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

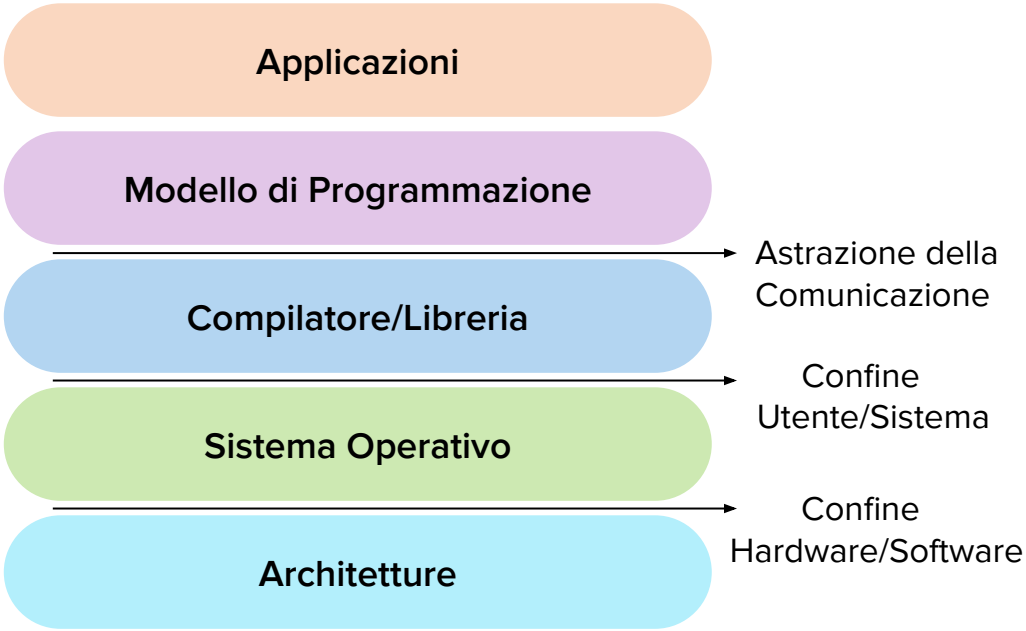
➤ Analisi delle Prestazioni
- Correttezza dei risultati e gestione degli errori
- Uso di strumenti di profiling (NVIDIA Nsight)

➤ 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.



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

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

Compilatore/Libreria: Strumenti che traducono il codice CUDA in istruzioni eseguibili dalla GPU, includendo ottimizzazioni specifiche.

Sistema Operativo: Gestisce le risorse del sistema, inclusa l'allocazione della GPU tra diverse applicazioni.

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:

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 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

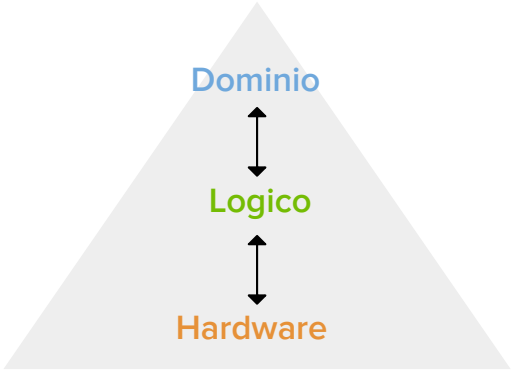
- Organizzazione e gestione dei thread.
- Implementazione della strategia di parallelizzazione.

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.

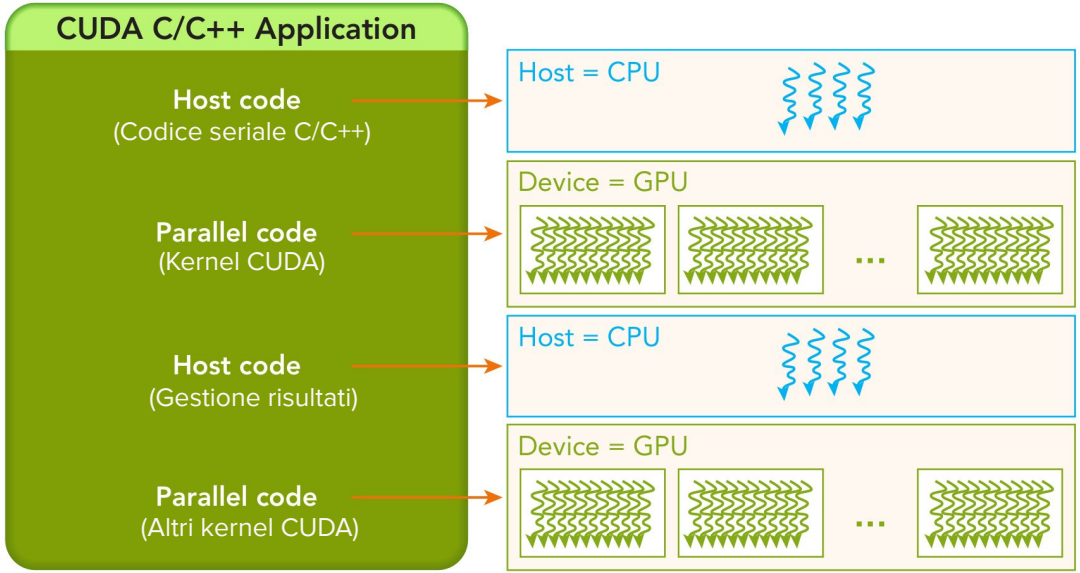
Cosa Fa un Thread CUDA?

- **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 (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

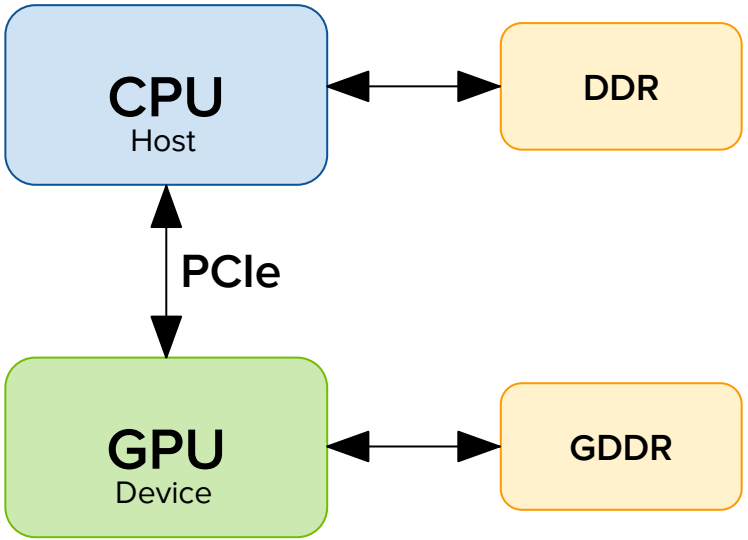
Gestione della Memoria in CUDA

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

- 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

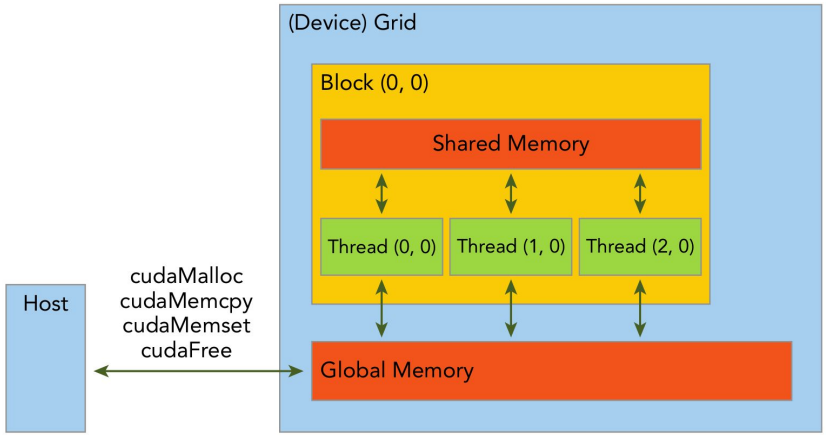
Shared Memory

- 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

- **`cudaMemCopy`** 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

- **cud**
  all'ir

Firma della

```
cudaErr
```

Parametri

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

Valore di R

- **cud**

Note importa

- 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).

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 deallocazioneha 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

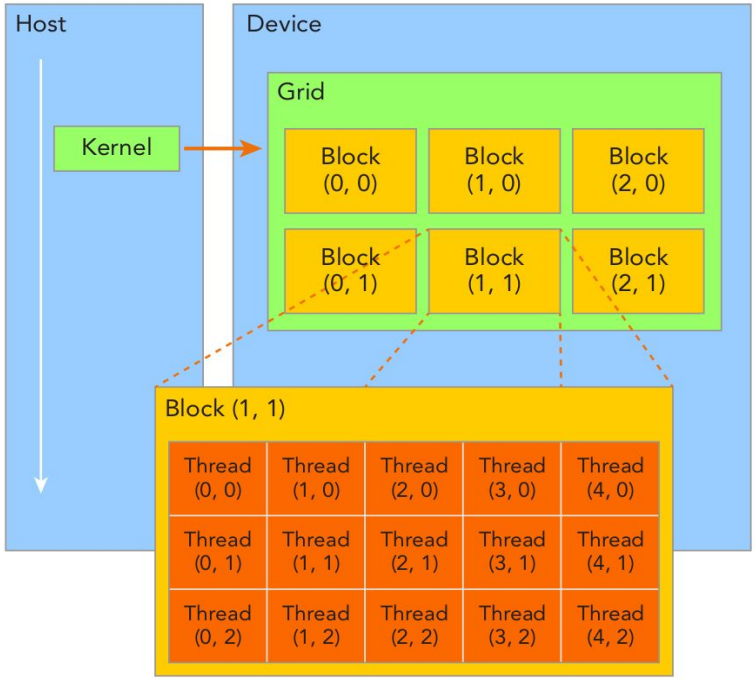
- CUDA adotta una **gerarchia a due livelli** per organizzare i thread basata su **blocchi di thread** e **griglie di blocchi**.

