

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

1	Introduzione	1
1.1	Perché Scegliere la Piattaforma CUDA?	1
1.2	Cos'è il CUDA Toolkit?	1
1.2.1	Componenti chiave del CUDA Toolkit	1
1.2.1.1	Strumenti di Debugging e Profiling	2
1.2.1.2	Relazione tra CUDA Toolkit e CUDA Version	2
1.2.1.3	Retrocompatibilità del CUDA Toolkit	2
1.3	CUDA Compute Capability (CC)	2
1.3.1	Relazione tra Compute Capability (CC) e CUDA Version	3
1.4	Evoluzione delle Architetture GPU NVIDIA	3
1.5	Anatomia di un Programma CUDA	3
1.5.1	Struttura del Codice Sorgente	3
1.5.2	Componenti Principali	3
1.5.3	Flusso di Compilazione	4
1.6	Compilazione di un Programma CUDA	5
1.7	Hello World in CUDA C	5
1.7.1	Passo 1: Creare il File Sorgente	5
1.7.1.1	Passo 2: Scrivere il codice	5
1.8	Hello World in CUDA C - Analisi	6
1.9	Hello World in CUDA C	6
1.9.0.1	Passo 3: Compilazione	6
1.10	Hello World in CUDA C	6
1.10.0.1	Passo 4: Esecuzione	6
1.11	Ottenere Informazioni sulla GPU tramite API CUDA	7
1.11.1	Utilizzo delle API CUDA	7
1.12	Ottenere Informazioni sulla GPU tramite API CUDA	7
1.12.0.1	Esempio di Output	7
1.13	Cosa Significa Programmare in CUDA?	8
1.13.1	Pensare in Parallello	8
1.13.2	Scrittura di codice in CUDA C	8
1.13.2.1	Testi Generali	8
1.13.2.2	NVIDIA Docs	8
1.13.2.3	Materiale di Approfondimento	8
2	Modello di Programmazione CUDA	9
2.1	La Struttura Stratificata dell'Ecosistema CUDA	9
2.2	Ruolo del Modello e del Programma	9
2.3	Livelli di Astrazione nella Programmazione Parallela CUDA	10
2.3.1	Livello Dominio	10
2.3.2	Livello Logico	10
2.3.2.1	Livello Hardware	10
2.4	Thread CUDA: L'Unità Fondamentale di Calcolo	10
2.4.1	Cos'è un Thread CUDA?	10
2.4.2	Cosa Fa un Thread CUDA?	10
2.5	Struttura di Programmazione CUDA	11
2.5.1	Caratteristiche Principali	11
2.6	Flusso Tipico di Elaborazione CUDA	11

2.7	Gestione della Memoria in CUDA	11
2.7.1	Modello di Memoria CUDA	11
2.7.2	Caratteristiche PCIe	12
2.8	Collegamento Fisico della GPU tramite PCIe	12
2.8.1	Connessione Fisica GPU	12
2.9	Modello di Memoria CUDA	12
2.10	Gerarchia di Memoria	13
2.11	Allocazione della Memoria sul Device	13
2.11.1	Esempio di Allocazione di Memoria sulla GPU	14
2.12	Trasferimento Dati	14
2.12.1	Spazi di Memoria Differenti	15
2.13	Deallocazione della Memoria sul Device	15
2.13.1	Esempio di Allocazione e Trasferimento Dati	15
2.14	Organizzazione dei Thread in CUDA	16
2.14.1	Struttura Gerarchica	16
2.14.2	Perché una Gerarchia di Thread?	17
2.15	Identificazione dei Thread in CUDA	17
2.15.1	Dimensione delle Griglie e dei Blocchi	18
2.16	Struttura <code>dim3</code>	18
2.17	Esecuzione di un Kernel CUDA	22
2.17.1	Cos'è un Kernel CUDA?	22
2.18	Qualificatori di Funzione in CUDA	22
2.18.1	Combinazione dei qualificatori host e device	23
2.19	Kernel CUDA: Regole e Comportamento	23
2.20	Configurazioni di un Kernel CUDA	23
2.20.1	Combinazioni di Griglia 1D (Esempi)	23
2.20.2	Combinazioni di Griglia 2D (Esempi)	23
2.20.3	Combinazioni di Griglia 3D (Esempi)	24
2.21	Numero di Thread per Blocco	24
2.22	Compute Capability (CC) - Limiti SM	24
2.23	Identificazione dei Thread in CUDA	25
2.23.1	Esempio Codice CUDA	25
2.24	Tecniche di Mapping e Dimensionamento	25
2.24.1	Somma di Array in CUDA	25
2.25	Confronto: Somma di Vettori in C vs CUDA C	26
2.26	Identificazione dei Thread e Mapping dei Dati in CUDA	27
2.26.1	Struttura dei Dati e Calcolo dell'Indice Globale	27
2.26.2	Calcolo dell'Indice Globale del Thread - Grid 1D, Block 1D	28
2.27	Calcolo dell'Indice Globale del Thread - Grid 1D, Block 2D	29
2.28	Calcolo dell'Indice Globale del Thread - Grid 2D, Block 2D	30
2.29	Metodo Basato su Coordinate per Indici Globali in CUDA	30
2.29.1	Calcolo degli Indici Coordinati	30
2.29.2	Esempio di Utilizzo (Caso 2D)	30
2.29.3	Come Calcolare la Dimensione della Griglia e del Blocco	31
2.29.4	Esempio 1 (Dati Residui): <code>dataSize = 1030, blockSize = 256</code>	31
2.29.5	Esempio 2 (Multiplo Perfetto): <code>dataSize = 1024, blockSize = 256</code>	31
2.29.6	Calcolo delle Dimensioni (Caso 2D)	31
2.29.7	Calcolo delle Dimensioni (Caso Generale 3D)	32

2.30	Analisi delle Prestazioni	32
2.30.1	Verifica del Kernel CUDA (Somma di Array)	32
2.30.2	Gestione degli Errori in CUDA	32
2.30.3	Profiling delle Prestazioni in CUDA	33
2.30.4	Metodi Principali	33
2.30.4.1	Timer CPU	33
2.30.4.2	Metodo 2: NVIDIA Profiler [$5.0 \leq \text{Compute Capability} < 8.0$] ..	37
2.30.4.3	NVIDIA Nsight Systems	38
2.30.4.4	NVIDIA Nsight Compute	39
2.30.5	Nvidia Nsight Systems vs. Compute	40
2.30.6	Ottimizzazione della Gestione della Memoria in CUDA	41
2.31	Applicazione Pratiche	41
2.31.1	Operazioni su Matrici in CUDA	41
2.31.1.1	Suddivisione della matrice in Griglia 2D e Blocchi 2D	43
2.31.1.2	Suddivisione della Matrice in Griglia 1D e Blocchi 1D	48
2.31.1.3	Confronto Kernel CUDA per la Somma fra Matrici	48
2.31.1.4	Suddivisione della Matrice in Griglia 1D e Blocchi 2D	51
2.31.1.5	Confronto Kernel CUDA per la Somma fra Matrici - 2D2D 1D2D ..	51
2.31.1.6	Suddivisione della Matrice in Griglia 2D e Blocchi 1D	53
2.31.1.7	Confronto Kernel CUDA per la Somma fra Matrici - 2D2D 2D1D ..	54
2.31.1.8	Confronto fra le Migliori Configurazioni di Blocchi e Griglie	55
2.32	Immagini come Matrici Multidimensionali	56
2.32.1	Memorizzazione Lineare di Immagini RGB in CUDA	56
2.32.2	Parallelismo GPU nella Conversione RGB a Grayscale	57
2.32.2.1	Suddivisione dell'Immagine in Blocchi per l'Elaborazione GPU ..	58
2.32.3	Confronto: Conversione RGB a Grayscale in C vs CUDA C	59
2.32.4	Conversione RGB a Grayscale in CUDA	59
2.32.5	Image Flipping con CUDA	61
2.32.6	Image Blur con CUDA	62
2.32.7	Introduzione alla Convoluzione 1D e 2D	64
2.32.7.1	Esempio di Convoluzione 1D	65
2.32.7.2	Perché la Convoluzione si Adatta al Calcolo Parallelo	65
2.32.7.3	Esempio di Convoluzione 2D	66
2.33	Riferimenti Bibliografici	67
3	Modello di Esecuzione CUDA	68
3.1	Introduzione al Modello di Esecuzione CUDA	68
3.2	Streaming Multiprocessor (SM)	68
3.3	CUDA Core - Unità di Elaborazione CUDA	70
3.4	Streaming Multiprocessor (SM) - Evoluzione	71
3.5	Tensor Core: Acceleratori per l'Intelligenza Artificiale (Volta+)	73
3.6	Evoluzione dei NVIDIA Tensor Core	73
3.7	Organizzazione e gestione dei thread	74
3.7.1	SM, Thread Blocks e Risorse	74
3.7.2	Corrispondenza tra Vista Logica e Vista Hardware	75
3.7.3	Distribuzione dei Blocchi su Streaming Multiprocessors	76
3.7.4	Concetto di Wave in CUDA	77
3.7.4.1	Numero di Waves per un Kernel CUDA	78
3.7.5	Scalabilità in CUDA	79

3.8	Modello di Esecuzione SIMT e Warp	79
3.8.1	Modello di Esecuzione: SIMD	79
3.8.2	Modello di Esecuzione: SIMT	80
3.8.3	Modello di Esecuzione Gerarchico di CUDA	81
3.8.4	Warp: L'Unità Fondamentale di Esecuzione nelle SM	82
3.8.4.1	Organizzazione dei Thread e Warp	82
3.8.4.2	Compute Capability (CC) - Limiti su Blocchi e Thread	84
3.8.4.3	Warp: Contesto di Esecuzione	85
3.8.4.4	Parallelismo a Livello di Warp nell'SM	85
3.8.5	Classificazione dei Thread Block e Warp	86
3.8.6	Classificazione degli Stati dei Thread	86
3.8.7	Scheduling dei Warp	87
3.8.7.1	Warp Scheduler e Dispatch Unit	88
3.8.7.2	Scheduling dei Warp: TLP e ILP	88
3.8.8	Esecuzione Parallela dei Warp - Esempio con Fermi SM	89
3.8.9	Scheduling Dinamico dell'Istruzioni - Fermi SM	89
3.8.10	Latency, Throughput e Concurrency	90
3.8.11	Latency Hiding nelle GPU	90
3.8.12	Legge di Little	91
3.8.13	Massimizzare il Parallelismo per Operazioni Aritmetiche	92
3.8.14	Massimizzare il Parallelismo per Operazioni di Memoria	92
3.8.15	Warp Divergence	93
3.8.16	CPU vs GPU: Gestione del Branching e della Warp Divergence	94
3.8.16.1	Warp Divergence: Analisi del Flusso di Esecuzione	94
3.8.16.2	Serializzazione nella Warp Divergence	95
3.8.16.3	Confronto delle Condizioni di Branch	96
3.8.17	Architetture Pre-Volta (< CC 7.0)	97
3.8.17.1	Architettura Volta (CC 7.0+) e Independent Thread Scheduling ..	97
3.8.17.2	Confronto Pre-Volta vs Post-Volta	98
3.8.18	Introduzione di <code>_syncwarp</code> in Volta	99
3.8.19	Confronto Pre-Volta vs Post-Volta	100
3.9	Sincronizzazione e Comunicazione	100
3.9.1	Sincronizzazione in CUDA - Motivazioni	100
3.9.2	Race Condition (Hazard)	101
3.9.3	Deadlock in CUDA	101
3.9.4	Sincronizzazione in CUDA	102
3.9.5	Operazioni Atomiche in CUDA	102
3.9.6	Operazioni Atomiche in CUDA - Esempi d'Uso	103
3.10	Ottimizzazione delle Risorse	104
3.10.1	Resource Partitioning in CUDA	104
3.10.2	Anatomia di un Thread Block	105
3.10.2.1	Compute Capability (CC) - Limiti SM	107
3.10.3	Occupancy	108
3.10.3.1	Occupancy Teorica vs Effettiva	109
3.10.3.2	Nota Importante sull'Occupancy	109
3.10.3.3	Nsight Compute: Occupancy Calculator	110
3.11	Parallelismo Avanzato	111
3.11.1	Introduzione al CUDA Dynamic Parallelism	111

3.11.1.1	Dynamic Parallelism: Eliminare il Round-trip CPU-GPU	111
3.11.2	Esecuzione Nidificata con CUDA Dynamic Parallelism	112
3.11.3	Esempio di CUDA Dynamic Parallelism	112
3.11.4	Memoria in CUDA Dynamic Parallelism	113
3.11.5	Memoria in CUDA Dynamic Parallelism	114
3.11.6	Gestione dello Scambio Dati nel Parallelismo Dinamico	114
3.11.7	Consistenza della Memoria nel Parallelismo Dinamico	114
3.11.8	Dipendenze Annidate in CUDA	115
3.11.9	Sincronizzazione con <code>cudaDeviceSynchronize()</code>	115
3.11.9.0.0.0.0.0.1	Funzione Principale	116
3.11.9.0.0.0.0.1.0.0.1	<code>Sincre</code> .	116
3.11.10	Esecuzione Nidificata con CUDA Dynamic Parallelism	116
3.11.11	Parallelismo Dinamico su GPU: Nested Hello World	117
3.11.12	Nested Hello World : Compilazione ed Esecuzione	118
3.11.13	Nested Hello World : Compilazione ed Esecuzione	118
3.11.13.0.0.0.0.0.0.0.1	Output (Terminale) .	119
3.11.14	Restrizioni sul Parallelismo Dinamico	120
3.11.15	Riferimenti Bibliografici	120
Appendices	121

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 Invisible: 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.

Compute Capability (Supporto CUDA SDK vs. Microarchitecture)

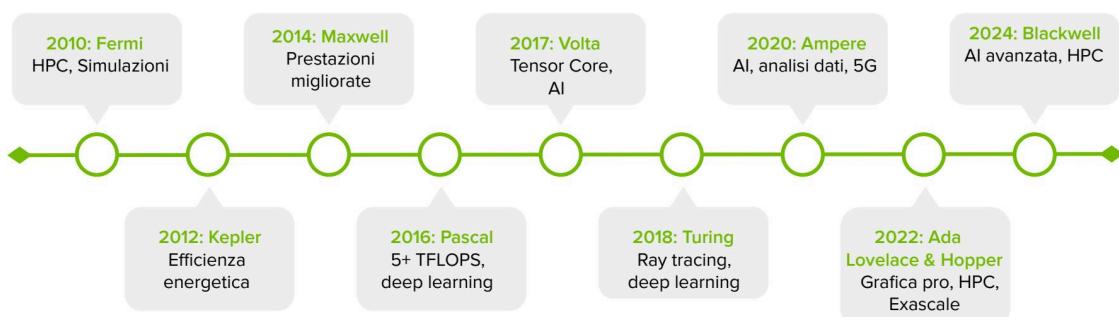
1.0 ^[41]	1.0 - 1.1					
1.1	1.0 - 1.1+x					
2.0	1.0 - 1.1+x					
2.1 - 2.3.1 ^{[42][43][44][45]}	1.0 - 1.3					
3.0 - 3.1 ^{[46][47]}	1.0	2.0				
3.2 ^[48]	1.0	2.1				
4.0 - 4.2	1.0	2.1				
5.0 - 5.5	1.0		3.0	3.5		
6.0	1.0		3.2	3.5		
6.5	1.1			3.7	5.x	
7.0 - 7.5		2.0			5.x	
8.0		2.0			6.x	
9.0 - 9.2			3.0			7.0 - 7.2
10.0 - 10.2			3.0			7.5
11.0 ^[49]				3.5		8.0
11.1 - 11.4 ^[50]				3.5		8.6
11.5 - 11.7.1 ^[51]				3.5		8.7
11.8 ^[52]				3.5		8.9, 9.0
12.0 - 12.6				5.0		9.0
12.8				5.0		12.0

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

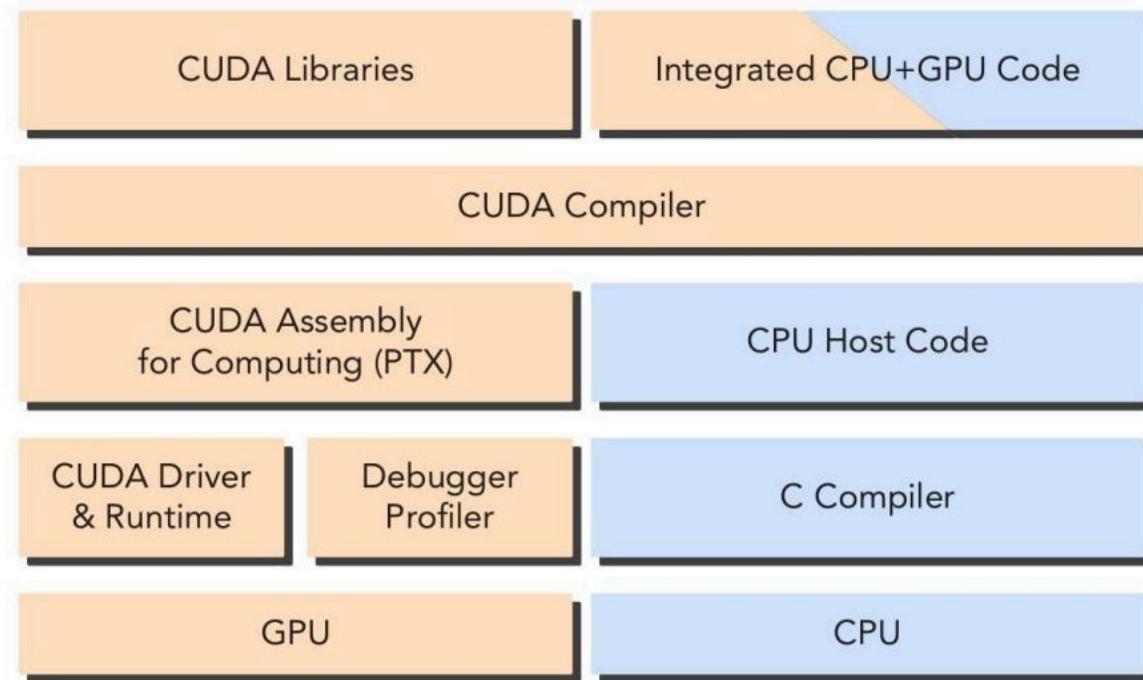
1.5.1 Struttura del Codice Sorgente

- 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

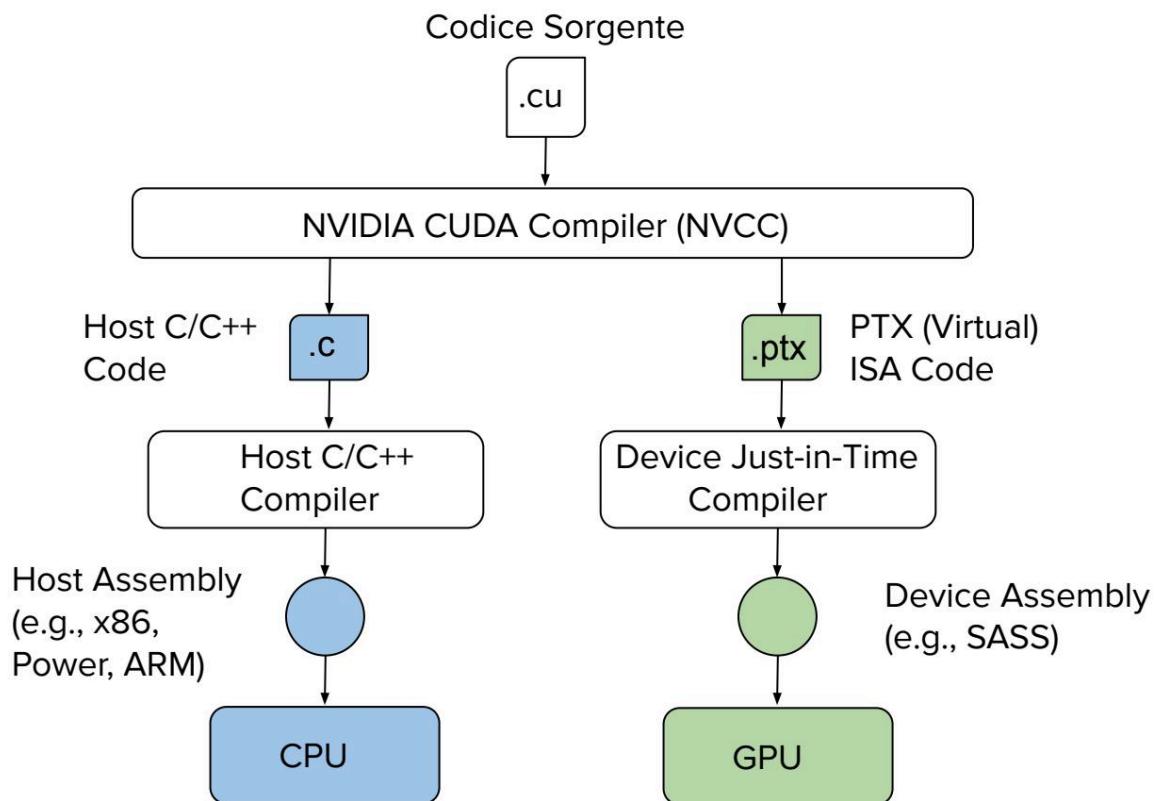
- Gestisce la sincronizzazione
- Codice Device:
 - Codice CUDA C eseguito sulla GPU
 - Contiene i kernel (funzioni parallele)
 - Esegue operazioni computazionali intensive in parallelo



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
- Esegibile 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

<pre> include <stdio.h> int main(void) { printf("Hello World from CPU!\n"); return 0; } </pre>	Linguaggio C
--	---------------------

```
Linguaggio CUDA C
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;
}
```

CUDA

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.

