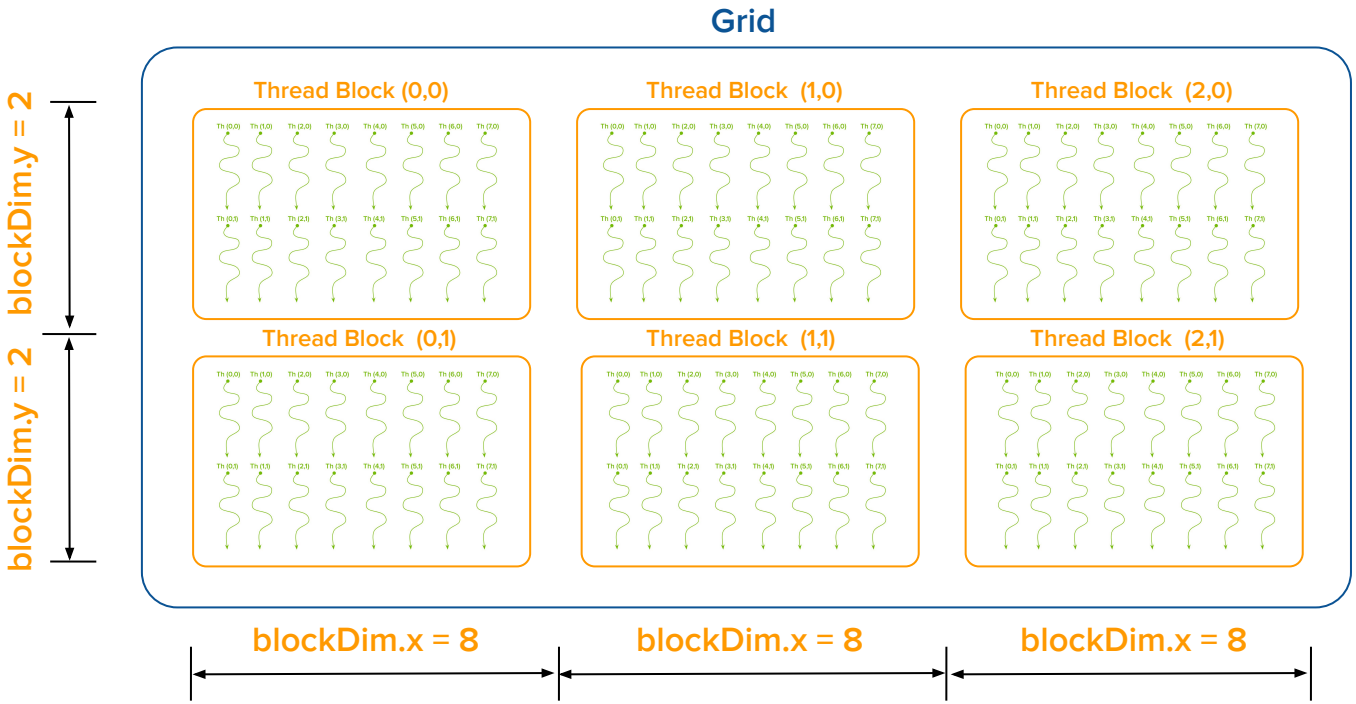
Identificazione dei Thread: Esempio Grid 2D, Block 2D

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



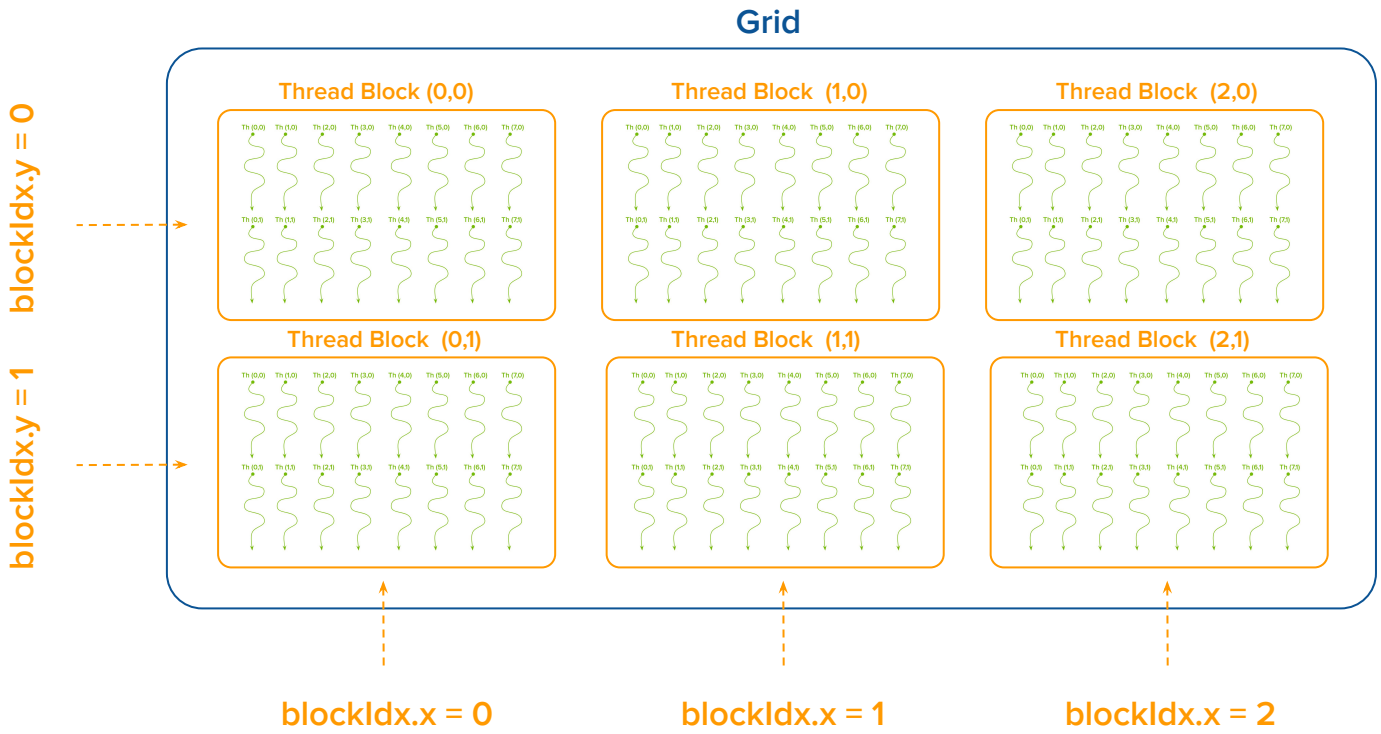
Identificazione dei Thread: Esempio Grid 2D, Block 2D

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



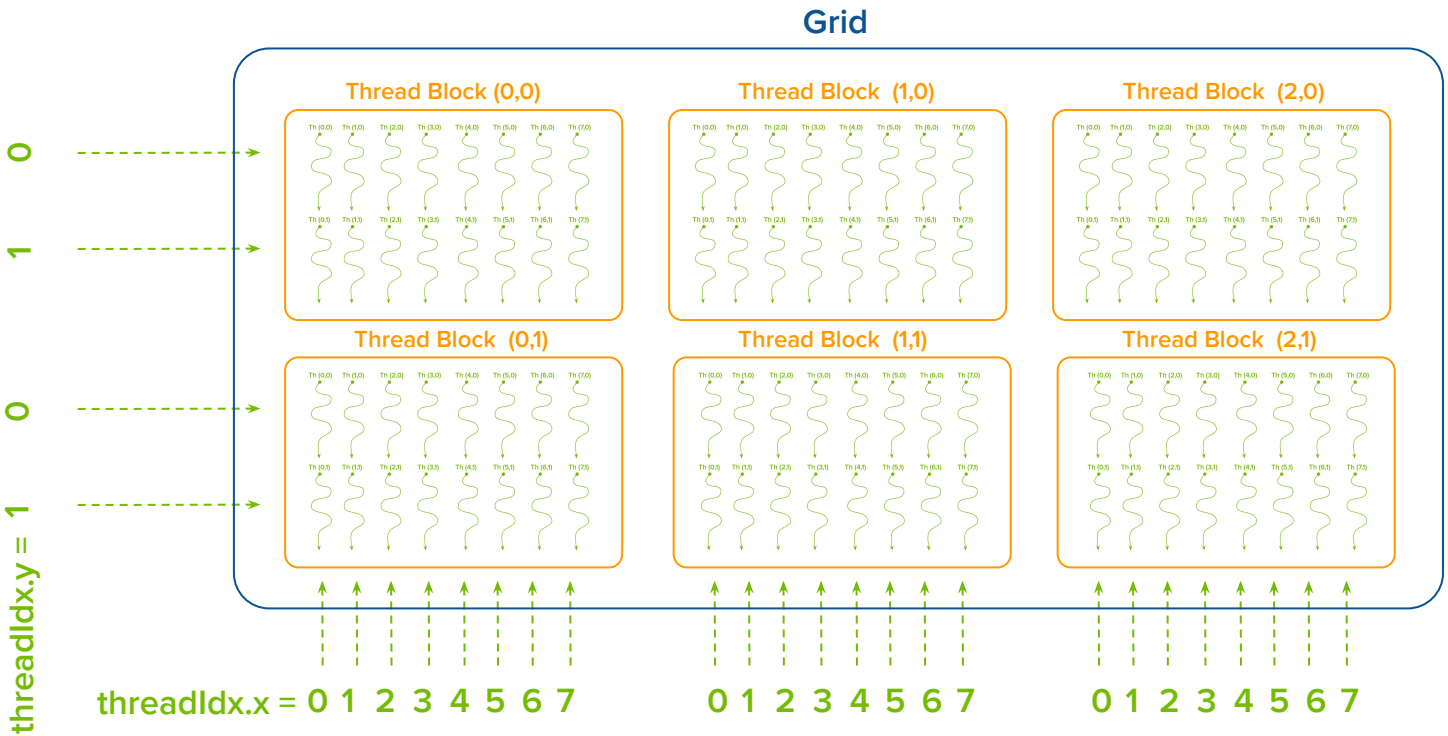
Identificazione dei Thread: Esempio Grid 2D, Block 2D

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 Modello di Programmazione CUDA

- 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
`__global__`	Sul Device	Dall'Host	Deve avere tipo di ritorno `void`
`__device__`	Sul Device	Solo dal Device	
`__host__`	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.