Struttura Gerarchica

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 cooperare.**

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 **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

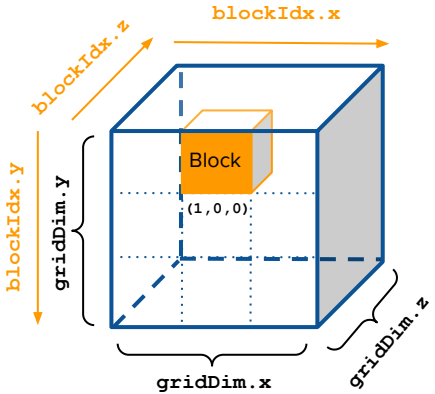
- La gerarchia è **scalabile** e permette di **adattare l'esecuzione** a GPU con diverse capacità e numero di core. Il codice CUDA, quindi, risulta più **portabile** e può essere eseguito su diverse architetture GPU.

➤ Sincronizzazione Granulare

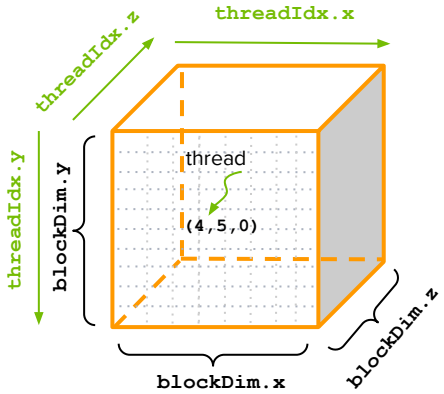
- I thread possono essere sincronizzati solo **all'interno del proprio blocco**, evitando costose sincronizzazioni globali che possono creare colli di bottiglia.

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**.



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 all'interno della gerarchia grid-block. Queste coordinate … **ai dati**.

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`

Variabili di Identi…

1. **blockIdx** (indice …)
   - Compo…
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**.

…read)
…e, 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`**

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
`__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.

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 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:

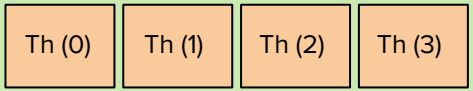
- Ottimale per problemi con **dati volumetrici**, come simulazioni fisiche o rendering 3D, dove ogni thread può operare su un voxel o una porzione dello spazio 3D.

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

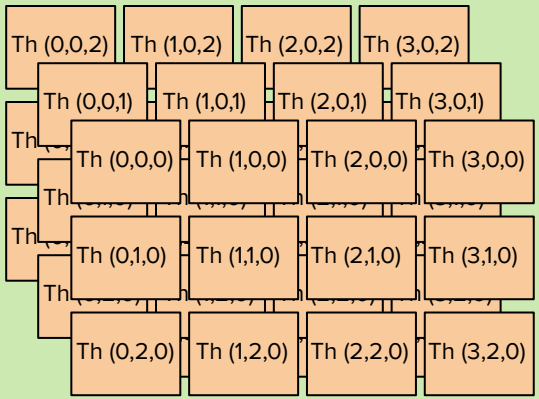
- (32, 1, 1)
- (96, 1, 1)
- (128, 1, 1)
- ...
- (1024, 1, 1)
- (2048, 1, 1) NO!

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

- 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

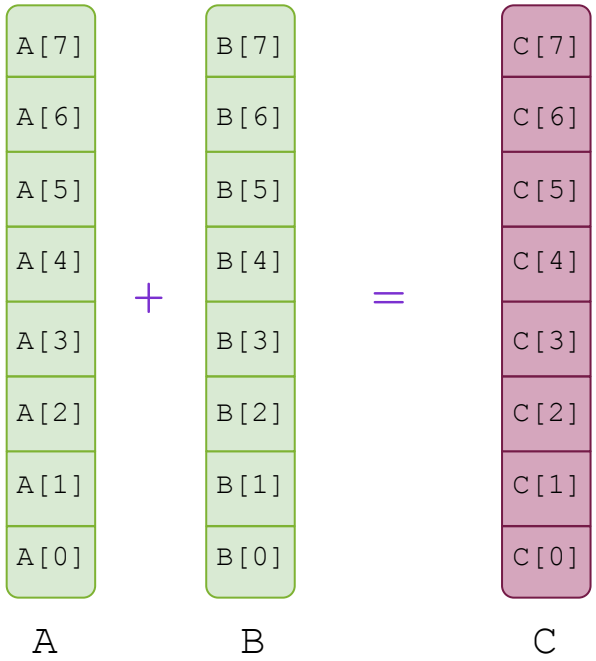
```cpp
#include <cuda_runtime.h>

// 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 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

```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

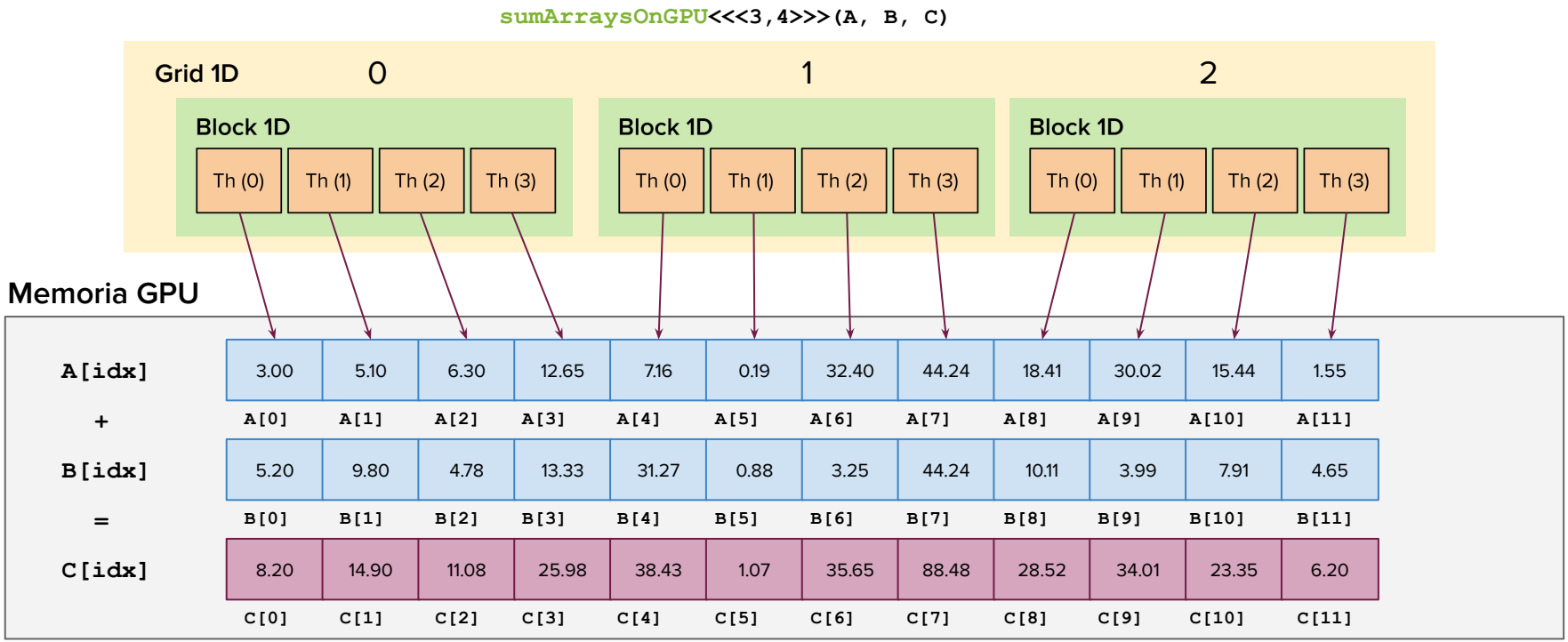
- **Esecuzione:** Parallela
- **Iterazione**: Implicita (un thread per elemento)
- **Indice**: ?
- **Scalabilità**: Elevata (sfrutta molti core GPU)

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?



`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** 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.

