Sistemi di Elaborazione Accelerata

Author: Bumma Giuseppe
Professors: Mattoccia Stefano, Tosi Fabio

Università degli studi di Bologna
Ingegneria Informatica Magistrale

2025/2026

The preface to your notes

Contents

Chapter 1
Introduzione

1.1 Perché Scegliere la Piattaforma CUDA?

Dominio di Mercato e Standard Industriale

Standard de facto per calcolo parallelo e GPU, ampiamente supportata da software e librerie come TensorFlow e PyTorch. Le GPU NVIDIA sono prevalenti in HPC, AI e simulazione scientifica.

Prestazioni Elevate

CUDA consente di sfruttare la potenza delle GPU NVIDIA per eseguire milioni di thread simultaneamente, migliorando significativamente le prestazioni rispetto ai processori CPU tradizionali.

Ampia Documentazione e Risorse

NVIDIA fornisce una documentazione dettagliata e risorse pratiche, mentre la comunità attiva consente il supporto e la condivisione di conoscenze tra sviluppatori.

Facilità d'Uso

CUDA estende i linguaggi di programmazione C, C++, e Fortran, permettendo agli sviluppatori di utilizzare sintassi e concetti già noti.

Versatilità

È utilizzato in vari campi, dalla grafica 3D alla simulazione scientifica, dall'elaborazione video al deep learning, rendendolo una scelta flessibile per molti progetti.

Ecosistema Ricco

CUDA offre un ampio set di librerie ottimizzate (cuBLAS, cuDNN, Thrust, etc.) e strumenti di sviluppo per facilitare l'ottimizzazione e il debugging.

1.2 Cos'è il CUDA Toolkit?

- Il CUDA Toolkit è un insieme completo di strumenti di sviluppo fornito da NVIDIA per creare applicazioni accelerate tramite GPU.
- È essenziale per lo sviluppo di applicazioni CUDA, poiché fornisce tutti gli strumenti necessari per scrivere, compilare e ottimizzare codice CUDA.

1.2.1 Componenti chiave del CUDA Toolkit

Driver NVIDIA

- Fondamento Invisibile: Essenziali per CUDA, ma gli sviluppatori raramente interagiscono direttamente con essi.
- Ruolo: Funzionano da ponte tra il sistema operativo e la GPU, gestendo l'hardware, il caricamento del codice e il trasferimento dati tra CPU e GPU.
- Installazione: Necessari per CUDA e di solito installati separatamente.
- Compatibilità: Devono essere compatibili sia con la versione del CUDA Toolkit utilizzata che con la GPU in uso.

CUDA Runtime / CUDA Driver API (Application Programming Interface)

- Gli sviluppatori possono scegliere tra due interfacce per interagire con la GPU:
  - CUDA Runtime API: Livello di astrazione più alto, più semplice da utilizzare.
  - CUDA Driver API: Livello di astrazione più basso, offre un controllo più granulare sulle operazioni.

Compilatore CUDA (NVIDIA CUDA Compiler - `nvcc`)

- Traduzione del codice: Compila il codice CUDA, scritto in linguaggi come C, C++ e Fortran, in un formato eseguibile dalle GPU NVIDIA.
- Fasi:
  - Separazione: Separa il codice destinato alla CPU da quello per la GPU.
  - Compilazione: Compila il codice GPU in PTX (linguaggio intermedio) o direttamente in codice macchina per l'architettura GPU target (tenendo conto della Compute Capability e della CUDA Version utilizzata).
  - Linking: Combina il codice CPU e GPU con le librerie CUDA per creare l'applicazione finale.

Librerie CUDA

- Le librerie forniscono implementazioni ottimizzate e parallele di operazioni comuni, così gli sviluppatori non devono reinventare algoritmi complessi da zero.
- Esempi:
  - cuBLAS: Algebra lineare (operazioni su matrici e vettori).
  - cuFFT: Fast Fourier Transform (analisi di segnali, elaborazione di immagini).
  - cuDNN: Primitive per deep neural networks (convoluzione, pooling, attivazione).
  - cuRAND: Generazione di numeri casuali (simulazioni Monte Carlo, crittografia).
  - cuSPARSE: Operazioni su matrici sparse (risoluzione di sistemi lineari sparsi).
  - Thrust: Algoritmi paralleli generici (ordinamento, ricerca, trasformazioni) su GPU.

Esempi di Codice (CUDA Samples)

- Utilità: Forniscono implementazioni concrete di algoritmi e applicazioni comuni che utilizzano CUDA.
- Scopo: Aiutano gli sviluppatori a imparare le best practice di programmazione CUDA e a iniziare rapidamente nuovi progetti.

1.2.1.1 Strumenti di Debugging e Profiling

- Nsight Systems:
  - Profiling a livello di sistema. Offre una visione d'insieme del comportamento dell'applicazione su CPU e GPU, evidenziando eventuali colli di bottiglia.
- Nsight Compute:
  - Profiling approfondito della GPU. Permette di analizzare le prestazioni dei kernel CUDA in dettaglio, identificando aree di ottimizzazione per la memoria e l'utilizzo dei core.
- CUDA-GDB:
  - Debugging a riga di comando. Permette di eseguire il debug del codice CUDA a livello di sorgente, sia sulla CPU che sulla GPU. Supporta breakpoint, ispezione di variabili e stack trace su thread GPU.
- NVIDIA Visual Profiler (NVVP) [non più supportato]:
  - Offre una rappresentazione grafica timeline delle attività della CPU e della GPU, facilitando l'identificazione dei colli di bottiglia. Oggi le sue funzionalità sono state integrate in Nsight.

1.2.1.2 Relazione tra CUDA Toolkit e CUDA Version

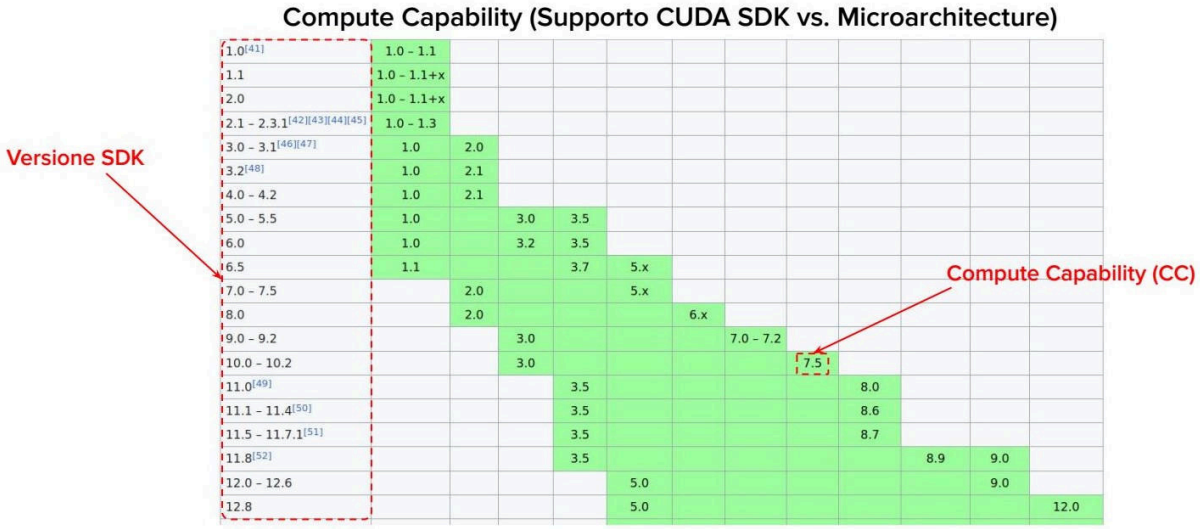
- Il CUDA Toolkit viene rilasciato in versioni numerate (es. CUDA 12.6, CUDA 11.0). La versione installata determina la CUDA Version in uso sul sistema.
- Ogni nuova CUDA Version include aggiornamenti software e bugfix, nuovi strumenti di sviluppo e librerie, supporto per le più recenti architetture GPU NVIDIA.
- Aggiornare il CUDA Toolkit alla versione più recente garantisce compatibilità con le GPU più recenti e l'accesso alle ultime ottimizzazioni, migliorando performance e stabilità del codice.

1.2.1.3 Retrocompatibilità del CUDA Toolkit

- Supporto per GPU Precedenti: Le nuove versioni del CUDA Toolkit mantengono la compatibilità con GPU più vecchie, anche se non tutte le funzionalità più recenti sono disponibili su queste architetture.
- Limitazioni: Alcune funzionalità avanzate introdotte nelle nuove versioni potrebbero non essere supportate su GPU datate, e il supporto per architetture molto vecchie può essere gradualmente ridotto o deprecato.
- Compatibilità del Codice: In generale, il codice scritto con versioni precedenti del Toolkit può essere eseguito su versioni più recenti, ma può richiedere piccoli adattamenti.

1.3 CUDA Compute Capability (CC)

- La Compute Capability (CC) è un numero in formato X.Y che identifica le caratteristiche e le capacità di una GPU NVIDIA in termini di funzionalità supportate e limiti hardware
- Il numero X (principale) indica la generazione dell'architettura (ad es. Turing, Ampere), mentre Y (secondario) indica una revisione della stessa architettura, con piccoli miglioramenti o varianti hardware.



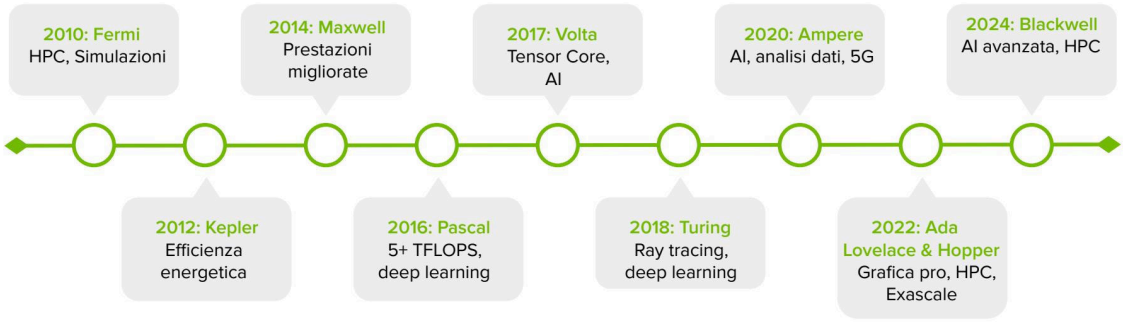
1.3.1 Relazione tra Compute Capability (CC) e CUDA Version

- **Compute Capability (CC)**
  - Indica le caratteristiche e i limiti hardware di una GPU.
  - È indipendente dalla CUDA Version, ma la CUDA Version deve supportare la Compute Capability della GPU per sfruttare appieno le sue capacità.
- **CUDA Version**
  - Determina quali funzionalità software, API e librerie sono disponibili per lo sviluppo.

- Può supportare più Compute Capabilities: una singola versione di CUDA Toolkit può essere compatibile con diverse generazioni di GPU (es. CUDA 11.x supporta Volta, Turing, Ampere).

1.4 Evoluzione delle Architetture GPU NVIDIA

- Progressione Tecnologica: Da Fermi a Blackwell, ogni generazione ha portato significativi avanzamenti nelle capacità di calcolo e nell'efficienza energetica.
- Adattamento al Mercato: L'evoluzione riflette il passaggio da un focus su grafica e HPC a un'enfasi crescente su AI, deep learning e calcolo ad alte prestazioni.