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


```

CUDA

- 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.

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

CUDA

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

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

\$ Shell

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 Ottenerne 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

```
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;
}
```

CUDA

- 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 Ottenerne 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 Parallello

- **Decomposizione del Problema:** Identificare le parti del problema che possono essere eseguite in parallello 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 parallello. 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 parallello 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^a edizione)
- Kirk, D. B., Hwu, W. W. (2022). Programming Massively Parallel Processors. Morgan Kaufmann (4^a 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/>

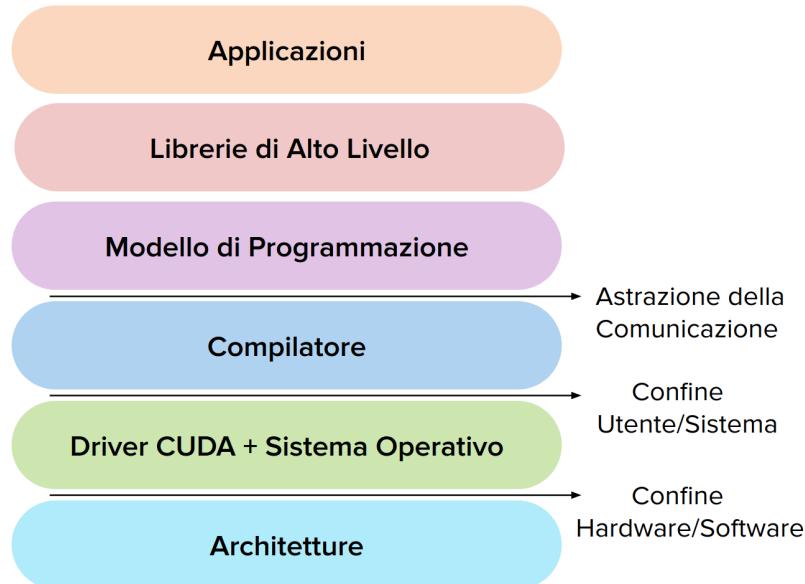
1.13.2.3 Materiale di Approfondimento

- 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 Parallelia 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

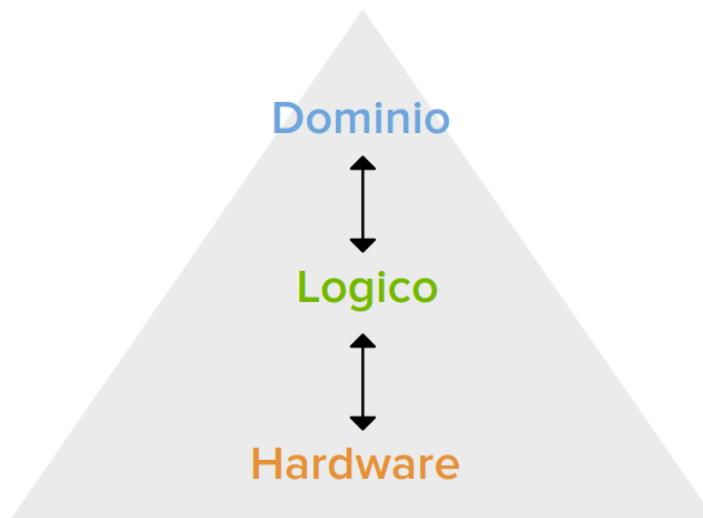
- Organizzazione e gestione dei thread.
- Implementazione della strategia di parallelizzazione.

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.

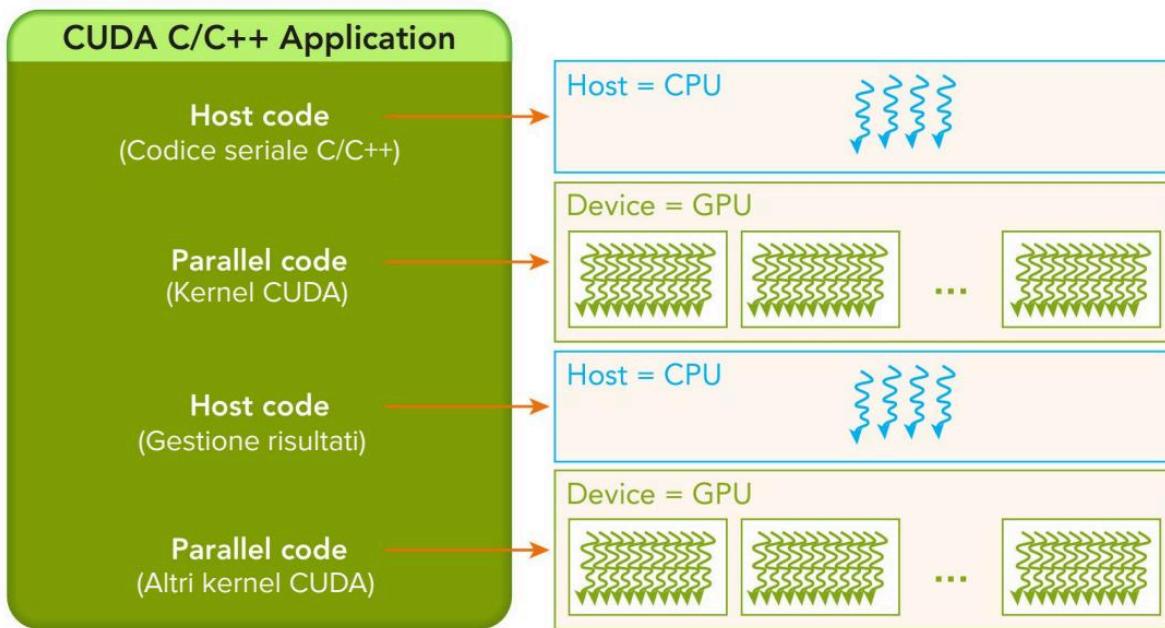
2.4.2 Cosa Fa un Thread CUDA?

- **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.

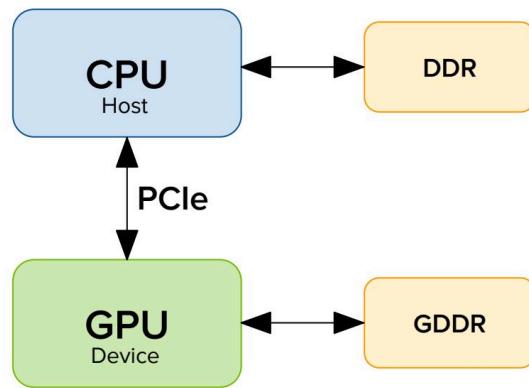
2.7 Gestione della Memoria in CUDA

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

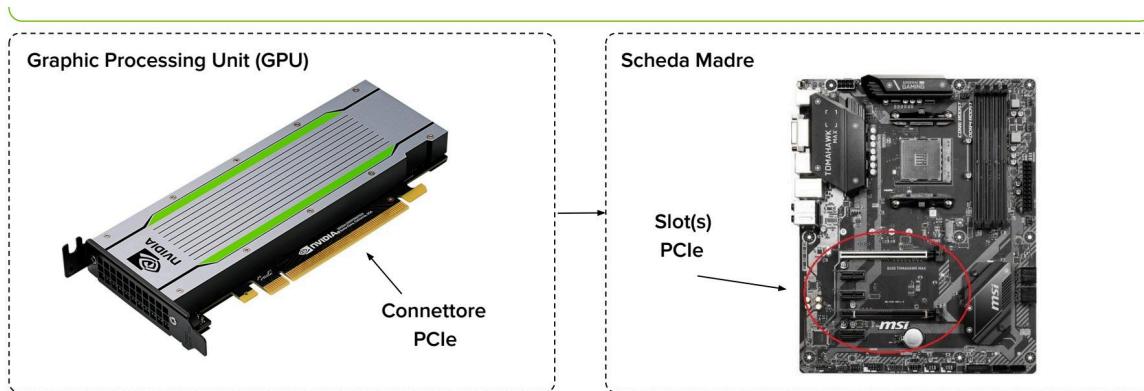
- **Lane:** Ogni lane (canale di trasmissione) è costituito da due copie di segnali differenziali (quattro fili), una per ricevere (RX) e una per trasmettere (TX) dati.
- **Full-Duplex:** Trasmette e riceve dati simultaneamente in entrambe le direzioni.
- **Scalabilità:** La larghezza di banda varia 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	Allocà 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

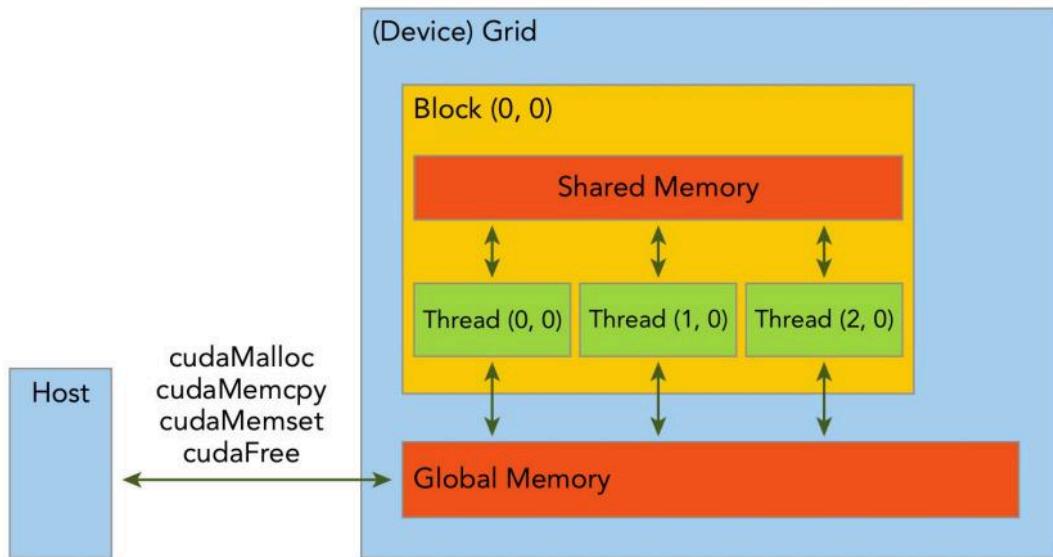
Shared Memory

- 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](#))

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

CUDA

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](#))

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

CUDA

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`.

```
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'allocatione 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");}
```

CUDA

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](#))

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

CUDA

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

```
host_array = dev_ptr
```

CUDA

invece di

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

CUDA

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](#))

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

CUDA

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.

```
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
```

CUDA

```

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)

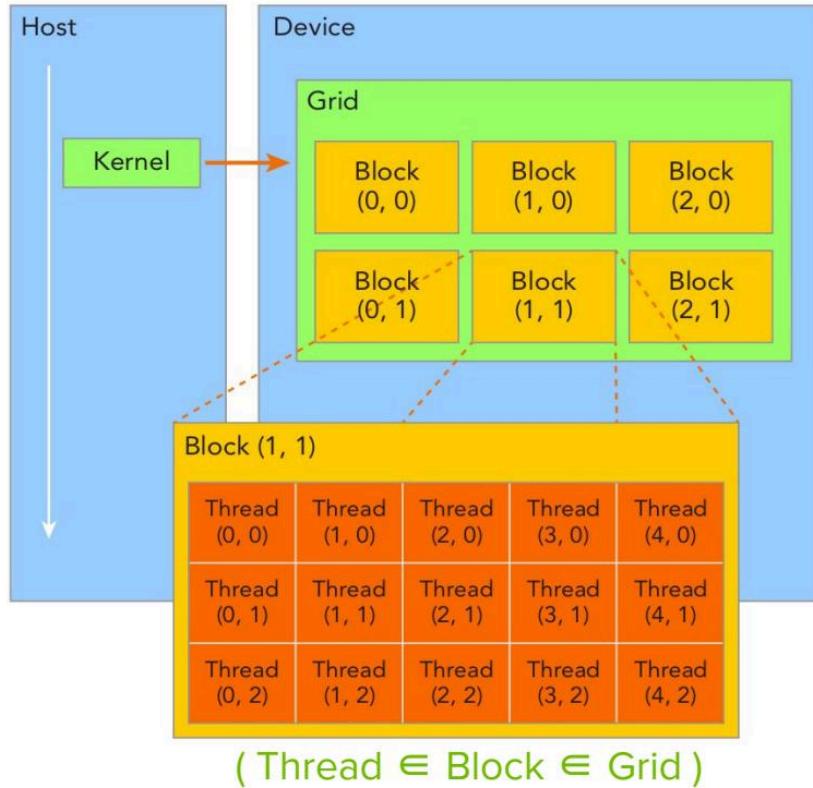
- Array di thread blocks.
- È organizzata in una struttura 1D, 2D o 3D.
- Rappresenta l'intera computazione di un kernel.
- Contiene tutti i thread che eseguono il singolo kernel.
- Condivide lo stesso spazio di memoria globale.

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

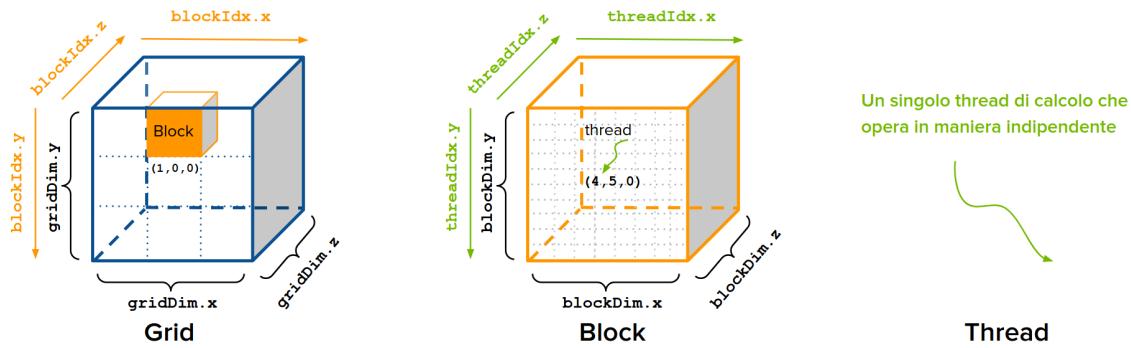
La gerarchia è scalabile e permette di adattare l'esecuzione a GPU con diverse capacità e numero di core. Il codice CUDA, quindi, risulta più portatile e può essere eseguito su diverse architetture GPU.

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

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

CUDA

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

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

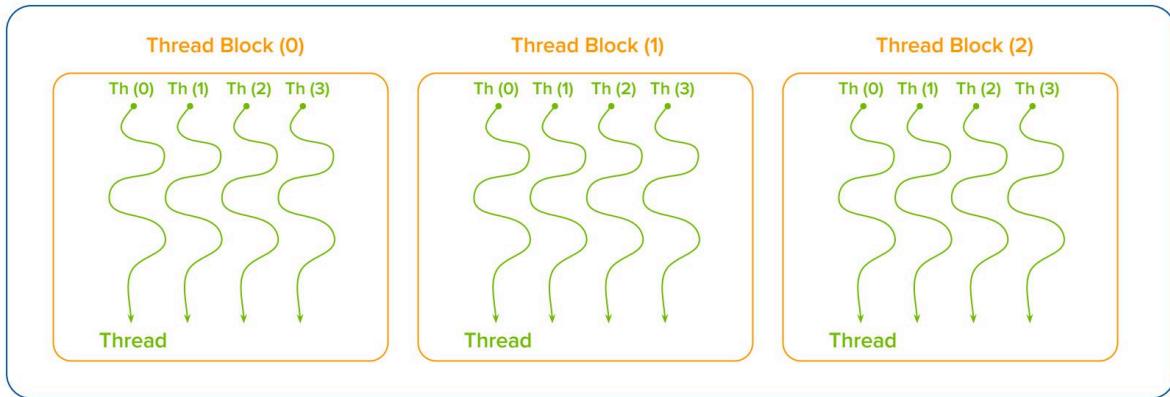
CUDA

Codice Originale:

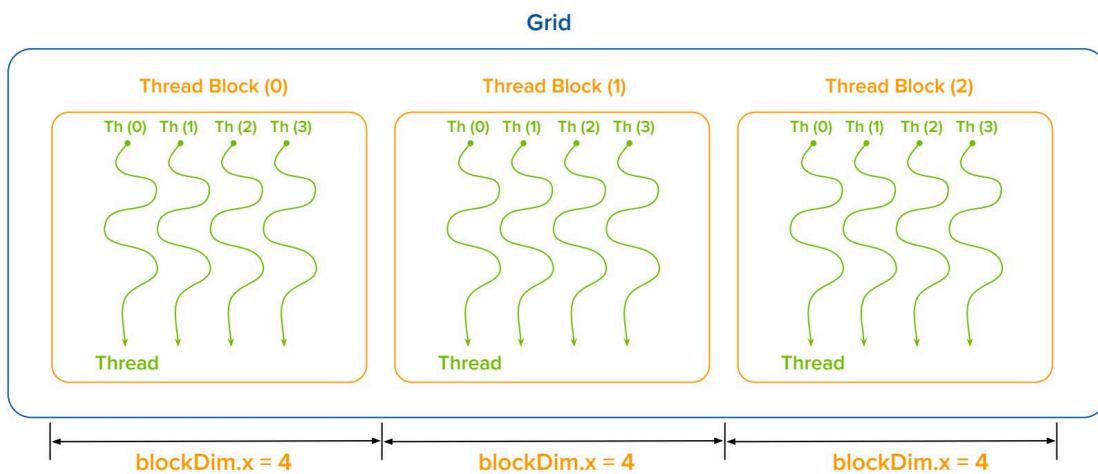
```
struct __device_builtin__ dim3
{
    unsigned int x, y, z;
    __if_defined(__cplusplus)
    __host__ __device__ dim3(unsigned int vx = 1, unsigned int vy = 1, unsigned int vz = 1) : x(vx),
    y(vy), z(vz) {}
    __host__ __device__ dim3(uint3 v) : x(v.x), y(v.y), z(v.z) {}
}
```

```
__host__ __device__ operator uint3(void) { uint3 t; t.x = x; t.y = y; t.z = z; return t; }
#endif / __cplusplus /
};
```

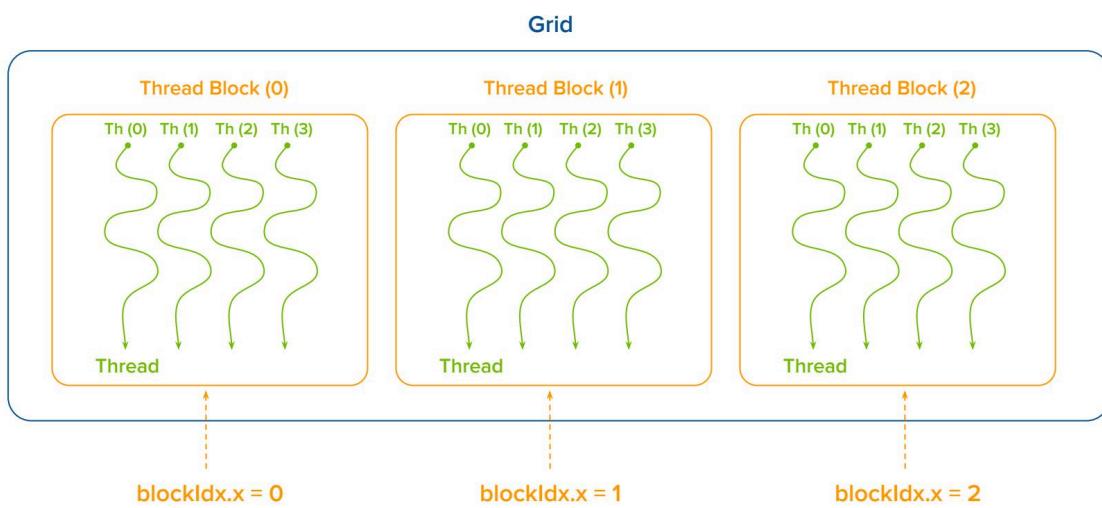
gridDim.x: Numero di blocchi nella griglia, in questo caso 3.



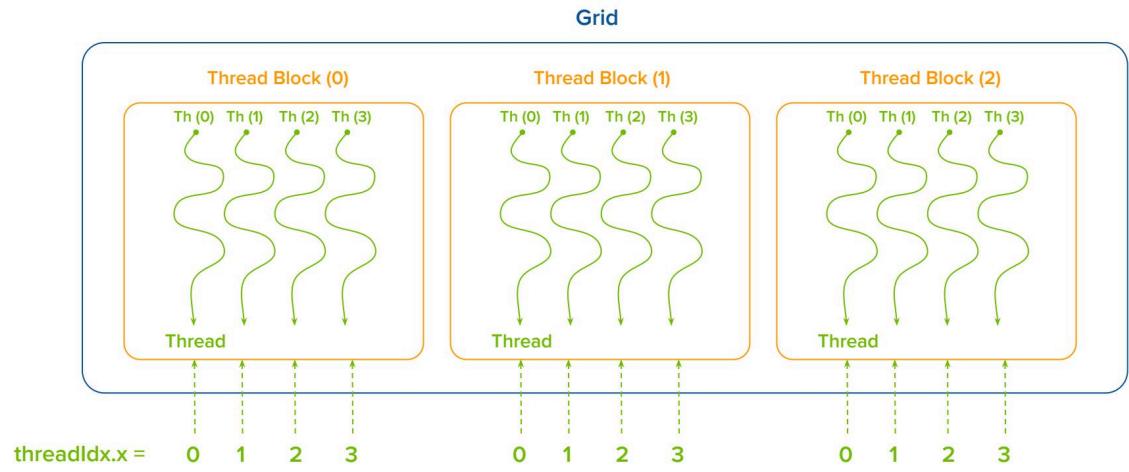
blockDim.x: Numero di thread per blocco, in questo caso 4.



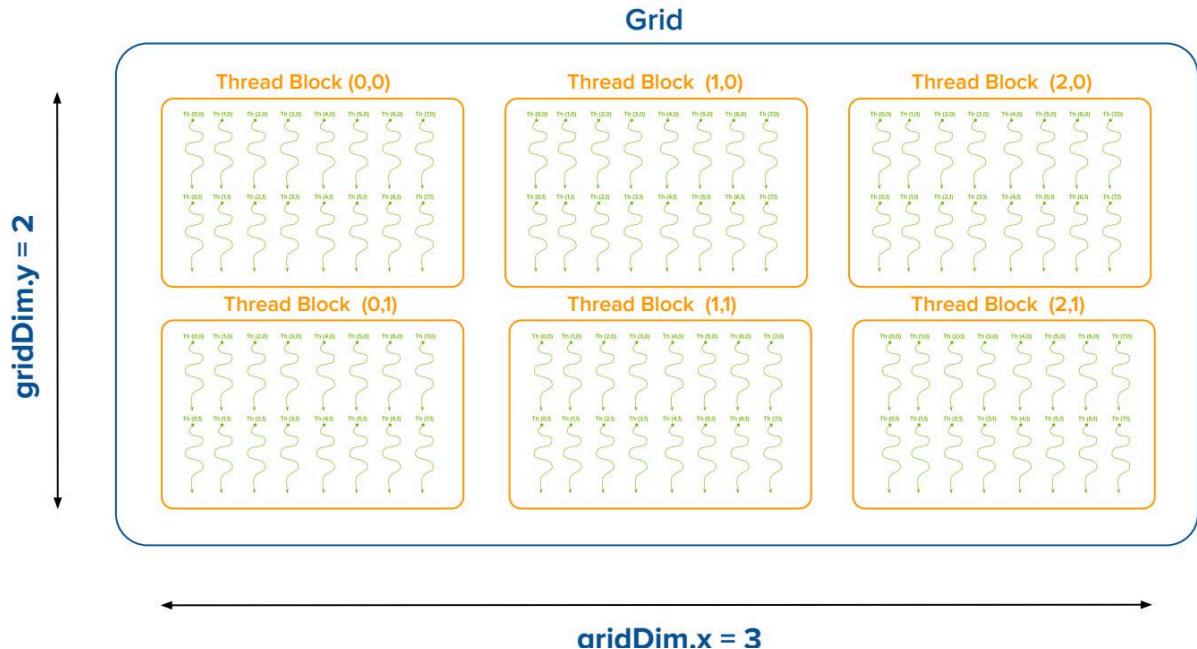
blockIdx.x: Indice di un blocco nella griglia.



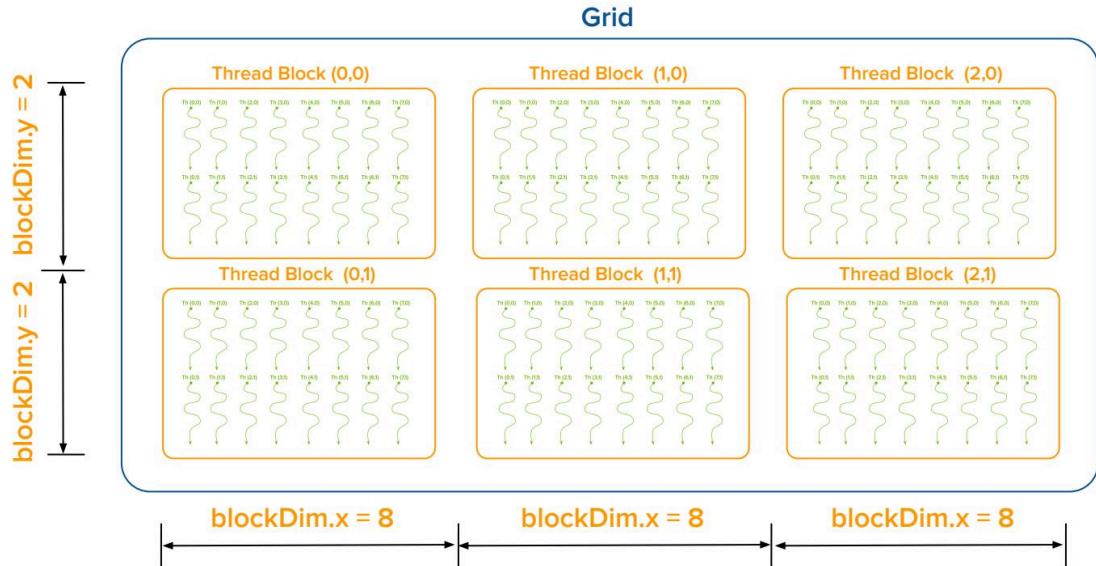
threadIdx.x: Indice del thread all'interno del blocco.



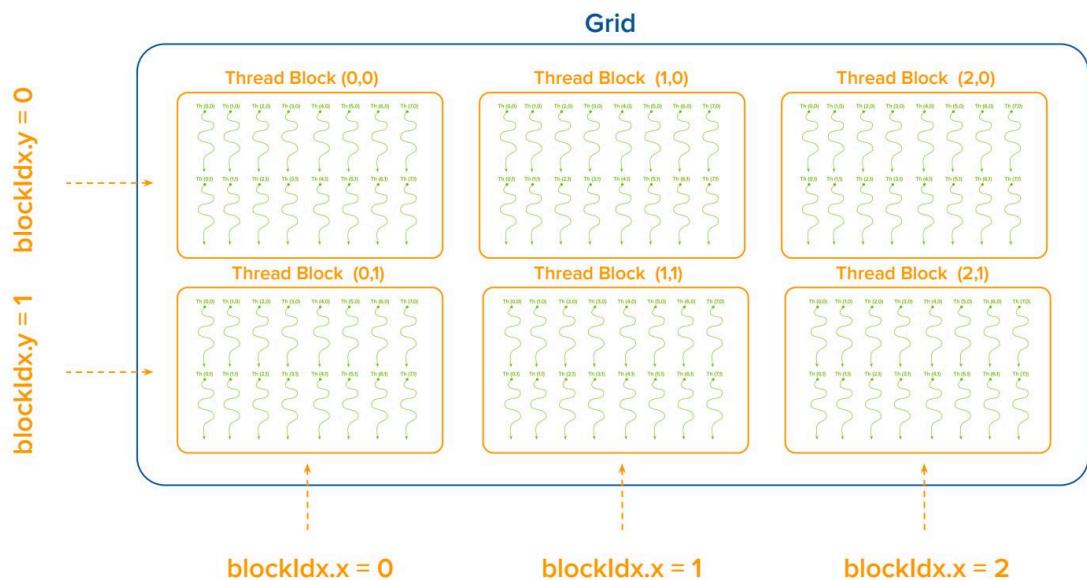
`gridDim.x, gridDim.y`: Numero di blocchi nella griglia lungo le dimensioni x e y.