Kernel CUDA: Regole e Comportamento

1. **Esclusivamente Memoria Device (`__global__` e `__device__`)**
   - Accesso consentito solo alla memoria della GPU. Niente puntatori verso la 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__`)**
   - I kernel non possono avere un numero variabile di argomenti.
4. **Nessun supporto per variabili statiche locali (`__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:

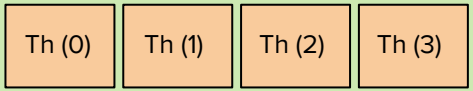
- Ottimale per problemi con **dati volumetrici**, come simulazioni fisiche o rendering 3D, dove ogni thread può operare su un voxel o una porzione dello spazio 3D.

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 >= 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

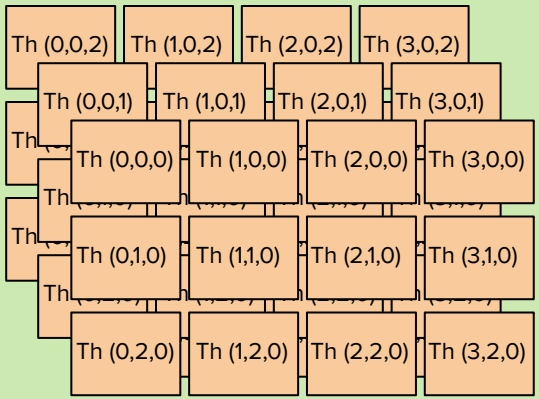
- (32, 1, 1)
- (96, 1, 1)
- (128, 1, 1)
- ...
- (1024, 1, 1)
- (2048, 1, 1) NO!

Block 2D

Esempi 2D

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

Block 3D



Esempi 3D

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

```
K = 1024
(blockDim.x * blockDim.y * blockDim.z) <= K
```

Compute Capability (CC) - Limiti SM

- La **Compute Capability (CC)** 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
10.x/12.x	Blackwell	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

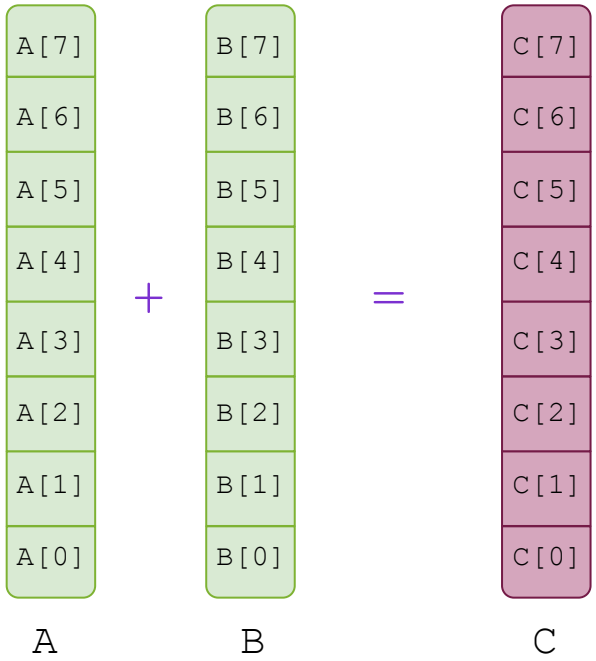
```cpp
#include <cuda_runtime.h>

// Kernel
__global__ void kernel_name() {
    // Accesso alle variabili built-in
    int blockId_x = blockIdx.x, blockId_y = blockIdx.y, blockId_z = blockIdx.z;
    int threadId_x = threadIdx.x, threadId_y = threadIdx.y, threadId_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 Modello di Programmazione CUDA

- ➤ 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 elaborati in sequenza, **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 **elaborati in parallelo**.
- 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

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

Vantaggi

- Portabilità su qualsiasi sistema
- Facilità di debugging

Codice CUDA C