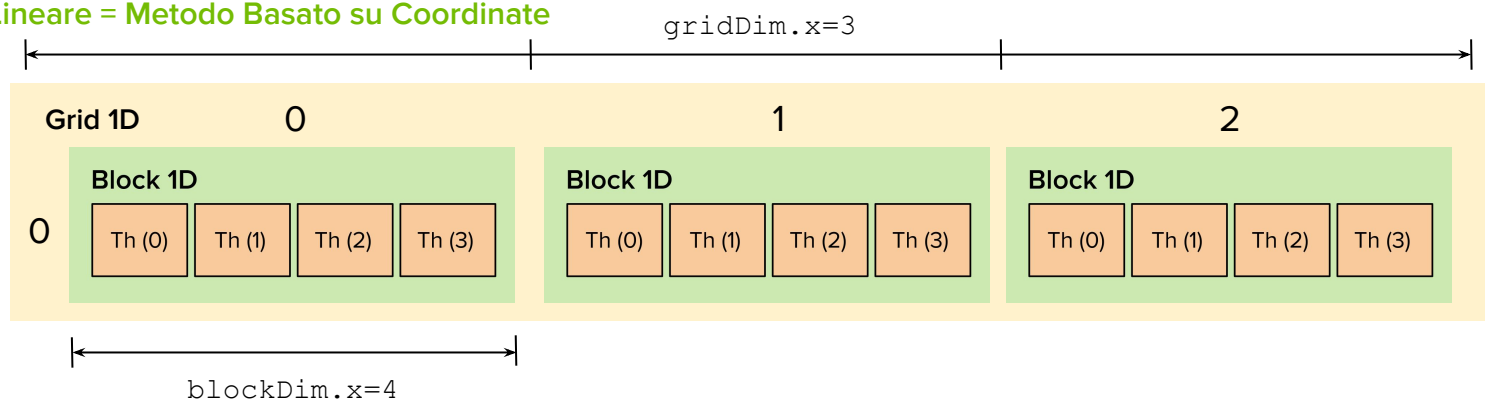
Struttura dei Dati e Calcolo dell'Indice Globale

- 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



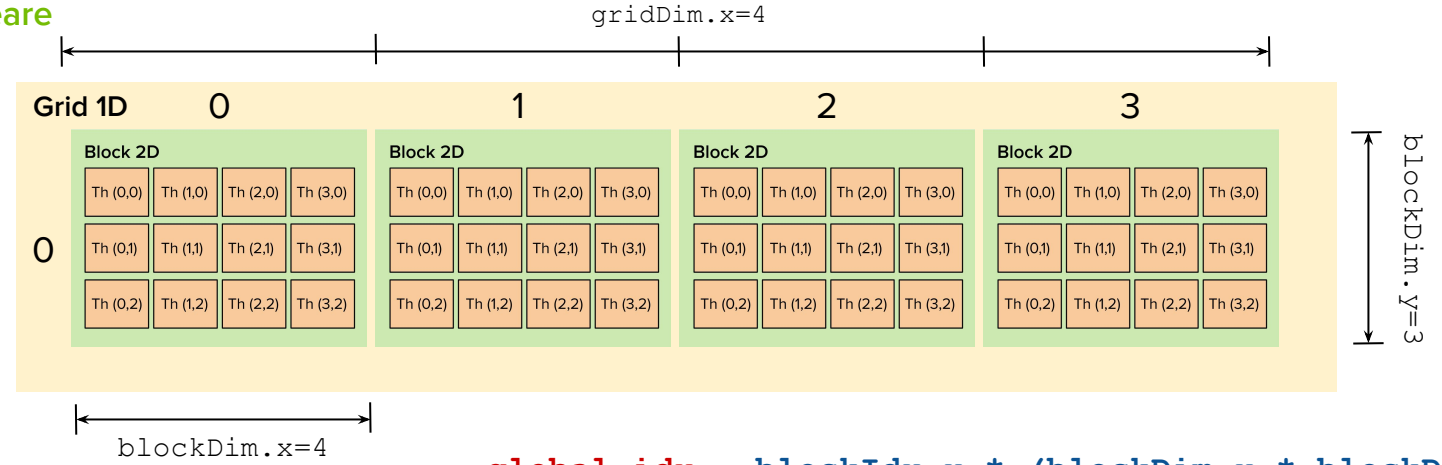
`global_idx` = **`blockIdx.x * blockDim.x`** + **`threadIdx.x`**

- 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 2D

Metodo Lineare



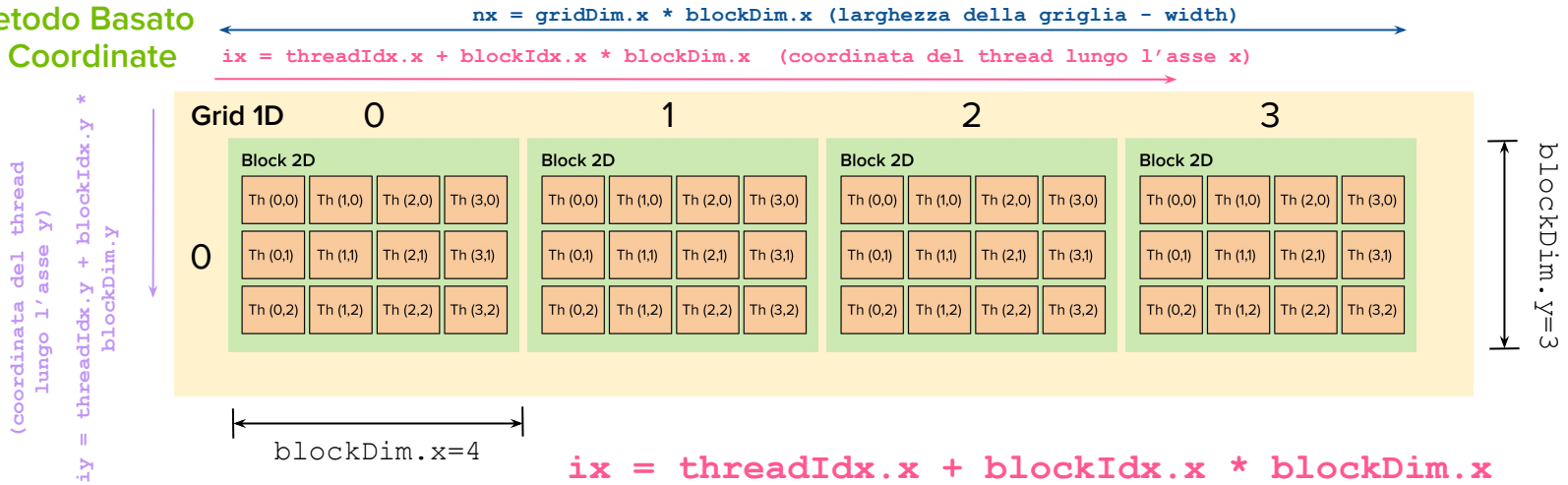

```
global_idx = blockIdx.x * (blockDim.x * blockDim.y)
           + threadIdx.y * blockDim.x
           + threadIdx.x
```