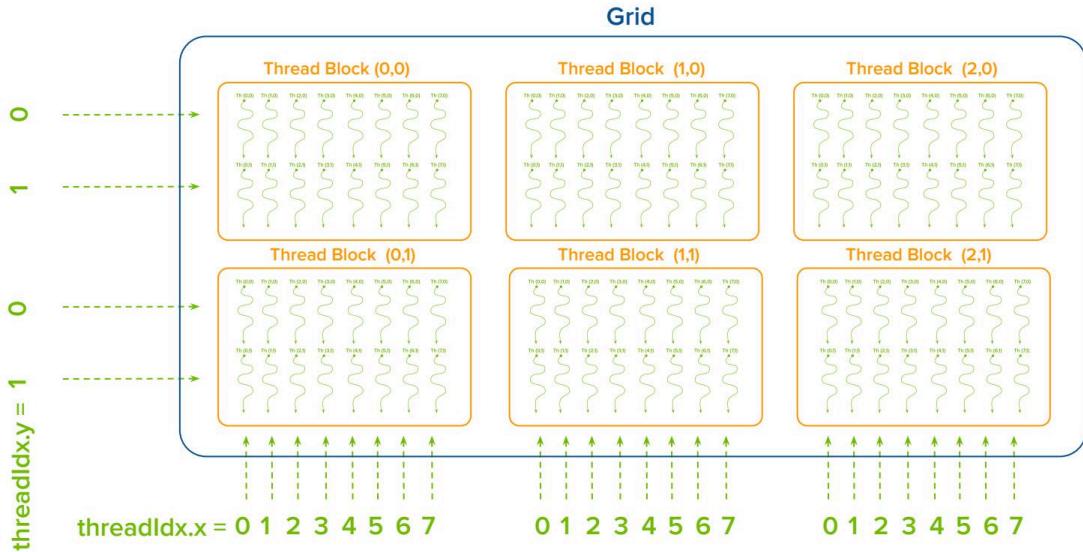
`blockDim.x, blockDim.y`: Numero di thread per blocco lungo le dimensioni x e y.



`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

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

CUDA

- 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);
```

C

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 Parallello sul Dispositivo?

- Sequenziale (non ottimale): `kernel_name<<<1, 1>>>(args); // 1 blocco, 1 thread per blocco`
- Parallello: `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
<code>__global__</code>	Sul Device	Dall'Host	Deve avere tipo di ritorno void
<code>__device__</code>	Sul Device	Solo dal Device	
<code>__host__</code>	Sull'Host	Solo dall'Host	Può essere omesso

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

CUDA

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

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

CUDA

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

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

CUDA

- 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.

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

CUDA

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.

```
// 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);
```

CUDA

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.

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

CUDA

```
// 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.

```
// 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);

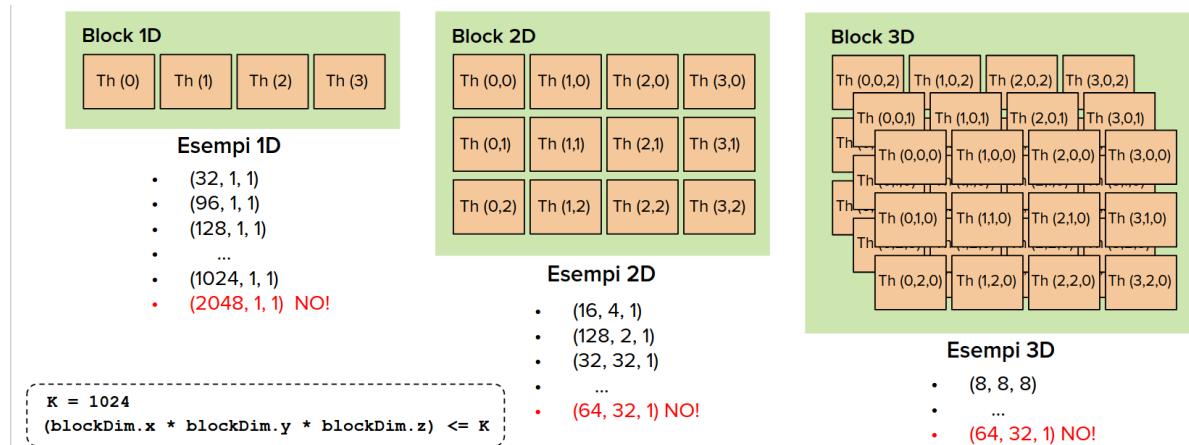
// 3D Grid, 3D Block
dim3 gridSize(4, 2, 2);
dim3 blockSize(8, 4, 2);
kernel_name<<<gridSize, blockSize>>>(args);
```

CUDA

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

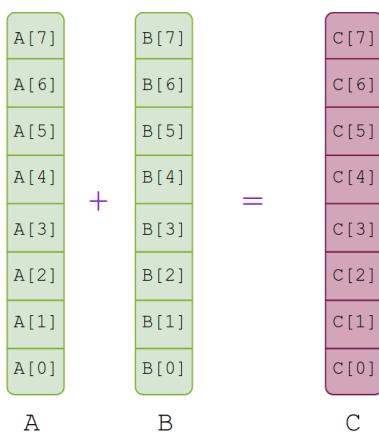
```
#include <cuda_runtime.h>
// Kernel
__global__ void kernel_name() {
    // Accesso alle variabili built-in
    int blockIdx_x = blockIdx.x, blockIdx_y = blockIdx.y, blockIdx_z = blockIdx.z;
    int threadIdx_x = threadIdx.x, threadIdx_y = threadIdx.y, threadIdx_z = threadIdx.z;
    int totalThreads_x = blockDim.x, totalThreads_y = blockDim.y, totalThreads_z = blockDim.z;
    int totalBlocks_x = gridDim.x, totalBlocks_y = gridDim.y, totalBlocks_z = gridDim.z;
    // Logica del kernel...
}
int main() {
    // Definizione delle dimensioni della griglia e del blocco (Caso 3D)
    dim3 gridDim(4, 4, 2); // 4x4x2 blocchi
    dim3 blockDim(8, 8, 4); // 8x8x4 thread per blocco
    // Lancio del kernel
    kernel_name<<<gridDim, blockDim>>>();
    // Resto del Programma
}
```

CUDA

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

```
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);
```

C

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

```
__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);
```

CUDA

Caratteristiche

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

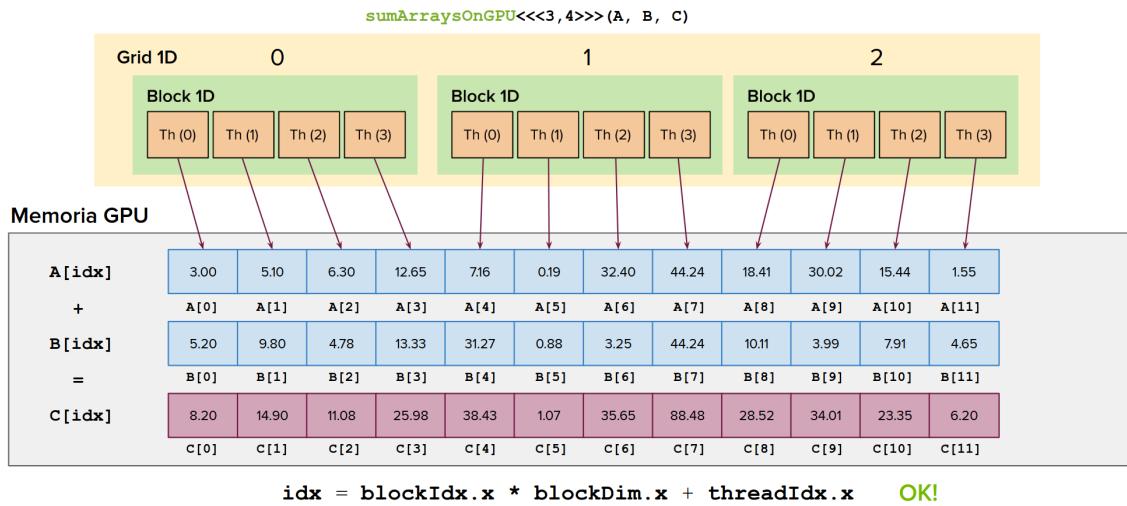
Vantaggi

- 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 adegu 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

```

__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);

```

CUDA

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?

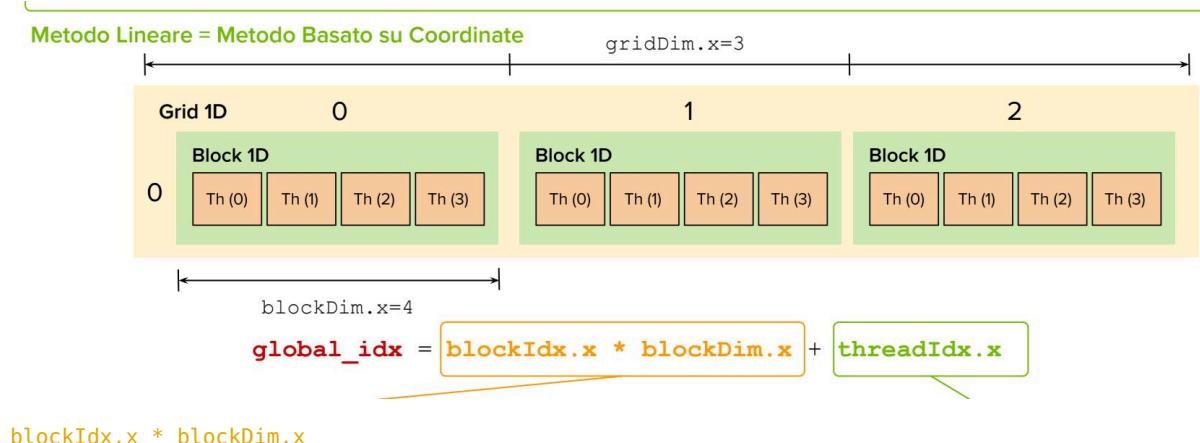
- L'indice globale di un thread identifica univocamente quale porzione di dati deve elaborare.
- Essenziale per gestire correttamente l'accesso alla memoria e coordinare l'esecuzione di algoritmi complessi.

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.

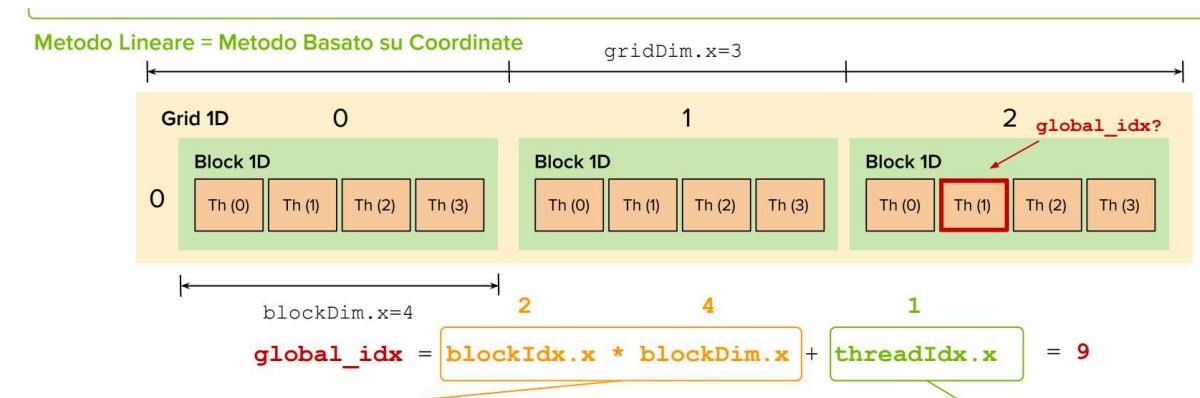


- Calcola l'offset di tutti i thread nei blocchi precedenti al blocco corrente.
- Moltiplicando blockIdx.x per blockDim.x, otteniamo il numero totale di thread che si trovano nei blocchi precedenti.

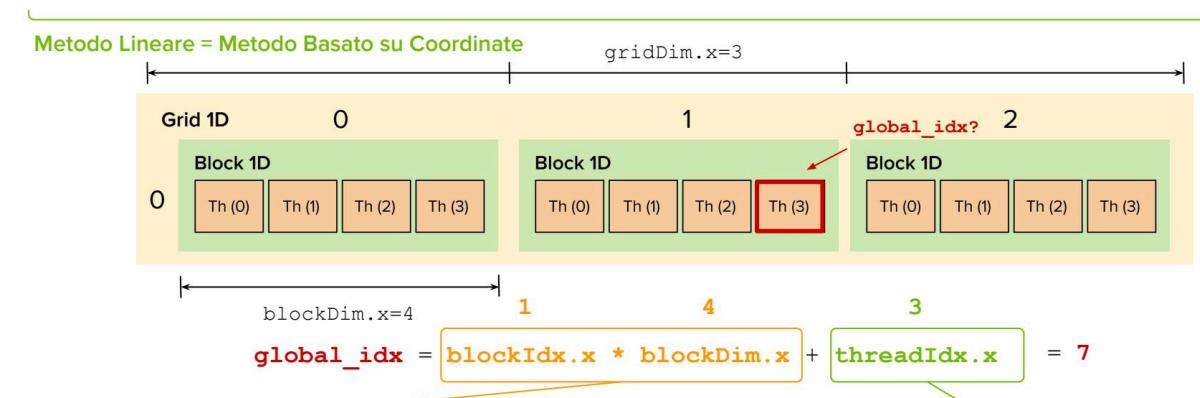
threadIdx.x

- 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.

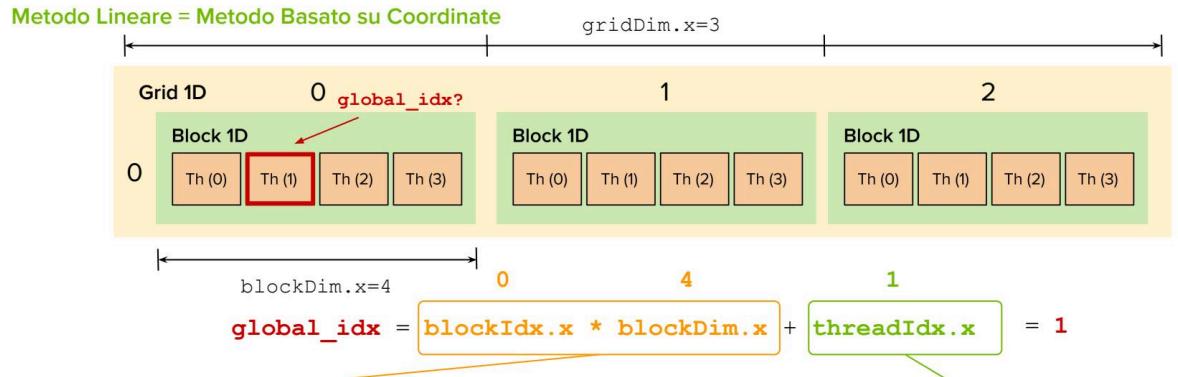
In questo esempio viene mostrato come calcolare l'indice globale per il thread Th(1) appartenente al blocco unidimensionale con indice blockIdx.x = 2



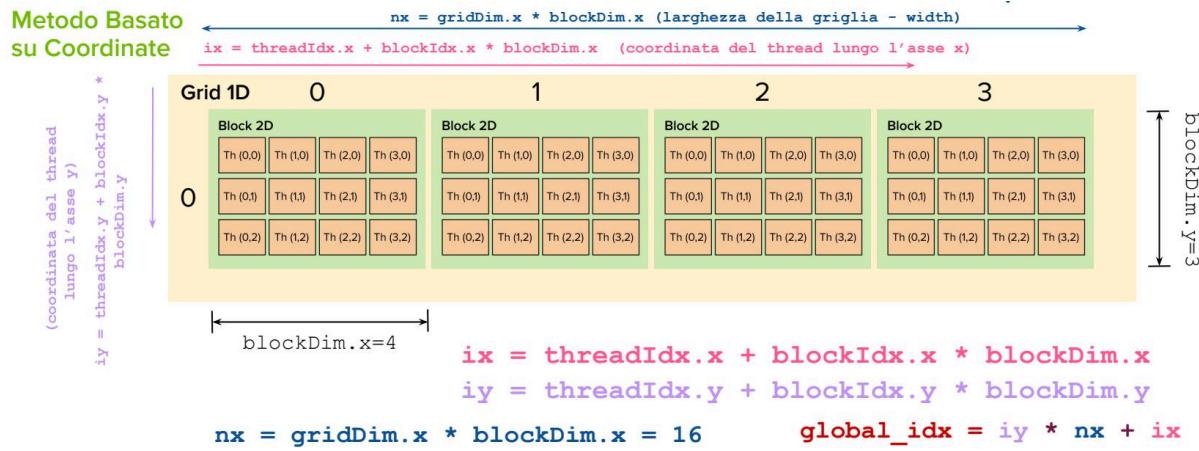
In questo esempio viene mostrato come calcolare l'indice globale per il thread Th(3) appartenente al blocco unidimensionale con indice blockIdx.x = 1



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

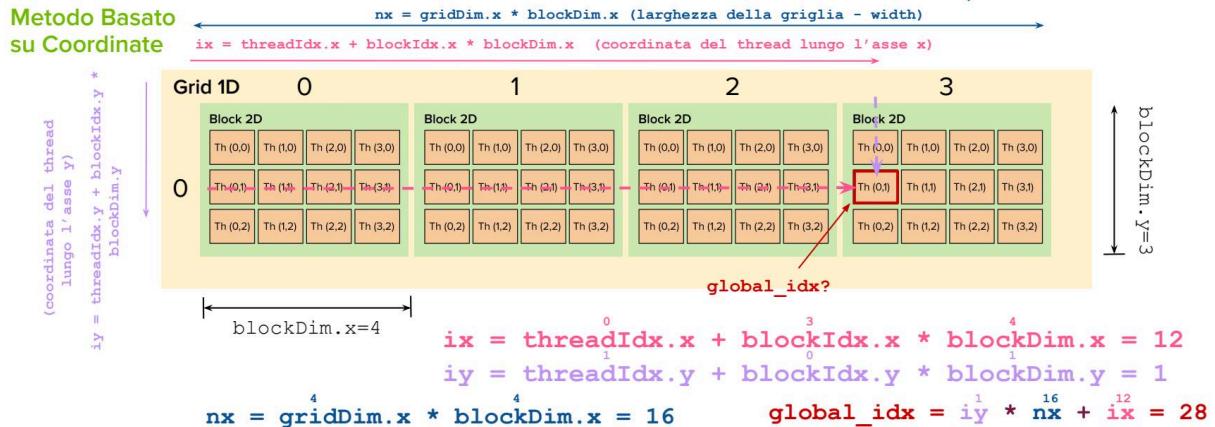


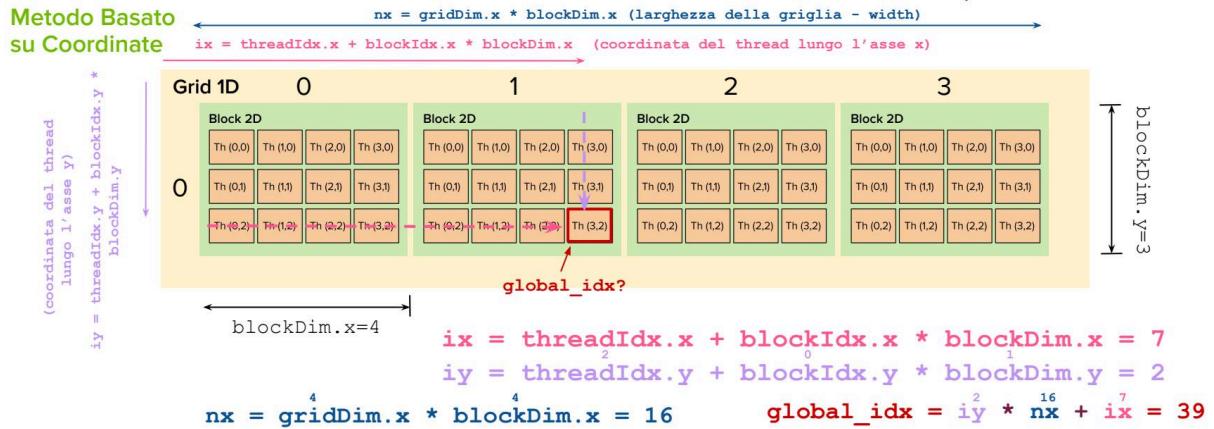
2.27 Calcolo dell'Indice Globale del Thread - Grid 1D, Block 2D



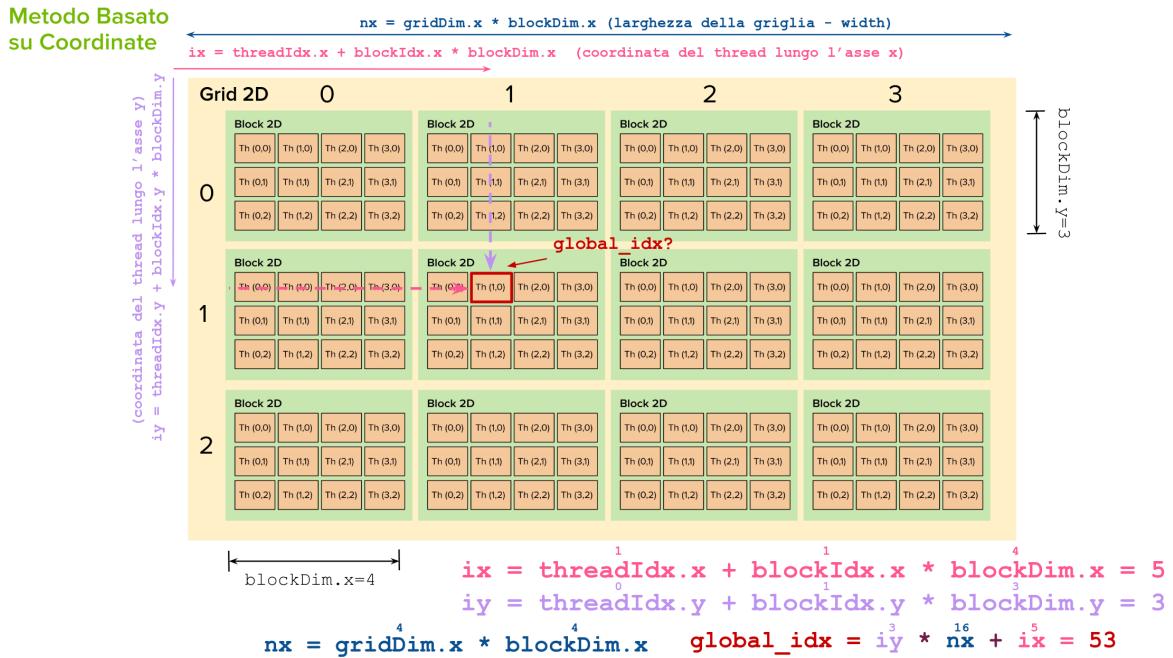
- $ix = \text{threadIdx.x} + \text{blockIdx.x} * \text{blockDim.x}$: determina l'indice del thread lungo l'asse x, prendendo in considerazione la posizione nel blocco (threadIdx.x) e il numero di blocchi precedenti ($\text{blockIdx.x} * \text{blockDim.x}$).
- $iy = \text{threadIdx.y} + \text{blockIdx.y}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

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

Calcolo degli Indici Coordinati

- Caso 1D) $x = blockIdx.x * blockDim.x + threadIdx.x \longrightarrow \text{idx} = x$ (equivalente al caso lineare)
- Caso 2D) $x = blockIdx.x * blockDim.x + threadIdx.x$
 $y = blockIdx.y * blockDim.y + threadIdx.y \longrightarrow \text{idx} = y * \frac{(nx)}{\text{width}} + x$
- Caso 3D) $x = blockIdx.x * blockDim.x + threadIdx.x$
 $y = blockIdx.y * blockDim.y + threadIdx.y$
 $z = blockIdx.z * blockDim.z + threadIdx.z \longrightarrow \text{idx} = z * \frac{(ny)}{\text{height} * \text{width}} + y * \frac{(nx)}{\text{width}} + x$

Calcolo dell'Indice Globale

2.29.2 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;
```

CUDA

```

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 $(\text{dataSize} + \text{blockSize} - 1) / \text{blockSize}$ 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. Equivalenti a calcolare la `ceil` della divisione.

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

Calcolo delle Dimensioni (Caso 1D)

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

$$\begin{aligned} 256 * 4 &= 1024 \\ 1030 - 1024 &= 6 \end{aligned}$$

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)

```

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)

```
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
```

CUDA

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.