```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);
```

Per evitare accessi non consentiti in memoria

Caratteristiche

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

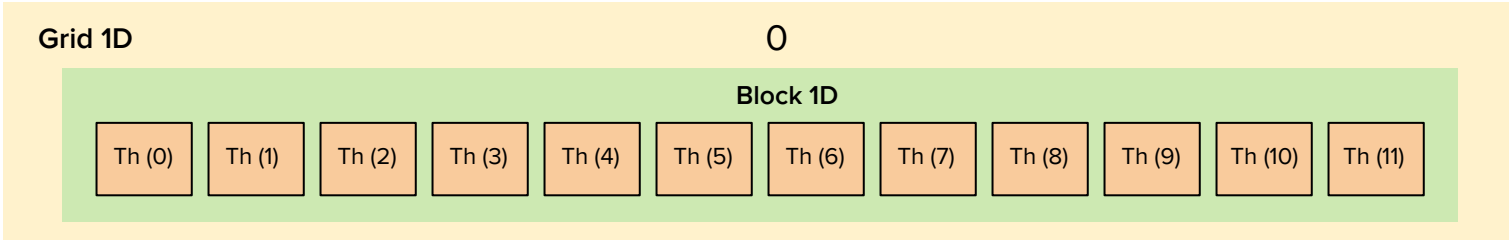
Vantaggi

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

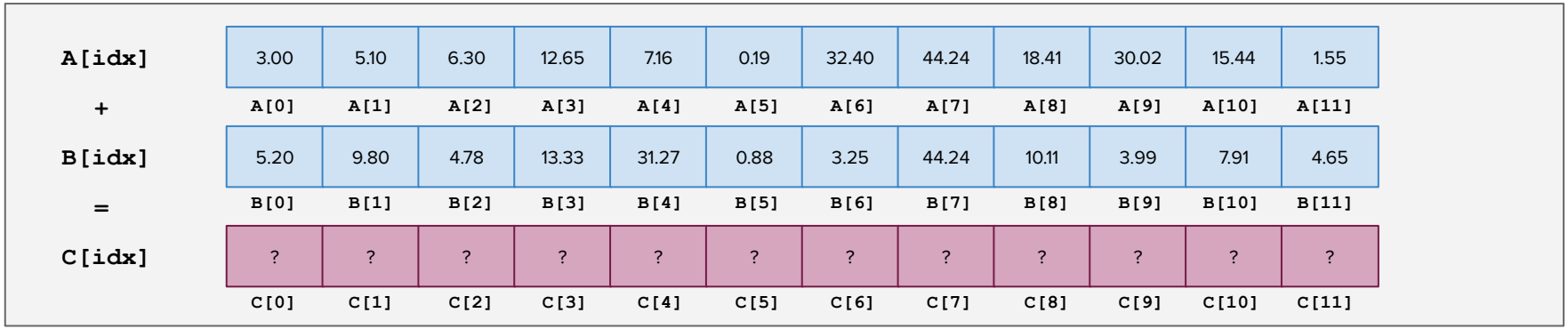
Mapping degli Indici ai Dati in CUDA - Esempio 1D

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

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



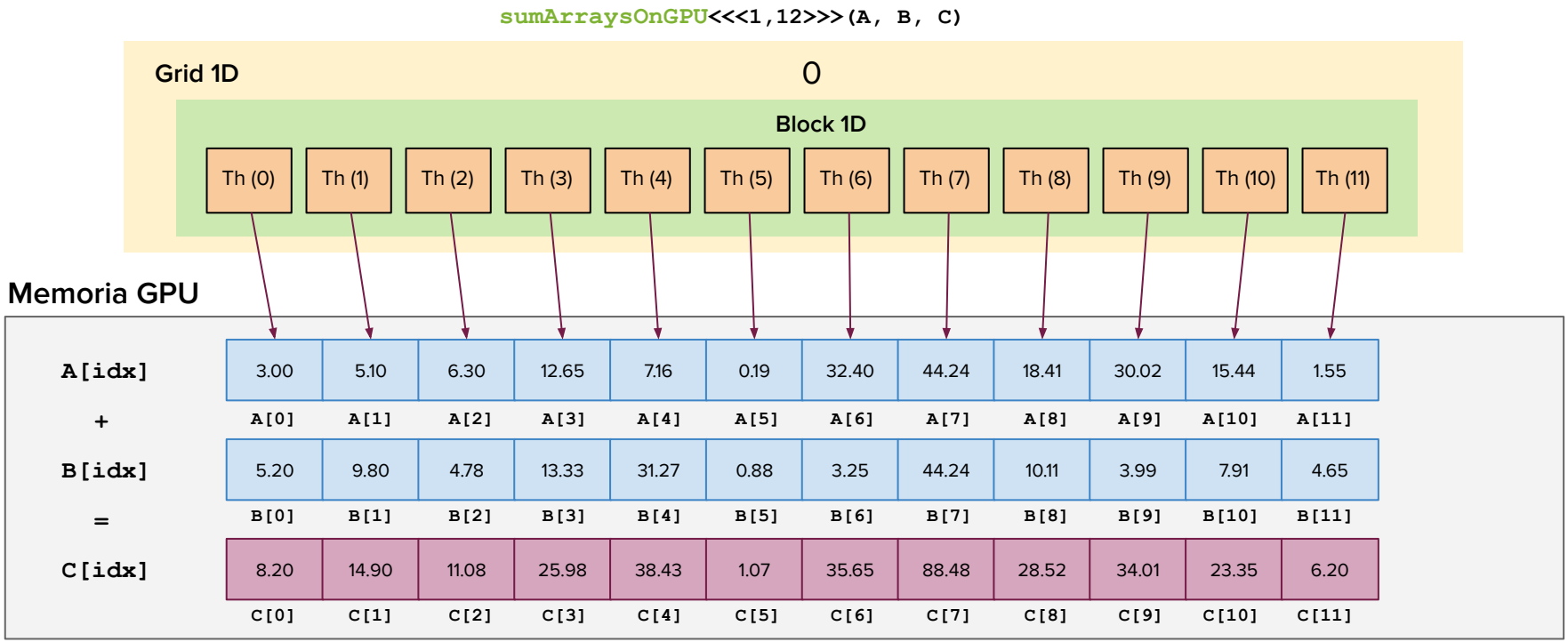
Memoria GPU



idx = ?

Mapping degli Indici ai Dati in CUDA - Esempio 1D

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



`idx = threadIdx.x` OK! Ma..

Mapping degli Indici ai Dati in CUDA - Esempio 1D

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

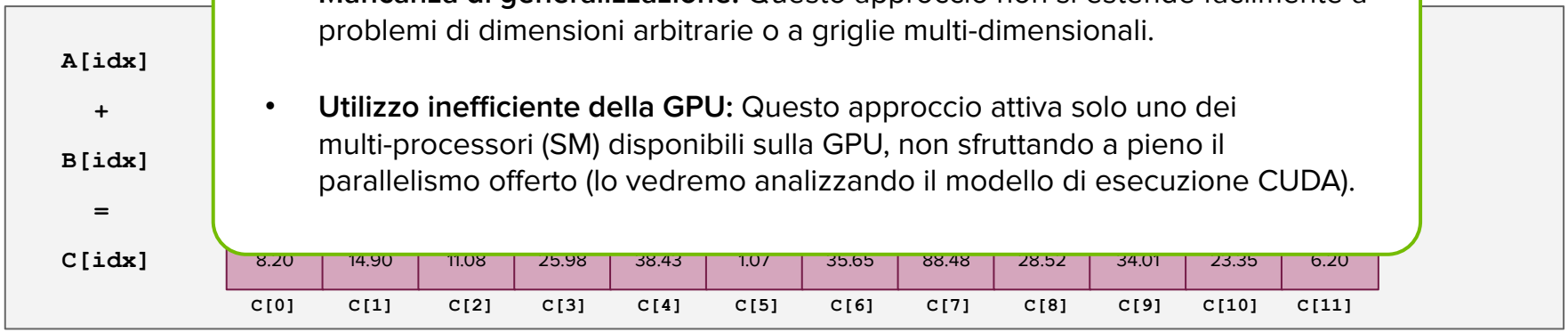
```
sumArraysOnGPU<<<1,12>>>(A, B, C)
```

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-dimensionali.
- **Utilizzo inefficiente della GPU:** Questo approccio attiva solo uno dei multi-processori (SM) disponibili sulla GPU, non sfruttando a pieno il parallelismo offerto (lo vedremo analizzando il modello di esecuzione CUDA).

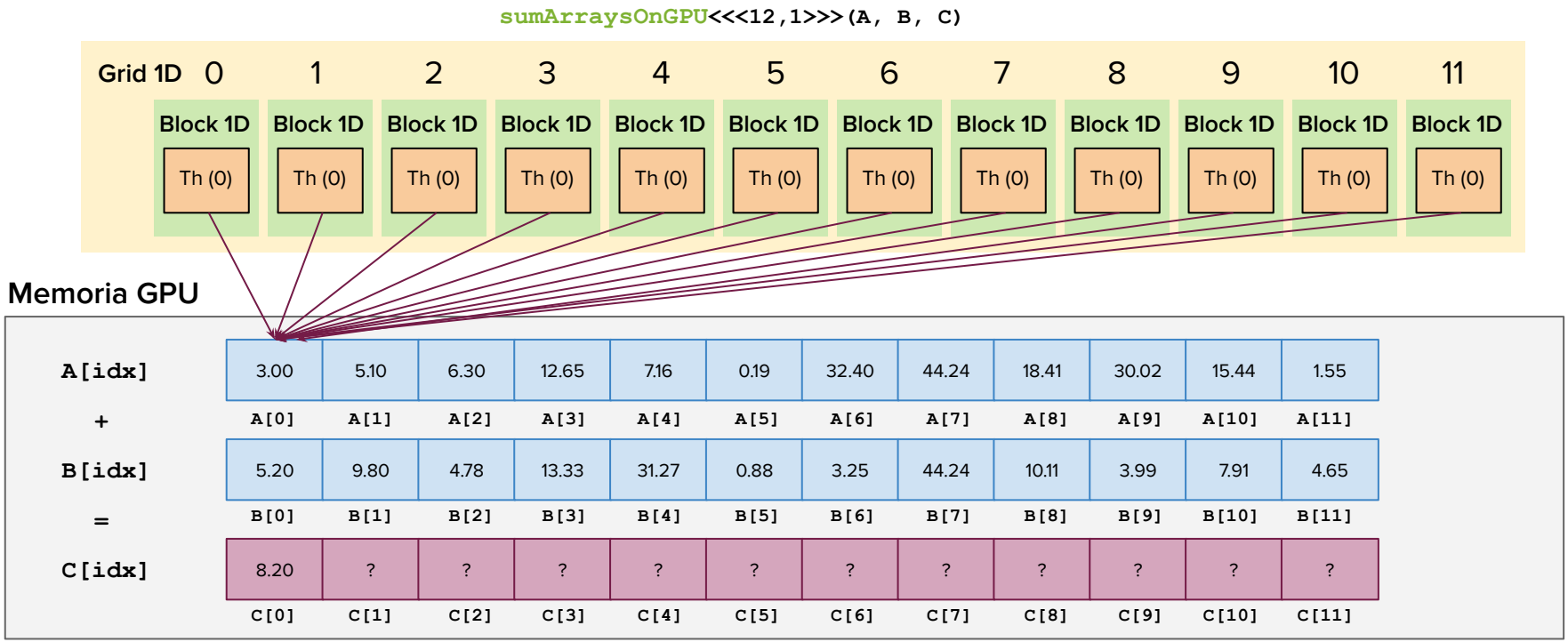
Memoria GPU



`idx = threadIdx.x` OK! Ma..

Mapping degli Indici ai Dati in CUDA - Esempio 1D

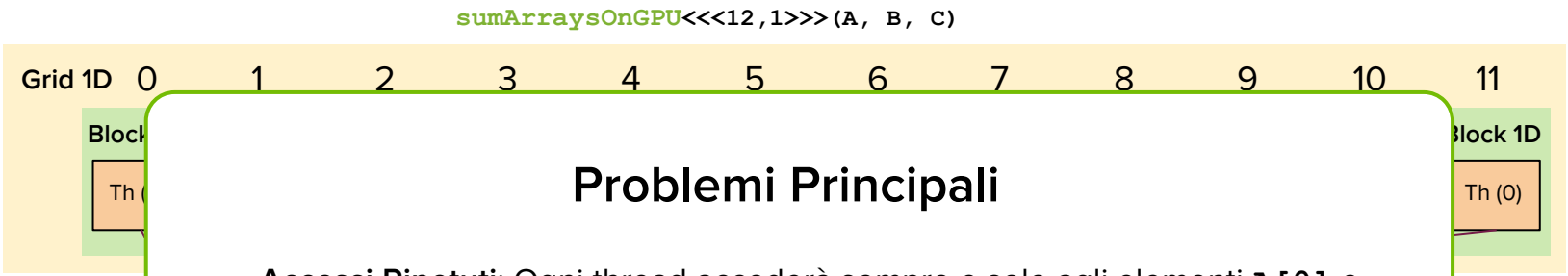
- Come **mappare** gli indici dei thread agli elementi dell'array?



`idx = threadIdx.x` NO!

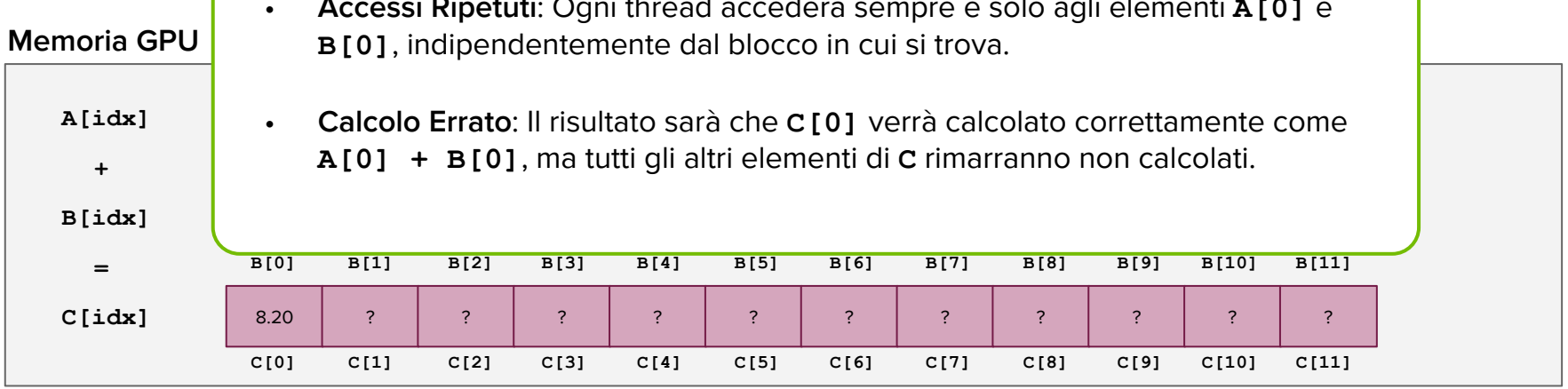
Mapping degli Indici ai Dati in CUDA - Esempio 1D