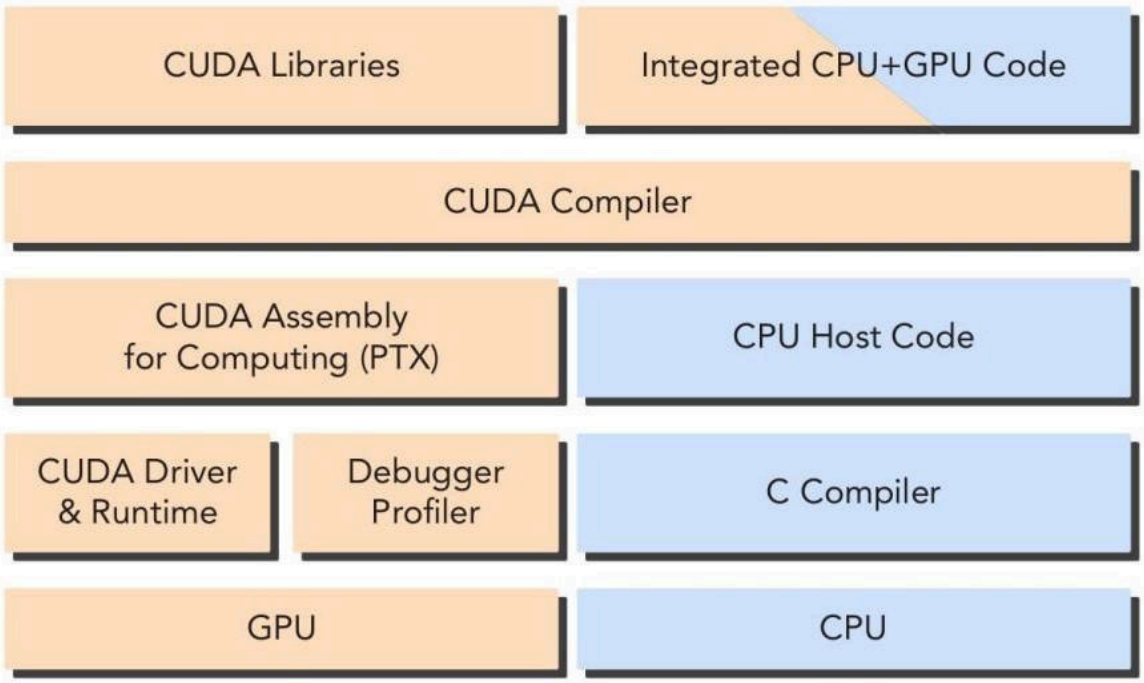
1.5 Anatomia di un Programma CUDA

1.5.1 Struttura del Codice Sorgente

- File Sorgente: Estensione `.cu`
- Codice host + Codice device

1.5.2 Componenti Principali

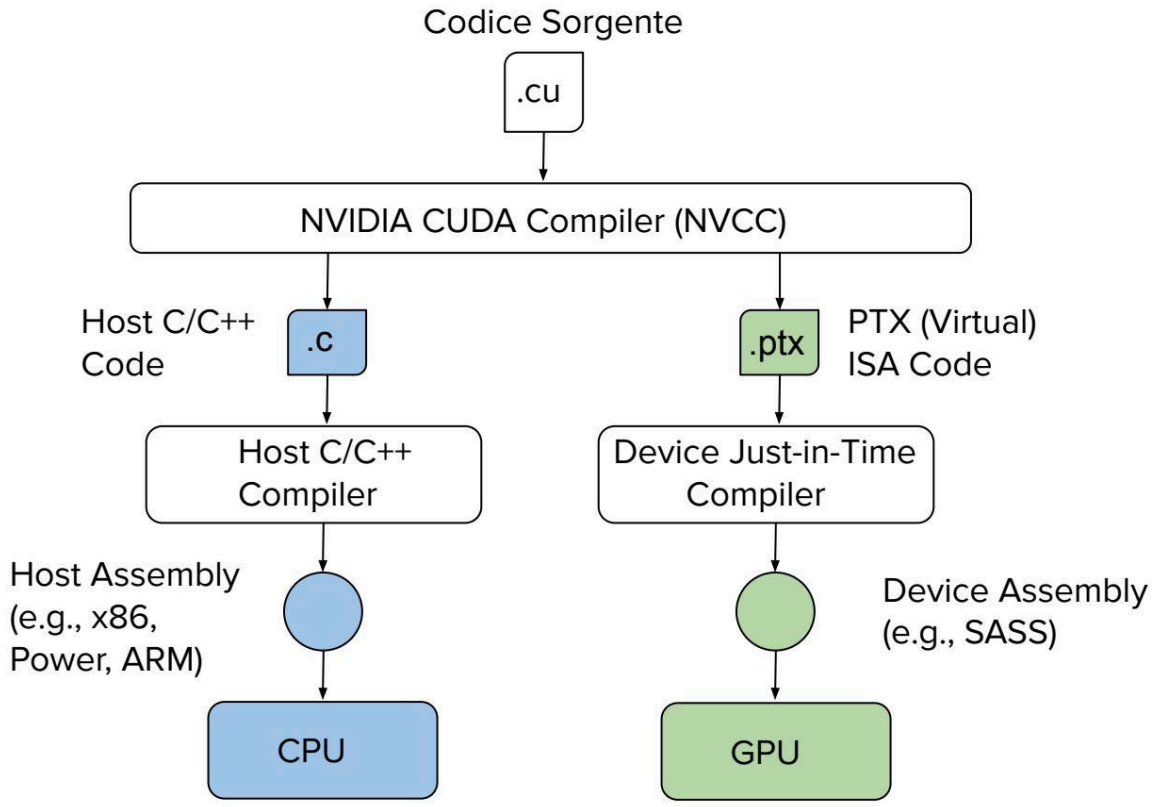
- Codice Host
  - Codice C/C++ eseguito sulla CPU
  - Gestisce la logica dell'applicazione
  - Alloca memoria sulla GPU
  - Trasferisce dati tra CPU e GPU
  - Lancia i kernel GPU
  - Gestisce la sincronizzazione
- Codice Device:
  - Codice CUDA C eseguito sulla GPU
  - Contiene i kernel (funzioni parallele)
  - Esegue operazioni computazionali intensive in parallelo



1.5.3 Flusso di Compilazione

- Separazione del codice
  - Il compilatore NVIDIA CUDA Compiler (`nvcc`) separa il codice device dal codice host
- Compilazione del codice host
  - È codice C standard o C++
  - Compilato con compilatori C tradizionali (gcc)
- Compilazione del codice device
  - Compilato da `nvcc` in formato intermedio PTX (Parallel Thread Execution) .
  - Il driver NVIDIA poi traduce il PTX in codice macchina specifico per la GPU (SASS - Streaming Assembly) al momento dell'esecuzione, usando un compilatore Just-In-Time (JIT).
- Linking
  - Aggiunta delle librerie runtime CUDA
  - Supporto per chiamate ai kernel e manipolazione esplicita della GPU
- Eseguibile finale
  - File unico con codice per CPU e GPU

1.6 Compilazione di un Programma CUDA



1.7 Hello World in CUDA C

1.7.1 Passo 1: Creare il File Sorgente

Nome file: `hello.cu`

1.7.1.1 Passo 2: Scrivere il codice

Codice C e CUDA C a confronto

Linguaggio C

```c
include <stdio.h>
  int main(void) {
    printf("Hello World from CPU!\n");
    return 0;
  }
```

Linguaggio CUDA C CUDA

```cuda
include <stdio.h>
__global__ void helloFromGPU()
{
  printf("Hello World from GPU thread
  %d!\n", threadIdx.x);
}
int main()
{
  // Lancio del kernel
  helloFromGPU<<<1, 10>>>();
  // Attendere che la GPU finisca
  cudaDeviceSynchronize();
  return 0;
}
```

Linguaggio CUDA C

1.8 Hello World in CUDA C - Analisi

- Definizione kernel GPU:
  - `__global__`: Qualificatore CUDA che indica una funzione eseguita sulla GPU, ma chiamata dalla CPU. (In C standard non esiste).
  - threadIdx.x: Variabile built-in CUDA che fornisce l'ID univoco del thread all'interno del blocco.

```cuda
__global__ void helloFromGPU(){
 printf("Hello World from GPU thread %d!\n", threadIdx.x);}
```

- Funzione `main`: Punto di ingresso del programma, eseguito sulla CPU come in C standard.
- Lancio del kernel
  - `<<<1, 10>>>`: Configurazione di esecuzione (1 blocco, 10 thread). Avvia 10 istanze parallele del kernel sulla GPU.
- Sincronizzazione GPU-CPU
  - `cudaDeviceSynchronize()`: la CPU attende che la GPU completi tutte le operazioni prima di proseguire/terminare.

```cuda
int main(){
  helloFromGPU<<<1, 10>>>();
  cudaDeviceSynchronize();
  return 0;
}
```

1.9 Hello World in CUDA C

1.9.0.1 Passo 3: Compilazione

- Salvare il codice nel file `hello.cu`
  - Un file `.cu` può contenere sia codice C/C++ standard che codice CUDA C
  - Questo permette di mescolare codice per CPU e GPU nello stesso file
- Compilare il programma usando il compilatore CUDA `nvcc`

- `nvcc` può gestire sia il codice C/C++ standard che le estensioni CUDA
- `nvcc` separa internamente il codice host e device, compilando ciascuno in modo appropriato

```shell
$ nvcc hello.cu -o hello
```

1.10 Hello World in CUDA C

1.10.0.1 Passo 4: Esecuzione

Eseguire il file eseguibile:

```
$ ./hello
```

Output:

Variante Linguaggio C

```
Hello World from CPU!
```

Variante Linguaggio CUDA C

```
Hello World from GPU thread 0!
Hello World from GPU thread 1!
Hello World from GPU thread 2!
Hello World from GPU thread 3!
Hello World from GPU thread 4!
Hello World from GPU thread 5!
Hello World from GPU thread 6!
Hello World from GPU thread 7!
Hello World from GPU thread 8!
Hello World from GPU thread 9!
```

1.11 Ottenere Informazioni sulla GPU tramite API CUDA

1.11.1 Utilizzo delle API CUDA

CUDA fornisce API per ottenere informazioni dettagliate sulle GPU direttamente dal codice

```cuda
int main() {
  cudaDeviceProp prop;
  cudaGetDeviceProperties(&prop, 0); // Ottieni proprietà del dispositivo 0
  printf("Nome Dispositivo: %s\n", prop.name);
  printf("Memoria Globale Totale: %.0f MB\n", prop.totalGlobalMem / 1024.0 / 1024.0);
  printf("Clock Core: %d MHz\n", prop.clockRate / 1000);
  printf("Compute Capability: %d.%d\n", prop.major, prop.minor);
  // Stampa di altre proprietà
  return 0;
}
```

- Utilizzo della struttura `cudaDeviceProp` per memorizzare le proprietà del device
- Utilizzo della funzione `cudaGetDeviceProperties` per ottenere le proprietà del device specificato

- Accesso alle proprietà della GPU, come nome, memoria totale, clock core e compute capability
- Per una lista completa delle proprietà disponibili, consulta la documentazione ufficiale di CUDA(https://docs.nvidia.com/cuda/cuda-runtime-api/structcudaDeviceProp.html)

1.12 Ottenere Informazioni sulla GPU tramite API CUDA

1.12.0.1 Esempio di Output

```
CUDA System Information:
CUDA Driver Version: 12.2
CUDA Runtime Version: 11.3
Numero di dispositivi CUDA: 2
Dispositivo 0: NVIDIA GeForce RTX 3090
1. Compute Capability: 8.6
2. Memoria Globale Totale: 23.69 GB
3. Numero di Multiprocessori: 82
4. Clock Core: 1695 MHz
5. Clock Memoria: 9751 MHz
6. Larghezza Bus Memoria: 384 bit
7. Dimensione Cache L2: 6144 KB
8. Memoria Condivisa per Blocco: 48 KB
9. Numero Massimo di Thread per Blocco: 1024
10. Dimensioni Massime Griglia: (2147483647, 65535, 65535)
11. Dimensioni Massime Blocco: (1024, 1024, 64)
12. Warp Size: 32
13. Memoria Costante Totale: 65536 bytes
14. Texture Alignment: 512 bytes
```

1.13 Cosa Significa Programmare in CUDA?

1.13.1 Pensare in Parallelo

- **Decomposizione del Problema:** Identificare le parti del problema che possono essere eseguite in parallelo per sfruttare al meglio le risorse della GPU.
- **Architettura della GPU:** Le GPU sono composte da migliaia di core in grado di eseguire thread in parallelo. CUDA fornisce gli strumenti per organizzare e gestire questi thread.
- **Scalabilità:** Progettare algoritmi che si adattano a diversi numeri di thread (e GPU).
- **Gerarchia di Thread:** Organizzare il lavoro in blocchi e griglie per massimizzare l'efficienza.
- **Gerarchia di Memoria:** Utilizzare strategicamente memoria globale, condivisa, locale e registri per ridurre i tempi di accesso.
- **Sincronizzazione:** Gestire la coordinazione tra thread e il trasferimento dati tra CPU e GPU senza conflitti.
- **Bilanciamento del Carico:** Distribuire il lavoro in modo uniforme fra thread per evitare colli di bottiglia.