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 Basato su Coordinate




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

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

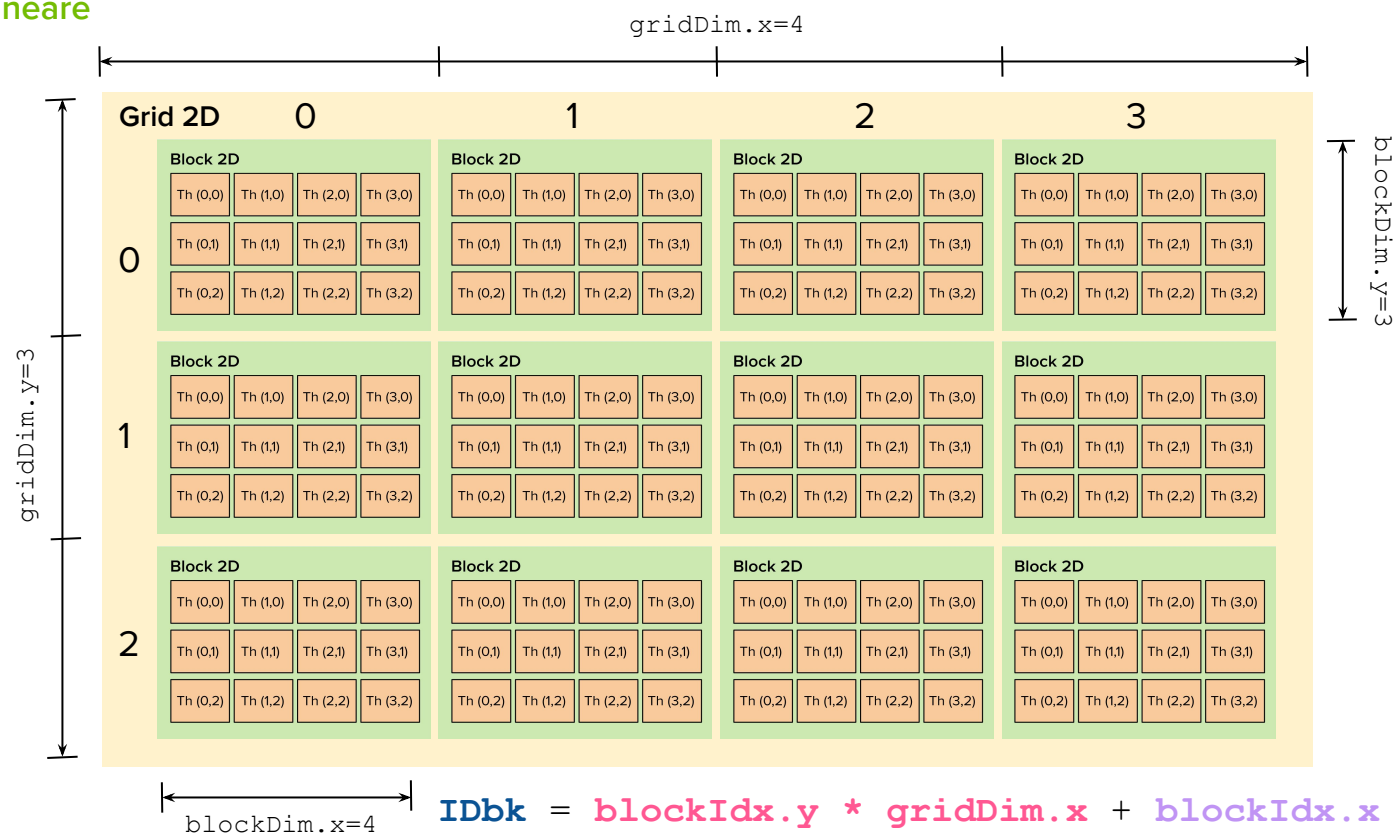
`global_idx = iy * nx + ix`

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 2D, Block 2D

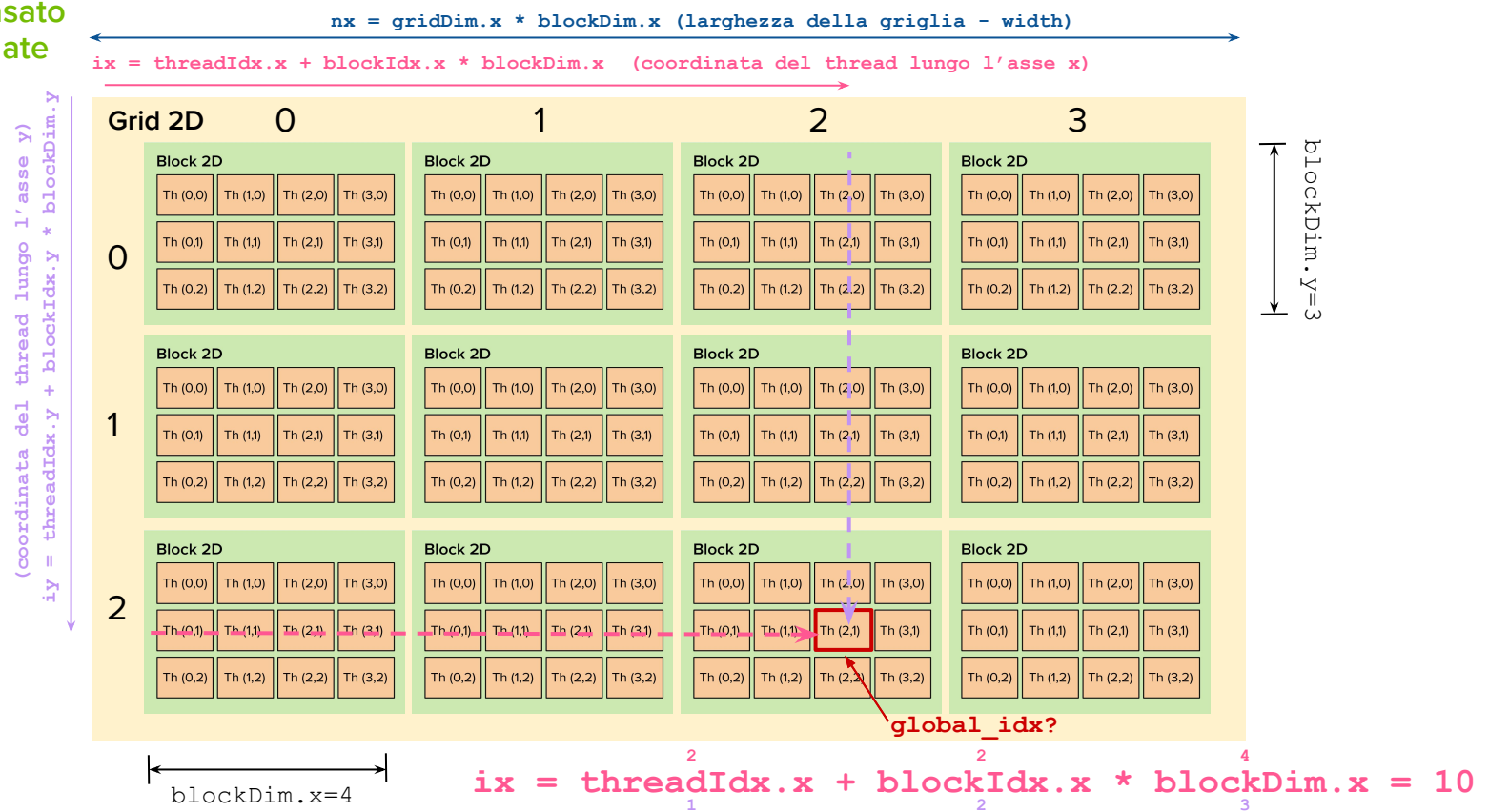
Metodo Lineare




```
IDbk = blockIdx.y * gridDim.x + blockIdx.x

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

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

Metodo Basato su Coordinate



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

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

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

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
```

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** (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 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
```

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,
                 float *gpuRef,
                 const int N) {
   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");}
```

Risultati attesi della somma

Risultati calcolati dal kernel

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,
                cudaGetErrorString(error));
        exit(1);
    }
}
```

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

```c
#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 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 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.

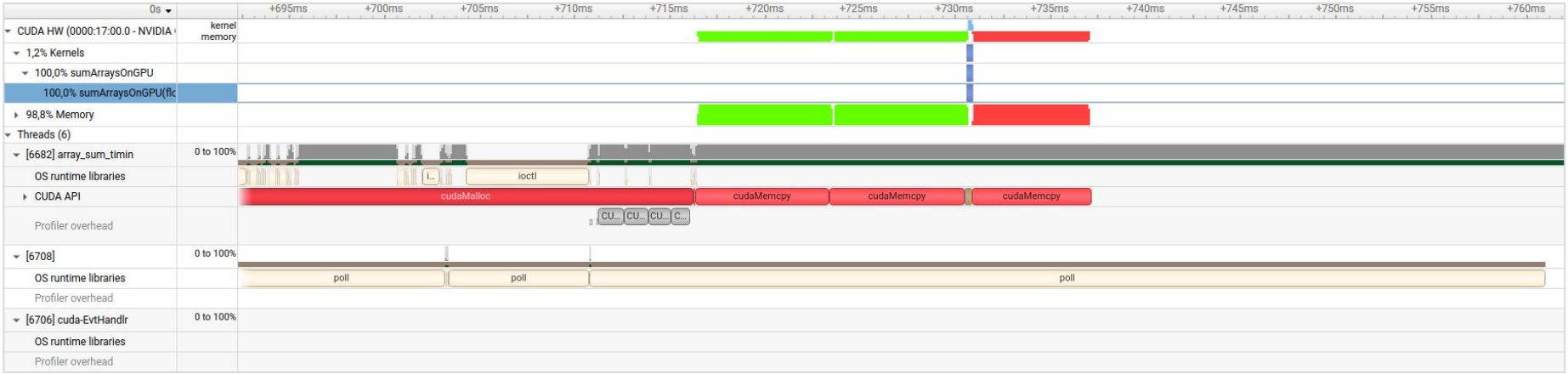
Come si usa?

```
$ 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

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.

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.

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.

Come si usa?

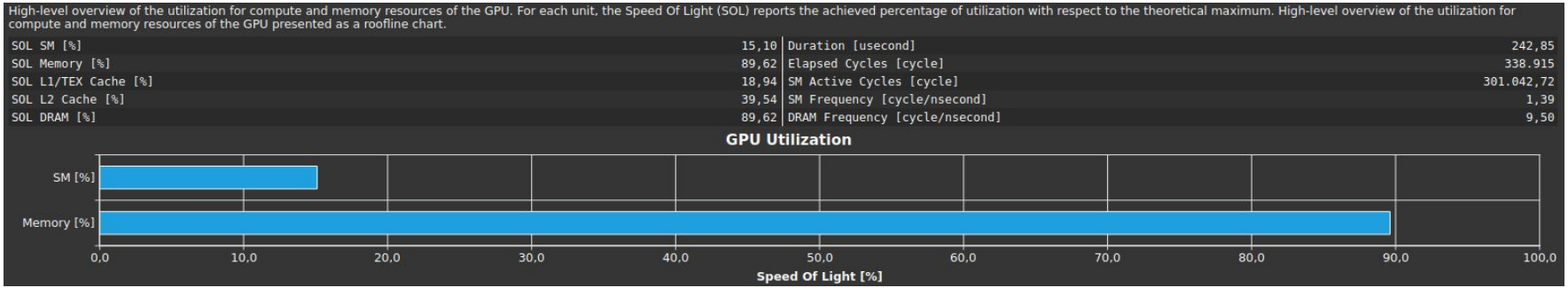
Necessario per generare file per la visualizzazione grafica.
Rimuovere per visualizzazione a terminale

```
$ 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 µs

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.

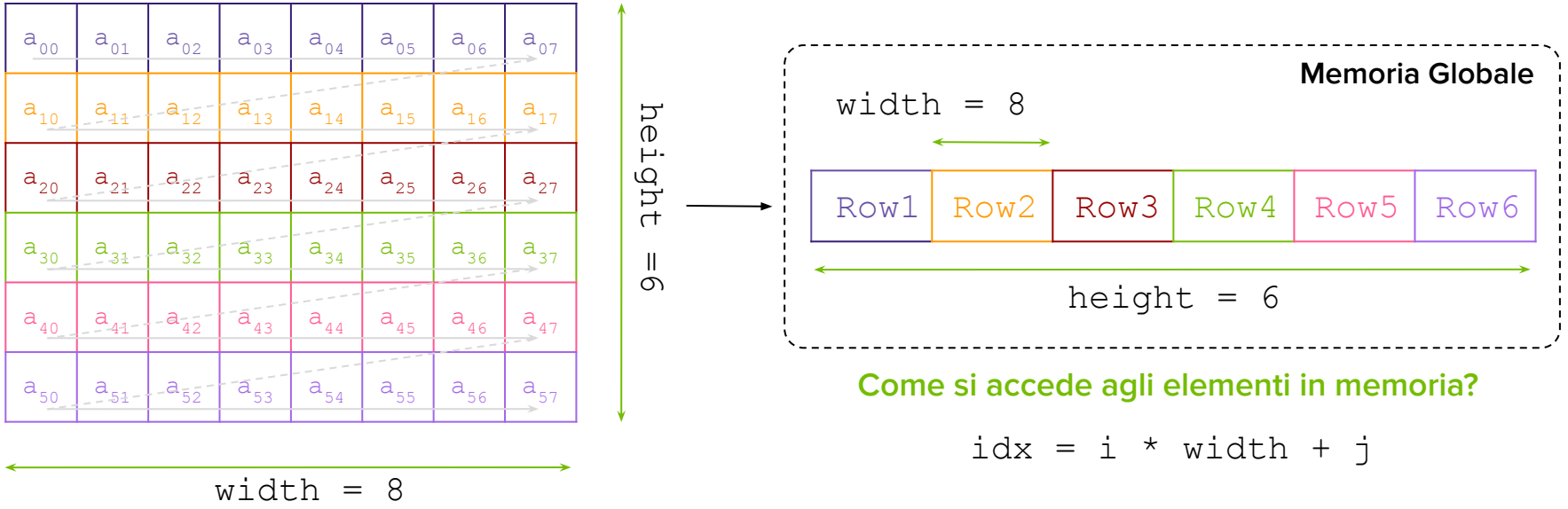
Panoramica del CUDA Programming Model

- **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)