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



Problemi Principali

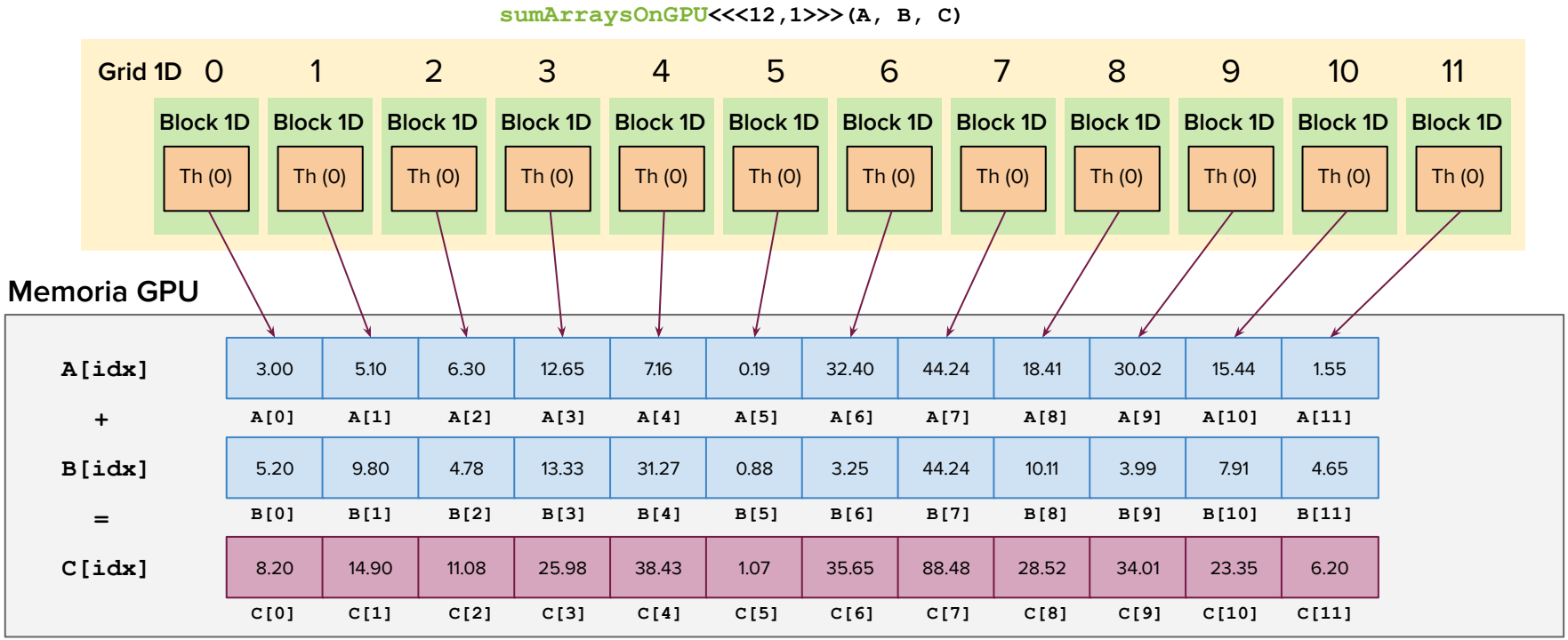
- **Accessi Ripetuti**: Ogni thread accederà sempre e solo agli elementi `A[0]` e `B[0]`, indipendentemente dal blocco in cui si trova.
- **Calcolo Errato**: Il risultato sarà che `C[0]` verrà calcolato correttamente come `A[0] + B[0]`, ma tutti gli altri elementi di `C` rimarranno non calcolati.



`idx = threadIdx.x` NO!

Mapping degli Indici ai Dati in CUDA - Esempio 1D

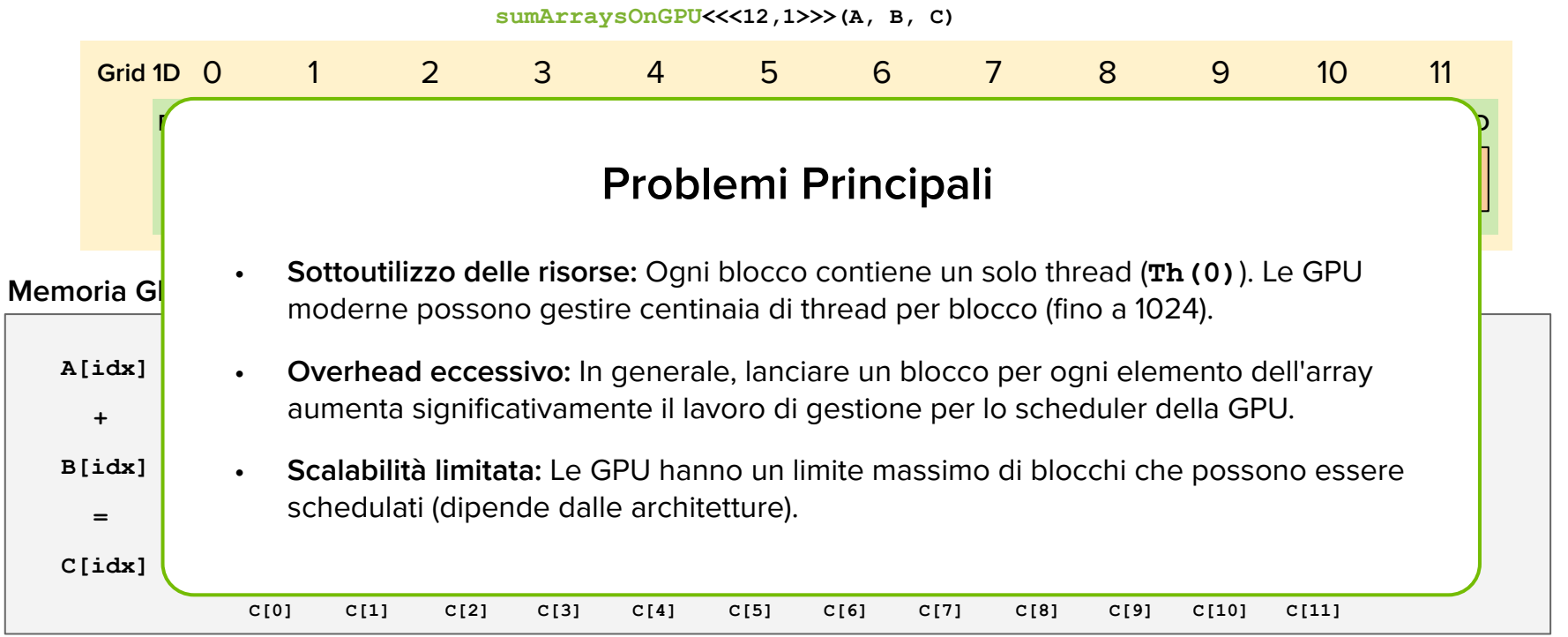
- Come **mappare** gli indici dei thread agli elementi dell'array?



`idx = blockIdx.x` OK! Ma..

Mapping degli Indici ai Dati in CUDA - Esempio 1D

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



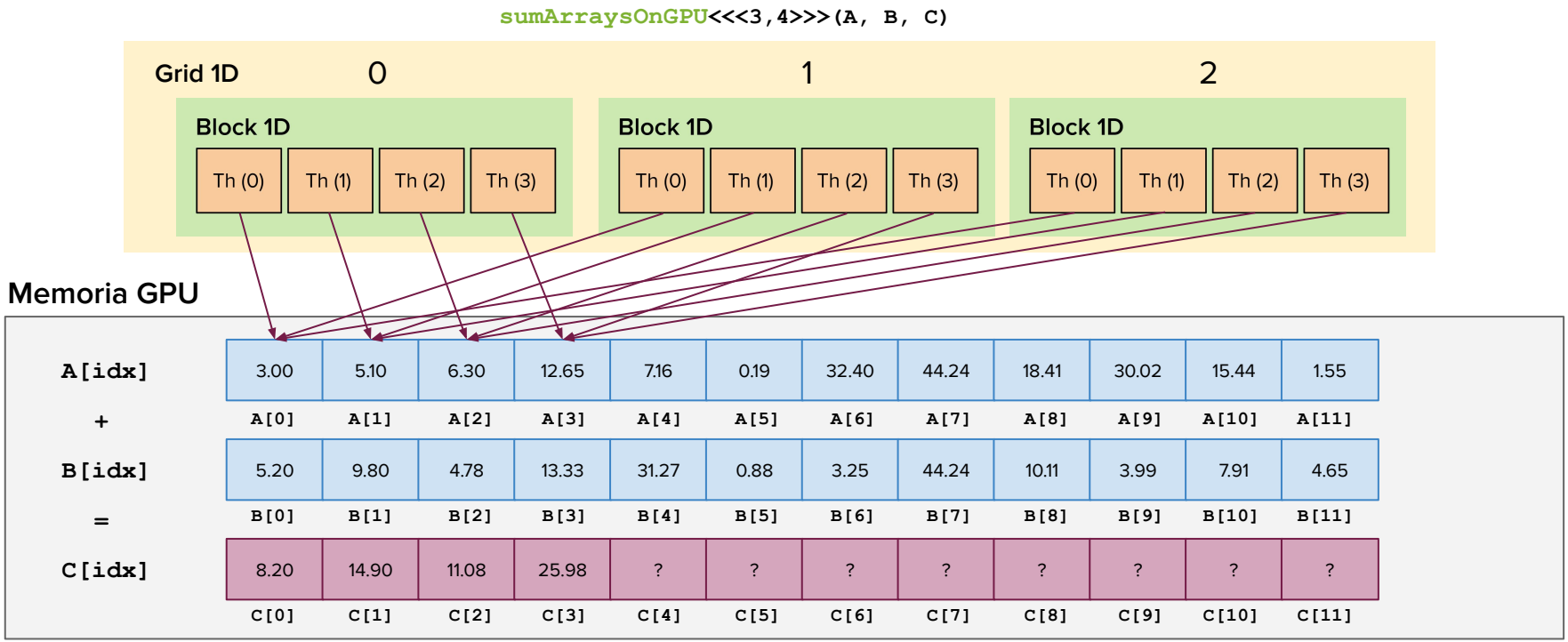
Problemi Principali

- **Sottoutilizzo delle risorse:** Ogni blocco contiene un solo thread (`Th(0)`). Le GPU moderne possono gestire centinaia di thread per blocco (fino a 1024).
- **Overhead eccessivo:** In generale, lanciare un blocco per ogni elemento dell'array aumenta significativamente il lavoro di gestione per lo scheduler della GPU.
- **Scalabilità limitata:** Le GPU hanno un limite massimo di blocchi che possono essere schedulati (dipende dalle architetture).

`idx = blockIdx.x` OK! Ma..

Mapping degli Indici ai Dati in CUDA - Esempio 1D

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

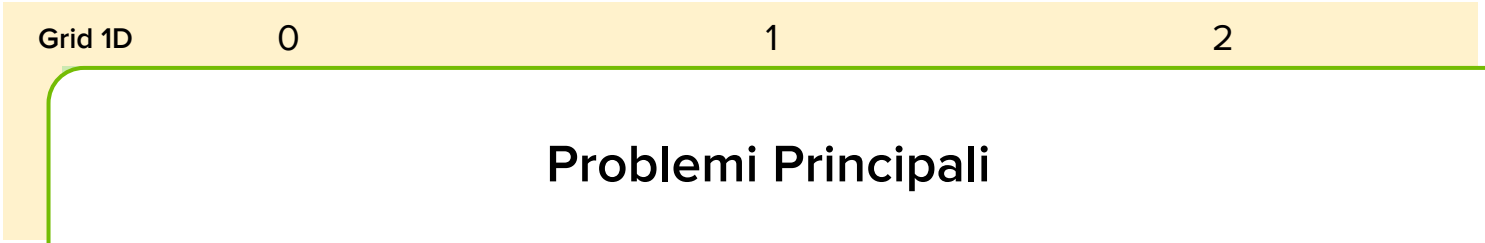


`idx = threadIdx.x` NO!

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)
```