1.13.2 Scrittura di codice in CUDA C

- CUDA estende C/C++ con costrutti specifici come `__global__`, `__shared__` e la sintassi `<<<...>>>` per lanciare kernel sulla GPU.

- Ogni kernel viene scritto come codice sequenziale, ma viene eseguito in parallelo da migliaia di thread, permettendo di pensare in modo semplice ma scalare.

1.13.2.1 Testi Generali

- Cheng, J., Grossman, M., McKercher, T. (2014). Professional CUDA C Programming. Wrox Pr Inc. (1^ edizione)
- Kirk, D. B., Hwu, W. W. (2022). Programming Massively Parallel Processors. Morgan Kaufmann (4^ edizione)

1.13.2.2 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/
- CUDA University Courses
  - https://developer.nvidia.com/educators/existing-courses=2
- An Even Easier Introduction to CUDA
  - https://developer.nvidia.com/blog/even-easier-introduction-cuda/

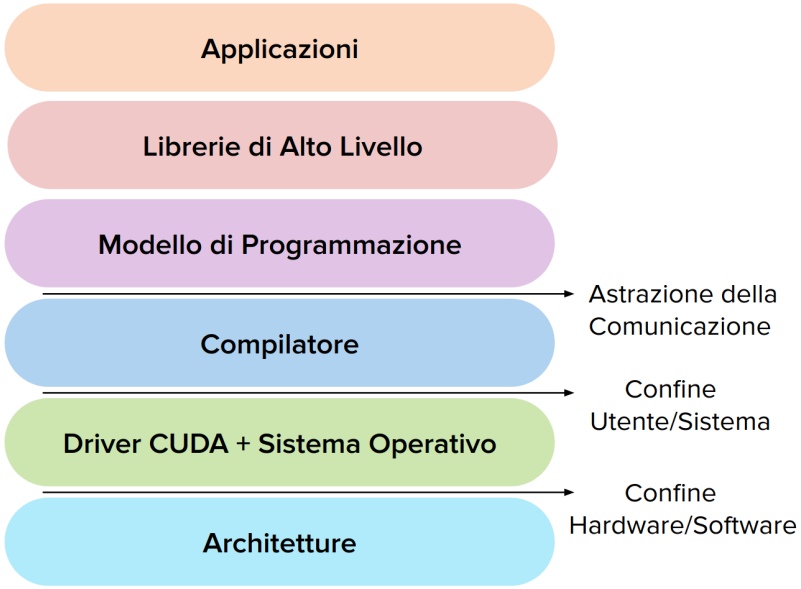
1.13.2.3 Materiale di Approfondimento

- Branch Education (canale YouTube con spiegazioni su GPU, ray tracing, hardware, ecc.)
  - https://www.youtube.com/@BranchEducation

Chapter 2

Modello di Programmazione CUDA

2.1 La Struttura Stratificata dell'Ecosistema CUDA



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

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

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

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

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

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

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

2.2 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.

2.3 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.

2.3.1 Livello Dominio

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

Chiave: Ottimizza la strategia di parallelizzazione.

2.3.2 Livello Logico

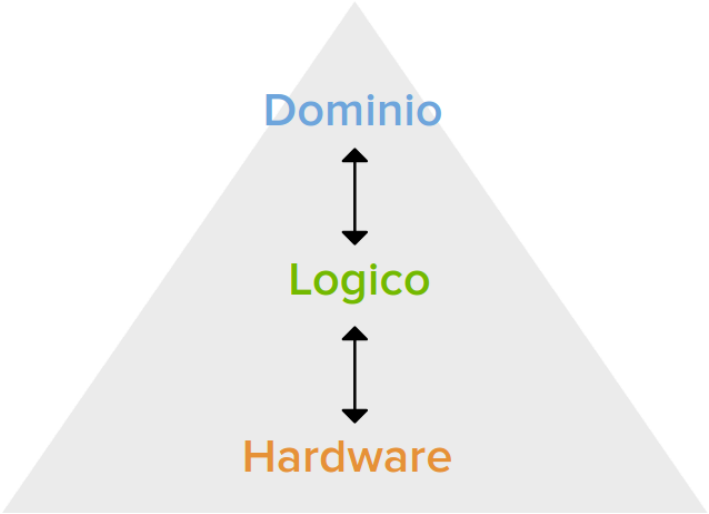
- Organizzazione e gestione dei thread.
- Implementazione della strategia di parallelizzazione.

Chiave: Massimizza l'efficienza del parallelismo.

2.3.2.1 Livello Hardware

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

Chiave: Sfrutta al meglio le risorse GPU.



2.4 Thread CUDA: L'Unità Fondamentale di Calcolo

2.4.1 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.

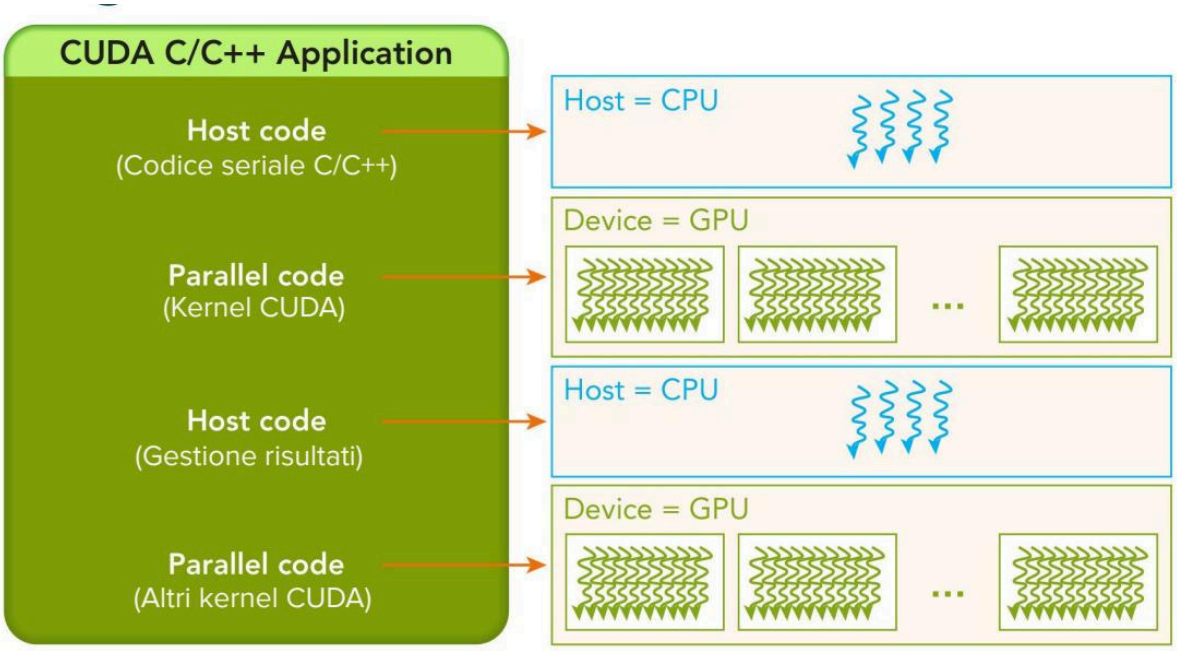
2.4.2 Cosa Fa un Thread CUDA?

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

Thread CUDA vs Thread CPU

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

2.5 Struttura di Programmazione CUDA



2.5.1 Caratteristiche Principali

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

2.6 Flusso Tipico di Elaborazione CUDA

1. Inizializzazione e Allocazione Memoria (Host)

   Preparazione dati e allocazione di memoria su CPU (host) e GPU (device).

2. Trasferimento Dati (Host → Device)

   Copia degli input dalla memoria host alla memoria device.

3. Esecuzione del Kernel (Device)

   La GPU esegue calcoli paralleli secondo la configurazione di griglia e blocchi.

4. Recupero Risultati (Device → Host)

   Copia dell'output dalla memoria device alla memoria host.

5. Post-elaborazione (Host)

   Analisi o elaborazione aggiuntiva dei risultati sulla CPU.

6. Liberazione Risorse

   Rilascio della memoria allocata su host e device.

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

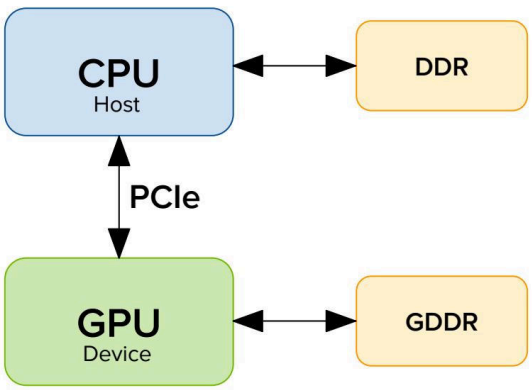
2.7 Gestione della Memoria in CUDA

2.7.1 Modello di Memoria CUDA

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

2.7.2 Caratteristiche PCIe

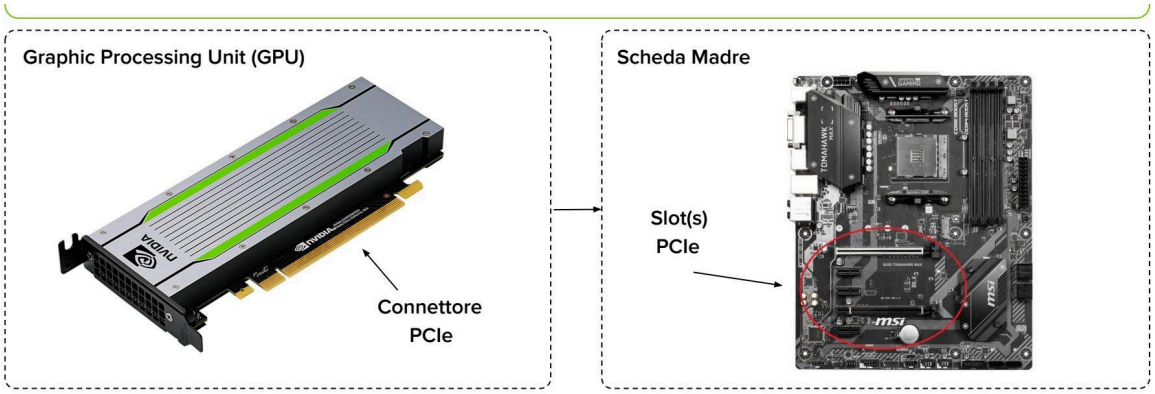
- **Lane:** Ogni lane (canale di trasmissione) è costituito da due coppie di segnali differenziali (quattro fili), una per ricevere (RX) e una per trasmettere (TX) dati.
- **Full-Duplex:** Trasmette e riceve dati simultaneamente in entrambe le direzioni.
- **Scalabilità:** La larghezza di banda varia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



2.8 Collegamento Fisico della GPU tramite PCIe

2.8.1 Connessione Fisica GPU

- La GPU si collega alla scheda madre attraverso uno slot PCI Express (PCIe).
- Il connettore, costituito da contatti metallici dorati sul bordo della scheda, si inserisce nello slot PCIe corrispondente.
- La maggior parte delle schede madri moderne ha uno o più slot PCIe, generalmente con almeno uno slot PCIe x16 destinato alla GPU.