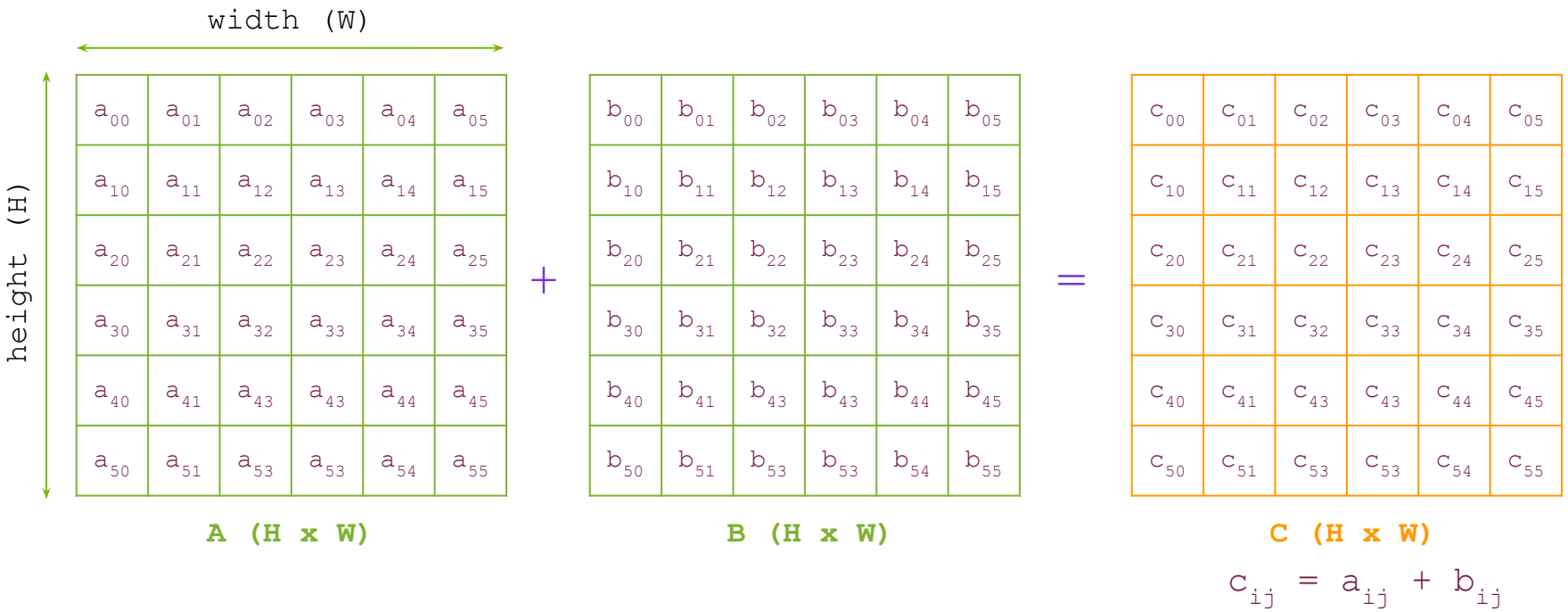


Come si accede agli elementi in memoria?

```
idx = i * width + j
```

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

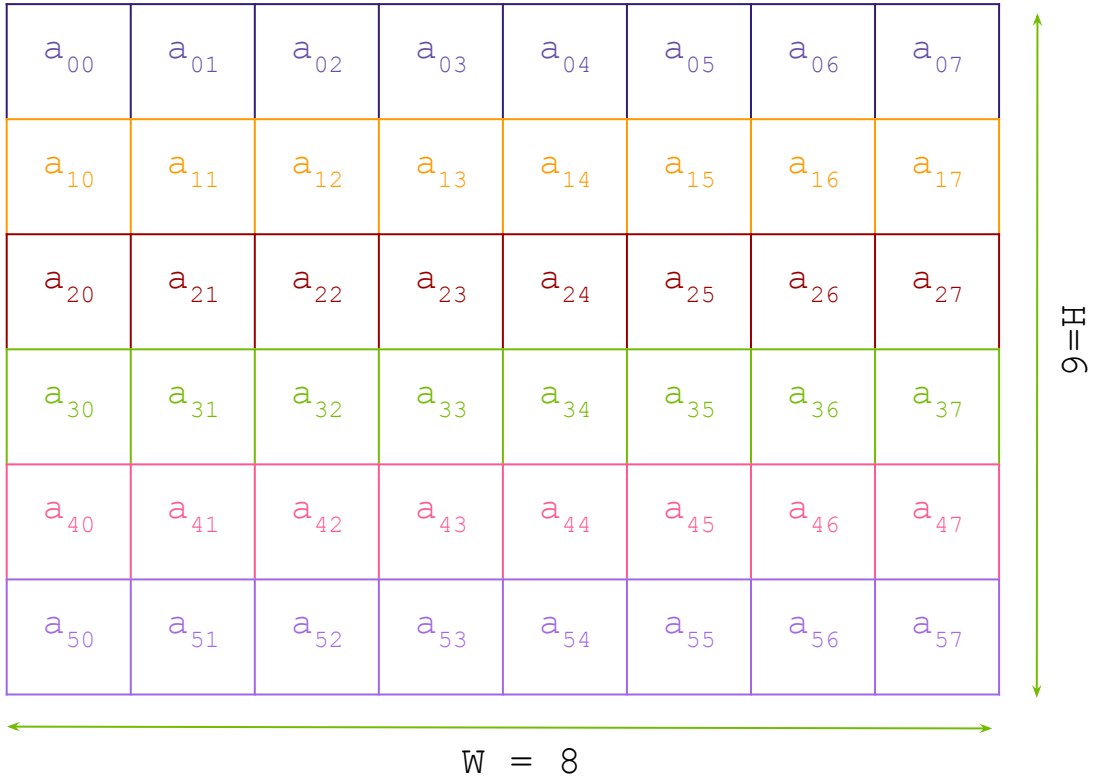
- Le matrici vengono linearizzate in memoria, quindi ogni elemento della matrice 2D deve essere mappato a un **indice lineare**: `idx = i * width + j` , dove `width` è il numero di colonne della matrice e `(i, j)` sono le coordinate dell'elemento.

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**?



Suddivisione

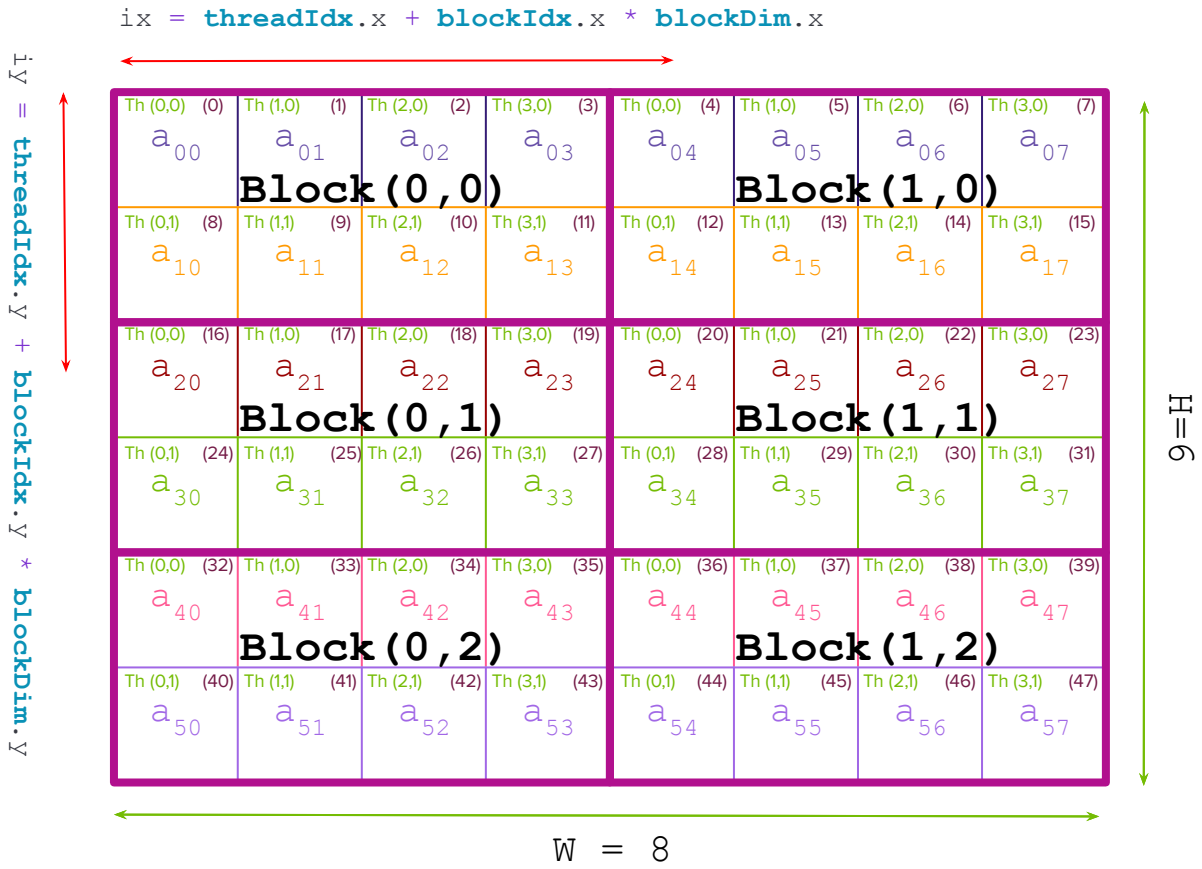
- La matrice può essere suddivisa in sottoblocchi di **dimensioni arbitrarie**.
- La scelta delle dimensioni dei blocchi influenza le **prestazioni**.