```
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 %.2f gpu %.2f at current %d\n", hostRef[i], gpuRef[i], i);
            break;
        }
    }
    if (match) printf("Arrays match.\n\n");
}
```

CUDA

- 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
// 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
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

- Timer CPU:** Semplice e diretto, utilizza funzioni di sistema per ottenere il tempo di esecuzione.
- NVIDIA Profiler** (deprecato): Strumento da riga di comando per analizzare attività di CPU e GPU.
- 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

```

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();
    initData(h_A, nElem); // Inizializza il vettore A
    initData(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;

```

Somma di due array

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(hostRef, d_C, nBytes, cudaMemcpyDeviceToHost)); // Copia dalla GPU

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

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

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

```

```
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:

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

CUDA

```
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)

```
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;
    }
}
```

CUDA

Compilazione con nvcc

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

\$ Shell

Esecuzione e Risultato

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

\$ Shell

WorkStation

Intel Core i9-10920X (CPU)

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

NVIDIA GeForce RTX 3090 (GPU)

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

Accesso ai Dati

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

Laptop

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

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

NVIDIA GeForce RTX 3070 (GPU)

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

Accesso ai Dati

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

2.30.4.2 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.

[Documentazione online](#)

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

\$ Shell

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

```
$ nvprof --help
```

\$ Shell

Output integrabile in Visual Profiler con: `nvprof -o file.nvvp ./app`

Nel nostro esempio:

```
$ nvprof ./array_sum
```

\$ Shell

Nota

- nvprof non è supportato su dispositivi con Compute Capability ≥ 8.0 (Ampere+).
- Per queste GPU, si consiglia di utilizzare **NVIDIA Nsight Systems** per il tracing della CPU/GPU, e **NVIDIA Nsight Compute** per il profiling della GPU.
- Ancora disponibile su ambienti come Google Colab (GPU NVIDIA Tesla T4 Compute Capability: 7.5).

Esempio di Profilazione su Google Colab

```

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

==1380== Profiling application: ./array_sum 16777216 256
==1380== Profiling result:
          Type  Time(%)      Time     Calls    Avg      Min      Max
GPU activities:   64.94% 29.658ms      2 14.829ms 14.756ms 14.983ms
                  33.36% 15.238ms      1 15.238ms 15.238ms 15.238ms
                  1.78% 776.75us      1 776.75us 776.75us 776.75us
API calls:       65.05% 95.410ms      1 95.410ms 95.410ms 95.410ms
                  31.16% 45.708ms      3 15.236ms 14.934ms 15.655ms
                  2.43% 3.5698ms      3 1.1899ms 266.77us 2.1799ms
                  0.53% 779.98us      1 779.98us 779.98us 779.98us
                  0.47% 695.52us      3 231.84us 146.46us 381.48us
                  0.16% 239.79us      1 239.79us 239.79us 239.79us
                  0.10% 142.14us      14 1.2460us 141ns 57.528us
                  0.07% 102.36us      1 102.36us 102.36us 102.36us
                  0.01% 13.085us      1 13.085us 13.085us 13.085us
                  0.00% 6.7260us      1 6.7260us 6.7260us 6.7260us
                  0.00% 4.7470us      1 4.7470us 4.7470us 4.7470us
                  0.00% 2.5600us      3 853ns 249ns 2.0180us
                  0.00% 1.0920us      2 546ns 164ns 928ns
                  0.00% 835ns        1 835ns 835ns 835ns
                  0.00% 236ns        1 236ns 236ns 236ns

```

NVIDIA Profiler

Include informazioni su:

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

2.30.4.3 NVIDIA Nsight Systems

Cos'è? ([Documentazione Online](#))

- Strumento avanzato di **profilazione** e **analisi** delle prestazioni a livello di sistema.
- Offre una **visione globale dell'applicazione**, inclusi CPU, GPU e interazioni con il 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** delle chiamate API CUDA e sincronizzazioni.
- **Monitoraggio** dell'utilizzo di memoria e cache.
- **Supporto** per sistemi multi-thread e 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.

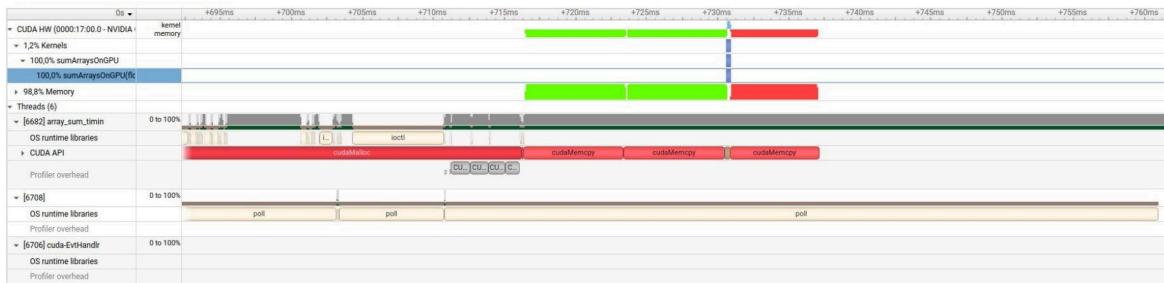
Come si usa?

```
nsys profile --stats=true ./array\_\_sum
```

\$ Shell

Avvia il profiler e produce un'analisi dettagliata (non disponibile su Google Colab per GPU Tesla T4).

Timeline View



CUDA Summary (API/Kernels/MemOps)

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

Analisi del profiling (prima colonna)

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

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

Precisione

- Nsight Systems:** Misurazioni precise, direttamente dalla GPU.
- Timer CPU:** Meno preciso, include overhead extra (lancio, sincronizzazione).

Contesto

- Nsight Systems:** Visione isolata del tempo di esecuzione del kernel GPU.
- Timer CPU:** Include attività di sistema e altri processi, meno preciso.

Affidabilità

- Nsight Systems:** Misurazioni stabili, meno influenzate da fattori esterni.
- Timer CPU:** Vulnerabile alle fluttuazioni del sistema, meno affidabile.

Implicazioni per lo sviluppo

- Nsight Systems:** Ottimizzazioni critiche, analisi approfondite, profiling accurato.
- Timer CPU:** Stime approssimative nelle fasi iniziali, non per analisi dettagliate.

2.30.4.4 NVIDIA Nsight Compute

Cos'è? ([Documentazione Online](#))

- Strumento di **profilazione** e **analisi** approfondita per singoli kernel CUDA.
- Fornisce **metriche dettagliate e mirate** alle prestazioni a livello di kernel.
- Permette di:
 - Analizzare** l'utilizzo delle risorse GPU.
 - Identificare **colli di bottiglia** nei 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.

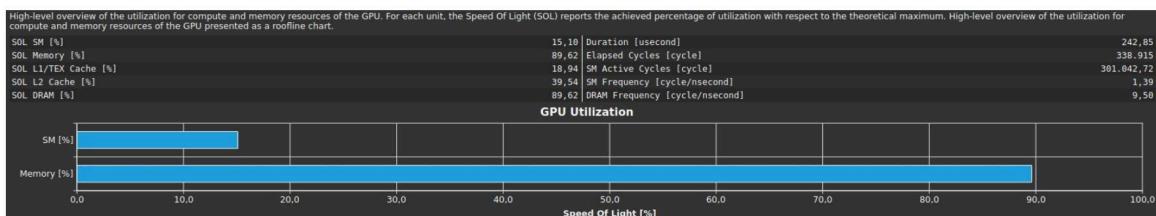
Come si usa?

```
ncu --set full -o test_report ./array_sum
```

\$ Shell

- `-o test_report` è necessario per generare file per la visualizzazione grafica.
- avvia il profiler Nsight Compute e fornisce un'analisi dettagliata delle prestazioni dei kernel CUDA.

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.

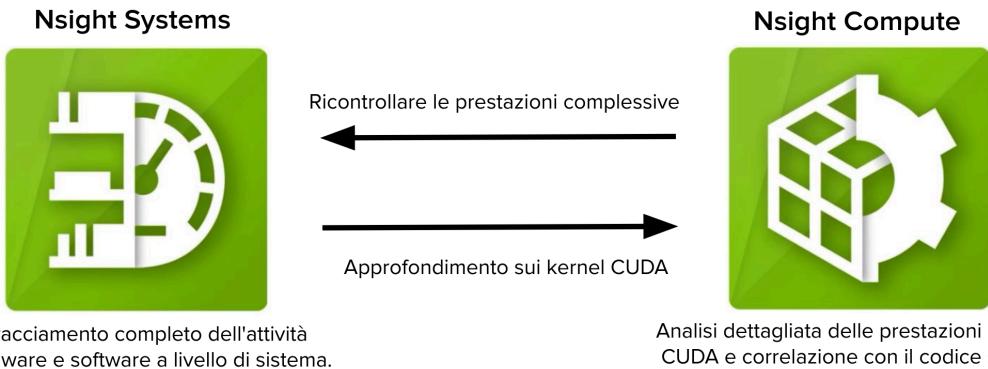
Considerazioni

- 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).

2.30.5 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.



2.30.6 Ottimizzazione della Gestione della Memoria in CUDA

- 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 Practice

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 leak e sprechi di risorse.

Sfruttare la Shared Memory (vedremo in seguito)

- 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.

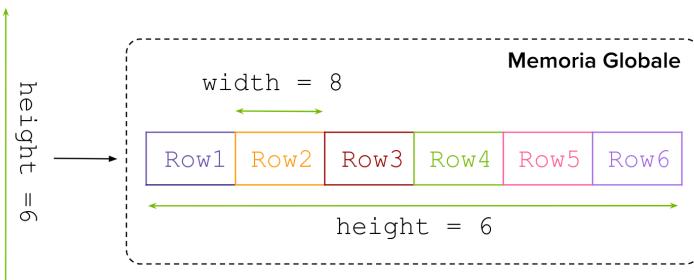
2.31 Applicazione Pratiche

2.31.1 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).

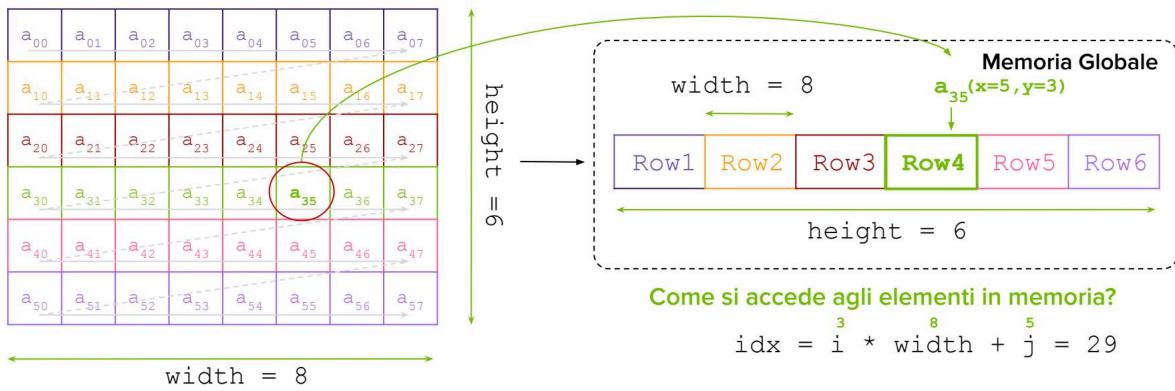
$A(i, j)$ (i : Indice di riga, j : Indice di colonna)

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

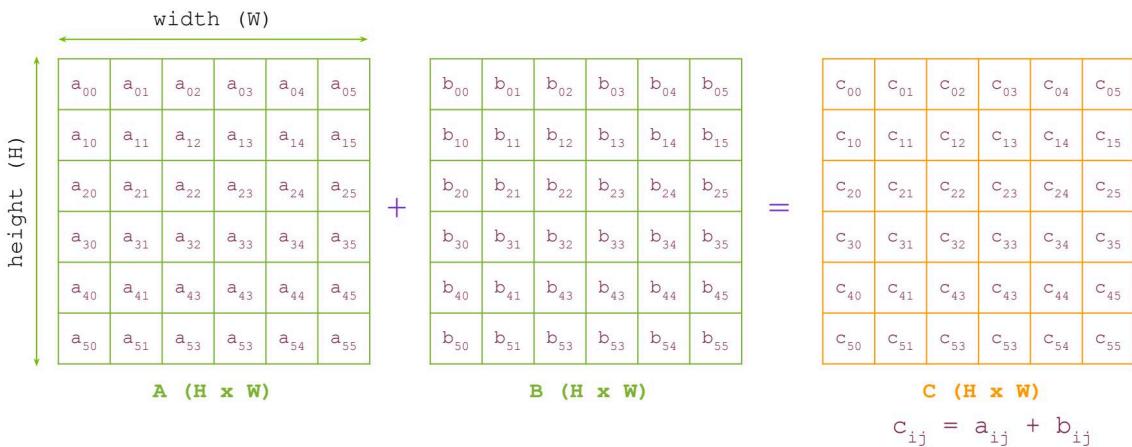


Come si accede agli elementi in memoria?

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



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.

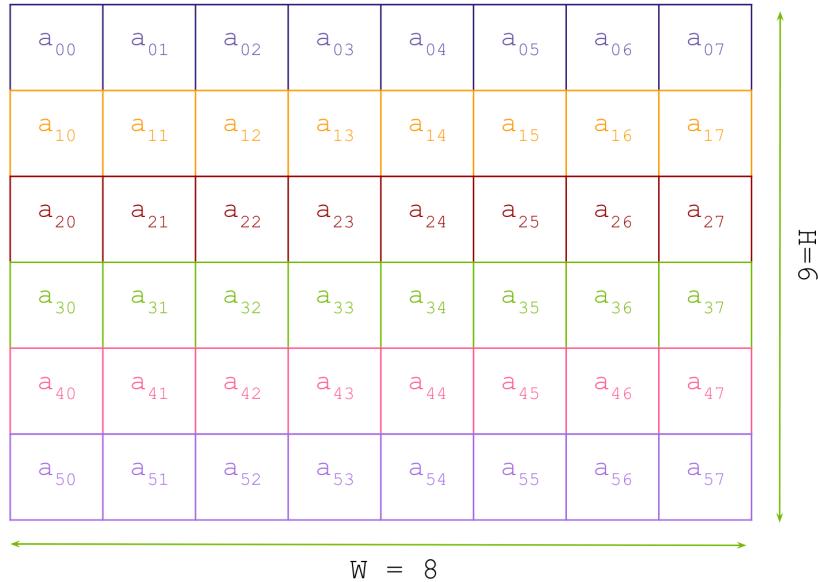
Impatto della Configurazione

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?



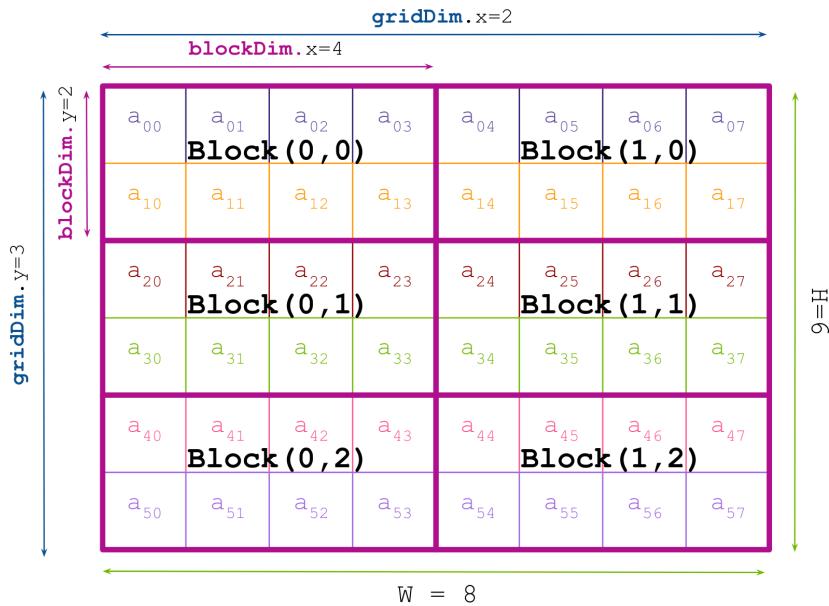
Suddivisione

- 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).

2.31.1.1 Suddivisione della matrice in Griglia 2D e Blocchi 2D



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.

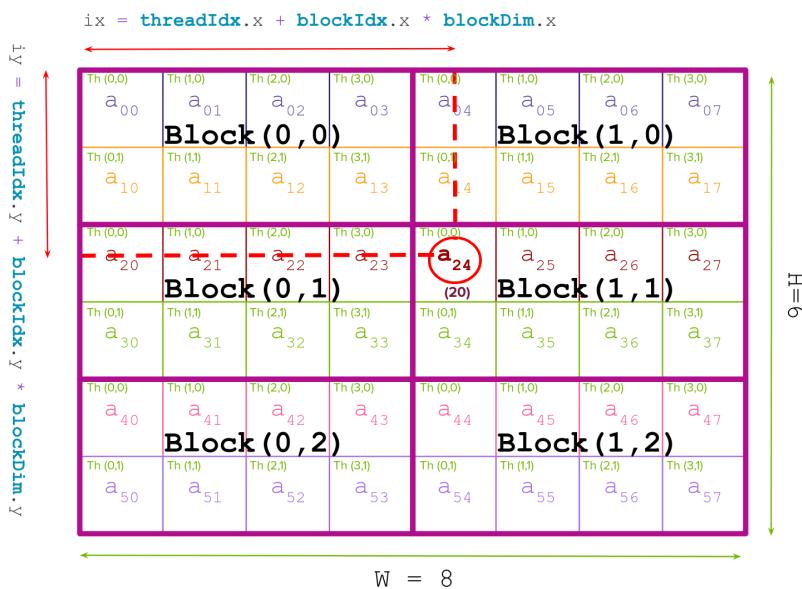
Attenzione!

Ambiguità tra Coordinate delle Matrici e dei Thread in CUDA

- Matrici: a_{ij} segue la convenzione riga, colonna (indice i per la riga e j per la colonna).
 - Esempio:** a_{31} - la prima coordinata i rappresenta la riga (3) mentre la seconda coordinata j rappresenta la colonna (1).
- Thread/Blocco CUDA:** `Th(x,y)/Block(x,y)` utilizza una convenzione basata su coordinata cartesiana, con x, y riferiti alla posizione all'interno del blocco/griglia.
 - Esempio:** `Th(3, 1)` - la prima coordinata x rappresenta la posizione lungo l'asse x (3), mentre la seconda coordinata y rappresenta la posizione lungo l'asse y (1).

Come calcolo l'indice globale?

Sceglieremo un metodo di mapping, ad esempio quello **basato su coordinate** - ci concentriamo su quest'ultimo per la sua maggiore **intuitività**.



Esempio di Mapping (in rosso)

Abbiamo che

- $ix = \text{threadIdx}.x + \text{blockIdx}.x * \text{blockDim}.x$
- $iy = \text{threadIdx}.y + \text{blockIdx}.y * \text{blockDim}.y$
- $idx = iy * W + ix$

quindi:

- Indice x** nella matrice
 - $ix = 0 + 1 * 4 = 4$
- Indice y** nella matrice
 - $iy = 0 + 1 * 2 = 2$
- Indice lineare**
 - $idx = 2 * 8 + 4 = 20$

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

Passi chiave

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

Confronto: Somma di Matrici in C vs CUDA C

Codice C Standard

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

C

Codice CUDA C

CUDA

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

Somma di due matrici

CUDA

```

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

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

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

    // Inizializza i dati delle matrici (casualmente)
    initData(h_A, size);
    initData(h_B, size);

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

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

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

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

```

Somma di due matrici

CUDA

```

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

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

```

```
iElaps = cpuSecond() - iStart;

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

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

// continue...
```

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

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

Osservazioni

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

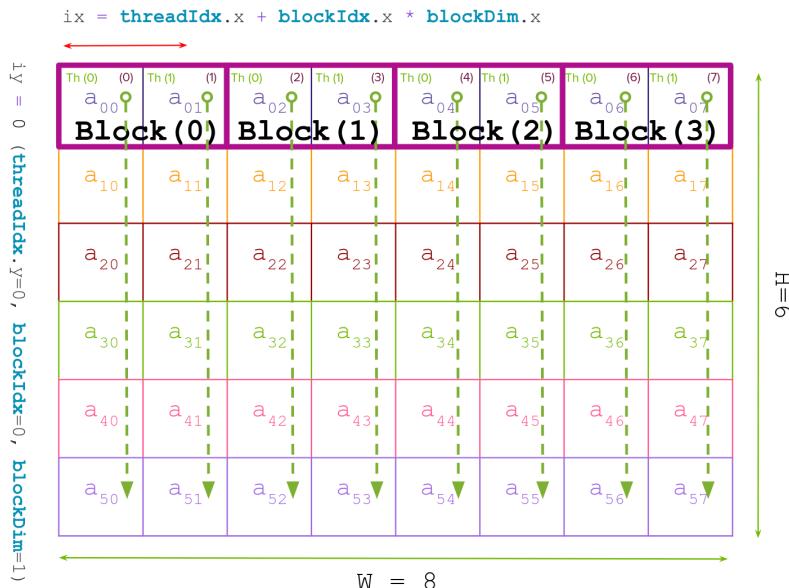
Perché il blocco di dimensioni (1,1) è inefficiente?

- Overhead di gestione:** Lanciare tanti blocchi singoli crea un enorme overhead di scheduling e gestione per la GPU.
- Mancato sfruttamento della località:** I thread non sono raggruppati in modo da sfruttare efficientemente la memoria cache e la memoria condivisa dei blocchi.
- Inefficienza nell'utilizzo dei warp:** Le GPU operano su gruppi di thread chiamati *warp* (tipicamente 32 thread). Con un thread per blocco, la maggior parte delle unità di elaborazione in ogni warp rimane inutilizzata.

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

- **Utilizzo della memoria (Memory [%])**
 - **Blocco 2D (1,1)**: 10,49%
 - **Blocco 2D (16,16)**: 94,28%
- **Throughput di memoria:**
 - **Blocco 2D (1,1)**: 14,42 GB/s
 - **Blocco 2D (16,16)**: 860,79 GB/s
- **SM Busy:**
 - **Blocco 2D (1,1)**: 5,64%
 - **Blocco 2D (16,16)**: 10,3%
- **Occupancy Risorse:** Aumento di 5,1 volte nell'occupancy, indicando un uso molto più efficiente delle risorse disponibili.
- **Utilizzo della memoria:** Miglioramento drammatico nell'utilizzo della larghezza di banda nel caso (16,16), ottimizzando gli accessi alla memoria.
- **Throughput di memoria:** Aumento di circa 60 volte, principale fattore del boost di performance complessivo.

2.31.1.2 Suddivisione della Matrice in Griglia 1D e Blocchi 1D



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 utilizzando gli indici del blocco e quelli locali lungo l'asse x
- ```
idx = ix
```
- Ogni thread **elabora una colonna** della matrice (parallelismo limitato)

### 2.31.1.3 Confronto Kernel CUDA per la Somma fra Matrici

#### Griglia 2D e Blocchi 2D

CUDA

```
// 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**

CUDA

```

__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 2D & Blocchi 2D**

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

**Griglia 1D & Blocchi 1D**

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

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*             | <b>71,68x</b>  | 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 griglia 2D e blocchi 2D 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.

**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)  |        |

| <b>Griglia 1D, Blocchi 1D (un thread elabora una colonna)</b> |              |             |              |                  |                |
|---------------------------------------------------------------|--------------|-------------|--------------|------------------|----------------|
| Dim. Matrice                                                  | Dim. Griglia | Dim. Blocco | Runtime (ms) | Speedup (vs CPU) | Device         |
| (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) |

| <b>Griglia 2D, Blocchi 2D (un thread elabora un singolo elemento)</b> |                     |             |              |                  |                |
|-----------------------------------------------------------------------|---------------------|-------------|--------------|------------------|----------------|
| Dim. Matrice                                                          | Dim. Griglia        | Dim. Blocco | Runtime (ms) | Speedup (vs CPU) | Device         |
| (16384, 16384)                                                        | (1024, 1024)        | (16, 16)    | 3,74*        | 137,99x          | RTX 3090 (GPU) |
| (32768, 8192)                                                         | (512, 2048)         | (16, 16)    | 3,73*        | 138,39x          | RTX 3090 (GPU) |
| (65536, 4096)                                                         | (256, 4096)         | (16, 16)    | 3,75*        | 137,86x          | RTX 3090 (GPU) |
| (1048576, 256)                                                        | (16, <u>65536</u> ) | (16, 16)    | x            | x                | RTX 3090 (GPU) |

- Limite dei blocchi:** Limite massimo dei 65535 thread per blocco sull'asse **y** (vedi compute capability GPU)
- Necessità di adattamento:** Per gestire matrici con un numero di righe superiori è necessario modificare la configurazione per suddividere il lavoro in modo diverso.