Problemi Principali

- **Accesso limitato:** Ogni blocco accederà solo ai primi 4 elementi dell'array (indici 0-3), indipendentemente dalla sua posizione nella griglia.
- **Sovrapposizione di calcoli:** Tutti i blocchi eseguiranno gli stessi calcoli sui primi 4 elementi, risultando in una ripetizione inutile del lavoro (spreco di risorse).
- **Elementi non elaborati:** Gli elementi dal 5° in poi (indici 4-11) non verranno mai elaborati, lasciando parte dell'array **C** non calcolata.

Memoria Gl

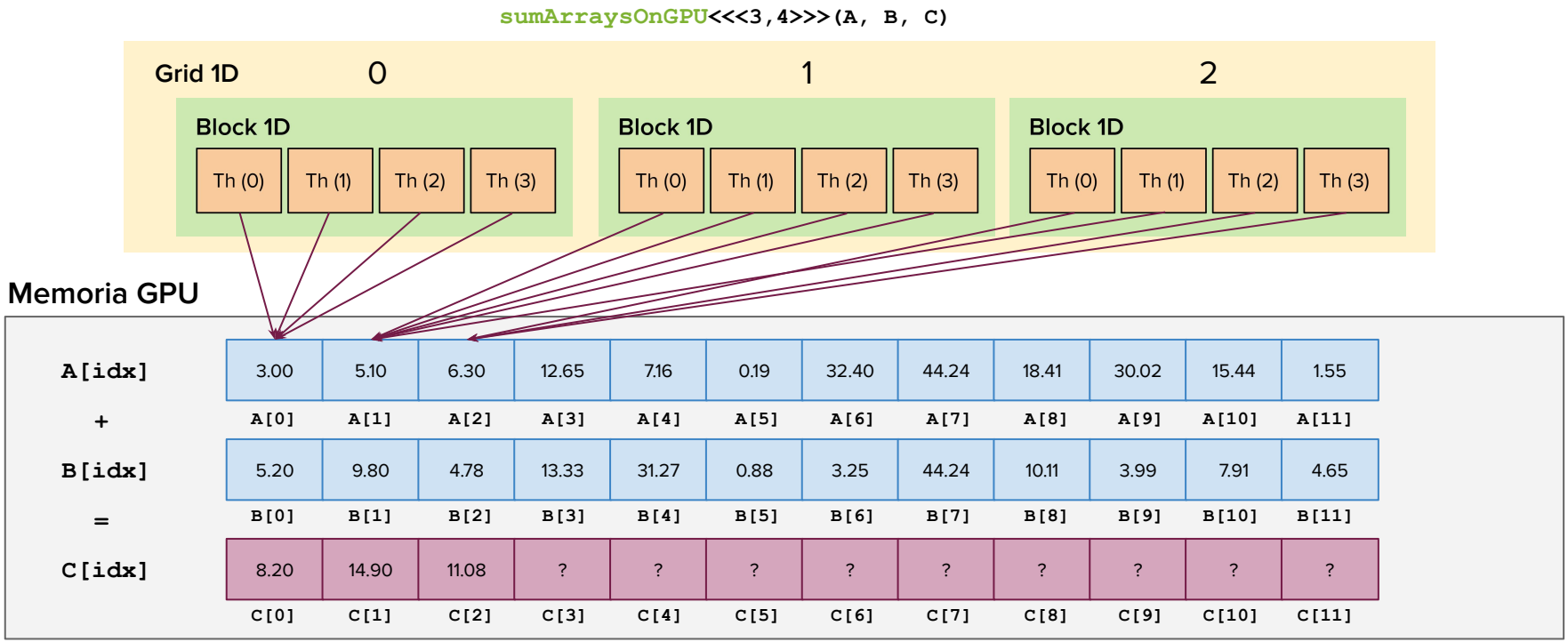
```
A[idx]
  +
B[idx]
  =
C[idx]
```

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 = threadIdx.x NO!

Mapping degli Indici ai Dati in CUDA - Esempio 1D

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

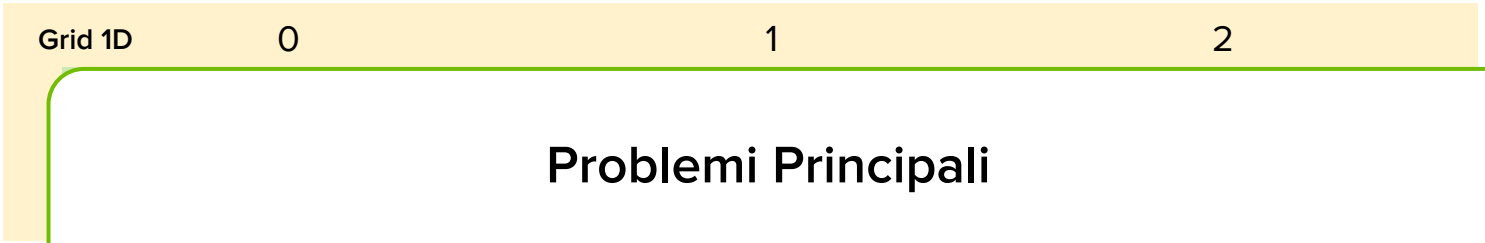


`idx = blockIdx.x` NO!

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)
```



Problemi Principali

- **Accesso limitato:** Solo 3 elementi degli array **`A`**, **`B`** e **`C`** verrebbero elaborati (indici 0, 1 e 2), corrispondenti ai tre blocchi nella griglia.
- **Spreco di risorse:** Come notato prima, 9 dei 12 thread totali stanno eseguendo calcoli ridondanti.
- **Elementi non elaborati:** Gli elementi dal 4° in poi (indici 3-11) non verranno mai elaborati, lasciando parte dell'array **`C`** non calcolata.

Memoria Gl

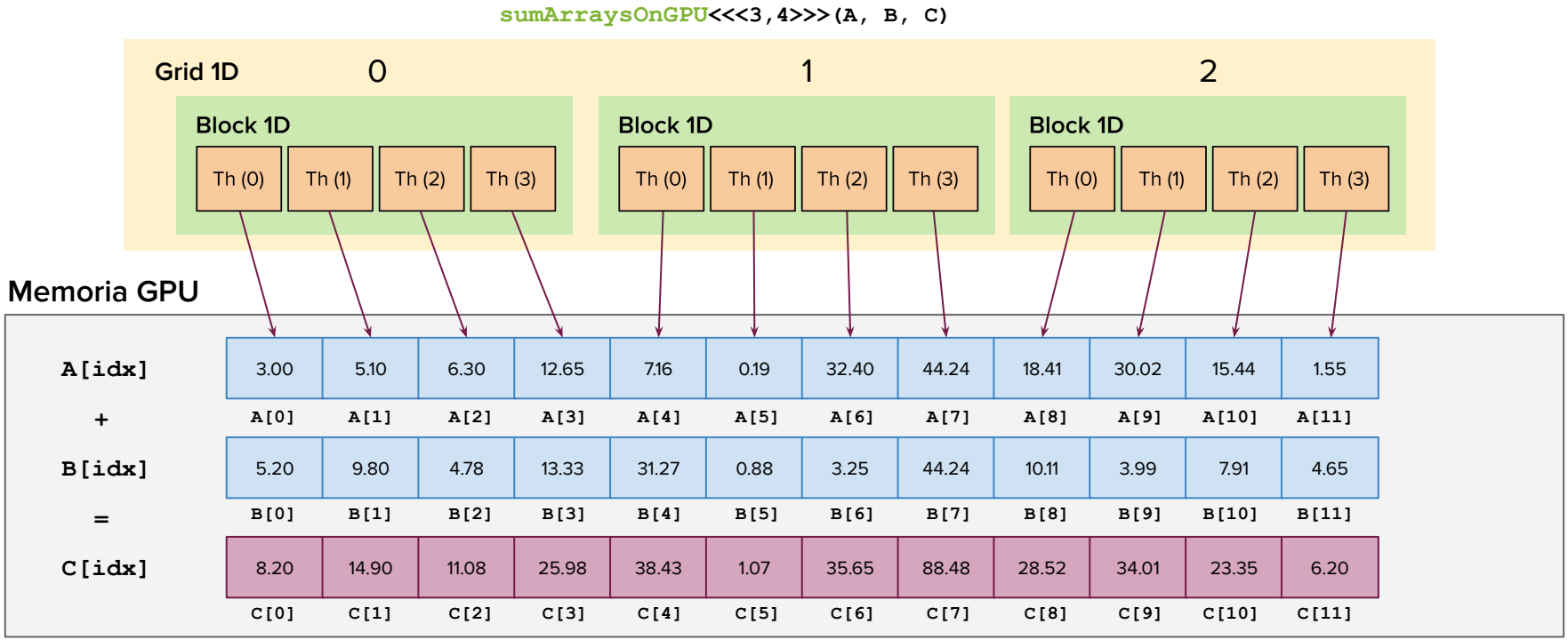
```
A[idx]
  +
B[idx]
  =
C[idx]
```

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 = blockIdx.x` NO!

Mapping degli Indici ai Dati in CUDA - Esempio 1D

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



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

Mapping degli Indici ai Dati in CUDA - Esempio 1D

- Com

Memoria Gl

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

Proprietà Chiave

- **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.

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

OK!

Confronto: Somma di Vettori in C vs CUDA C

Codice C Standard

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

Vantaggi

- Portabilità su qualsiasi sistema
- Facilità di debugging

Codice CUDA C