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).

Calcolo dell'Indice Global - 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 2x3 (`gridDim.x = 2, gridDim.y = 3`)
- Ogni blocco è di dimensione 4x2, 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
```

Confronto: Somma di Matrici in C vs CUDA C

Codice C Standard

```c
// 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

```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
}}
```

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)

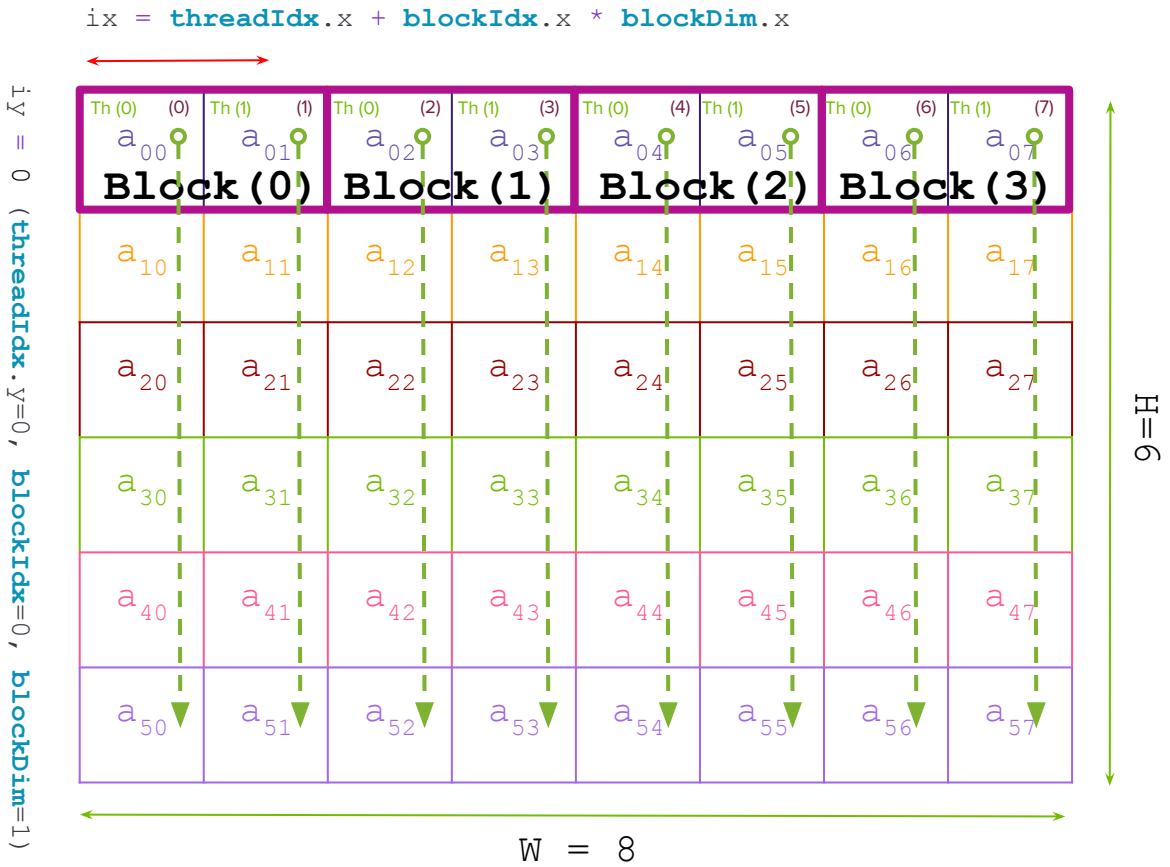
Osservazioni

- 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

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

Metodo Basato su Coordinate



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`

  `idx = 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
       }}}
```

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*

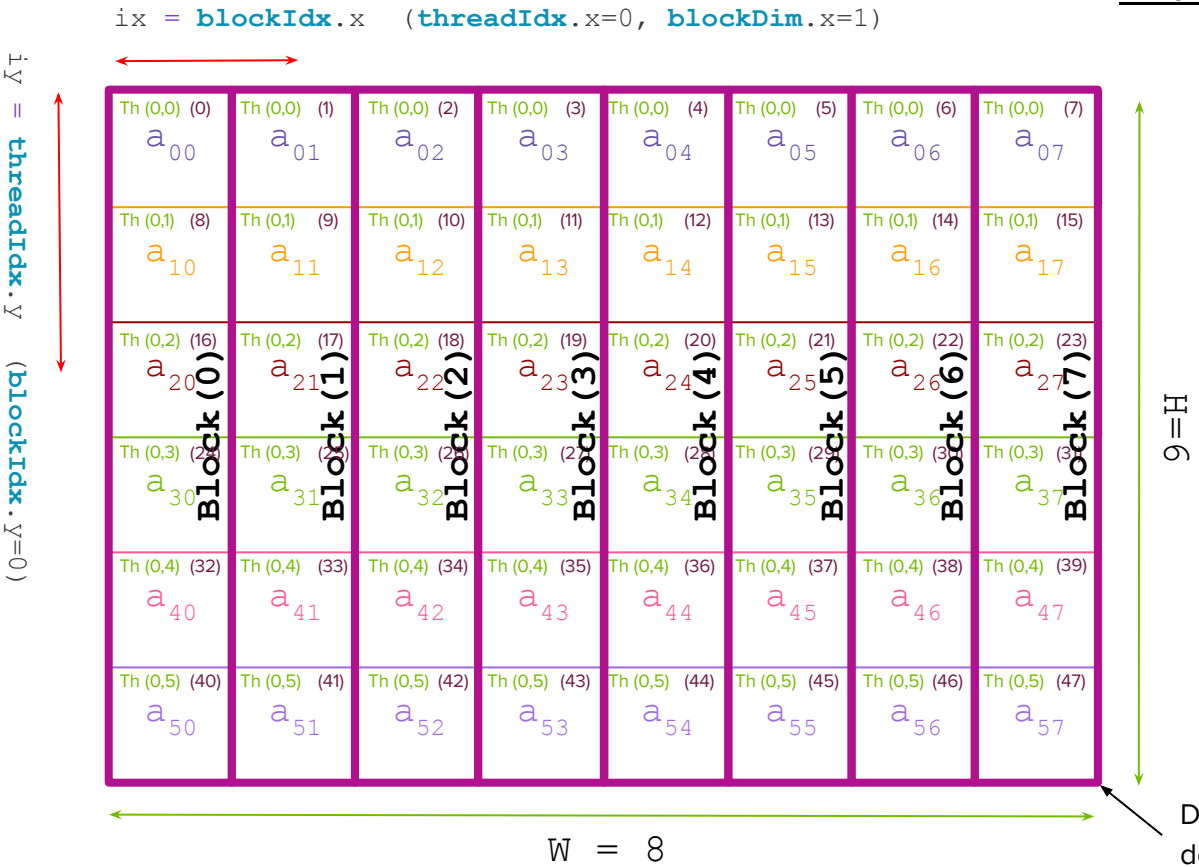
Se aumentassimo il numero di righe?

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)
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**.

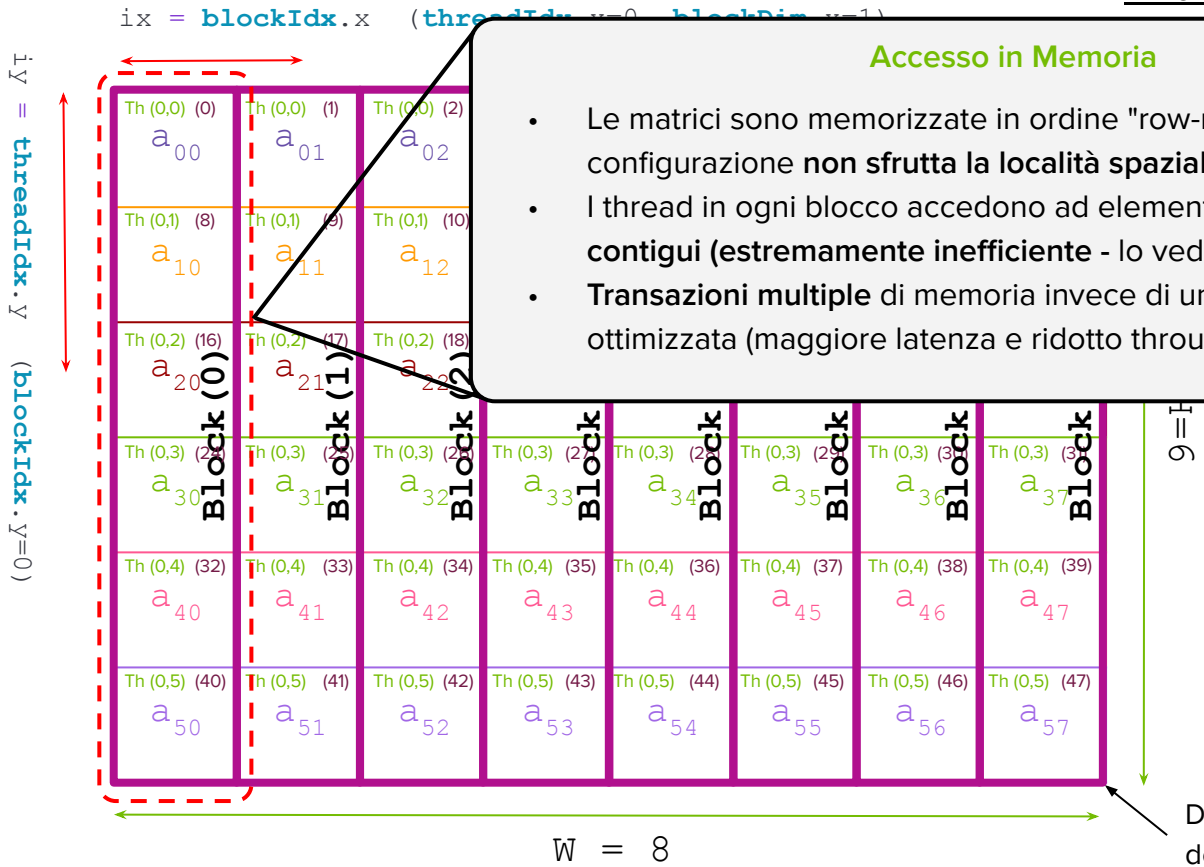
```
idx = iy * W + ix
```

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`).
figurazione 2D e
`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**.