**NVIDIA Nsight Compute\*****Se aumentiamo solo le righe mantenendo size costante?**

| 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)  |        |

| <b>Griglia 1D, Blocchi 1D (un thread elabora una colonna)</b> |              |             |              |                  |                |
|---------------------------------------------------------------|--------------|-------------|--------------|------------------|----------------|
| Dim. Matrice                                                  | Dim. Griglia | Dim. Blocco | Runtime (ms) | Speedup (vs CPU) | Device         |
| (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) |

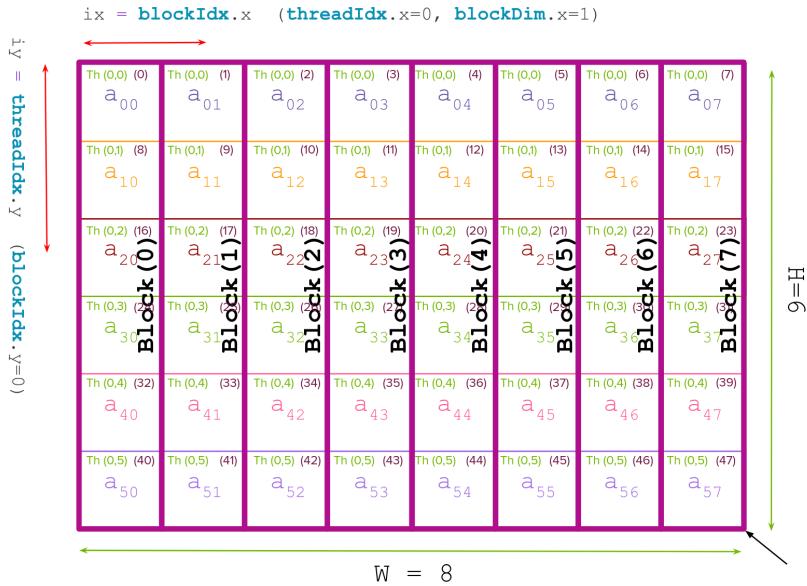
  

| <b>Griglia 2D, Blocchi 2D (un thread elabora un singolo elemento)</b> |                    |             |              |                  |                |
|-----------------------------------------------------------------------|--------------------|-------------|--------------|------------------|----------------|
| Dim. Matrice                                                          | Dim. Griglia       | Dim. Blocco | Runtime (ms) | Speedup (vs CPU) | Device         |
| (16384, 16384)                                                        | (1024, 1024)       | (16, 16)    | 3,74*        | 137,99x          | RTX 3090 (GPU) |
| (32768, 8192)                                                         | (512, 2048)        | (16, 16)    | 3,73*        | 138,39x          | RTX 3090 (GPU) |
| (65536, 4096)                                                         | (256, 4096)        | (16, 16)    | 3,75*        | 137,86x          | RTX 3090 (GPU) |
| (1048576, 256)                                                        | (8, <u>32768</u> ) | (16, 16)    | 4,00*        | 131,09x          | RTX 3090 (GPU) |

**Tendenza nelle Prestazioni**

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

### 2.31.1.4 Suddivisione della Matrice in Griglia 1D e Blocchi 2D



#### Organizzazione della Griglia

- La matrice è divisa, in questo caso specifico, in **8 blocchi**, in una configurazione 1D ( $\text{gridDim.x} = 8$ ).
- Ogni blocco ha configurazione 2D e contiene 6 thread ( $\text{blockDim.x} = 1, \text{blockDim.y} = 6$ ) - 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
```

#### 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)

### 2.31.1.5 Confronto Kernel CUDA per la Somma fra Matrici - 2D2D 1D2D

#### 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 + blockDim.x * blockIdx.x; // Calcola indice x globale
 unsigned int iy = threadIdx.y + blockDim.y * 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
 }
}
```

CUDA

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

```
__global__ void sumMatrixOnGPU1D2D(float MatA, float MatB, float MatC, int W, int H) {
 unsigned int ix = blockIdx.x; // Calcola indice x globale
 unsigned int iy = threadIdx.y; // Calcola indice y globale
 if (ix < W && iy < H) { // Controlla limiti matrice
```

CUDA

```

 unsigned int idx = iy * W + ix; // Calcola indice lineare
 MatC[idx] = MatA[idx] + MatB[idx]; // Somma elementi corrispondenti
}
}

```

### Griglia 2D & Blocchi 2D

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

### Griglia 1D & Blocchi 1D (degenero)

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

**Nota:** Il primo kernel (2D-2D) è **più generale**: può gestire qualsiasi configurazione di griglia/blocchi. Il secondo (1D-2D) è una **versione specializzata**:

**Esempio:** Configurando il lancio del primo kernel con dim3 **block** (1, blockDim\_y); dim3 **grid** (W, 1); le formule diventano:

```

ix = threadIdx.x + blockIdx.x * blockDim.x → ix = 0 + blockIdx.x * 1
iy = threadIdx.y + blockIdx.y * blockDim.y → iy = threadIdx.y + 0

```

### NVIDIA Nsight Compute\*

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

**Note sulla configurazione (2048, 16384)**

- Limite dei thread:** Limite massimo dei 1024 thread per blocco (vedi compute capability GPU)
- Necessità di adattamento:** Per gestire matrici così grandi con questa configurazione, sarebbe necessario modificare il codice per suddividere il lavoro in modo diverso. Come?

**Analisi Dettagliata da Nsight Compute (2D2D vs 1D2D)****Utilizzo della memoria (Memory [%])**

- 1D2D:** 83,95%
- 2D2D:** 89,40%

**SM Busy:**

- 1D2D:** 1,66%
- 2D2D:** 11,28%

**Throughput di memoria:**

- 1D2D:** 164,20 GB/s
- 2D2D:** 810,80 GB/s

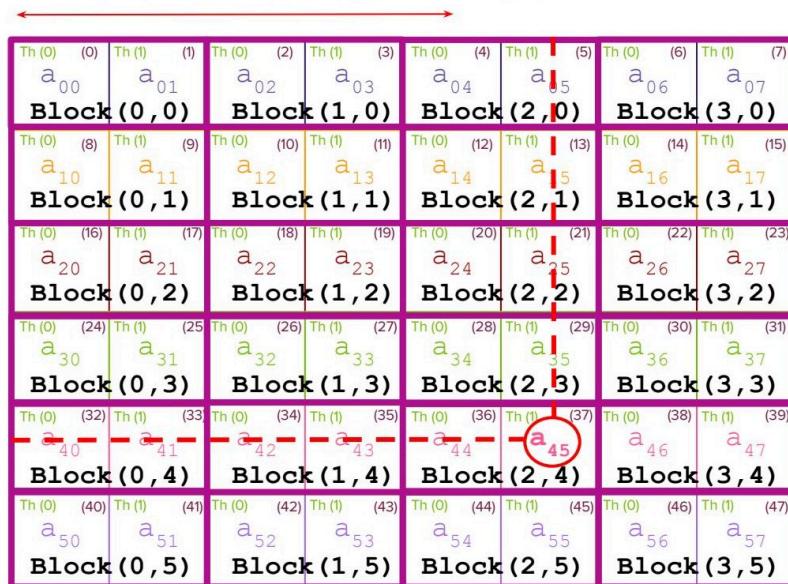
**Cicli di stallo per istruzione:**

- 1D2D:** 361,92 cicli
- 2D2D:** 50,85 cicli

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

**2.31.1.6 Suddivisione della Matrice in Griglia 2D e Blocchi 1D**

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



### 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**

$$idx = iy * W + ix = 4 * 8 + 5 = 37$$

L'indice 37 corrisponde all'elemento  $a_{24}$ .

### 2.31.1.7 Confronto Kernel CUDA per la Somma fra Matrici - 2D2D 2D1D

#### 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
 }
}
```

CUDA

#### 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
 }
}
```

CUDA

**Nota:** Anche in questo caso il primo kernel (2D-2D) è **più generale** - può gestire qualsiasi configurazione di griglia/blocchi.

Il secondo (2D-1D) è una **versione specializzata**:

**Esempio:** Configurando il lancio del primo kernel con `cudaLaunchKernel` (`dim3 block (blockDim_x, 1); dim3 grid (gridDim_x, H)`); le formule diventano:

```
ix = threadIdx.x + blockIdx.x * blockDim.x → rimane invariata
iy = threadIdx.y + blockIdx.y * 1 → iy = threadIdx.y * 0
```

**Strategia inversa:** Qui usiamo threading solo su X e griglia su Y, mentre nel caso precedente (1D-2D) era il contrario.

**NVIDIA Nsight Compute\*****Dim. Matrice** (16384, 16384)

| <b>Dim. Griglia</b> | <b>Dim. Blocco</b> | <b>Runtime (ms)</b> | <b>Speedup vs CPU</b> | <b>Device</b>   |
|---------------------|--------------------|---------------------|-----------------------|-----------------|
| --                  | --                 | 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*               | <b>137,62x</b>        | RTX 3090 (GPU)  |
| (128, 16384)        | 128                | 3,75*               | <b>137,62x</b>        | RTX 3090 (GPU)  |
| (64, 16384)         | 256                | 3,75*               | <b>137,62x</b>        | RTX 3090 (GPU)  |
| (32, 16384)         | 512                | 3,76*               | 137,25x               | RTX 3090 (GPU)  |

**Osservazioni**

- Le migliori prestazioni si raggiungono con configurazioni a più di 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)

**2.31.1.8 Confronto fra le Migliori Configurazioni di Blocchi e Griglie****NVIDIA Nsight Compute\*****Dim. Matrice** (16384, 16384)

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

**Osservazioni**

- L'approccio **Grid 1D e Blocchi 1D** mostra prestazioni generalmente inferiori, con uno speedup massimo di **71,68x** rispetto alla CPU (il loop per thread limita le prestazioni).
- L'approccio **Grid 1D e Blocchi 2D** (degenero) non è in grado di gestire queste dimensioni della matrice (righe > 1024) senza modifiche al codice. Ogni thread dovrebbe processare più elementi della matrice.
- L'approccio **Grid 2D e Blocchi 1D** raggiunge prestazioni equivalenti, ma sacrifica la semplicità concettuale del mapping diretto matrice ↔ griglia/blocchi 2D.
- 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.

## 2.32 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**.



Figure 49: Immagine a Colore (RGB)

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



Figure 50: Immagine Grayscale

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

| Immagine a Colore (RGB) |                 |                 |                 |                 |                 |                 |                 |
|-------------------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| B <sub>00</sub>         | B               | B               | B               | B               | B               | B               | B               |
| B <sub>10</sub>         | G <sub>00</sub> | G               | G               | G               | G               | G               | G               |
| B <sub>20</sub>         | G <sub>10</sub> | R <sub>00</sub> | R <sub>01</sub> | R <sub>02</sub> | R <sub>03</sub> | R <sub>04</sub> | R <sub>05</sub> |
| B <sub>30</sub>         | G <sub>20</sub> | R <sub>10</sub> | R <sub>11</sub> | R <sub>12</sub> | R <sub>13</sub> | R <sub>14</sub> | R <sub>15</sub> |
| B <sub>40</sub>         | G <sub>30</sub> | R <sub>20</sub> | R <sub>21</sub> | R <sub>22</sub> | R <sub>23</sub> | R <sub>24</sub> | R <sub>25</sub> |
| B <sub>50</sub>         | G <sub>40</sub> | R <sub>30</sub> | R <sub>31</sub> | R <sub>32</sub> | R <sub>33</sub> | R <sub>34</sub> | R <sub>35</sub> |
|                         | G <sub>50</sub> | R <sub>40</sub> | R <sub>41</sub> | R <sub>43</sub> | R <sub>43</sub> | R <sub>44</sub> | R <sub>45</sub> |
|                         |                 | R <sub>50</sub> | R <sub>51</sub> | R <sub>53</sub> | R <sub>53</sub> | R <sub>54</sub> | R <sub>55</sub> |
|                         |                 |                 |                 |                 |                 | R <sub>56</sub> |                 |

Figure 51: Immagine a Colore (RGB)

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

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

Figure 52: Immagine Grayscale

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

### 2.32.1 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 (ogni pixel occupa 3 posizioni in memoria).

### Approccio di Memorizzazione (Caso RGB)

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

#### 1. Planar:

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

|                 |                 |                 |                 |    |                 |                 |                 |                 |    |                 |                 |                 |                 |    |
|-----------------|-----------------|-----------------|-----------------|----|-----------------|-----------------|-----------------|-----------------|----|-----------------|-----------------|-----------------|-----------------|----|
| R <sub>00</sub> | R <sub>01</sub> | R <sub>02</sub> | R <sub>03</sub> | .. | G <sub>00</sub> | G <sub>01</sub> | G <sub>02</sub> | G <sub>03</sub> | .. | B <sub>00</sub> | B <sub>01</sub> | B <sub>02</sub> | B <sub>03</sub> | .. |
|-----------------|-----------------|-----------------|-----------------|----|-----------------|-----------------|-----------------|-----------------|----|-----------------|-----------------|-----------------|-----------------|----|

#### 2. Interleaved (più comune):

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

|                 |                 |                 |                 |                 |                 |                 |                 |                 |                 |                 |                 |    |    |    |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|----|----|----|
| R <sub>00</sub> | G <sub>00</sub> | B <sub>00</sub> | R <sub>01</sub> | G <sub>01</sub> | B <sub>01</sub> | R <sub>02</sub> | G <sub>02</sub> | B <sub>02</sub> | R <sub>03</sub> | G <sub>03</sub> | B <sub>03</sub> | .. | .. | .. |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|----|----|----|



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

#### • Calcola l'indice di base:

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

#### • Accesso ai canali:

- **R**: baseIndex
- **G**: baseIndex + 1
- **B**: baseIndex + 2

#### • Dimensioni: Larghezza x Altezza

- Ogni elemento della matrice è un singolo valore di intensità [0..255]

### 2.32.2 Parallelismo GPU nella Conversione RGB a Grayscale

#### Perché le GPU sono Ideali per l'Elaborazione delle Immagini

##### • Struttura delle Immagini

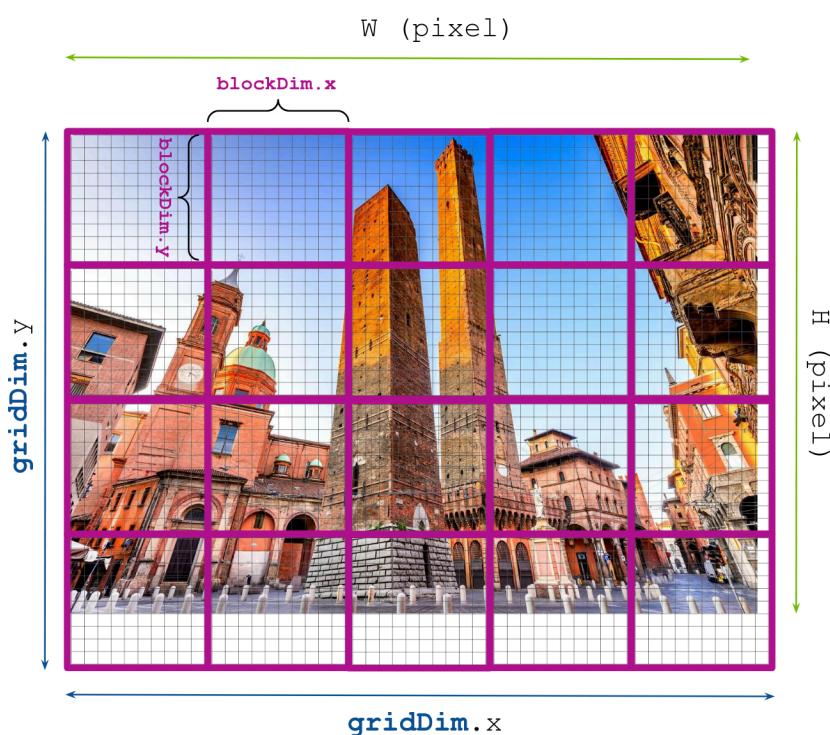
- Le immagini sono composte da molti **pixel indipendenti**.
- Ogni pixel può essere elaborato **separatamente**.
- Operazioni Uniformi**
  - La **stessa operazione** viene spesso applicata a tutti i pixel.
  - Perfecto per il paradigma **SIMD** (Single Instruction, Multiple Data).

### Esempio: Conversione RGB a Grayscale



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

#### 2.32.2.1 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
```

CUDA

### Threads Oltre i Limiti dell'Immagine

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

### 2.32.3 Confronto: Conversione RGB a Grayscale in C vs CUDA C

#### Codice C Standard

```
// Funzione host per la conversione RGB->Gray
void rgbToGrayCPU(unsigned char *rgb, unsigned char *gray, int width, int height) {
 for (int y = 0; y < height; y++) { // Ciclo su tutte le righe dell'immagine
 for (int x = 0; x < width; x++) { // Ciclo su tutti i pixel di una riga
 int rgbOffset = (y * width + x) * 3; // Calcola l'offset per il pixel RGB
 int grayOffset = y * width + x; // Calcola l'offset per il pixel in scala di grigi
 unsigned char r = rgb[rgbOffset]; // Legge il valore rosso
 unsigned char g = rgb[rgbOffset + 1]; // Legge il valore verde
 unsigned char b = rgb[rgbOffset + 2]; // Legge il valore blu
 gray[grayOffset] = (unsigned char)(0.299f * r + 0.587f * g + 0.114f * b); // RGB->Gray
 }
 }
}
```

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
 }
}
```

CUDA

### 2.32.4 Conversione RGB a Grayscale in CUDA

#### Conversione RGB -> Grayscale

CUDA

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

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

 // Carica l'immagine usando "stb_image.h" e "stb_image_write.h"
```

```

int width, height, channels;
unsigned char *rgb = stbi_load(argv[1], &width, &height, &channels, 3);
if (!rgb) {
 printf("Error loading image %s\n", argv[1]);
 return 1;
}
printf("Image loaded: %dx%d with %d channels\n", width, height, channels);

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

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

// Alloca la memoria del device
unsigned char *d_rgb, *d_gray;
CHECK(cudaMalloc((void *)&d_rgb, rgbSize)); // Alloca memoria GPU per l'immagine RGB
CHECK(cudaMalloc((void *)&d_gray, imageSize)); // Alloca memoria GPU per l'output

// Trasferisce i dati dall'host al device
CHECK(cudaMemcpy(d_rgb, rgb, rgbSize, cudaMemcpyHostToDevice));

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

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

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

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

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

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

```

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

**Nota:** vedi documentazione [stb\\_image.h](#) e [stb\\_image\\_write.h](#).

### 2.32.5 Image Flipping con CUDA

- L'image flipping è una tecnica di elaborazione delle immagini che **inverte l'ordine dei pixel lungo un asse** specifico per ciascun canale di colore, creando un **effetto specchio**.

Il flipping può essere:

- **Orizzontale:** Invertendo l'ordine dei pixel da sinistra a destra.
- **Verticale:** Invertendo l'ordine dei pixel dall'alto verso il basso.



#### 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  $(i, width - 1 - j)$ .
- Per un **flip verticale**, la nuova posizione è calcolata come  $(height - 1 - i, j)$ .
- Il thread **copia i valori** dei canali RGB del pixel originale nella nuova posizione calcolata.



#### 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;
```

CUDA

```
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
}
}
```

## 2.32.6 Image Blur con CUDA

## Introduzione all'Image Blurring

L'immagine 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

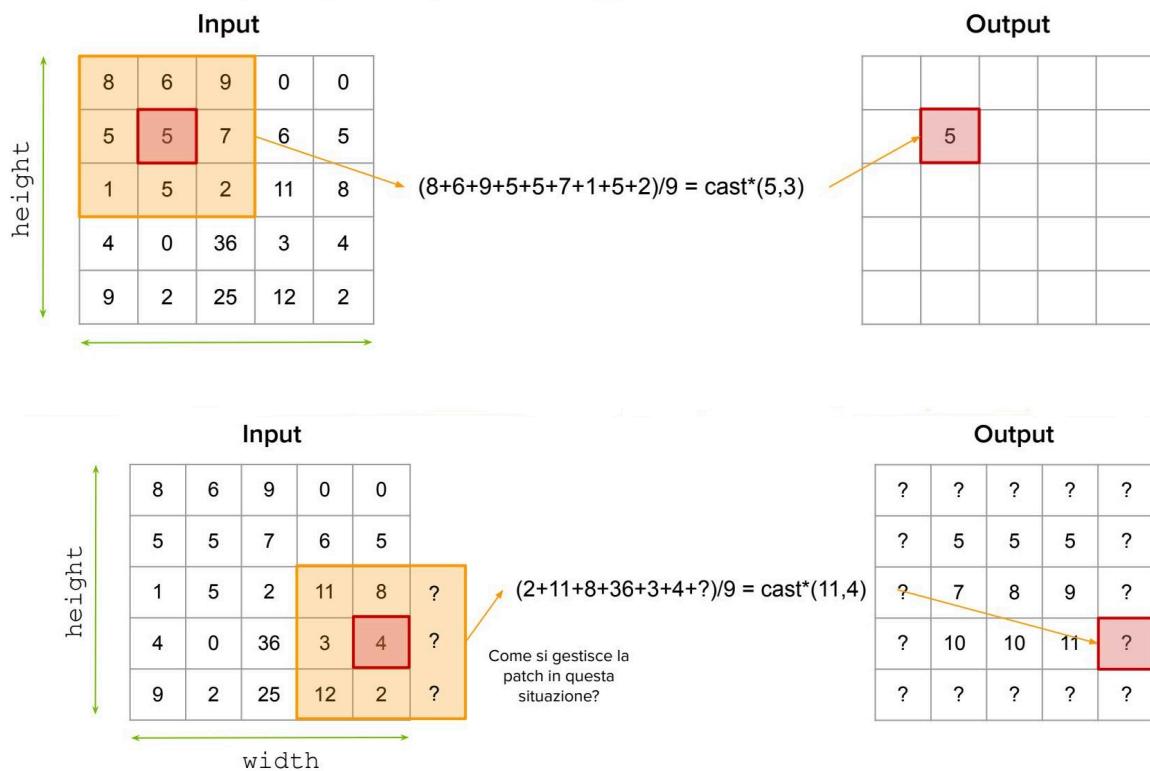


### Blurred Image (window size=25)

## 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 NxN:** 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 3x3:** Include il pixel centrale più gli 8 pixel che lo circondano, formando una matrice



### 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

CUDA

```
#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);
}
}

```

### 2.32.7 Introduzione alla Convoluzione 1D e 2D

#### Che cos'è la Convoluzione?

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

#### Convoluzione 1D

- Applicata a **dati unidimensionali** (segnali audio, serie temporali, sequenze di testo)
- Il filtro è un vettore che **scorre** sul segnale
- L'output ad ogni punto è la **somma dei prodotti elemento per elemento (prodotto scalare)** tra il filtro e la porzione dell'immagine 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. Perché?)
  - Fondamentale nelle **reti neurali convoluzionali (CNN)** per l'elaborazione di immagini

#### Concetti aggiuntivi

- **Padding:** consiste nell'aggiungere un **bordo di valori** (solitamente zeri) attorno all'input prima di applicare la convoluzione. Utile per controllare
- **Stride:** definisce il **passo** con cui il kernel si sposta sull'input durante la convoluzione. Uno stride maggiore comporta sia una riduzione delle dimensioni dell'output che un aumento della velocità di elaborazione

### 2.32.7.1 Esempio di Convoluzione 1D

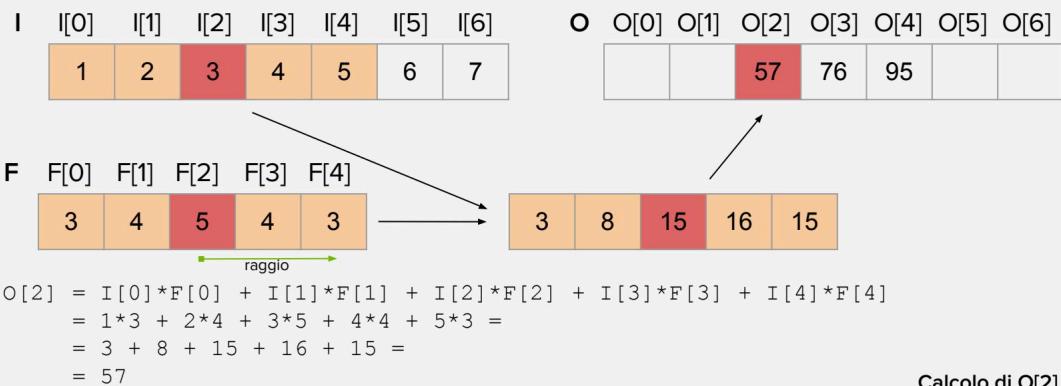
#### Descrizione

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

#### Formalmente

$$O[i] = \sum_{j=-r}^r F[j], I[i+j]$$

$r$ : raggio del filtro 1D



### 2.32.7.2 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)

CUDA

```
__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
}
}

```

### 2.32.7.3 Esempio di Convoluzione 2D

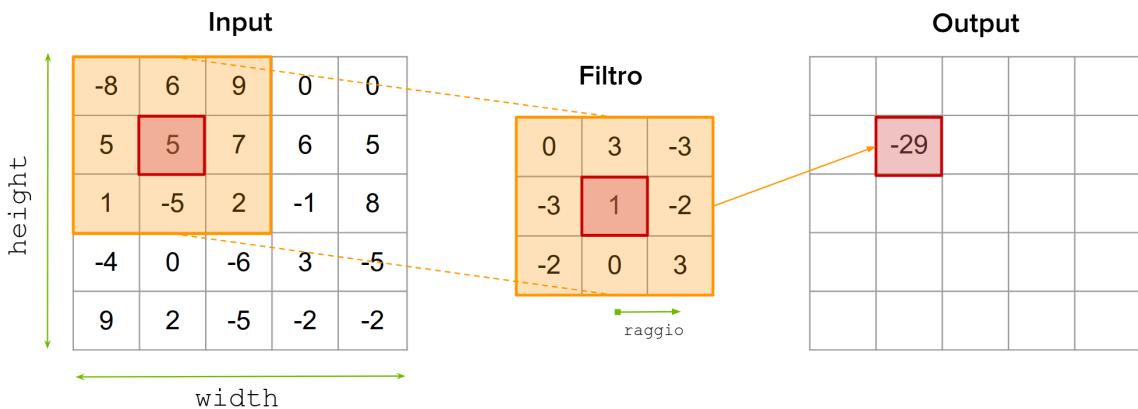
#### Descrizione

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

#### Formalmente

$$O(x, y) = \sum_{m=-r_x}^{r_x} \sum_{n=-r_y}^{r_y} F[m, n] \cdot I[x + m, y + n]$$

$r_x, r_y$ : raggio del filtro 2D nelle due direzioni



CUDA Convoluzione 2D: Soluzione (non ottimale)

CUDA

```

__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;
 }
}

```

```
 output[y * W + x] = result; // Salva il risultato
}
}
```

## 2.33 Riferimenti Bibliografici

### Testi Generali

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

### NVIDIA Docs

- Cuda Programming: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/>
- CUDA C Best Practice Guide: <https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/>
- CUDA University Courses: <https://developer.nvidia.com/educators/existing-courses#2>

# Chapter 3

## Modello di Esecuzione CUDA

### 3.1 Introduzione al Modello di Esecuzione CUDA

#### Modello di Esecuzione CUDA

In generale, un **modello di esecuzione** fornisce una **visione operativa** di come le istruzioni vengono eseguite su una specifica architettura di calcolo (nel nostro caso, le GPU).