2.9 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.

2.10 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

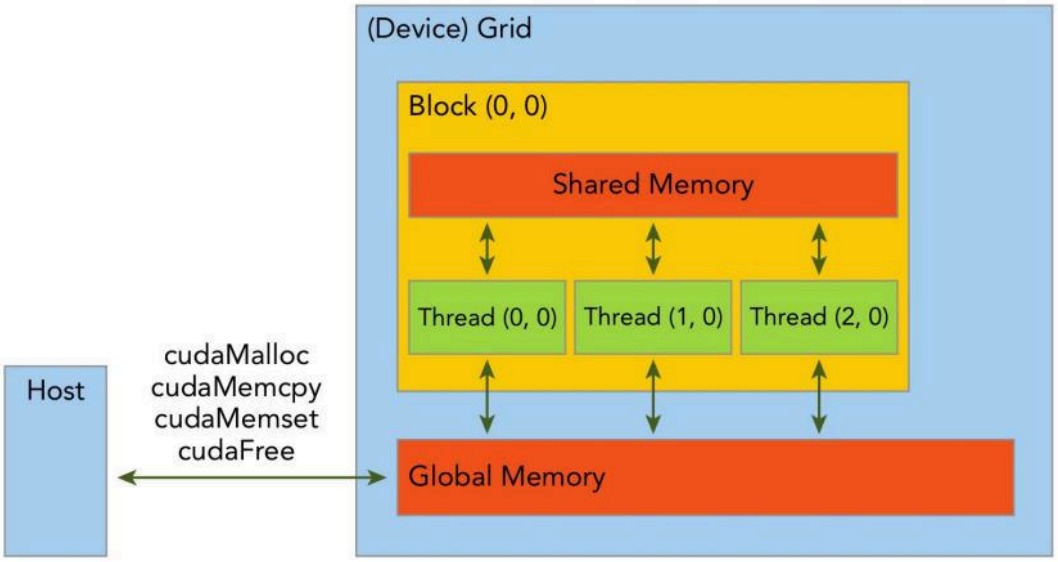
Shared Memory

- 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.



2.11 Allocazione della Memoria sul Device

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

Firma della Funzione (Documentazione Online)

```cuda
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.

`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)

```cuda
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.

2.11.1 Esempio di Allocazione di Memoria sulla GPU

Mostra come allocare memoria sulla GPU utilizzando `cudaMalloc`.

```cuda
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");}
```

2.12 Trasferimento Dati

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

Firma della Funzione (Documentazione Online)

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

Parametri

- `dst`: Puntatore alla memoria di destinazione.
- `src`: Puntatore alla memoria sorgente.

- `count`: Numero di byte da copiare.
- `kind`: Direzione della copia (cudaMemcpyKind).

Tipi di Trasferimento (kind)

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

Valore di Ritorno

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

Note importanti

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

2.12.1 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

```cuda
host_array = dev_ptr
```

invece di

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

Conseguenza dell'errore: Accesso a indirizzi non validi → possibile blocco o crash dell'applicazione.

Soluzione: La Unified Memory, introdotta in CUDA 6 e oggi ottimizzata, consente di usare un unico puntatore valido per CPU e GPU, con gestione automatica della migrazione dati (vedremo in seguito).

2.13 Deallocazione della Memoria sul Device

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

Firma della Funzione (Documentazione Online)

```cuda
cudaError_t cudaFree(void devPtr)
```

Parametri

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

Valore di Ritorno

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

Note Importanti

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

2.13.1 Esempio di Allocazione e Trasferimento Dati

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

```cuda
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);
}

// 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
```

2.14 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.

2.14.1 Struttura Gerarchica

1. **Grid (Griglia)**

- Array di thread blocks.
- È organizzata in una struttura 1D, 2D o 3D.
- Rappresenta l'intera computazione di un kernel.

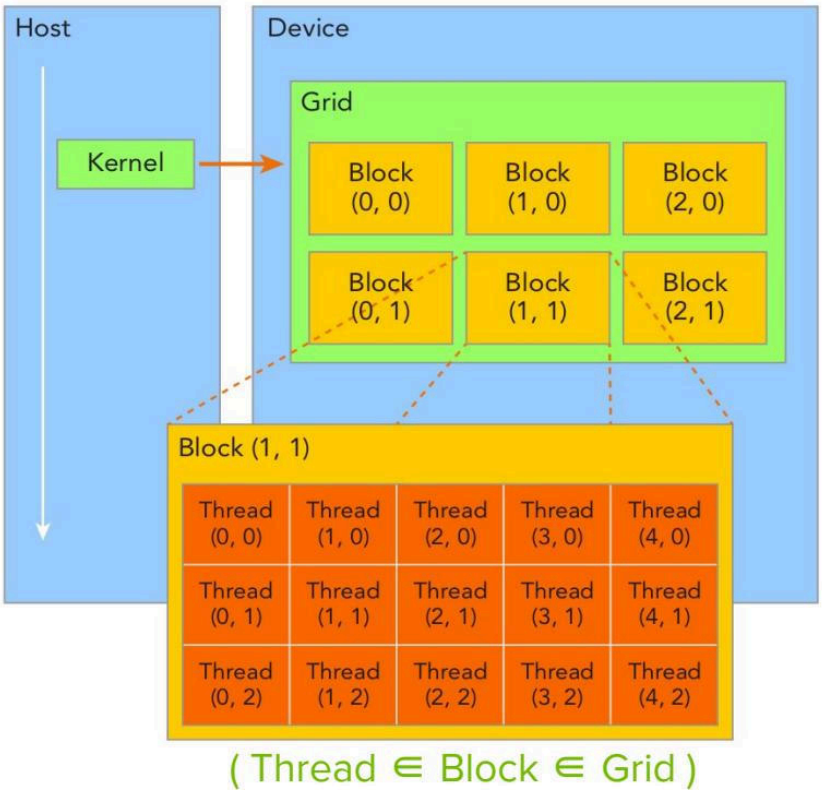
- Contiene tutti i thread che eseguono il singolo kernel.
- Condivide lo stesso spazio di memoria globale.

2. **Block (Blocco)**

- Un thread block è un gruppo di thread eseguiti logicamente in parallelo.
- Ha un ID univoco all'interno della sua griglia.
- I blocchi sono organizzati in una struttura 1D, 2D o 3D.
- I thread di un blocco possono sincronizzarsi (non automaticamente) e condividere memoria.
- I thread di blocchi diversi non possono sincronizzarsi direttamente (solo tramite memoria globale o kernel successivi)

3. **Thread**

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



2.14.2 Perché una Gerarchia di Thread?

Mappatura Intuitiva

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

Organizzazione e Ottimizzazione

Il programmatore può definire le dimensioni dei blocchi e della griglia per adattare l'esecuzione alle caratteristiche specifiche dell'hardware e del problema, ottimizzando l'utilizzo delle risorse.

Efficienza nella Memoria

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

Scalabilità e Portabilità

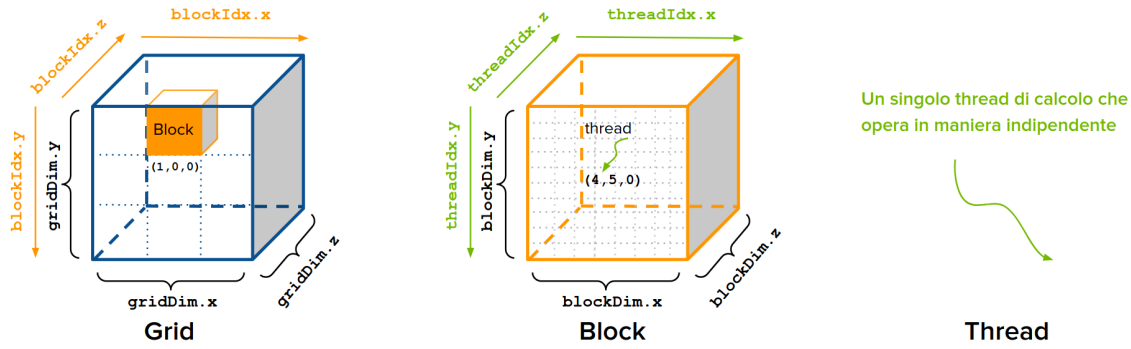
La gerarchia è scalabile e permette di adattare l'esecuzione a GPU con diverse capacità e numero di core. Il codice CUDA, quindi, risulta più portabile e può essere eseguito su diverse architetture GPU.

Sincronizzazione Granulare

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

2.15 Identificazione dei Thread in CUDA

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



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`

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

2.15.1 Dimensione delle Griglie e dei Blocchi

- La scelta delle dimensioni ottimali dipende dalla struttura dati del problema e dalle capacità hardware/risorse della GPU.

- Le dimensioni di griglia e blocchi vengono definite nel codice host prima del lancio del kernel.
- Sia le griglie che i blocchi utilizzano il tipo dim3 (lato host) con tre campi unsigned int. I campi non utilizzati vengono inizializzati a 1 e ignorati.
- 9 possibili configurazioni (1D, 2D, 3D per griglia e blocco) in tutto anche se in genere si usa la stessa per entrambi.

2.16 Struttura `dim3`

Definizione

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

Esempi CUDA

```
dim3 gridDim(256); // Definisce una griglia di 256x1x1 blocchi.
dim3 blockDim(512, 512);` // Definisce un blocco di 512x512x1 threads.
```

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

CUDA

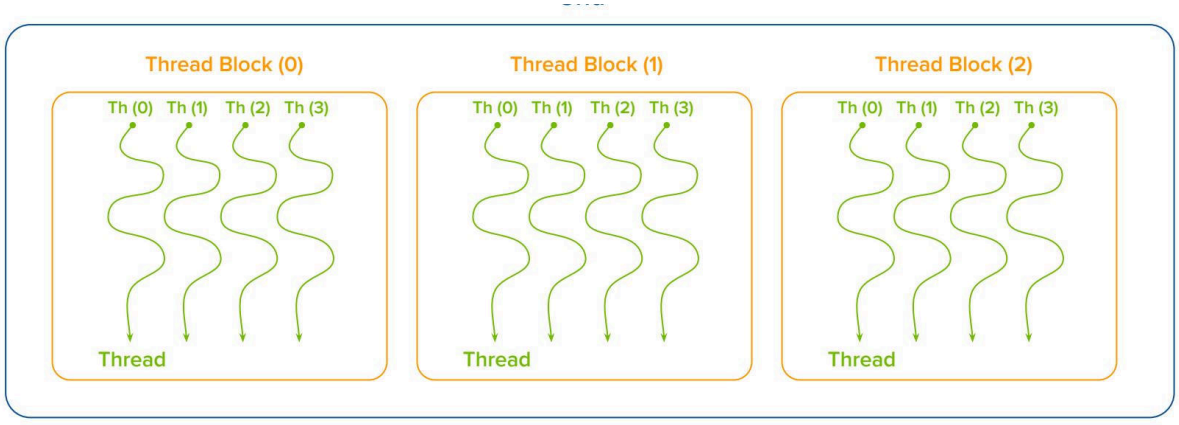
```
kernel_name<<<gridDim, blockDim>>>(...);
```

Codice Originale:

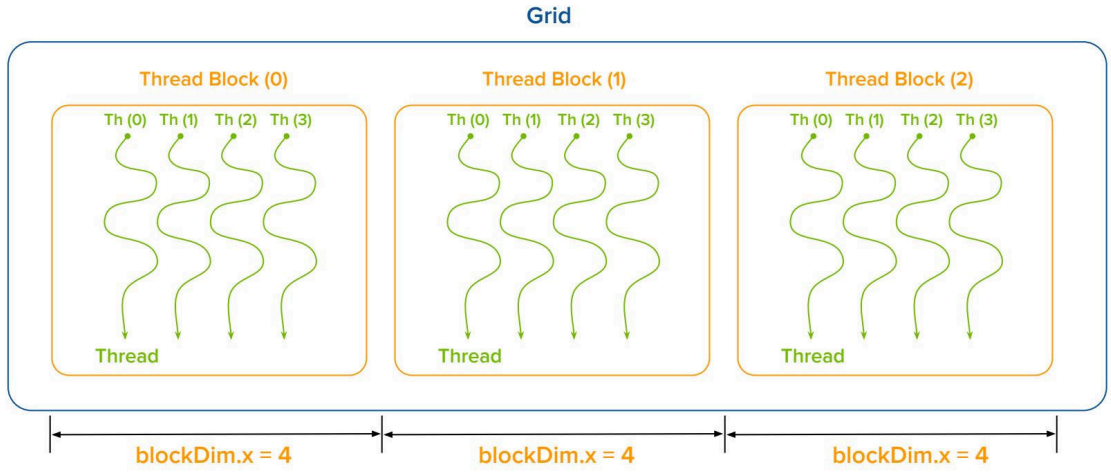
CUDA

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

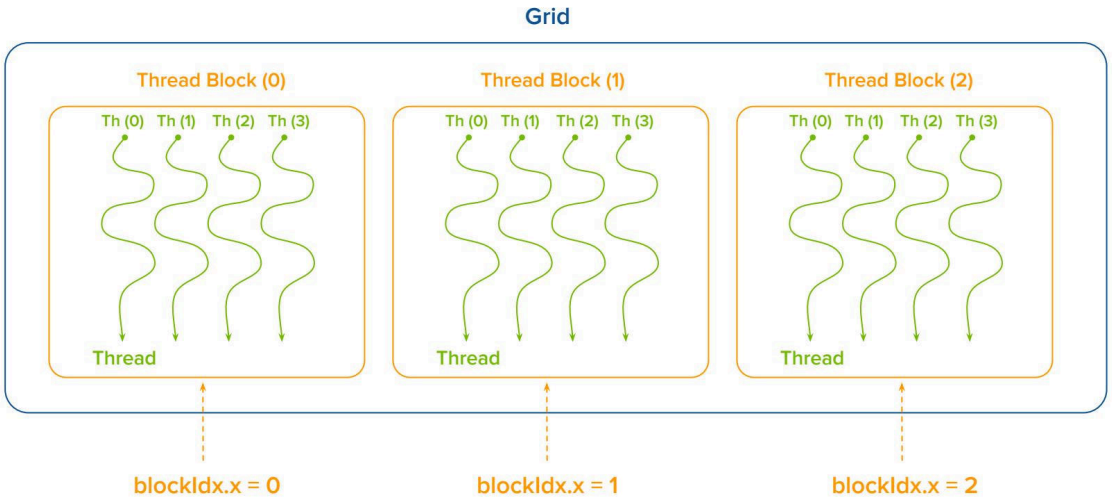
`gridDim.x`: Numero di blocchi nella griglia, in questo caso 3.



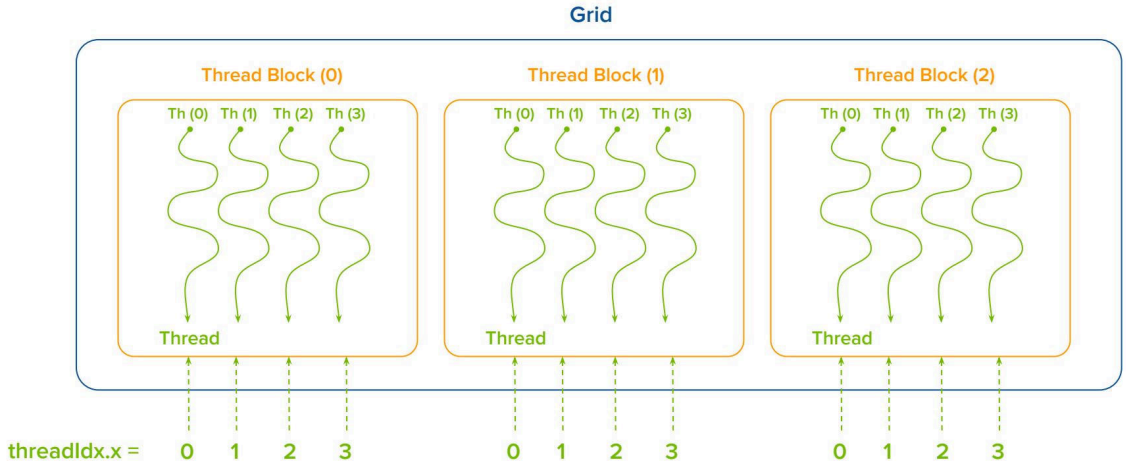
`blockDim.x`: Numero di thread per blocco, in questo caso 4.



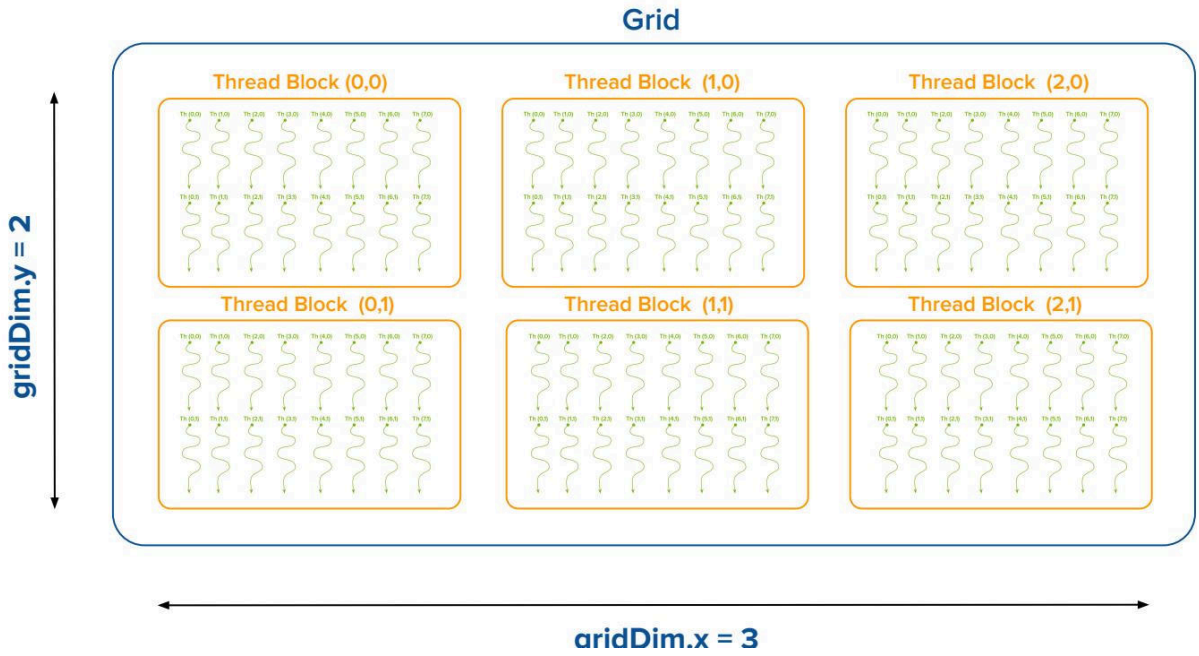
`blockIdx.x`: Indice di un blocco nella griglia.



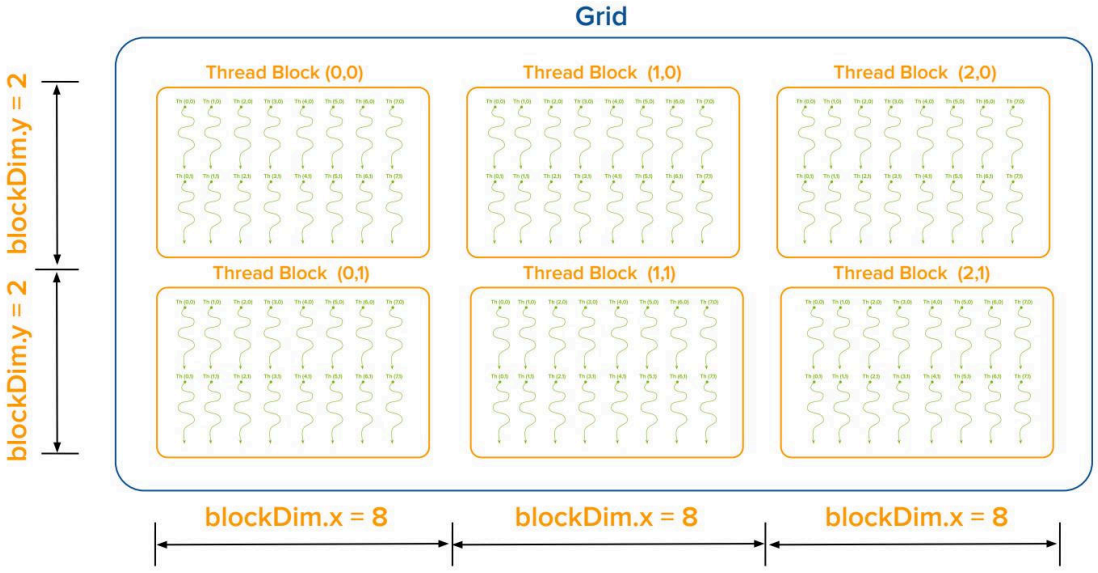
`threadIdx.x`: Indice del thread all'interno del blocco.



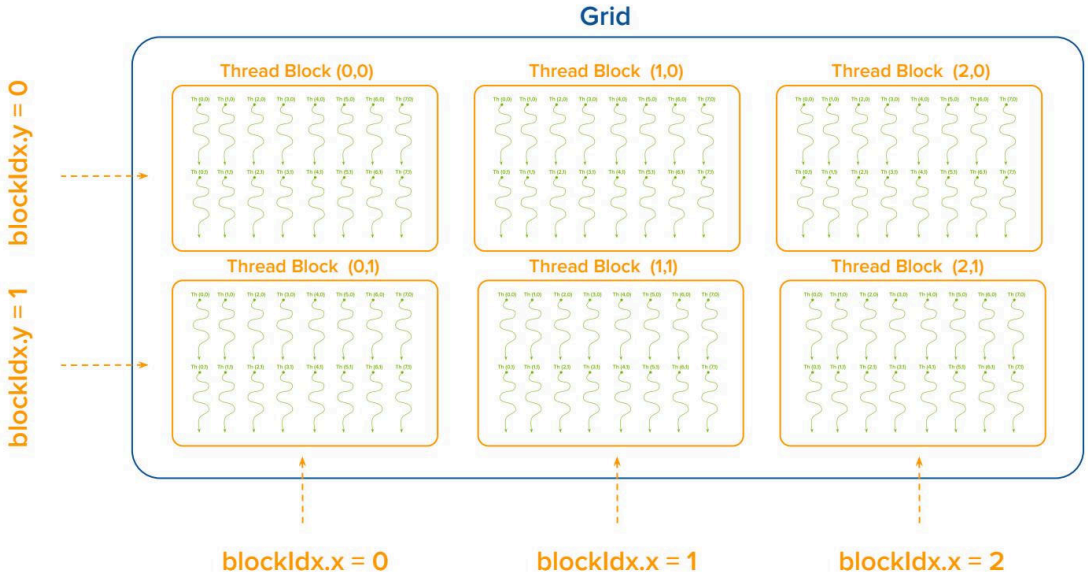
`gridDim.x`, `gridDim.y`: Numero di blocchi nella griglia lungo le dimensioni x e y.