```
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 degenere)

```
__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
    }
}
```

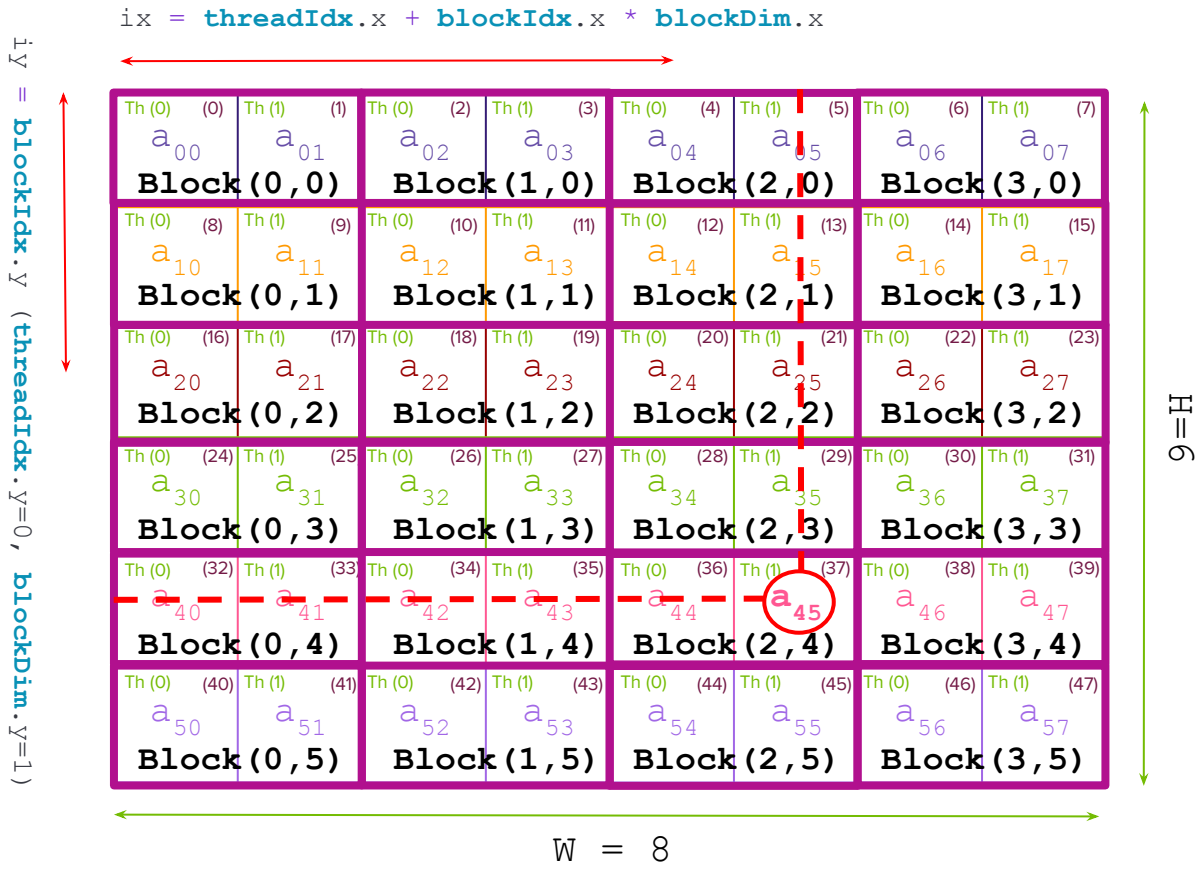
Griglia 1D e Blocchi 2D - Confronto fra Diverse Configurazioni

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 (degenere)					
(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)	×	×	RTX 3090 (GPU)
(1024,16384)	16384	(1,1024)	1,21*	28,03x	RTX 3090 (GPU)
(2048,16384)	16384	(1, ×)	×	×	RTX 3090 (GPU)
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 4x6 (`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)

1. **Indice x** nella matrice
   - `ix = 1 + 2 * 2 = 5`
2. **Indice y** nella matrice
   - `iy = 0 + 4 * 1 = 4`
3. **Indice lineare**
   - **idx** = iy*W+ix=37 (4, 8, 5)

L'indice **37** corrisponde all'elemento a_{24}

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	(1,16384) **(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** (degenere) 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)

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

Immagine Grayscale

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

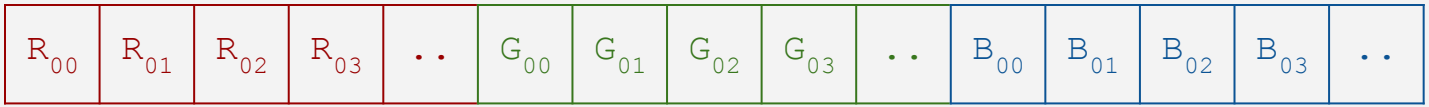
Memorizzazione Lineare di Immagini RGB in CUDA

- 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.

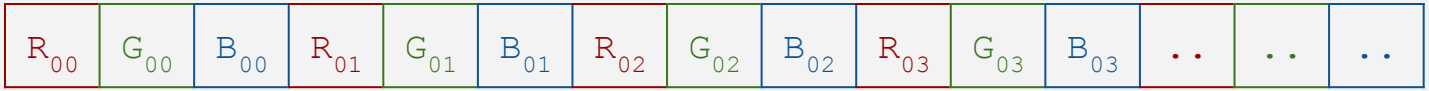
Approccio di Memorizzazione (Caso RGB)

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



2. **Interleaved** (più comune):
   - I valori R, G, B per ogni pixel sono memorizzati consecutivamente



Accesso agli Elementi dell'Immagine

RGB

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

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

  ```
  baseIndex = (i * width + j) * 3
  ```

- **Accesso ai canali:**
  - **R**: `baseIndex`
  - **G**: `baseIndex + 1`
  - **B**: `baseIndex + 2`

Grayscale

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

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

  ```
  baseIndex = i * 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

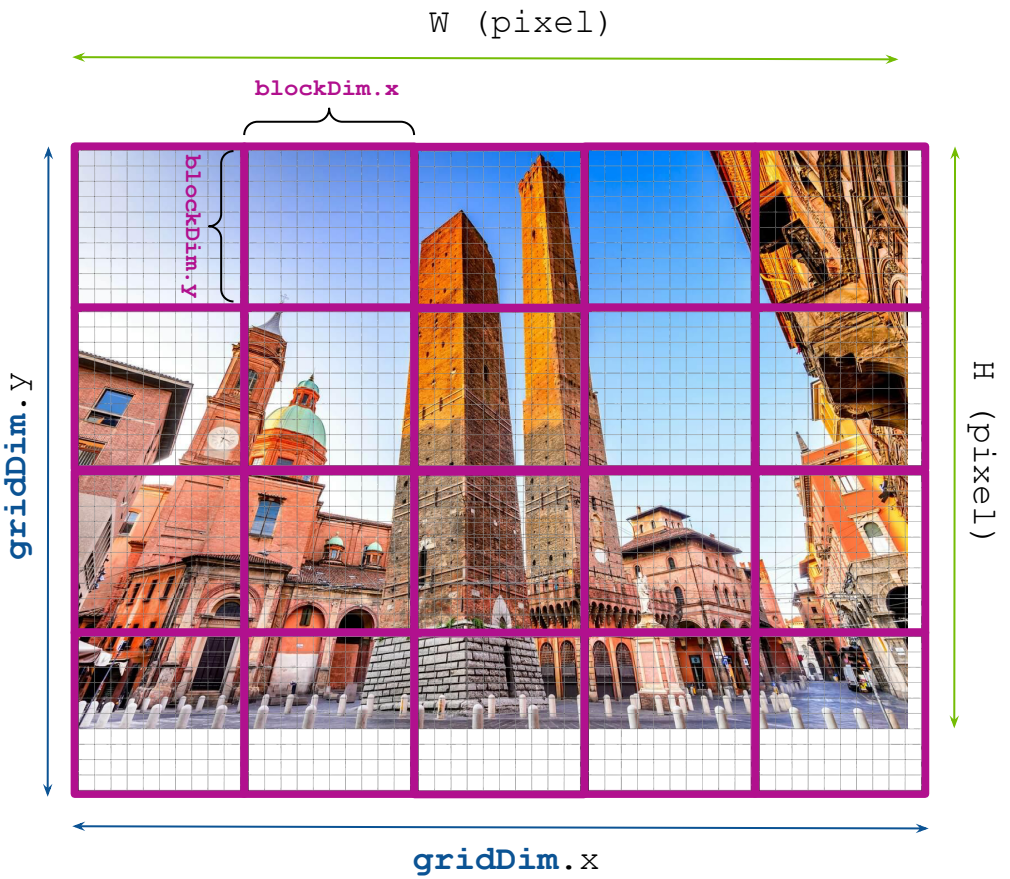
Formula: Gray = 0.299R + 0.587G + 0.114B (per pixel)

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 + 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 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
  }
}
```