```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);
```

Per evitare accessi non consentiti in memoria

Tutto ruota intorno a questa linea di codice

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** di un thread identifica univocamente **quale porzione di dati** deve elaborare.
- Essenziale per gestire correttamente l'**accesso alla memoria** e **coordinare** l'esecuzione di algoritmi complessi.

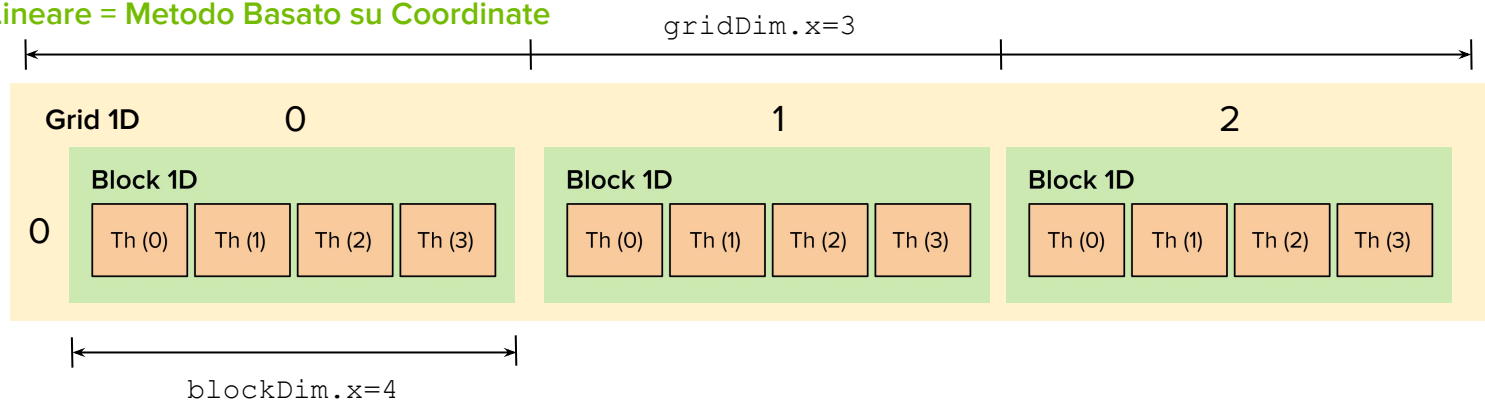
Struttura dei Dati e Calcolo dell'Indice Globale

- 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** in base al proprio indice globale.
- Esistono **diversi metodi** per calcolare l'**indice globale** di un thread (es. **Metodo Lineare**, **Coordinate-based**).
- Metodi diversi possono produrre **mappature diverse** tra thread e dati, **influenzando prestazioni** (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



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

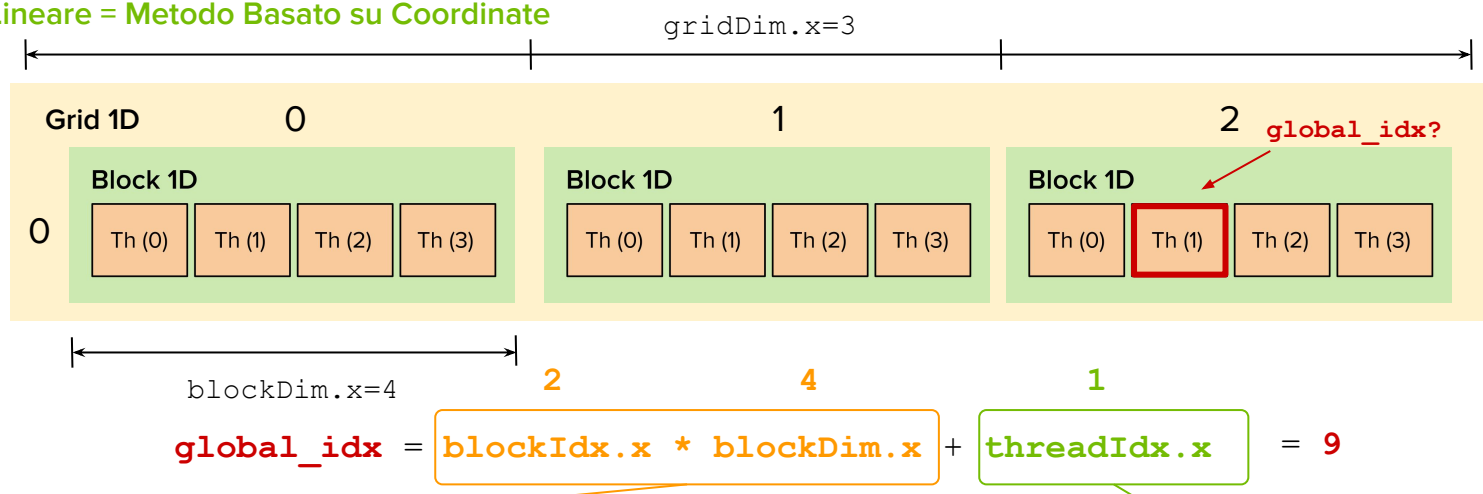
- Calcola l'offset di tutti i thread nei blocchi precedenti al blocco corrente.
- Moltiplicando **`blockIdx.x`** per **`blockDim.x`**, otteniamo il numero totale di thread che si trovano nei blocchi precedenti.

- 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



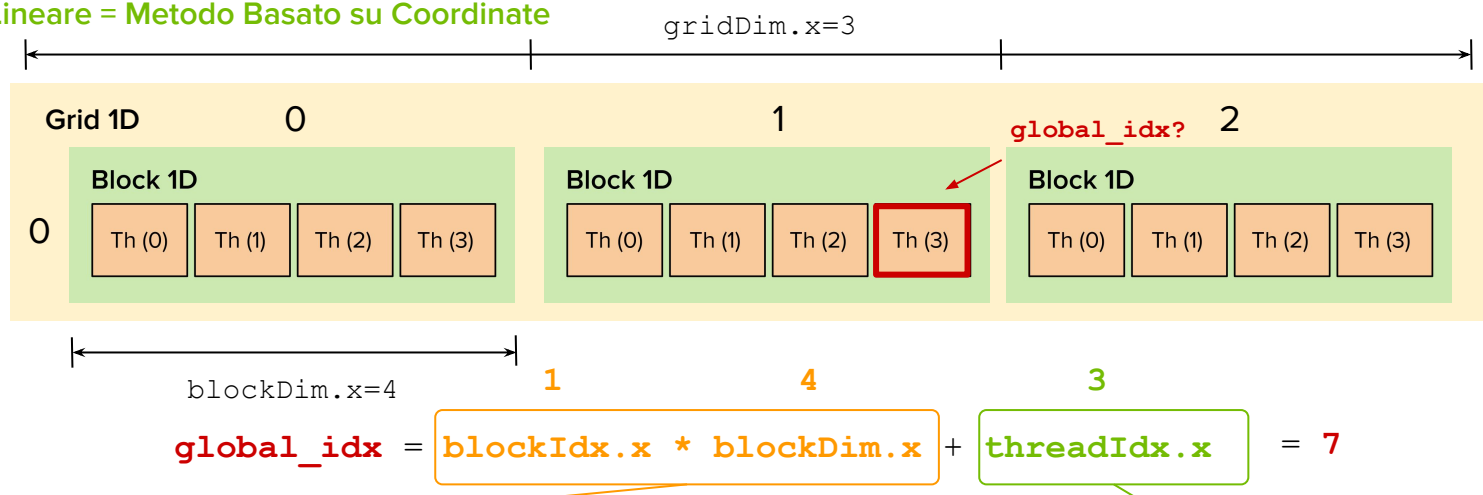
- Calcola l'offset di tutti i thread nei blocchi precedenti al blocco corrente.
- Moltiplicando **`blockIdx.x`** per **`blockDim.x`**, otteniamo il numero totale di thread che si trovano nei blocchi precedenti.

- 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



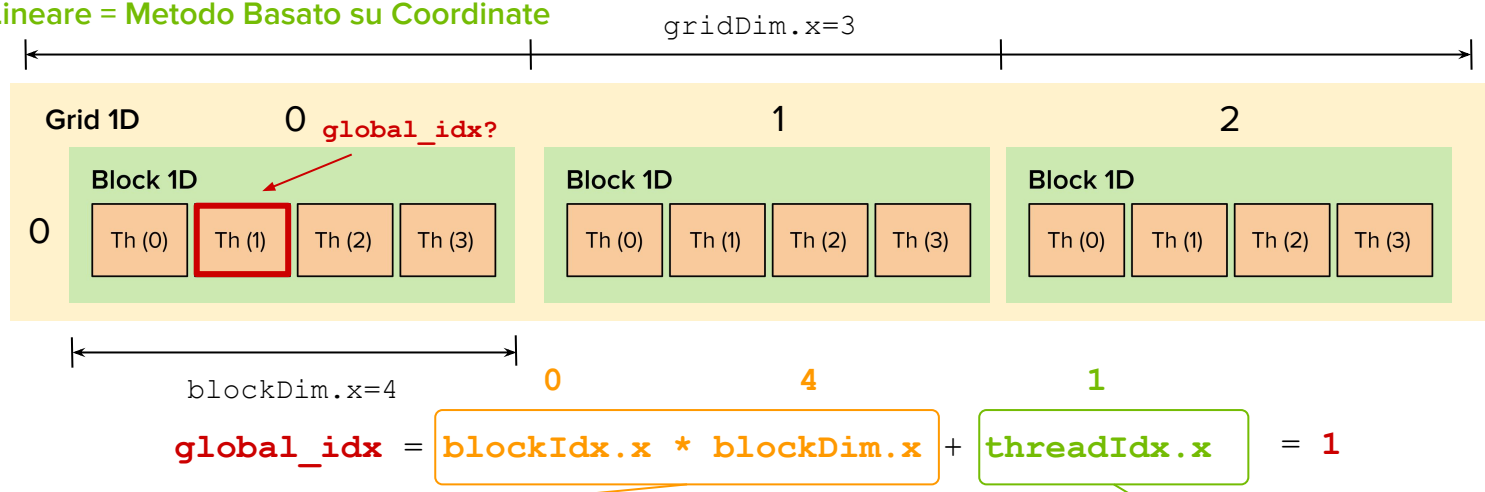
- Calcola l'offset di tutti i thread nei blocchi precedenti al blocco corrente.
- Moltiplicando **`blockIdx.x`** per **`blockDim.x`**, otteniamo il numero totale di thread che si trovano nei blocchi precedenti.

- 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

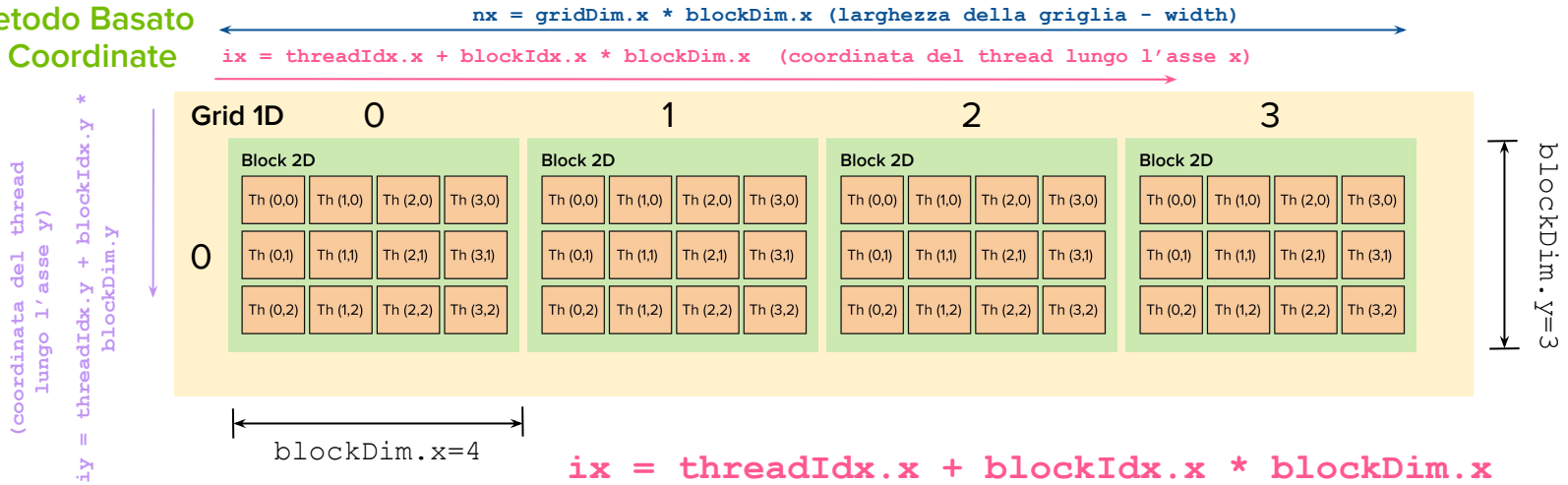


- Calcola l'offset di tutti i thread nei blocchi precedenti al blocco corrente.
- Moltiplicando **`blockIdx.x`** per **`blockDim.x`**, otteniamo il numero totale di thread che si trovano nei blocchi precedenti.

- 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 Basato su Coordinate