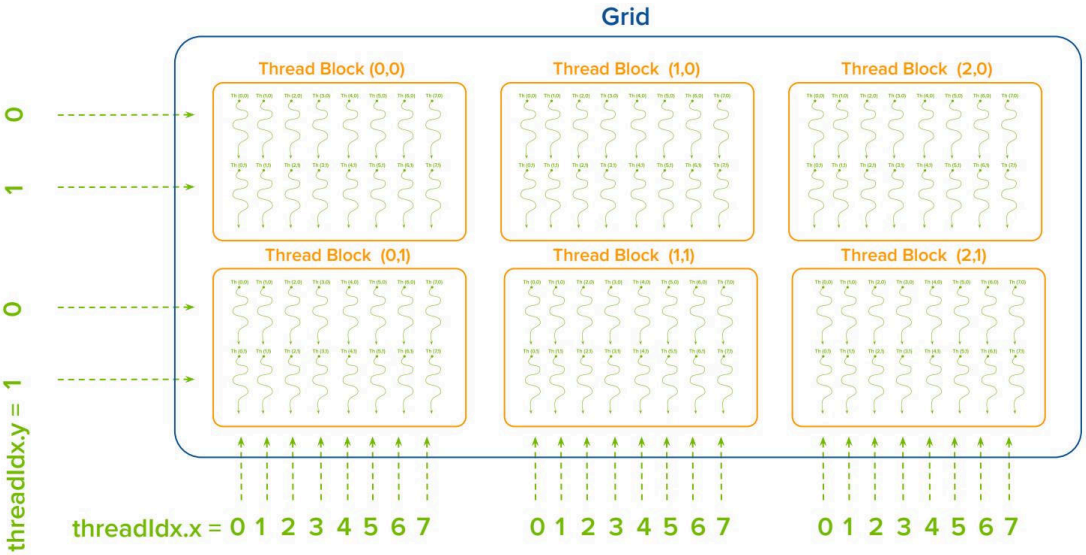
`blockDim.x`, `blockDim.y`: Numero di thread per blocco lungo le dimensioni x e y.



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



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



2.17 Esecuzione di un Kernel CUDA

2.17.1 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

```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`

```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`

2.18 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

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

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

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

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

```cuda
__host__int hostFunction(int x);
```

- Funzione host (eseguibile su CPU).

2.18.1 Combinazione dei qualificatori host e device

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

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

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

2.19 Kernel CUDA: Regole e Comportamento

1. **Esclusivamente Memoria Device** (`__global__` e `__device__`)
   - Accesso consentito solo alla memoria della GPU. Niente puntatori verso la memoria host.
2. **Ritorno void** (`__global__`)
   - I kernel non restituiscono valori direttamente. La comunicazione con l'host avviene tramite la memoria.

3. **Nessun supporto per argomenti variabili** (`__global__` e `__device__`)
   - I kernel non possono avere un numero variabile di argomenti.
4. **Nessun supporto per variabili statiche locali** (`__global__` e `__device__`)
   - Tutte le variabili devono essere passate come argomenti o allocate dinamicamente.
5. **Nessun supporto per puntatori a funzione** (`__global__` e `__device__`)
   - Non è possibile utilizzare puntatori a funzione all'interno di un kernel.
6. **Comportamento asincrono** (`__global__`)
   - I kernel vengono lanciati in modo asincrono rispetto al codice host, salvo sincronizzazioni esplicite.

2.20 Configurazioni di un Kernel CUDA

2.20.1 Combinazioni di Griglia 1D (Esempi)

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.

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

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

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

2.20.2 Combinazioni di Griglia 2D (Esempi)

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.

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

```
dim3 blockSize(8, 4, 2);
kernel_name<<<gridSize, blockSize>>>(args);
```

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

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

2.20.3 Combinazioni di Griglia 3D (Esempi)

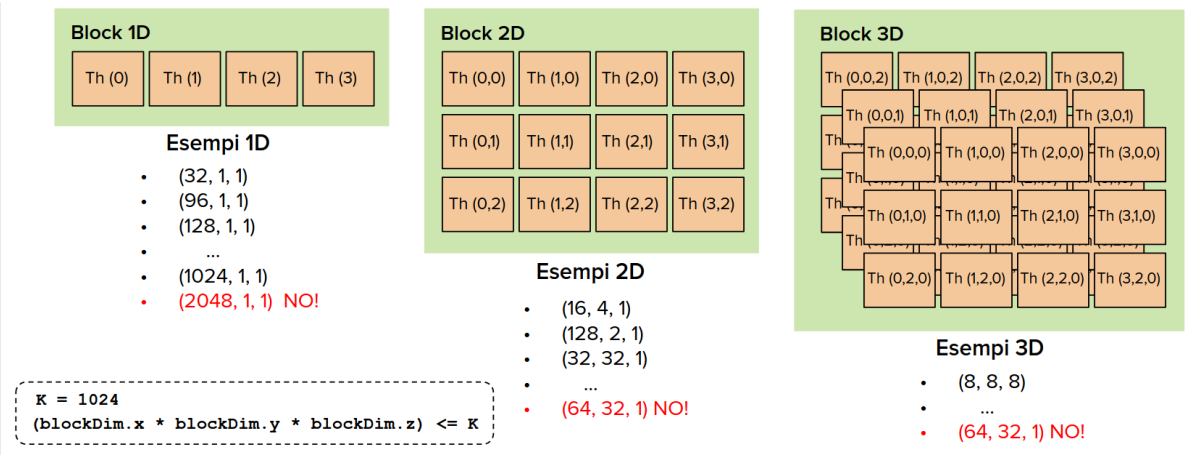
La configurazione di griglia e blocchi può essere 1D, 2D o 3D (9 combinazioni in totale), permettendo una mappatura efficiente (ed intuitiva) su array, matrici o dati volumetrici.

```cuda
// 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);
```

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.

2.21 Numero di Thread per Blocco

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



2.22 Compute Capability (CC) - Limiti SM

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

2.23 Identificazione dei Thread in CUDA

2.23.1 Esempio Codice CUDA

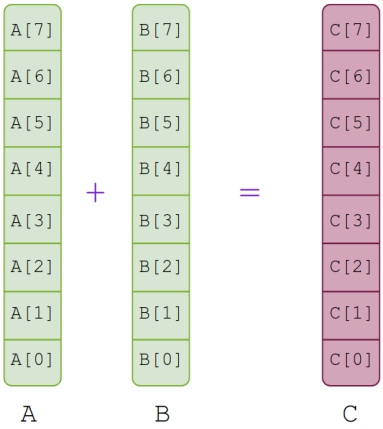
CUDA

```cuda
#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
}
```

2.24 Tecniche di Mapping e Dimensionamento

2.24.1 Somma di Array in CUDA

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



Approccio Tradizionale (CPU)

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

Approccio CUDA (GPU)

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

Approccio Tradizionale (CPU)

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

Approccio CUDA (GPU)

- Gli elementi degli array vengono elaborati in parallelo.
- La GPU è progettata per eseguire calcoli paralleli su larga scala.
- Migliaia di core della GPU lavorano insieme,

2.25 Confronto: Somma di Vettori in C vs CUDA C

Codice C Standard C

```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 CUDA

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

Caratteristiche

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

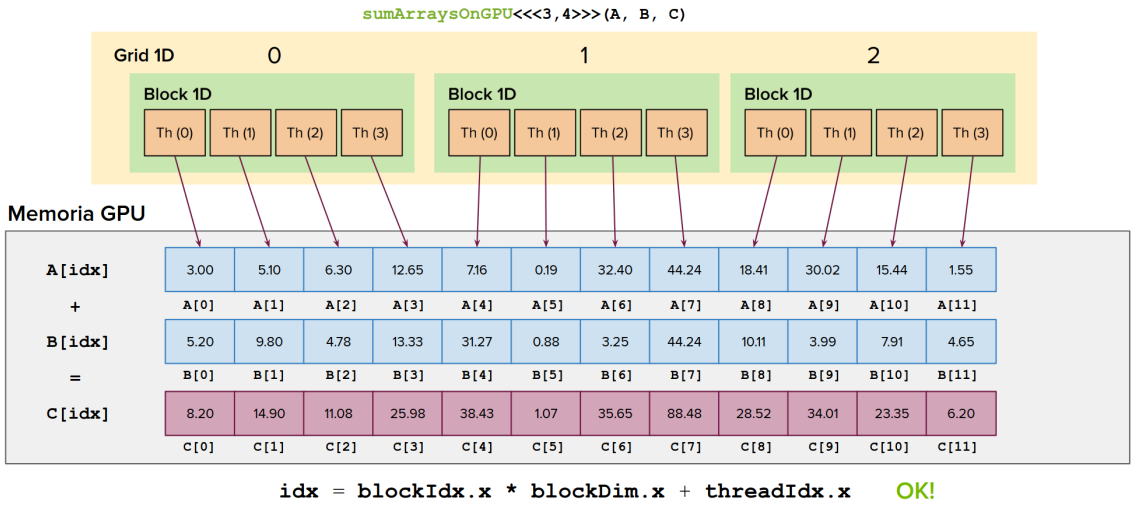
Vantaggi

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

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

idx = threadIdx.x OK! Ma..

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



Proprietà chiave di questo approccio

- **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.

Quindi il codice CUDA sarà il seguente:

Codice CUDA C CUDA