#### Caratteristiche Principali

- Fornisce un'**astrazione portabile dell'architettura** (Grid, Block, Thread, Warp, SM).
- Preserva **concetti fondamentali** tra generazioni differenti di GPU.
- Esposizione delle **funzionalità architettoniche** chiave per la programmazione CUDA.
- Descrive come kernel, griglie e blocchi vengono effettivamente **mappati** sull'hardware GPU.
- Basato sul **parallelismo massivo** e sul **modello SIMT** (Single Instruction, Multiple Thread).

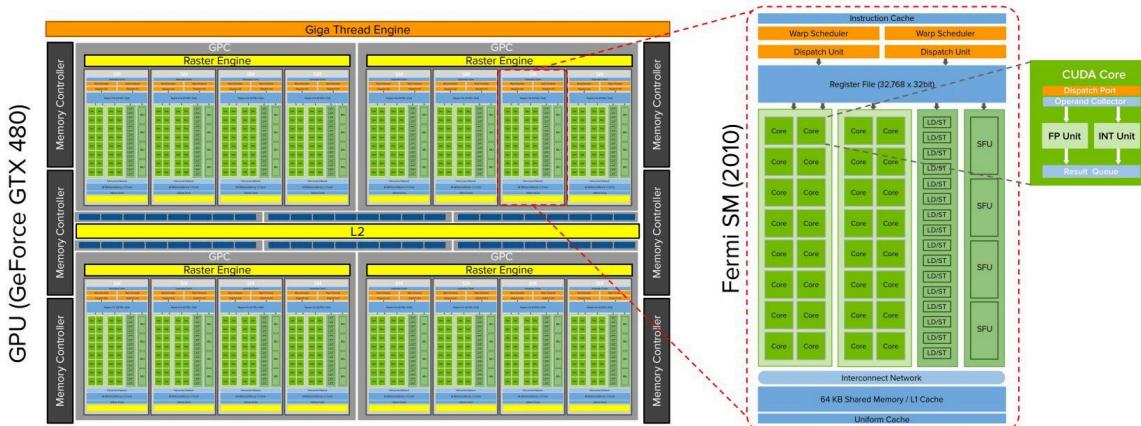
#### Importanza

- Offre una **visione unificata** dell'esecuzione su diverse GPU.
- Fornisce indicazioni utili per l'**ottimizzazione** del codice in termini di:
  - **Throughput** delle istruzioni.
  - **Accessi alla memoria**.
- Facilita la comprensione della **relazione** tra il modello di programmazione e l'esecuzione effettiva.
- Permette di interpretare correttamente i risultati dei profiler CUDA, collegando i fenomeni osservati (latenze, occupancy, conflitti di memoria) alla struttura del modello di esecuzione.

### 3.2 Streaming Multiprocessor (SM)

#### Cosa sono?

- Gli **Streaming Multiprocessors** (SM) sono le unità fondamentali di elaborazione all'interno delle GPU.
- Ogni SM contiene diverse **unità di calcolo, memoria condivisa e altre risorse essenziali** per gestire l'esecuzione concorrente e parallela di migliaia di thread.
- Il parallelismo hardware delle GPU è ottenuto attraverso la **replica** di questo blocco architettonico.



**1. CUDA Cores**

- Unità di elaborazione che eseguono istruzioni aritmetico/logiche.

**2. Shared Memory/L1 Cache**

- Memoria ad alta velocità condivisa tra i thread di un blocco.

**3. Register Files**

- Memoria privata di ogni thread per dati temporanei.

**4. Load/Store Units (LD/ST)**

- Gestiscono il trasferimento dati da/verso la memoria.

**5. Special Function Units (SFU)**

- Accelerano calcoli matematici complessi (funzioni trascendenti).

**6. Warp Scheduler**

- Seleziona thread pronti per l'esecuzione nell'SM.

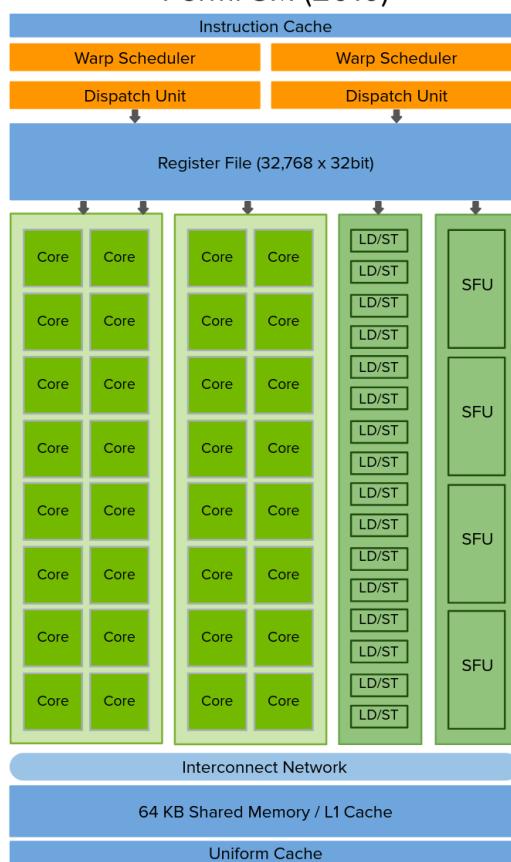
**7. Dispatch Unit**

- Assegna i thread selezionati alle unità di esecuzione.

**8. Instruction Cache**

- Memorizza temporaneamente le istruzioni usate di frequente.

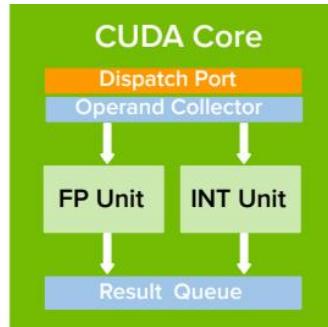
Fermi SM (2010)



### 3.3 CUDA Core - Unità di Elaborazione CUDA

#### Cos'è un CUDA Core?

- Un **CUDA Core** è l'**unità di elaborazione** di base all'interno di un SM di una GPU NVIDIA.
- L'architettura e la funzionalità dei CUDA Core sono evolute nel tempo, passando da unità generiche a unità specializzate.



#### Composizione e Funzionamento (Architettura Fermi e Precedenti)

- Inizialmente, i CUDA Core erano unità di calcolo relativamente semplici, in grado di eseguire sia operazioni intere (INT) che in virgola mobile (FP) in un ciclo di clock (fully pipelined, non simultaneamente).
  - **ALU (Arithmetic Logic Unit):** Ogni CUDA Core contiene un'unità logico-aritmetica che esegue operazioni matematiche di base come addizioni, sottrazioni, moltiplicazioni e operazioni logiche.
  - **FPU (Floating Point Unit):** Include anche una FPU per gestire le operazioni in virgola mobile, supportando principalmente calcoli a precisione singola (FP32).
- I CUDA Core usano **registri condivisi** a livello di Streaming Multiprocessor per memorizzare temporaneamente dati durante l'esecuzione dei thread.

#### Evoluzione dell'Architettura (Kepler e successive)

- Dall'architettura Kepler, NVIDIA ha introdotto la **specializzazione delle unità di calcolo** all'interno di uno SM:

|                                                                                                                                                                                                                                                                                                                                                                                                                           |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>General</b> <ul style="list-style-type: none"> <li>◦ <b>Unità FP64:</b> dedicate alle operazioni in virgola mobile a <i>doppia precisione</i>.</li> <li>◦ <b>Unità FP32:</b> dedicate alle operazioni in virgola mobile a <i>singola precisione</i>.</li> <li>◦ <b>Unità INT:</b> dedicate alle <i>operazioni intere</i>.</li> </ul>                                                                                   |
| <b>AI</b> <ul style="list-style-type: none"> <li>◦ <b>Tensor Core - TC</b> (Architettura Volta e successive): Unità specializzate particolarmente ottimizzate per moltiplicazioni fra matrici in <i>precisione ridotta/mista</i> (FP32, FP16, TF32, INT8, etc.).</li> </ul>                                                                                                                                               |
| <b>Grafica</b> <ul style="list-style-type: none"> <li>◦ <b>Ray Tracing Core - RT</b> (Ampere e successive): Unità dedicate per l'accelerazione del <i>ray tracing</i>.</li> <li>◦ <b>Unità di Texture:</b> ottimizzate per gestire <i>texture</i> e <i>operazioni di filtraggio</i>.</li> <li>◦ <b>Unità di Rasterizzazione:</b> utilizzate per la <i>rasterizzazione</i> delle immagini durante il rendering.</li> </ul> |

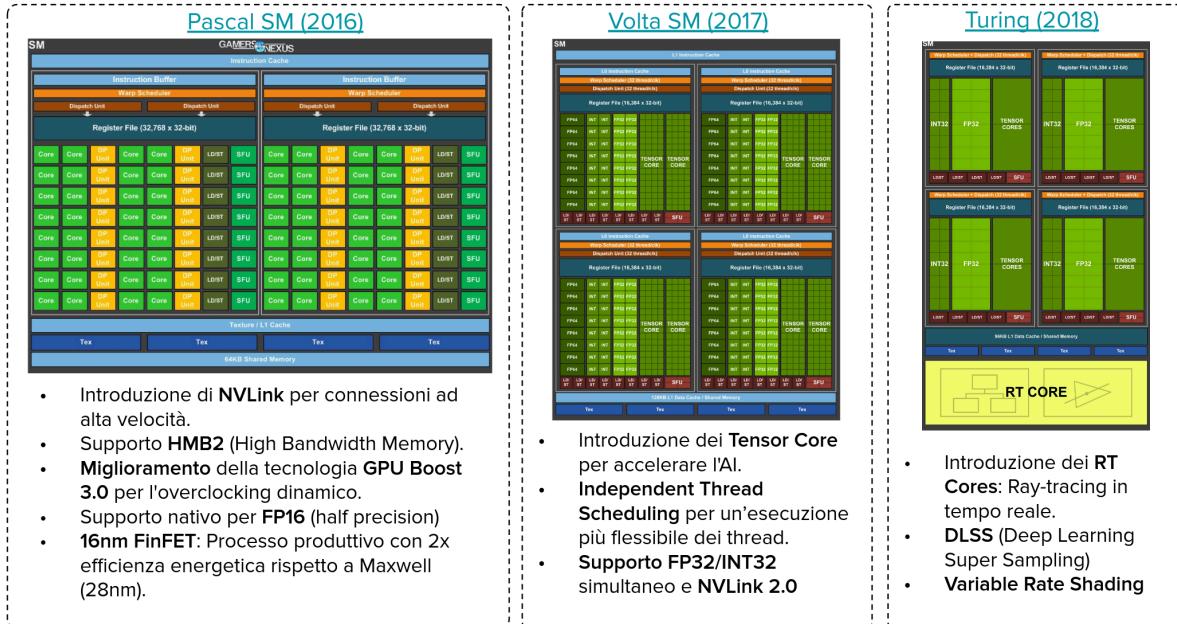
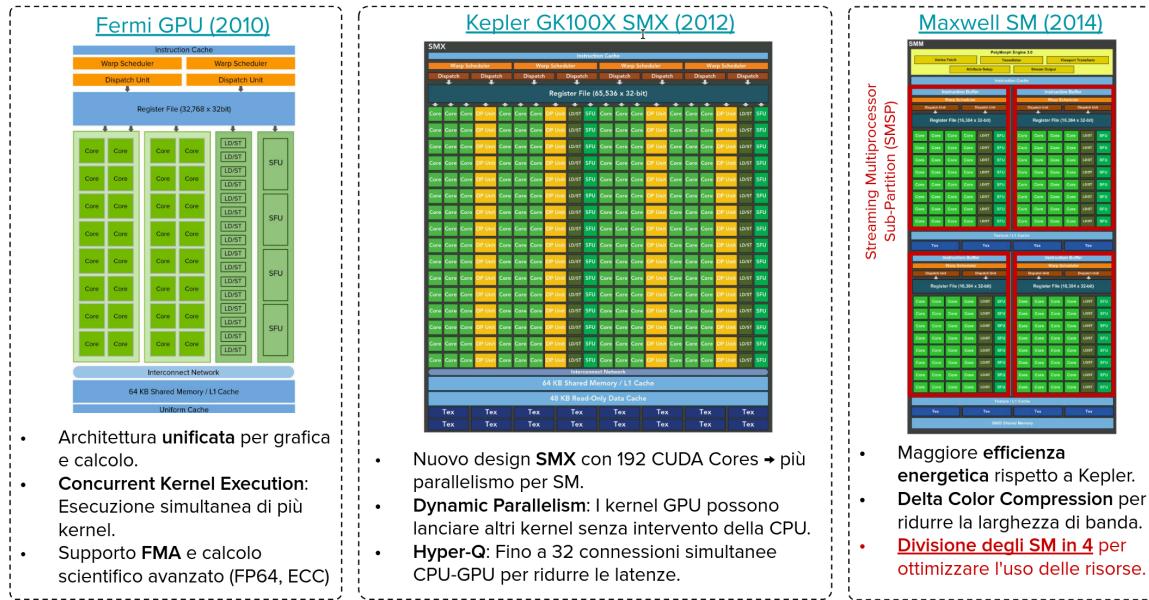
#### Ruolo del Modello CUDA

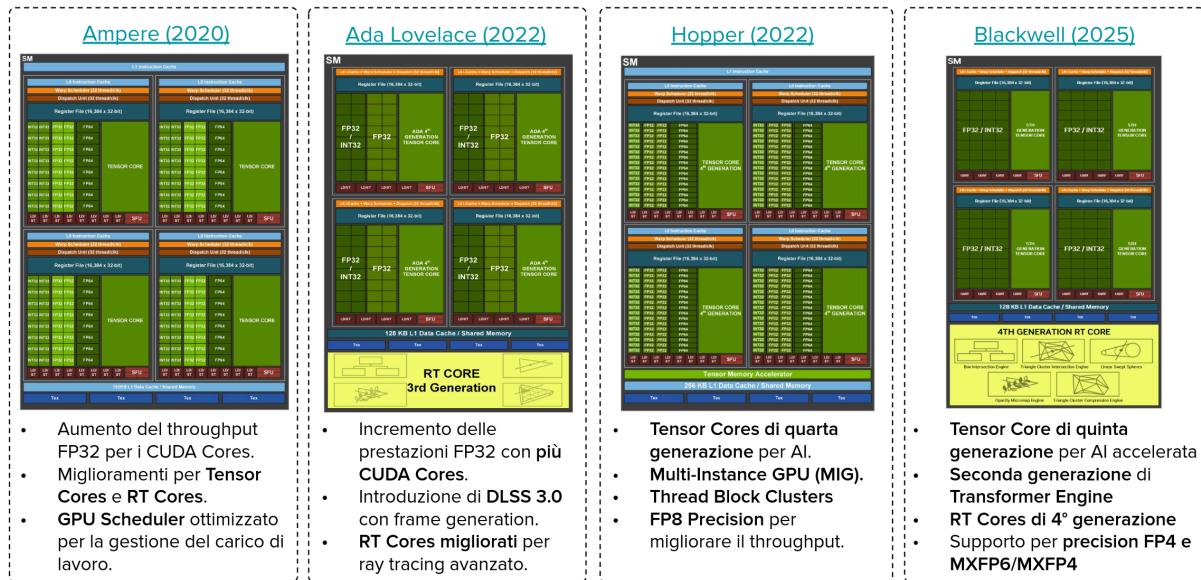
- **Esecuzione Parallela:** Ogni unità di elaborazione esegue un thread in parallelo con altri nel medesimo SM.

#### Differenze rispetto alle CPU

- **Semplicità Architetturale:** Le varie unità di gestione all'interno di un SM sono più semplici rispetto ai core delle CPU, senza unità di controllo complesse, permettendo una maggiore densità di unità specializzate.
- **Specializzazione:** Mentre le CPU sono general purpose, le GPU, attraverso i CUDA Core e le unità specializzate, offrono performance elevate anche per compiti specifici come l'**Intelligenza Artificiale** ed il **rendering grafico**.

### 3.4 Streaming Multiprocessor (SM) - Evoluzione





- Aumento di SM e CUDA Core:** Ogni generazione ha generalmente aumentato il numero di SM e CUDA Core.
- Miglioramento del Parallelismo:** L'aumento delle unità di elaborazione permettono un maggior parallelismo, migliorando le prestazioni complessive della GPU.
- Calcolo CUDA Core Totali:** Totale CUDA Core = (SM per GPU) x (CUDA Core per SM)

| Architettura                        | SM per GPU | CUDA Cores FP32 per SM | Totale CUDA FP32 Cores |
|-------------------------------------|------------|------------------------|------------------------|
| <b>Tesla (GTX 200 series)</b>       | 30         | 8                      | 240                    |
| <b>Fermi (GTX 400/500 series)</b>   | 16         | 32                     | 512                    |
| <b>Kepler (GTX 600/700 series)</b>  | 15         | 192                    | 2880                   |
| <b>Maxwell (GTX 900 series)</b>     | 16         | 128                    | 2048                   |
| <b>Pascal (GTX 10 series)</b>       | 20         | 128                    | 2560                   |
| <b>Volta (Tesla V100)</b>           | 80         | 64                     | 5120                   |
| <b>Turing (RTX 20 series)</b>       | 72         | 64                     | 4608                   |
| <b>Ampere (RTX 30 series)</b>       | 84         | 128                    | 10752                  |
| <b>Ada Lovelace (RTX 40 series)</b> | 128        | 128                    | 16384                  |
| <b>Hopper (GH series)</b>           | 144        | 128                    | 18432                  |
| <b>Blackwell (RTX 50 series)</b>    | 170        | 128                    | 21760                  |

**Nota:** I valori mostrati sono tipici dei modelli di punta. Possono esserci variazioni tra i diversi modelli di una stessa serie.

### 3.5 Tensor Core: Acceleratori per l'Intelligenza Artificiale (Volta+)

#### Cosa sono i Tensor Core?

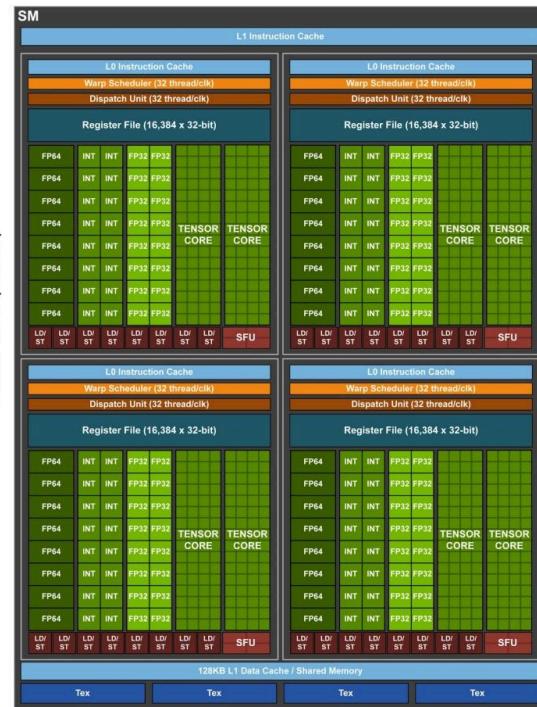
- **Unità di elaborazione specializzata** per operazioni tensoriali (array multidimensionali).
- Progettata per accelerare calcoli di **AI** e **HPC** (Riduzione dei tempi di training e inferenza).
- Presenti in GPU NVIDIA RTX da Volta (2017) in poi.

#### Caratteristiche

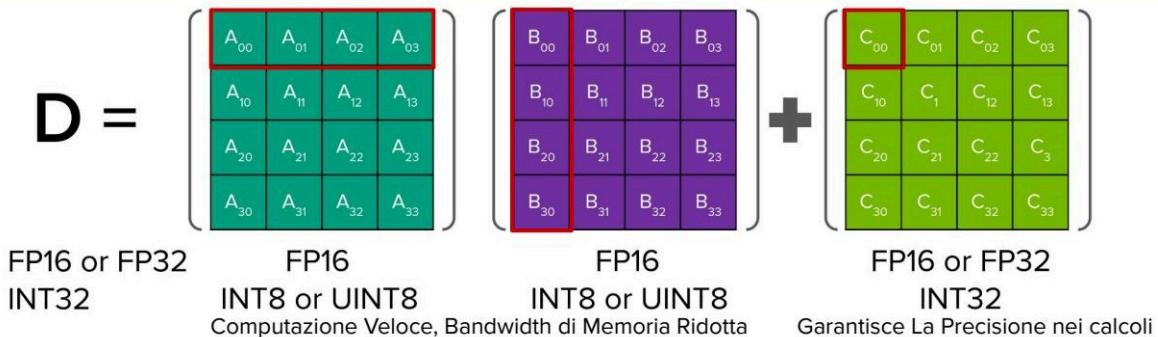
- Esegue operazioni **matrice-matrice** (es. GEMM General Matrix Multiply) in **precisione mista**.
- Supporta formati **FP8, FP16, FP32, FP64, INT8, INT4, BF16** e nuovi formati come **TF32 (TensorFloat-32)**.
- Offrono un significativo **speedup** nel calcolo senza compromettere l'accuratezza.

#### Evoluzione

- **Miglioramenti:** Volta → ... → Hopper → Blackwell
- Integrazione con CUDA, cuDNN, TensorRT



- **Fused Multiply-Add (FMA):** Un'operazione che combina una moltiplicazione e un'addizione di **scalar**i in un unico passo, eseguendo . Un CUDA core esegue 1 FMA per ciclo di clock in FP32.
- **Matrix Multiply-Accumulate (MMA):** Operazione che calcola il prodotto di due **matrici** e somma il risultato a una terza matrice, eseguendo
- Per matrici ( ), ( ) e ( ), l'operazione produce ( ) e richiede operazioni FMA, dove ogni elemento di necessita di moltiplicazioni-addizioni.

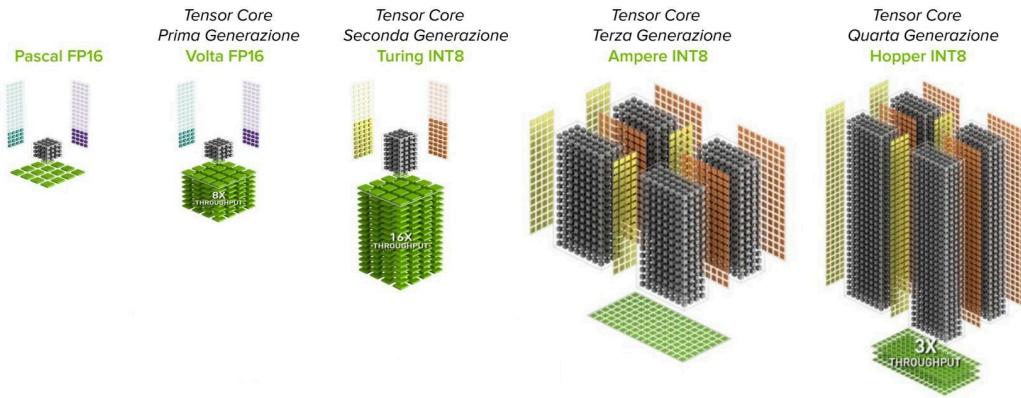


#### Esecuzione Parallela

- Ogni Tensor Core esegue **64 operazioni FMA (4x4x4)** in un singolo ciclo di clock, grazie al parallelismo interno.
- Per operazioni su matrici più grandi, queste vengono **decomposte in sottomatrici 4x4**.
- Più operazioni 4x4 vengono eseguite **in parallelo su diversi Tensor Cores**.