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

`nx = gridDim.x * blockDim.x = 16`

`global_idx = iy * nx + ix`

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

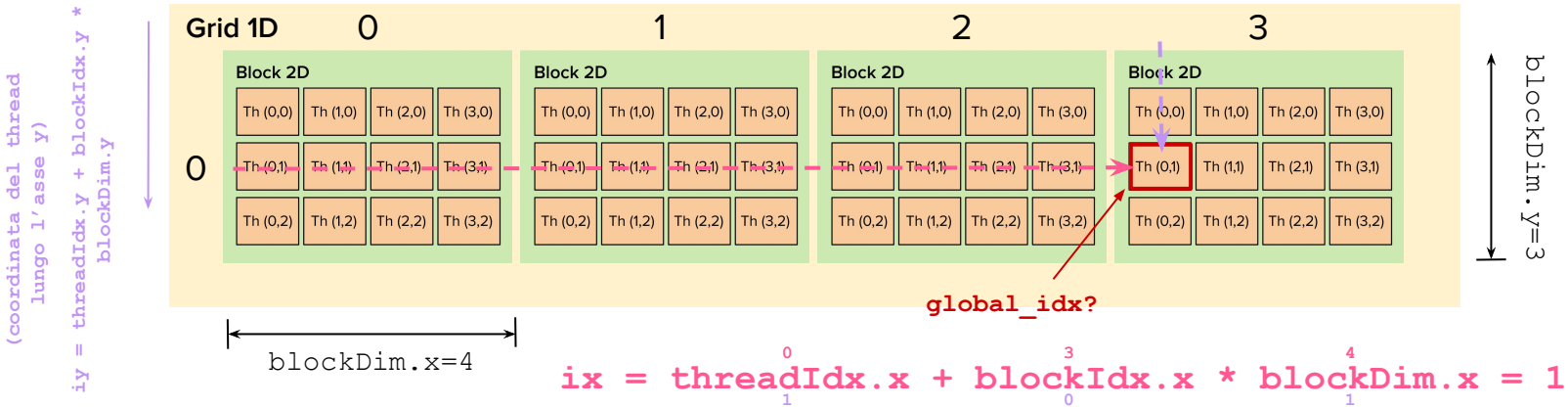
- `ix = threadIdx.x + blockIdx.x * blockDim.x`: Determina l'indice del thread lungo l'asse **x**, prendendo in considerazione la posizione nel blocco (`threadIdx.x`) e il numero di blocchi precedenti (`blockIdx.x * blockDim.x`).
- `iy = threadIdx.y + blockIdx.y * blockDim.y`: Determina l'indice del thread 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 = gridDim.x * blockDim.x`).

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

Metodo Basato su Coordinate

`nx = gridDim.x * blockDim.x (larghezza della griglia - width)`

`ix = threadIdx.x + blockIdx.x * blockDim.x (coordinata del thread lungo l'asse x)`



$ix = threadIdx.x + blockIdx.x * blockDim.x = 0 + 3 * 4 = 12$

$iy = threadIdx.y + blockIdx.y * blockDim.y = 1 + 0 * 1 = 1$

$nx = gridDim.x * blockDim.x = 4 * 4 = 16$

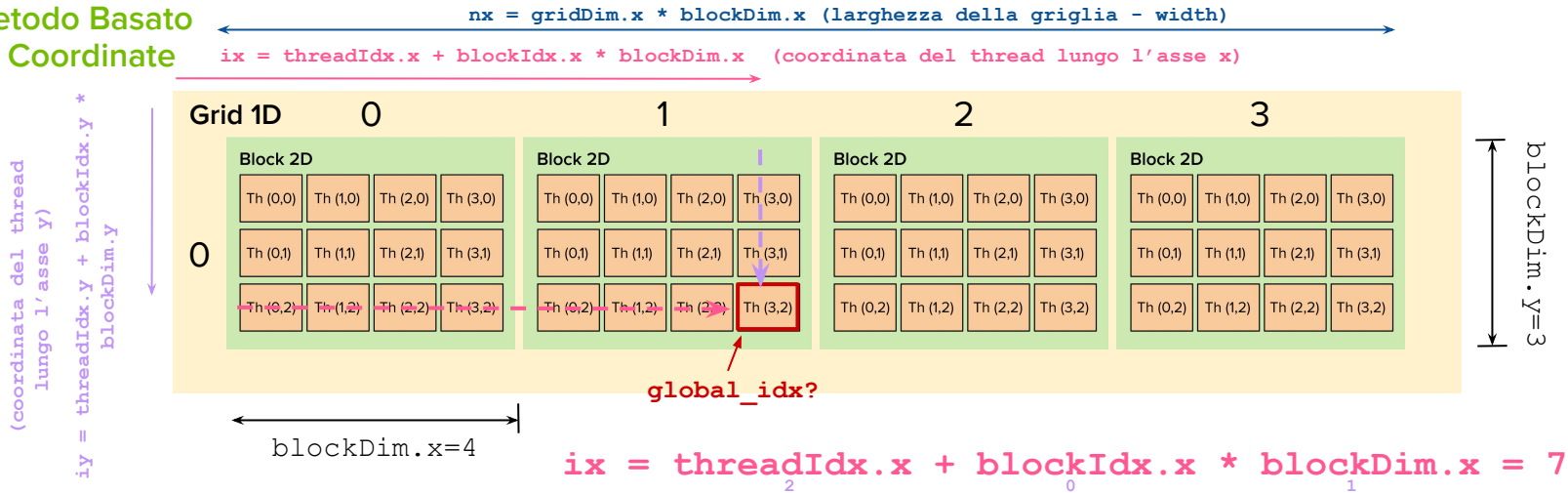
$global\_idx = iy * nx + ix = 1 * 16 + 12 = 28$

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 = gridDim.x * blockDim.x`).

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

Metodo Basato su Coordinate



$$ix = threadIdx.x + blockIdx.x * blockDim.x = 7$$

$$iy = threadIdx.y + blockIdx.y * blockDim.y = 2$$

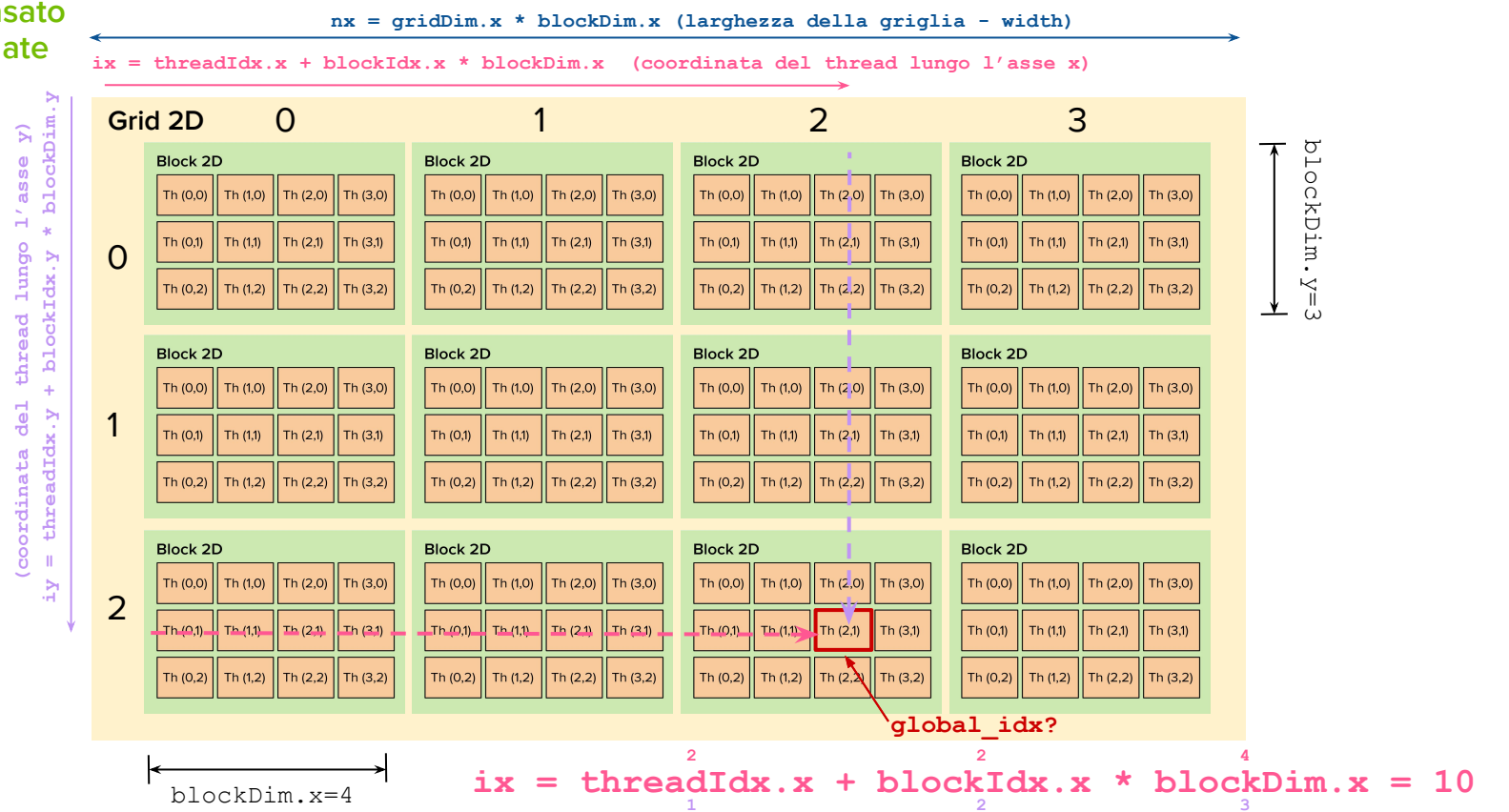
$$nx = gridDim.x * blockDim.x = 16 \qquad global\_idx = iy * nx + ix = 39$$

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 = gridDim.x * blockDim.x`).

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

Metodo Basato su Coordinate



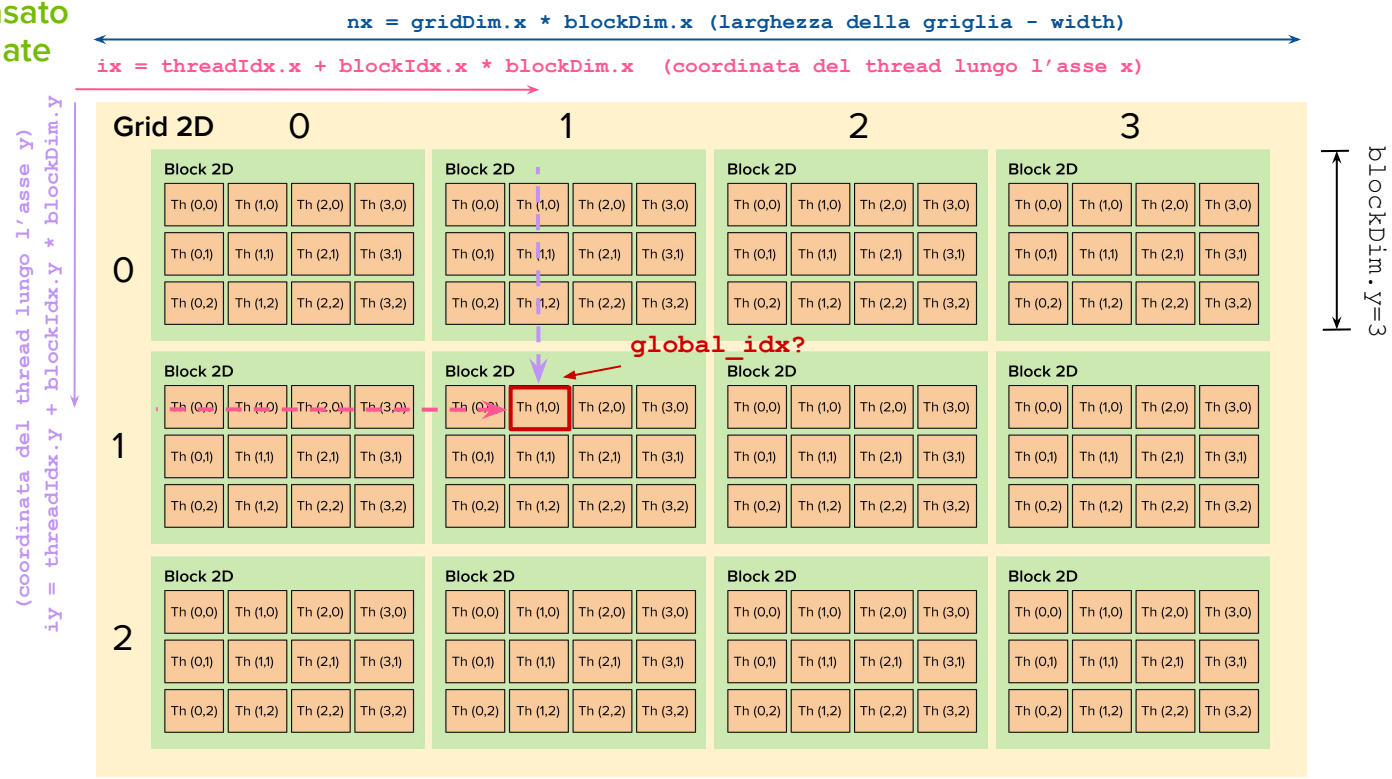
$$ix = threadIdx.x^{2} + blockIdx.x^{2} * blockDim.x^{4} = 10$$

$$iy = threadIdx.y^{1} + blockIdx.y^{2} * blockDim.y^{3} = 7$$

$$nx = gridDim.x^{4} * blockDim.x^{4} \qquad global\_idx = iy^{7} * nx^{16} + ix^{10} = 114$$

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

Metodo Basato su Coordinate



$$ix = threadIdx.x + blockIdx.x * blockDim.x = 5$$

$$iy = threadIdx.y + blockIdx.y * blockDim.y = 3$$

$$nx = gridDim.x * blockDim.x \qquad global\_idx = iy * nx + ix = 53$$

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
```