```
__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
sumArraysOnGPU<<<gridDim,blockDim>>>(A, B, C, N);
```

2.26 Identificazione dei Thread e Mapping dei Dati in CUDA

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

Perché Identificare i Thread?

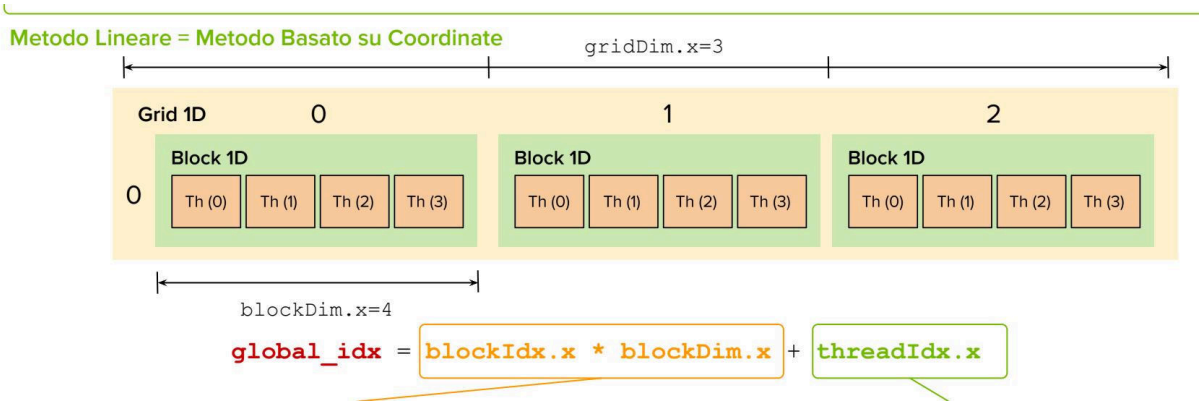
- L'indice globale di un thread identifica univocamente quale porzione di dati deve elaborare.
- Essenziale per gestire correttamente l'accesso alla memoria e coordinare l'esecuzione di algoritmi complessi.

2.26.1 Struttura dei Dati e Calcolo dell'Indice Globale

- Anche le strutture più complesse, come matrici (2D) o array tridimensionali (3D), vengono memorizzate come una sequenza di elementi contigui in memoria nella GPU, tipicamente organizzati in array lineari.
- Ogni thread elabora uno o più elementi in base al proprio indice globale.
- Esistono diversi metodi per calcolare l'indice globale di un thread (es. Metodo Lineare, Coordinate-based).
- Metodi diversi possono produrre mappature diverse tra thread e dati, influenzando prestazioni (come la coalescenza degli accessi in memoria) e la leggibilità del codice.

2.26.2 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.



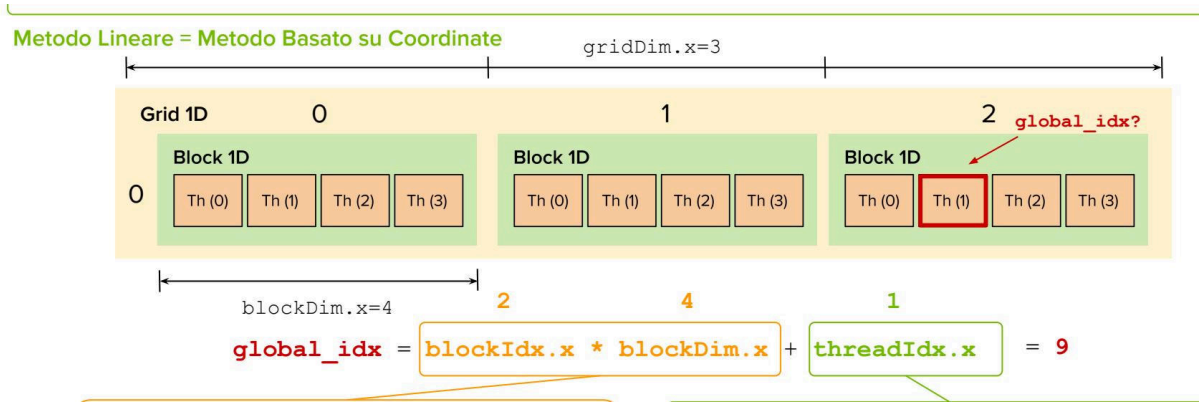
`blockIdx.x * blockDim.x`

- 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.

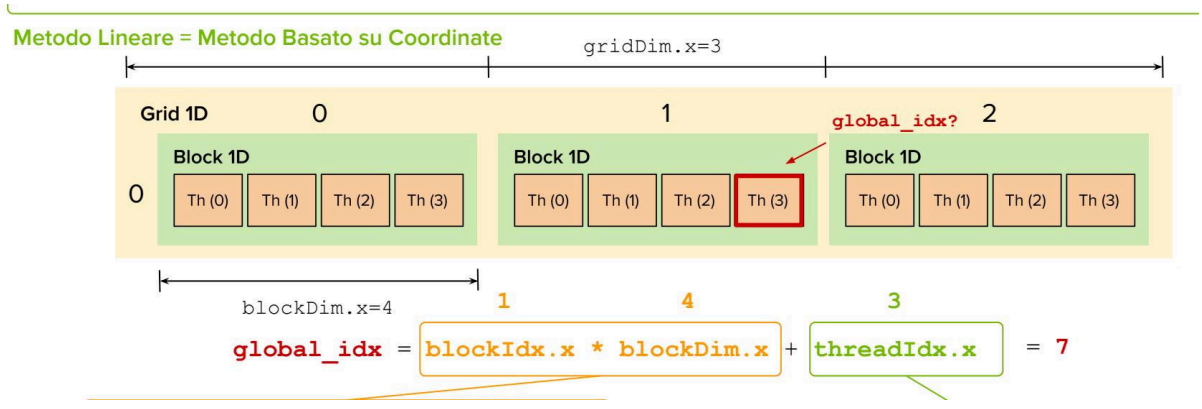
`threadIdx.x`

- 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`.

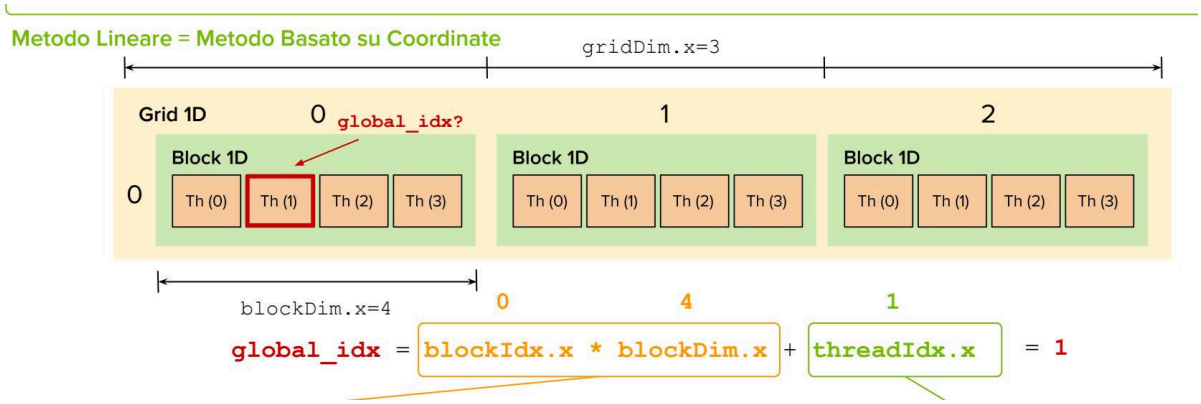
In questo esempio viene mostrato come calcolare l'indice globale per il thread `Th(1)` appartenente al blocco unidimensionale con indice `blockIdx.x = 2`



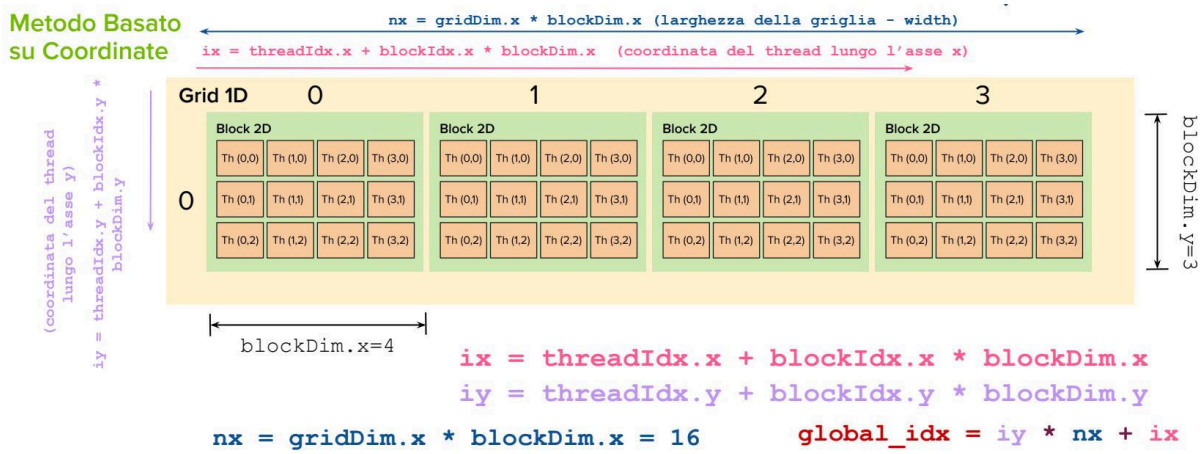
In questo esempio viene mostrato come calcolare l'indice globale per il thread `Th(3)` appartenente al blocco unidimensionale con indice `blockIdx.x = 1`



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

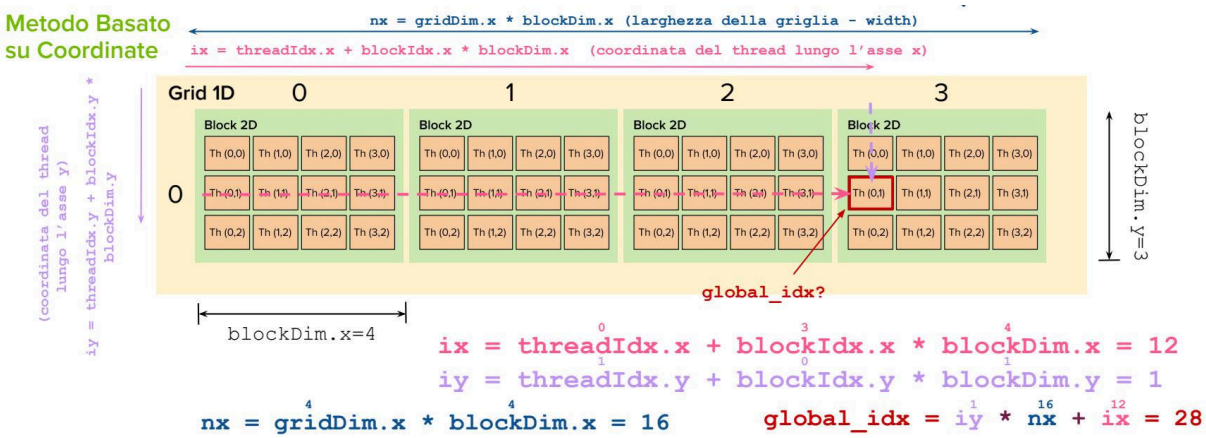


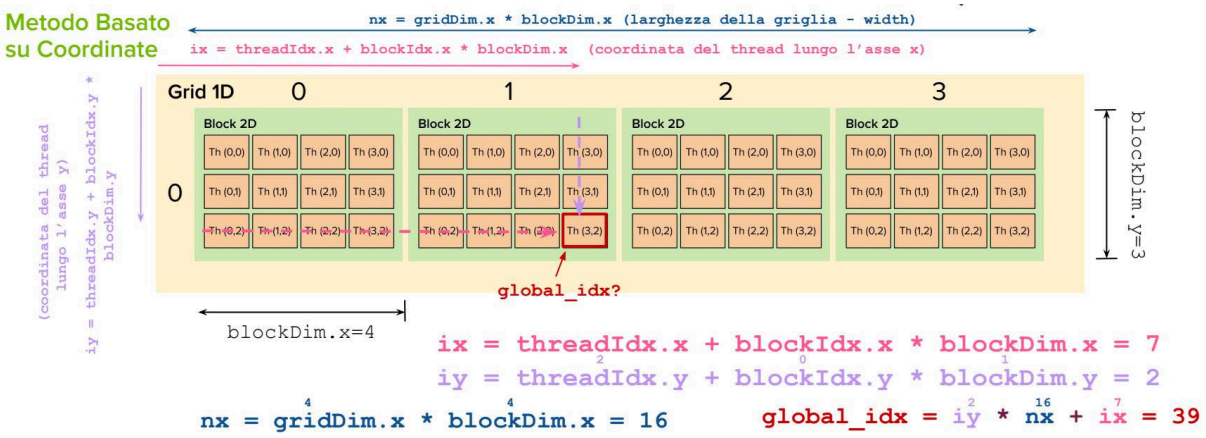
2.27 Calcolo dell'Indice Globale del Thread - Grid 1D, Block 2D



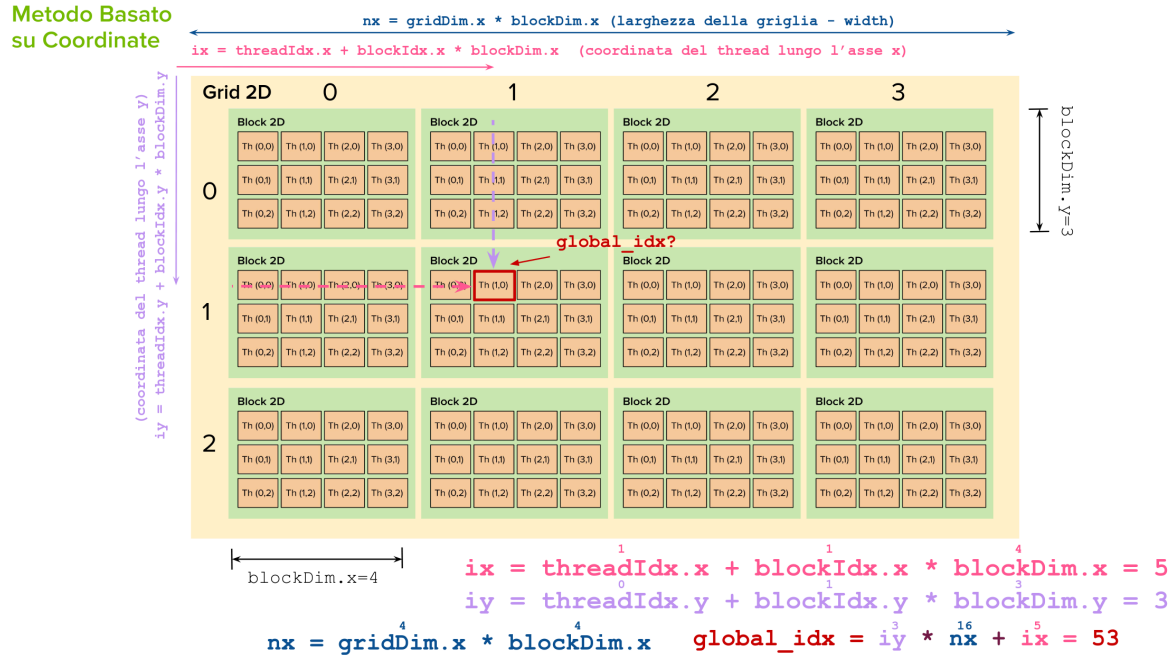
- `ix = threadIdx.x + blockIdx.x * blockDim.x`: determina l'indice del thread lungo l'asse `x`, prendendo in considerazione la posizione nel blocco (`threadIdx.x`) e il numero di blocchi precedenti (`blockIdx.x * blockDim.x`).
- `iy = threadIdx.y + blockIdx.y * blockDim.y`: determina l'indice del thread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

Seguono alcuni esempi.





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



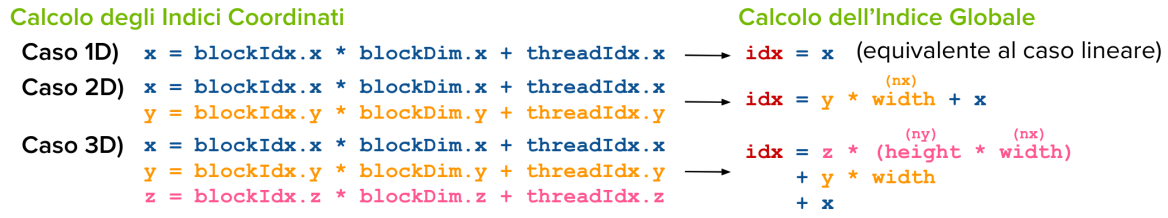
2.29 Metodo Basato su Coordinate per Indici Globali in CUDA

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.

2.29.1 Calcolo degli Indici Coordinati



2.29.2 Esempio di Utilizzo (Caso 2D)