### 3.6 Evoluzione dei NVIDIA Tensor Core

Le generazioni più recenti di GPU hanno ampliato la flessibilità dei Tensor Cores, supportando **dimensioni di matrici più grandi e/o sparse** con un maggiore numero di formati numerici.



### Impatto delle Approssimazioni

- **Accelerazione** significativa dei calcoli
- **Riduzione** del consumo di memoria
- **Perdita di Precisione**: di è dimostrato che ha un impatto minimo sull'accuratezza finale dei modelli di deep learning

## 3.7 Organizzazione e gestione dei thread

### 3.7.1 SM, Thread Blocks e Risorse

#### Parallelismo Hardware

- Più SM per GPU permettono l'**esecuzione simultanea** di migliaia di thread (anche da kernel differenti).

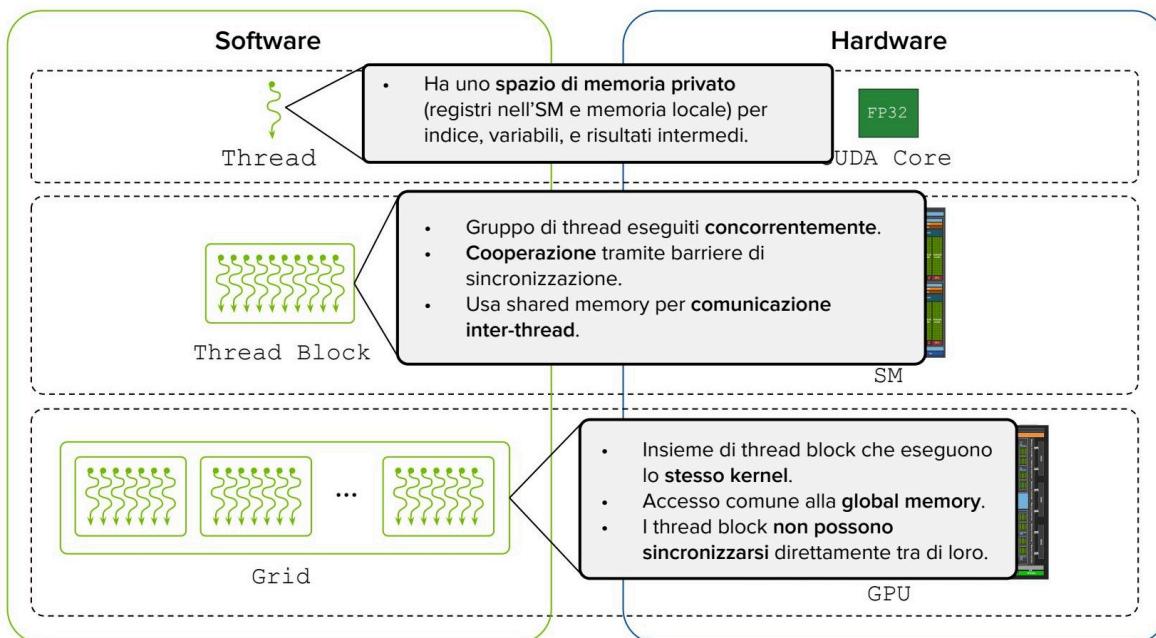
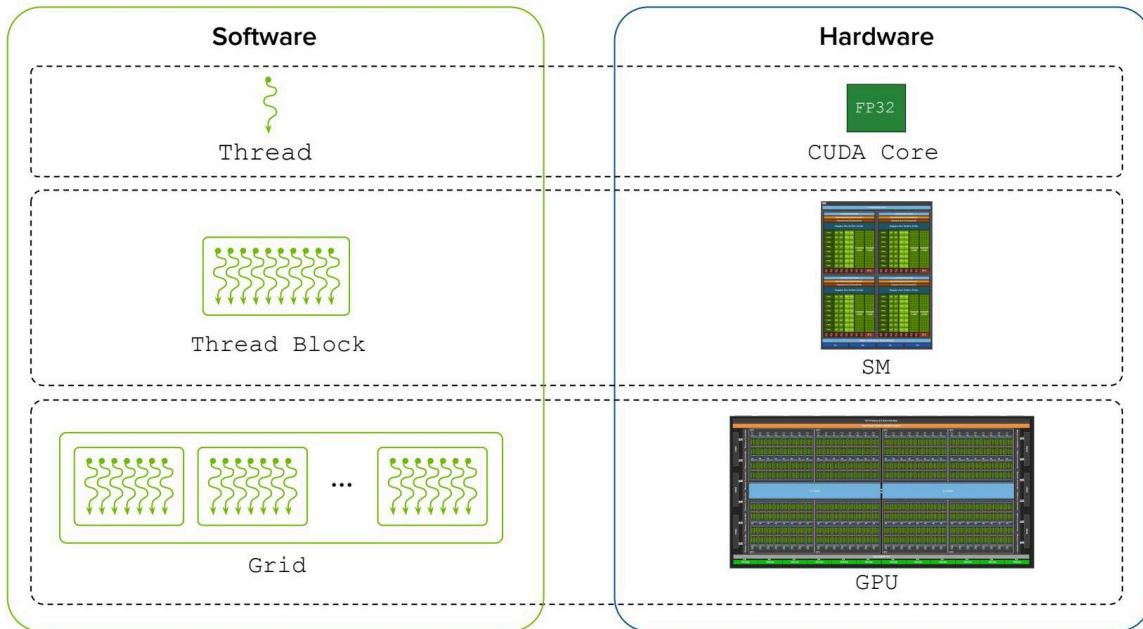
#### Distribuzione dei Thread Blocks

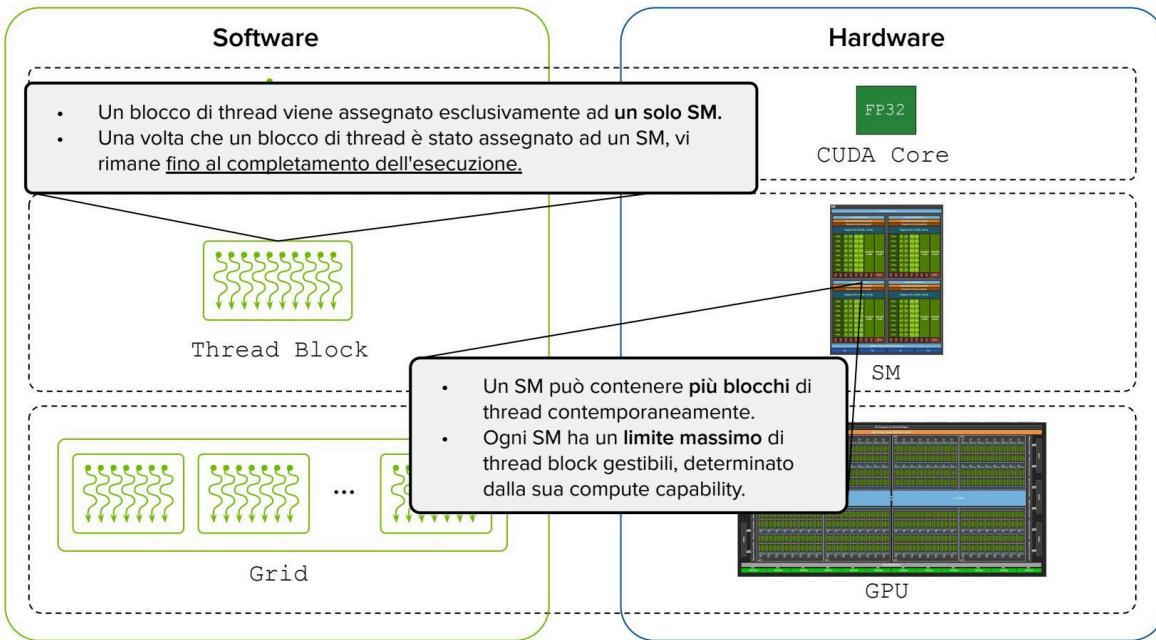
- Quando un kernel viene lanciato, i blocchi di vengono **automaticamente e dinamicamente distribuiti dal GigaThread Engine** (scheduler globale) agli SM disponibili.
- Le variabili di identificazione e dimensione (**gridDim**, **blockIdx**, **blockDim**, e **threadIdx**) sono rese disponibili ad ogni thread e condivise nello stesso SM.
- Una volta assegnato a un SM, un blocco rimane vincolato a quell'SM **per tutta la durata dell'esecuzione**.

#### Gestione delle Risorse e Scheduling

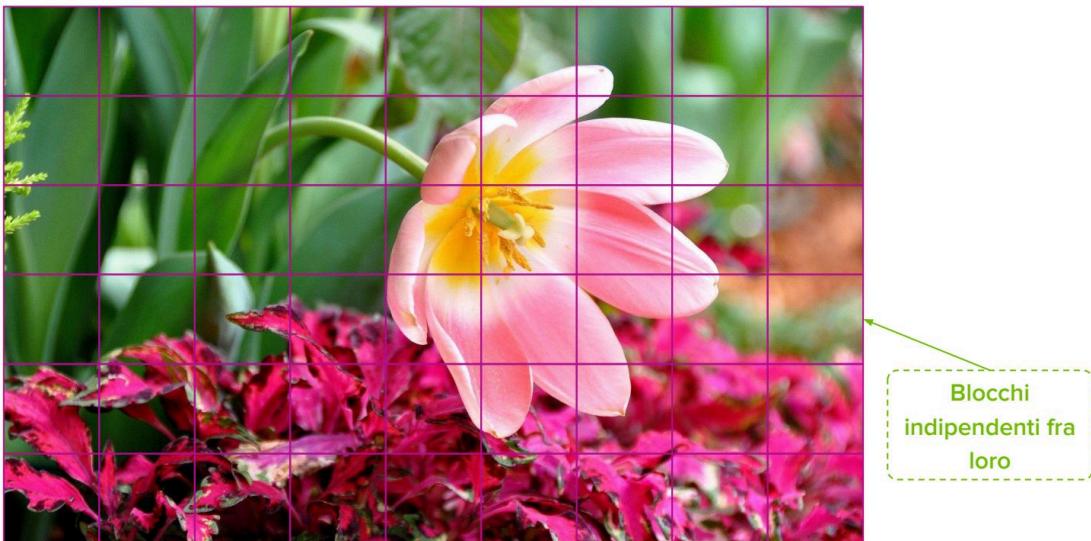
- **Più blocchi di thread** possono essere assegnati **allo stesso SM** contemporaneamente.
- Lo scheduling dei blocchi dipende dalla **disponibilità delle risorse** dell'SM (registri, memoria condivisa) e dai **limiti architetturali** di ciascun SM (max blocks, max threads, etc.).
- Tipicamente, la maggior parte delle grid contiene **molti più blocchi di quanti possano essere eseguiti** in parallelo sugli SM disponibili.
- Il **runtime system** mantiene quindi una coda di blocchi in attesa, assegnandone di nuovi agli SM non appena quelli precedenti terminano l'esecuzione.

### 3.7.2 Corrispondenza tra Vista Logica e Vista Hardware



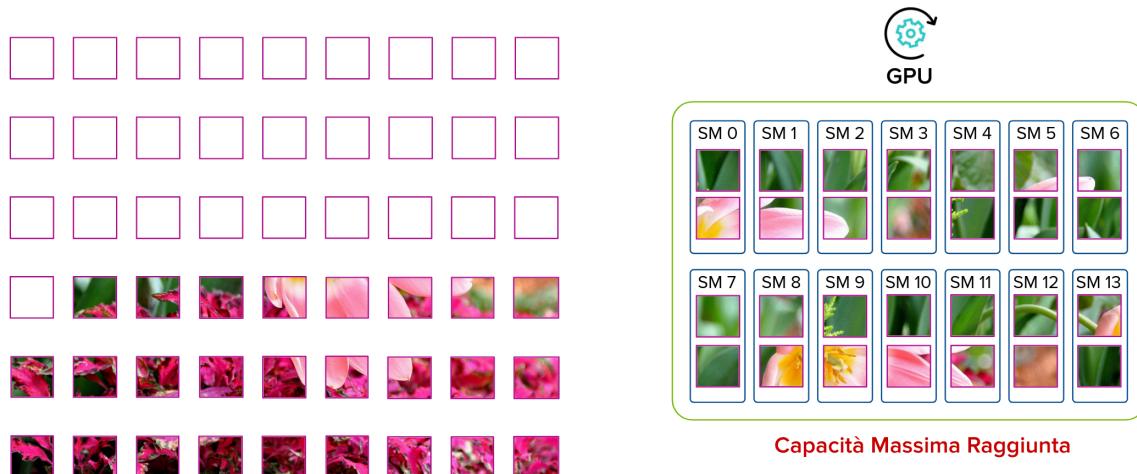


### 3.7.3 Distribuzione dei Blocchi su Streaming Multiprocessors

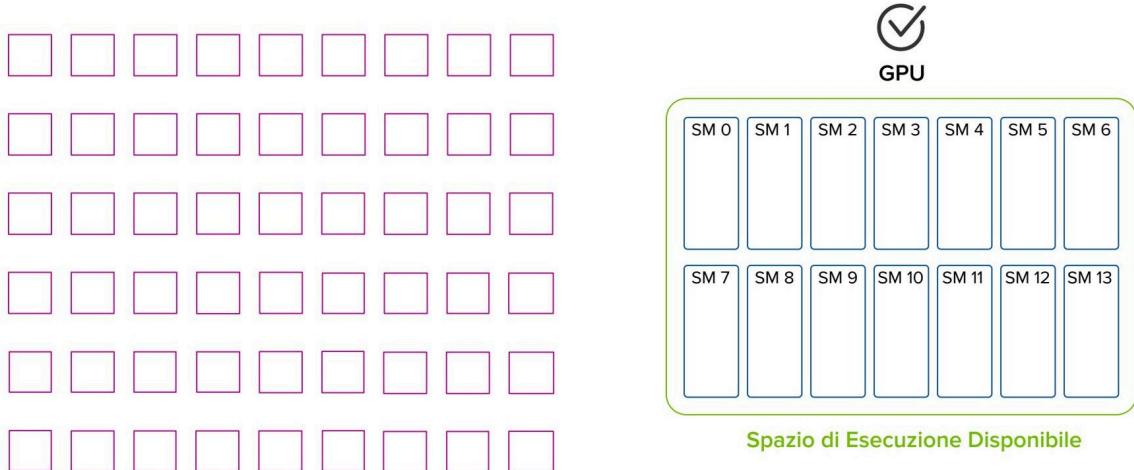


Supponiamo di dover realizzare un algoritmo parallelo che effettui il calcolo parallelo su un'immagine.

- Il Gigathread Engine **smista** i blocchi di thread agli SM in base alle risorse disponibili.
- CUDA non garantisce l'ordine di esecuzione e non è possibile scambiare dati tra i blocchi.
- Ogni blocco viene elaborato in modo **indipendente**.



Quando un blocco completa l'esecuzione e libera le risorse, un nuovo blocco viene schedulato al suo posto nell'SM, e questo processo continua fino a quando tutti i blocchi del grid non sono stati elaborati.



### 3.7.4 Concetto di Wave in CUDA

#### Cosa si intende per Wave?

- Un **"Wave"** rappresenta l'insieme dei blocchi di thread che vengono eseguiti simultaneamente su tutti gli SM della GPU in un dato momento.
- La **Full Wave Capacity**, invece, rappresenta la capacità teorica massima della GPU, ossia il numero totale di blocchi che possono essere residenti simultaneamente su tutti gli SM.

$$\text{Full Wave Capacity} = (\text{Numero di SM}) * (\text{Numero massimo di blocchi attivi per SM})$$

**⚠ Attenzione:** Questo numero massimo di blocchi dipende dall'architettura GPU (Compute Capability) e dalle risorse richieste da ciascun blocco (come registri, memoria condivisa), che influenzano l'occupancy (lo vedremo).

### Full Wave vs Partial Wave

- **Full Wave:** tutti gli SM sono occupati al massimo della loro capacità → utilizzo 100%
- **Partial Wave:** solo parte degli SM è occupata, oppure non tutti al massimo → utilizzo < 100%
- **Esempio:** GPU con 80 SM e fino a 4 blocchi attivi per SM → **Full Wave Capacity** =  $80 \times 4 = 320$  blocchi simultanei.
  - Se il kernel lancia:
    - **320 blocchi** → esegue in **1 full wave** ( $320/320 = 100\%$  utilizzo)
    - **100 blocchi** → esegue in **1 partial wave** ( $100/320 = 31\%$  utilizzo)
    - Cosa succede se il numero di blocchi è **superiore alla capacità massima?** (es. 500 blocchi)

#### 3.7.4.1 Numero di Waves per un Kernel CUDA

##### Calcolo del Numero di Waves

- Quando si lanciano più blocchi di quelli che la GPU può gestire simultaneamente, l'esecuzione avviene in più **ondate successive (waves)**:
- Il numero di blocchi attivi è una proprietà **statica**, determinata dall'architettura e dalle risorse richieste dal kernel.

Numero di waves =  $\lceil (\text{Blocchi totali}) / (\text{Full wave capacity}) \rceil$

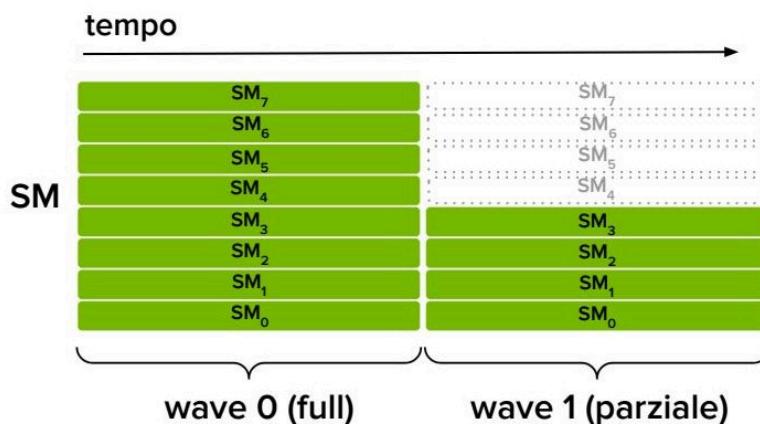
##### Esempio

Consideriamo una GPU con **8 SM** e un kernel con **12 blocchi totali** che consente **1 solo blocco attivo per SM**.

- **Full wave capacity** =  $8 \text{ SM} \times 1 \text{ blocco/SM} = 8 \text{ blocchi}$
- **Numero di waves** =  $\lceil 12 / 8 \rceil = 2 \text{ waves}$

##### Esecuzione:

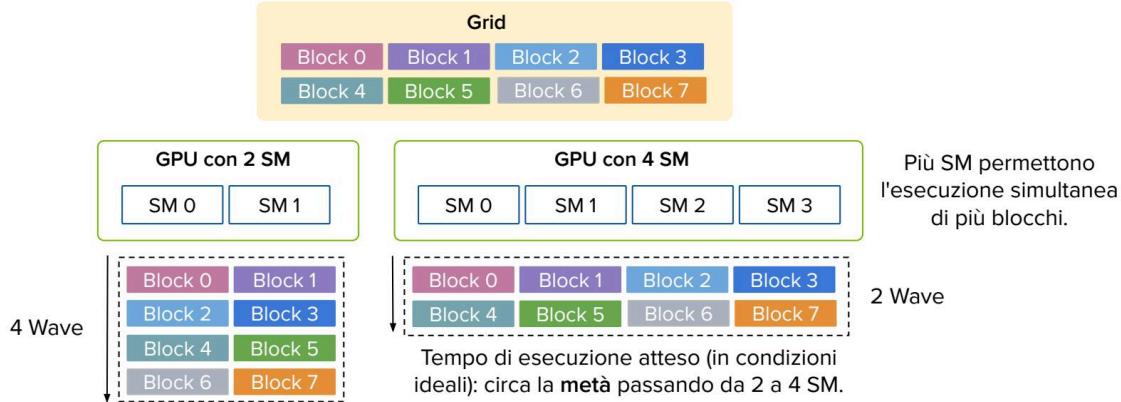
- **Wave 1 (full wave):** 8 blocchi su 8 SM → **utilizzo 100%**
- **Wave 2 (partial wave):** 4 blocchi su 8 SM → **utilizzo 50%**
- **Efficienza media di esecuzione:**  $(8 + 4) / (8 + 8) = 12/16 = 75\%$
- Il secondo wave è un esempio di **tail effect**.



### 3.7.5 Scalabilità in CUDA

#### Cosa si intende?

- Per **scalabilità** in CUDA ci si riferisce alla capacità di un'applicazione di migliorare le prestazioni proporzionalmente all'**aumento delle risorse** hardware disponibili.
- **Più SM disponibili = Più blocchi eseguiti contemporaneamente = Maggiore Parallelismo.**
- **Nessuna modifica al codice** richiesta per sfruttare hardware più potente.



## 3.8 Modello di Esecuzione SIMD e Warp

### 3.8.1 Modello di Esecuzione: SIMD

#### SIMD (Single Instruction, Multiple Data)

- È un **modello di esecuzione parallela** in cui una singola istruzione opera simultaneamente su più elementi di dato, utilizzando **unità vettoriali dedicate** (vector units) presenti nei core della CPU
- Utilizza **registri vettoriali** che possono contenere più elementi (es. 4 float, 8 int16, 16 byte).
- Il programma segue un **flusso di controllo centralizzato** (singolo thread di controllo).
- **Limitazioni:**
  - Larghezza vettoriale **fissa** nell'hardware (es. AVX-512 consente 512 bit), limitando gli elementi per istruzione.
  - Tutti gli elementi vettoriali in un vettore vengono elaborati in **lockstep** (perfettamente sincroni).
  - **La divergenza non è ammessa:** se occorrono percorsi condizionali (**if-else**), si impiegano **maschere vettoriali** che selezionano gli elementi su cui applicare l'operazione.

```
Somma di Due Array (SIMD con Neon intrinsics - ARM)
void array_sum(uint32_t *a, uint32_t *b, uint32_t *c, int n){
 for(int i=0; i<n; i+=4) {
 //calcola c[i], c[i+1], c[i+2], c[i+3]
 uint32x4_t a4 = vld1q_u32(a+i);
 uint32x4_t b4 = vld1q_u32(b+i);
 uint32x4_t c4 = vaddq_u32(a4,b4);
 vst1q_u32(c+i,c4);
 }
}
```

**N.B.** I dati vengono suddivisi in vettori di dimensione fissa e il loop elabora questi vettori utilizzando istruzioni *intrinsics* con nomenclatura specifica dell'architettura.

### 3.8.2 Modello di Esecuzione: SIMT

#### SIMT (Single Instruction, Multiple Thread)

- **Modello ibrido** adottato in CUDA che combina parallelismo multi-thread con esecuzione SIMD-like.
- **Caratteristiche Chiave:**
  - A differenza del SIMD, non ha un controllo centralizzato delle istruzioni.
  - Ogni thread possiede un proprio **Program Counter (PC)**, **registri** e **stato** indipendenti (maggiore flessibilità).
  - **Supporta divergenza** del flusso di controllo (thread possono avere percorsi di esecuzione indipendenti).
  - Hardware gestisce **automaticamente** la divergenza (trasparente al programmatore).
- **Implementazione**
  - In CUDA, i thread sono organizzati in gruppi di 32 chiamati **warps** (unità minima di esecuzione in un SM).
  - I thread in un warp iniziano insieme allo **stesso indirizzo del programma (PC)**, ma possono divergere.
  - Divergenza in un warp causa **esecuzione seriale dei percorsi diversi**, riducendo l'efficienza (da evitare).
  - La divergenza è **gestita automaticamente dall'hardware**, ma con un impatto negativo sulle prestazioni.

#### Somma di Due Array (SIMT)

```
__global__ void array_sum(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]; //Questa riga rappresenta l'essenza del SIMT
}
```

CUDA

#### Perché 32 Thread in un Warp CUDA?

- **Efficienza Hardware:** Massimizza l'utilizzo delle risorse hardware dell'SM
  - Warp troppo piccolo sarebbe inefficiente, mentre uno troppo grande complicherebbe lo scheduling e potrebbe sovraccaricare gli SM / la memoria.
- **Efficienza della Memoria:** Un warp di 32 thread accede a indirizzi di memoria consecutivi, permettendo aggregazioni in poche transazioni e massimizzando l'efficienza delle linee di connessione per evitare accessi parziali.
- **Flessibilità Software:** Offre una granularità gestibile per il controllo della divergenza e per il bilanciamento del carico di lavoro tra thread.
- **Adattabilità:** Questa dimensione si è dimostrata efficace per varie generazioni di GPU NVIDIA, pur rimanendo aperta a future evoluzioni.