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

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

Calcolo dell'Indice Globale

```
idx = x
```
(equivalente al caso lineare)

```
idx = y * width + x
```

```
idx = z * (height * width)
    + y * width
    + x
```

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** (numero di 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`** assicura un numero sufficiente di 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`** "compensa" questi dati residui, includendo 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** (numero di 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 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** (numero di thread per blocco).
- Poi, calcolare **automaticamente** la dimensione della griglia in base ai **dati** e alla **dimensione del blocco.**

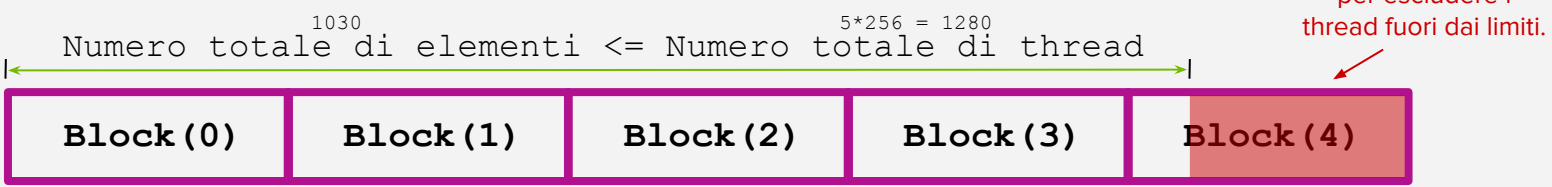
Motivazioni

- 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



Come Calcolare la Dimensione della Griglia e del Blocco?

Approccio Generale

- Definire **manualmente prima la dimensione del blocco** (numero di 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** (numero di 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**.

```
#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 di elementi
  dim3 block(3);                              // Definiamo un blocco 1D con 3 thread
  dim3 grid((nElem+block.x-1)/block.x);       // Calcolo dei blocchi necessari

  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);}
```

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;                                  // Numero
  dim3 block(3);                                  // Defini
  dim3 grid((nElem+block.x-1)/block.x);           // Calcol

  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

  checkIndex<<<grid, block>>>();                  // Esecuzio

  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, %
         "gridDim:(%d, %d, %d)\n", threadIdx.x, thre
                                   blockIdx.x, block
                                   blockDim.x, block
                                   gridDim.x, gridDi


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

  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

  checkIndex<<<grid, block>>>();              // Esecuzi

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

  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

  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)
```