```cuda
__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]
  }
}
```

2.29.3 Come Calcolare la Dimensione della Griglia e del Blocco

Approccio Generale

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

Motivazioni

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

Calcolo delle Dimensioni (Caso 1D)

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

Spiegazione del Calcolo

La formula `(dataSize + blockSize 1) / blockSize` assicura un numero sufficiente di blocchi per coprire tutti i dati, anche se `dataSize` non è un multiplo esatto di `blockSize`.

- **Divisione semplice**: `dataSize / blockSize` fornisce il numero di blocchi completamente pieni.
- Se ci sono dati residui che non riempiono un intero blocco, la divisione semplice li ignorerebbe.
- Aggiungere `blockSize - 1` a `dataSize` "compensa" questi dati residui, includendo l'ultimo blocco parziale. Equivalente a calcolare la ceil della divisione.

2.29.4 Esempio 1 (Dati Residui): `dataSize = 1030`, `blockSize = 256`

Calcolo delle Dimensioni (Caso 1D) CUDA

```cuda
int blockSize = 256; int dataSize = 1030; // Dimensione del blocco e dei dati
dim3 blockDim(blockSize); dim3 gridDim((dataSize + blockSize - 1) / blockSize);
kernel_name<<<gridDim, blockDim>>>(args); // Lancio del kernel
```

- **Divisione semplice**: $1030/256 = 4$ blocchi; **ignorerebbe** l'ultimo blocco parziale perché

$$256 * 4 = 1024$$
$$1030 - 1024 = 6$$

quindi 6 elementi residui ❌

- Con la formula $(1030 + 256 - 1)/256 = 1285/256 = 5$ blocchi; nessun elemento residuo ✅
- In questo caso, la divisione semplice avrebbe dato 4 blocchi, ma c'è un residuo di 6 elementi ($1030 \bmod 256 = 6$); la formula include anche il blocco parziale, quindi otteniamo 5 blocchi.

2.29.5 Esempio 2 (Multiplo Perfetto): `dataSize = 1024`, `blockSize = 256`

- **Divisione semplice**: $1024/256 = 4 \rightarrow$ copre esattamente 1024 elementi ✅
- Con la formula $(1024 + 256 - 1)/256 = 1279/256 = 4$ blocchi ✅
- **Spiegazione**: $1279/256 = 4.996$, ma essendo divisione intera $\rightarrow$ 4 blocchi; aggiungere 255 non basta per raggiungere 1280 (soglia del 5° blocco), quindi non si arrotonda per eccesso.
- **Importante**: La formula funziona correttamente sia con residui che senza, garantendo sempre il numero esatto di blocchi necessari senza sprechi

2.29.6 Calcolo delle Dimensioni (Caso 2D)

CUDA

```cuda
int blockSizeX = 16, blockSizeY = 16; // Dimensione del blocco
int dataSizeX = 1024, dataSizeY = 512; // Dimensione dei dati
dim3 blockDim(blockSizeX, blockSizeY); // Definizione del blocco 2D
dim3 gridDim( // Calcolo della griglia 2D
  (dataSizeX + blockSizeX - 1) / blockSizeX, // Numero di blocchi in X
  (dataSizeY + blockSizeY - 1) / blockSizeY // Numero di blocchi in Y
);
kernel_name<<<gridDim, blockDim>>>(args); // Lancio del kernel
```

2.29.7 Calcolo delle Dimensioni (Caso Generale 3D)

CUDA

```cuda
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
```

2.30 Analisi delle Prestazioni

2.30.1 Verifica del Kernel CUDA (Somma di Array)

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

```cuda
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 (fabsf(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");
}
```

- `hostRef`: risultati attesi dalla somma
- `gpuRef`: risultati calcolati dal kernel
- `fabsf`: funzione della libreria C che calcola il valore assoluto di un numero in virgola mobile a precisione singola

Suggerimenti per la Verifica (basic)

- Verifica ogni elemento degli array per assicurarsi che i risultati del kernel corrispondano ai valori attesi.
- Usa una piccola tolleranza (`epsilon`) per confronti in virgola mobile, in quanto ci possono essere 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.

2.30.2 Gestione degli Errori in CUDA

Problema

- **Asincronicità**: molte chiamate CUDA sono asincrone, rendendo difficile associare un errore alla specifica chiamata che lo ha causato.
- **Complessità di Debugging**: l'errore può emergere in una parte del programma diversa e lontana dal punto in cui è stato generato, rendendo l'individuazione della causa complicata.
- **Gestione Manuale**: controllare ogni chiamata CUDA manualmente è tedioso e soggetto a errori.

Macro CHECK CUDA

```cuda
// 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);
  }
}
```

Esempi di Utilizzo CUDA

```cuda
CHECK(cudaMalloc(&d_input, size)); // Allocazione
CHECK(cudaMemcpy(d_input, h_input, size, cudaMemcpyHostToDevice));
kernel_function <<<numBlocks, blockSize >>>(argument list); // Lancia il kernel
CHECK(cudaGetLastError()); // Primo controllo: errori di lancio del kernel
CHECK(cudaDeviceSynchronize()); // Secondo controllo: errori durante l'esecuzione del kernel. Usare solo in DEBUG (Overhead di performance!)
```

2.30.3 Profiling delle Prestazioni in CUDA

Introduzione al Profiling

- Misurare e ottimizzare le prestazioni è fondamentale per garantire l'efficienza del codice.
- Il profiling permette di misurare le prestazioni, analizzare l'uso delle risorse e individuare possibili ottimizzazioni.

Importanza della Misurazione del Tempo

- **Identificazione dei Colli di Bottiglia**: Individuare le sezioni di codice che limitano le prestazioni (un'implementazione naïve raramente è ottimale)
- **Analisi degli Effetti delle Modifiche**: valutare l'impatto delle modifiche sul tempo di esecuzione.
- **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 thread, blocchi e host-device.

2.30.4 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 moderni avanzati per analisi approfondita e ottimizzazione a livello di sistema e kernel.

2.30.4.1 Timer CPU

- Timer eseguito dall'host, misura il tempo **wall-clock** visto dalla CPU.
- Soluzione semplice e pratica basata su funzioni di sistema standard.
- **Può misurare qualsiasi operazione**: kernel GPU, trasferimenti memoria, codice CPU.

- Il tempo include anche gli overhead: lancio dei kernel, sincronizzazione, latenze di comunicazione.
- Per operazioni GPU asincrone (kernel), richiede sincronizzazione esplicita.

Funzione del Timer della CPU CUDA

```
#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.
- Precisione teorica fino al **nanosecondo**.

Utilizzo Per Misurare un Kernel CUDA 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 sia l'**esecuzione** sia l'**overhead** di lancio e 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 ($< 1ms$).
- 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**.

Somma di due array CUDA

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

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

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

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

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

  // Somma dei vettori su host per verifica
  iStart = cpuSecond();
  sumArraysOnHost(h_A, h_B, hostRef, nElem); // Calcola la somma su CPU per confronto
  iElaps = cpuSecond() - iStart;
  printf("sumArraysOnHost elapsed %f sec\n", iElaps);
  // Allocazione della memoria su device
  float *d_A, *d_B, *d_C;
```

```cuda
CHECK(cudaMalloc((float**)&d_A, nBytes)); // Alloca memoria per A su GPU
CHECK(cudaMalloc((float**)&d_B, nBytes)); // Alloca memoria per B su GPU
CHECK(cudaMalloc((float**)&d_C, nBytes)); // Alloca memoria per il risultato su GPU
// Copia dei dati dall'host al device
CHECK(cudaMemcpy(d_A, h_A, nBytes, cudaMemcpyHostToDevice)); // Copia A su GPU
CHECK(cudaMemcpy(d_B, h_B, nBytes, cudaMemcpyHostToDevice)); // Copia B su GPU

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

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

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

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

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

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

```cuda
int dev = 0
```

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

Alternative e Pratiche Comuni

1. Selezione del dispositivo tramite argomenti:
   `if (argc > 1) dev = atoi(argv[1]);`
2. Utilizzo di variabili d'ambiente:

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

```
cudaSetDevice(dev)
```

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

```
initialData(h_A, nElem); // Inizializza il vettore A
initialData(h_B, nElem); // Inizializza il vettore B
```

- Inizializza un array di float con valori casuali (compresi fra 0 e 25.5)

```cuda
void initialData(float *ip, int size){
  // Genera dati casuali
  time_t t;
  srand((unsigned int) time(&t));
  for (int i = 0; i < size; i++){
  ip[i] = (float)(rand() & 0xFF) / 10.0f;
  }
}
```

Appendices