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 `(width - i - 1, j)`.
  - Per un **flip verticale**, la nuova posizione è calcolata come `(i, 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

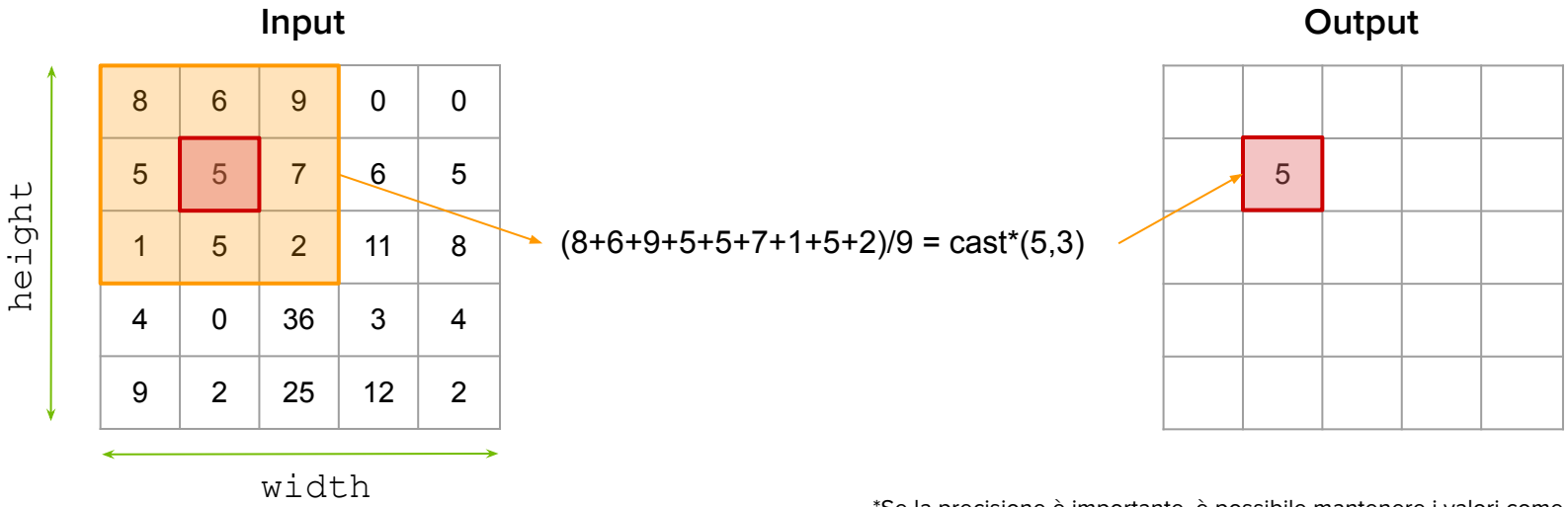
Blurred Image (`window_size=25`)

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

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: 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 **sovrapposizione** 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.

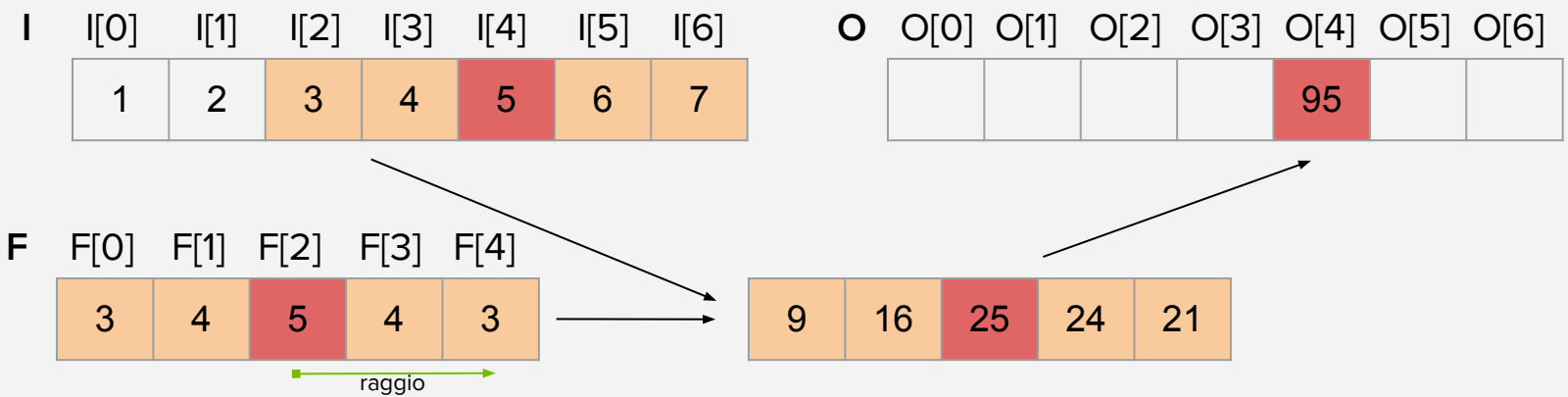
Convoluzione 2D

- 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.

Esempio di Convoluzione 1D

Descrizione

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




```
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
```

Calcolo di O[4]

Perché la Convoluzione si Adatta al Calcolo Parallelo

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.

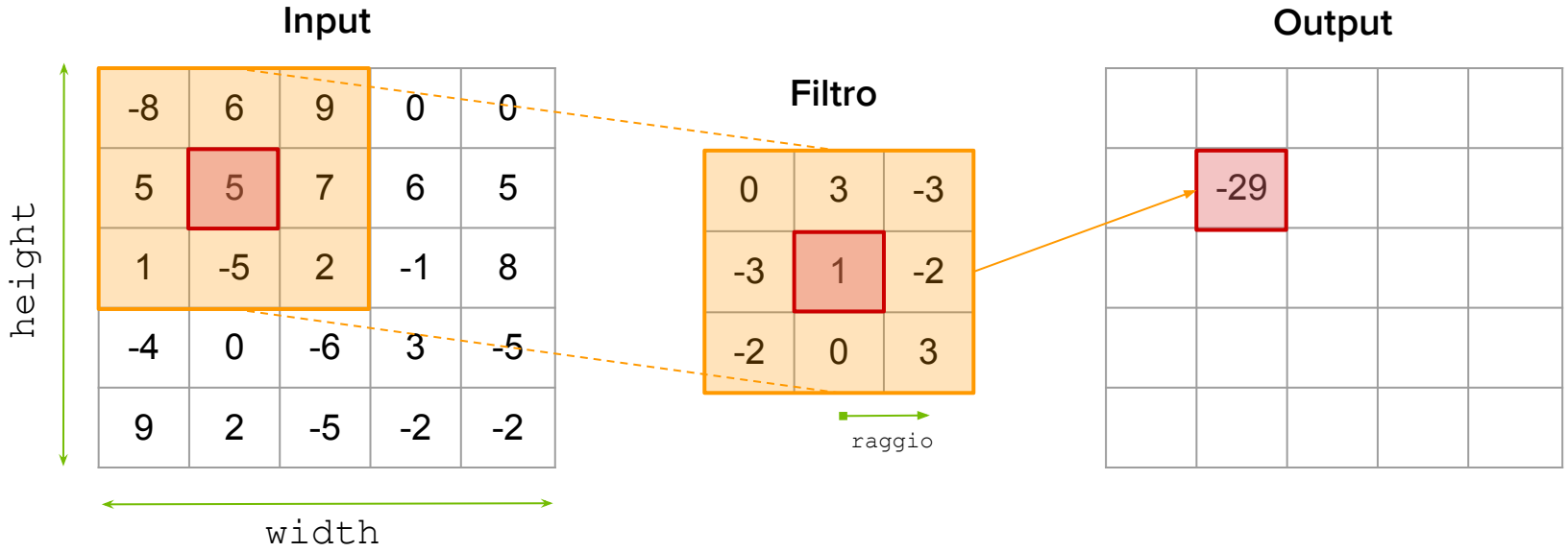
CUDA Convoluzione 1D: Soluzione (non ottimale)


```
__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]...I[4,4]).
- **Filtro (F):** Matrice di 9 elementi (F[0,0]...F[2,2]).
- **Output (O):** Matrice risultante dalla convoluzione di I con F.



CUDA Convoluzione 2D: Soluzione (non ottimale)


```
__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
    }
}
```

Riferimenti Bibliografici

Testi Generali

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