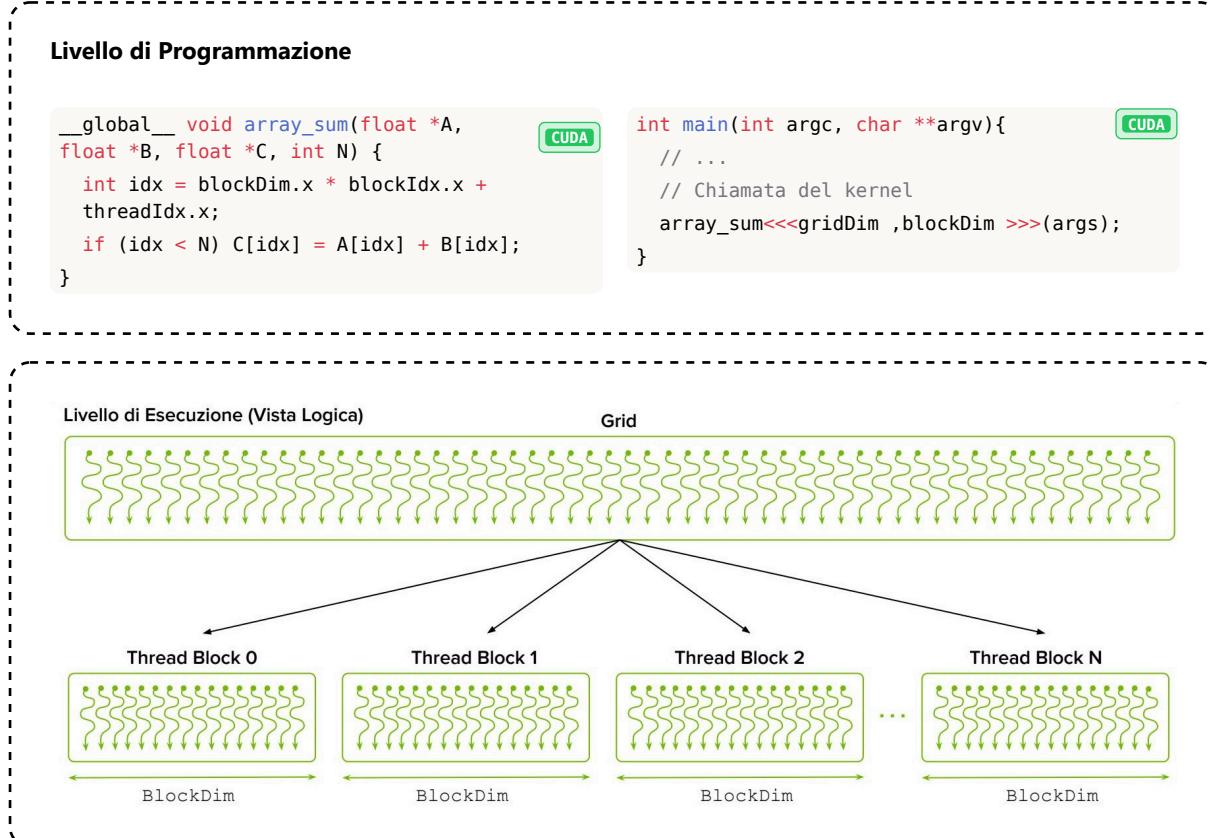
## Modello di Esecuzione: SIMD vs. SIMT

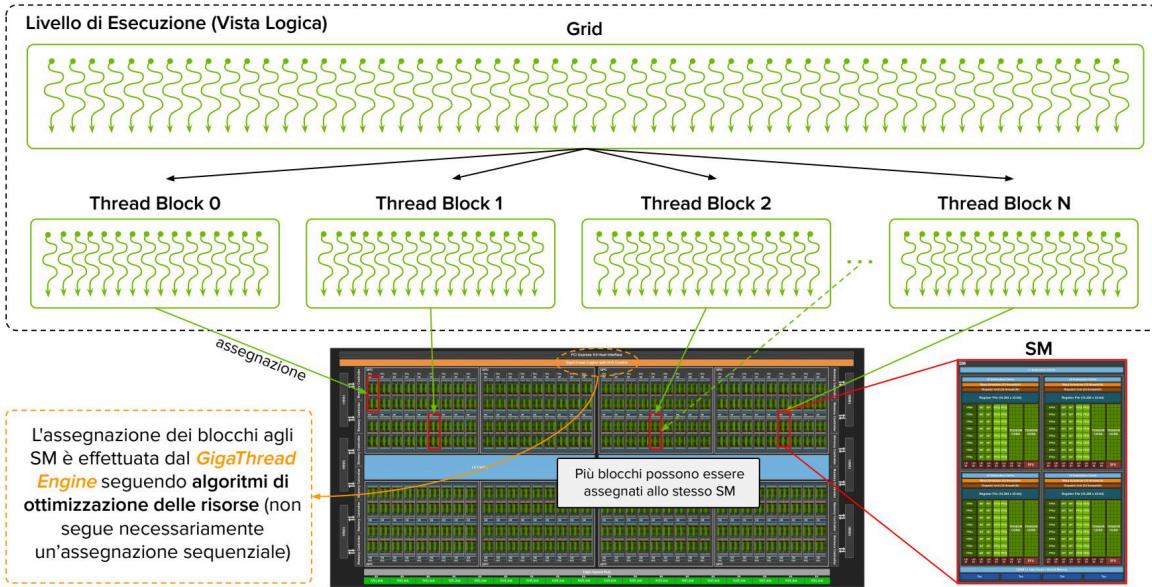
|                            | SIMD                                                              | SIMT                                                                      |
|----------------------------|-------------------------------------------------------------------|---------------------------------------------------------------------------|
| <b>Unità di Esecuzione</b> | Un <b>singolo thread</b> controlla vettori di dimensione fissa    | <b>Molti thread</b> leggeri raggruppati in warp (32 thread)               |
| <b>Registri</b>            | <b>Registri vettoriali</b> condivisi tra le unità di calcolo      | Set completo di <b>registri per thread</b>                                |
| <b>Flessibilità</b>        | <b>Bassa</b> : Stessa operazione per tutti gli elementi vettore   | <b>Alta</b> : Ogni thread può eseguire operazioni e percorsi indipendenti |
| <b>Indipendenza</b>        | <b>Non applicabile</b> , controllo centralizzato                  | Ogni thread mantiene il <b>proprio stato di esecuzione</b> (vedi nota*)   |
| <b>Branching</b>           | Gestito <b>esplicitamente</b> con <b>maschere</b> (no divergenza) | Gestito <b>via hardware</b> con <b>thread masking automatico</b>          |
| <b>Scalabilità</b>         | <b>Limitata</b> dalla larghezza vettoriale                        | <b>Massiva</b> (migliaia/milioni di thread)                               |
| <b>Sincronizzazione</b>    | <b>Intrinseca</b> (lock-step automatico)                          | <b>Esplicita</b> (es., <b>syncthreads</b> )                               |
| <b>Utilizzo Tipico</b>     | <b>Estensioni CPU</b> (SSE, AVX, NEON)                            | <b>GPU Computing</b> (CUDA, OpenCL)                                       |

**Nota:** Con l'architettura Volta, NVIDIA ha introdotto l'Independent Thread Scheduling.

- **Pre-Volta**, un Program Counter (PC) era condiviso per warp;
- **Post-Volta**, ogni thread ha il proprio PC, migliorando la gestione della divergenza e la flessibilità d'esecuzione.

### 3.8.3 Modello di Esecuzione Gerarchico di CUDA





### 3.8.4 Warp: L'Unità Fondamentale di Esecuzione nelle SM

#### Distribuzione dei Thread Block

- Quando si lancia una griglia di thread block, questi vengono **distribuiti** tra i diversi SM disponibili.

**Partizionamento in Warp** I thread di un thread block vengono suddivisi in **warp di 32 thread (con ID consecutivi)**.

#### Esecuzione SIMT

- I thread in un warp eseguono la **stessa istruzione** su **dati diversi**, con possibilità di **divergenza**.

#### Esecuzione Logica vs Fisica

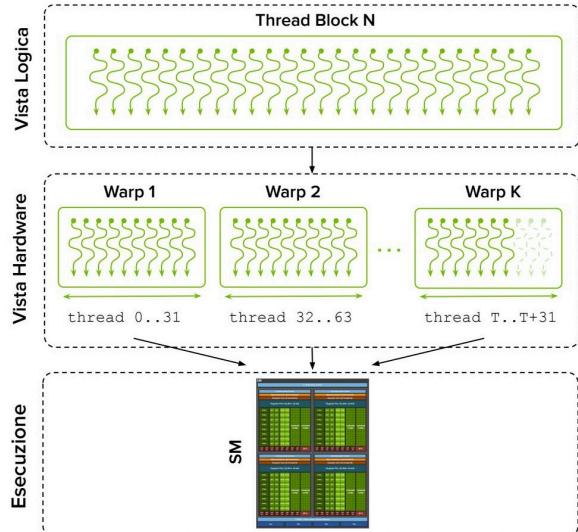
- Thread eseguiti in parallelo **logicamente**, ma non sempre fisicamente.

#### Scheduling Dinamico (Warp Scheduler)

- L'SM gestisce **dinamicamente** l'esecuzione di un numero limitato di warp, switchando efficientemente tra di essi.

#### Sincronizzazione

- Possibile all'interno di un thread block, ma non tra thread block diversi.



### 3.8.4.1 Organizzazione dei Thread e Warp

#### Thread Blocks e Warp

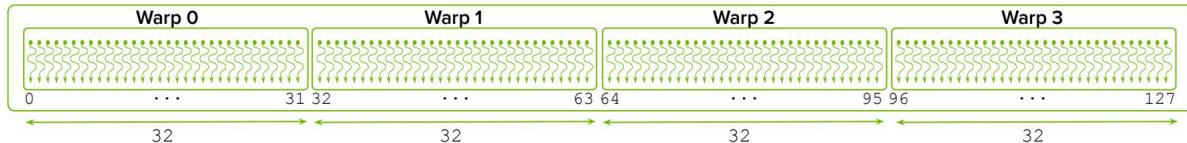
- **Punto di Vista Logico:** Un blocco di thread è una collezione di thread organizzati in un layout 1D, 2D o 3D.
- **Punto di Vista Hardware:** Un blocco di thread è una collezione 1D di warp. I thread in un blocco sono organizzati in un layout 1D e ogni insieme di 32 thread consecutivi (con ID consecutivi) forma un warp.

### Esempio 1D

Un blocco 1D con 128 thread viene suddiviso in 4 warp, ognuno composto da 32 thread (**ID Consecutivi**).

- Warp 0: thread 0, thread 1, thread 2, ... thread 31
- Warp 1: thread 32, thread 33, thread 34, ... thread 63
- Warp 2: thread 64, thread 65, thread 66, ... thread 95
- Warp 3: thread 96, thread 97, thread 98, ... thread 127

### Thread Block N (Caso 1D)

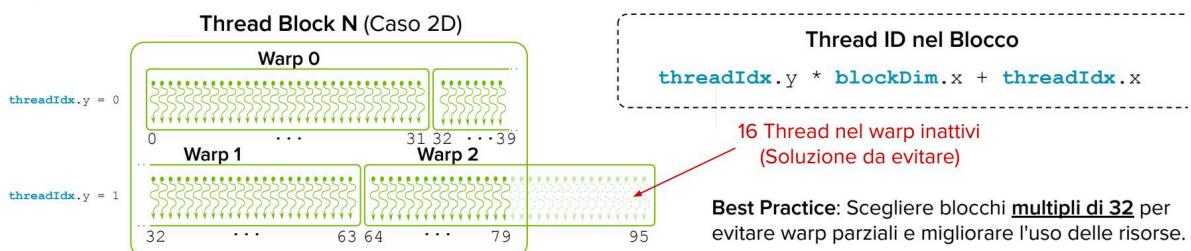


### Mapping Multidimensionale (2D e 3D)

- Il programmatore usa **threadIdx** e **blockDim** per identificare i thread nel layout logico.
- Il runtime CUDA si occupa automaticamente di linearizzare gli indici multidimensionali in ordine row-major, raggruppare i thread in warp, gestire il mapping hardware.
- L'ID di un thread in un blocco multidimensionale viene calcolato usando **threadIdx** e **blockDim**
- Caso 2D:**  $\text{threadIdx.y} * \text{blockDim.x} + \text{threadIdx.x}$
- Caso 3D:**  $\text{threadIdx.z} * \text{blockDim.y} * \text{blockDim.x} + \text{threadIdx.y} * \text{blockDim.x} + \text{threadIdx.x}$
- Calcolo del Numero di Warp:**  $\text{ceil}(\text{ThreadsPerBlock}/\text{warpSize})$
- L'hardware alloca sempre un numero **discreto** di warp.

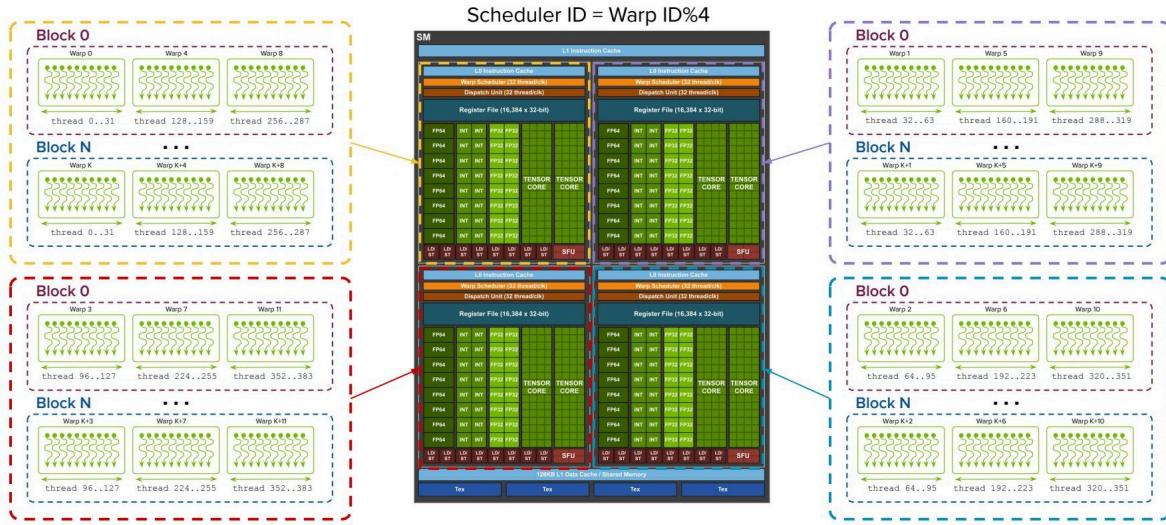
### Mapping Multidimensionale (Caso 2D)

- Esempio 2D: Un thread block 2D con 40 thread in x e 2 in y (80 thread totali) richiederà 3 warp (96 thread hardware). L'ultimo semi-warp (16 thread) sarà inattivo, consumando comunque risorse.



### Warp: L'Unità Fondamentale di Esecuzione nell'SM

- Un warp viene **assegnato** a una sub-partition, solitamente in base al suo ID, dove rimane fino al completamento.
- Una sub-partition gestisce un **“pool” di warp concorrenti** di dimensione fissa (es., Turing 8 warp, Volta 16 warp).



### 3.8.4.2 Compute Capability (CC) - Limiti su Blocchi e Thread

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

| Compute Capability | Architettura | Warp Size | Max Blocchi per SM* | Max Warp per SM* | Max Threads per SM* |
|--------------------|--------------|-----------|---------------------|------------------|---------------------|
| 1.x                | Tesla        | 32        | 8                   | 24/32            | 768/1024            |
| 2.x                | Fermi        | 32        | 8                   | 48               | 1536                |
| 3.x                | Kepler       | 32        | 16                  | 64               | 2048                |
| 5.x                | Maxwell      | 32        | 32                  | 64               | 2048                |
| 6.x                | Pascal       | 32        | 32                  | 64               | 2048                |
| 7.x                | Volta/Turing | 32        | 16/32               | 32/64            | 1024/2048           |
| 8.x                | Ampere/Ada   | 32        | 16/24               | 48/64            | 1536/2048           |
| 9.x                | Hopper       | 32        | 32                  | 64               | 2048                |
| 10.x/12.x          | Blackwell    | 32        | 32                  | 64/48            | 2048/1536           |

[https://en.wikipedia.org/wiki/CUDA#Version\\_features\\_and\\_specifications](https://en.wikipedia.org/wiki/CUDA#Version_features_and_specifications)

### 3.8.4.3 Warp: Contesto di Esecuzione

- Il contesto di **esecuzione locale** di un warp in un SM contiene:

**Volta- : Per warp**  
**Volta+ : Per thread**

- Program Counter (PC):** Indica l'indirizzo della prossima istruzione da eseguire.
- Call Stack:** Struttura dati che memorizza le informazioni sulle chiamate di funzione, inclusi gli indirizzi di ritorno, gli argomenti, array e strutture dati più grandi.

**Registri:** Memoria veloce e privata per ogni thread, utilizzata per memorizzare variabili e dati temporanei.

**Memoria Condivisa:** Memoria veloce e condi-visa tra i thread di un blocco utile per comunicare.

- Thread Mask:** Indica quali thread del warp sono attivi o inattivi durante l'esecuzione di un'istruzione.
- Stato di Esecuzione:** Informazioni sullo stato corrente del warp (es. in esecuzione/in stallo/eleggibile).
- Warp ID:** Identificatore che consente di distinguere i warp e calcolare l'offset nel register file per ogni thread nel warp.
- L'SM mantiene **on-chip** il contesto di ogni warp per tutta la sua durata, quindi il **cambio di contesto è senza costo**.



### 3.8.4.4 Parallelismo a Livello di Warp nell'SM

#### Parallelismo a Livello di Warp nell'SM

##### Codice CUDA

```
__global__ void array_sum(float *A, float *B, float *C, int N) {
 int i = blockIdx.x * blockDim.x + threadIdx.x;
 if (i < N) C[i] = A[i] + B[i];
```

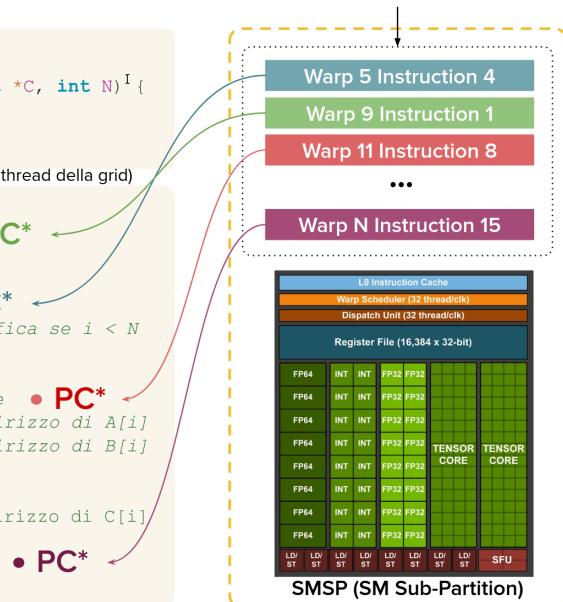
Compilazione (Codice comune a tutti i thread della grid)

##### Codice SASS (Assembly)

```
MOV R1, c[0x0][0x28] // Carica il parametro N • PC*
S2R R6, SR_CTAID.X // Ottiene blockIdx.x
S2R R3, SR_TID.X // Ottiene threadIdx.x
IMAD R6, R6, c[0x0][0x0], R3 // Calcola i • PC*
ISETP.GE.AND P0, PT, R6, c[0x0][0x178], PT // Verifica se i < N
@P0 EXIT // Esce se i >= N
MOV R7, 0x4 // Dimensione di un float (4 byte)
ULDC.64 UR4, c[0x0][0x118] // Carica indirizzo base • PC*
IMAD.WIDE R4, R6, R7, c[0x0][0x168] // Calcola indirizzo di A[i]
IMAD.WIDE R2, R6, R7, c[0x0][0x160] // Calcola indirizzo di B[i]
LDG.E R4, [R4.64] // Carica A[i]
LDG.E R3, [R2.64] // Carica B[i]
IMAD.WIDE R6, R6, R7, c[0x0][0x170] // Calcola indirizzo di C[i]
FADD R9, R4, R3 // Esegue A[i] + B[i]
STG.E [R6.64], R9 // Salva il risultato in C[i]
EXIT // Termina il kernel
```

\*Volta- : Per warp, Volta+ : Per thread

I warp possono appartenere anche a blocchi differenti



### 3.8.5 Classificazione dei Thread Block e Warp

#### Thread Block Attivo (Active Block)

- Un thread block viene considerato **attivo** (o **residente**) quando gli vengono allocate risorse di calcolo di un SM come registri e memoria condivisa (non significa che tutti i suoi warp siano in esecuzione simultaneamente sulle unità).
- I warp contenuti in un thread block attivo sono chiamati **warp attivi**.
- Il numero di blocchi/warp attivi in ciascun istante è **limitato dalle risorse** dell'SM (compute capability).

#### Tipi di Warp Attivi

##### 1. Warp Selezionato (Selected Warp)

- Un warp in esecuzione attiva su un'unità di elaborazione (FP32, INT32, Tensor Core, etc.).

##### 2. Warp in Stallo (Stalled Warp)

- Un warp in attesa di dati o risorse, impossibilitato a proseguire l'esecuzione.
- Cause comuni: latenza di memoria, dipendenze da istruzioni, sincronizzazioni.

##### 3. Warp Eleggibile/Candidato (Eligible Warp)

- Un warp pronto (ma ancora non scelto) per l'esecuzione, con tutte le risorse necessarie disponibili.
- Condizioni per l'eleggibilità:
  - **Disponibilità Risorse**: I thread del warp devono essere allocabili sulle unità di esecuzione disponibili.
  - **Prontezza Dati**: Gli argomenti dell'istruzione corrente devono essere pronti (es. dati dalla memoria).
  - **Nessuna Dipendenza Bloccante**: Risolte tutte le dipendenze con le istruzioni precedenti.

### 3.8.6 Classificazione degli Stati dei Thread

#### Thread all'interno di un Warp

- Un warp contiene sempre 32 thread, ma non tutti potrebbero essere logicamente attivi.
- Lo stato di ogni thread è tracciato attraverso una **thread mask** o **maschera di attività** (un registro hardware a 32 bit).

#### Stati dei Thread



##### 1. Thread Attivo (Active Thread)

- Esegue l'istruzione corrente del warp.
- o Contribuisce attivamente all'esecuzione **SIMT**.

##### 2. Thread Inattivo (Inactive Thread)

- **Divergenza**: Ha seguito un percorso diverso nel warp per istruzioni di controllo flusso, come salti condizionali.
- **Terminazione**: Ha completato la sua esecuzione prima di altri thread nel warp.
- **Padding**: I thread di padding sono utilizzati in situazioni in cui il numero totale di thread nel blocco non è un multiplo di 32, per garantire che il warp sia completamente riempito (puro overhead).

### Gestione degli Stati

- Gli stati sono **gestiti automaticamente dall'hardware** attraverso maschere di esecuzione.
- La transizione tra stati è dinamica durante l'esecuzione, quindi il numero di thread attivi può variare nel tempo.

## 3.8.7 Scheduling dei Warp

### Introduzione al Warp Scheduler

- Un'**unità hardware** presente in più copie all'interno di ogni SM, responsabile della **selezione** e **assegnazione** dei warp alle unità di calcolo CUDA.
- **Obiettivo:** Massimizzare l'utilizzo delle risorse di calcolo dell'SM, selezionando in modo efficiente i warp pronti e minimizzando i tempi di inattività.
- **Latency Hiding:** Contribuiscono a nascondere la latenza eseguendo warp alternativi quando altri sono in stallo, garantendo un utilizzo efficace delle risorse computazionali (prossime slide).

### Funzionamento Generale

- **Processo di Schedulazione:** I warp scheduler all'interno di un SM selezionano i warp eleggibili ad ogni ciclo di clock e li inviano alle dispatch unit, responsabili dell'assegnazione effettiva alle unità di esecuzione.
- **Gestione degli Stalli:** Se un warp è in stallo, il warp scheduler seleziona un altro warp eleggibile per l'esecuzione, garantendo consentendo l'esecuzione continua e l'uso ottimale delle risorse di calcolo.
- **Cambio di Contesto:** Il cambio di contesto tra warp è estremamente rapido (on-chip per tutta la durata del warp) grazie alla partizione delle risorse di calcolo e alla struttura hardware della GPU.

### Limiti Architettonici

- Il numero di warp attivi su un SM è limitato dalle risorse di calcolo. (Esempio: 64 warp concorrenti su un SM Kepler).
- Il numero di warp selezionati ad ogni ciclo è limitato dal numero di scheduler di warp. (Esempio: 4 su un SM Kepler).

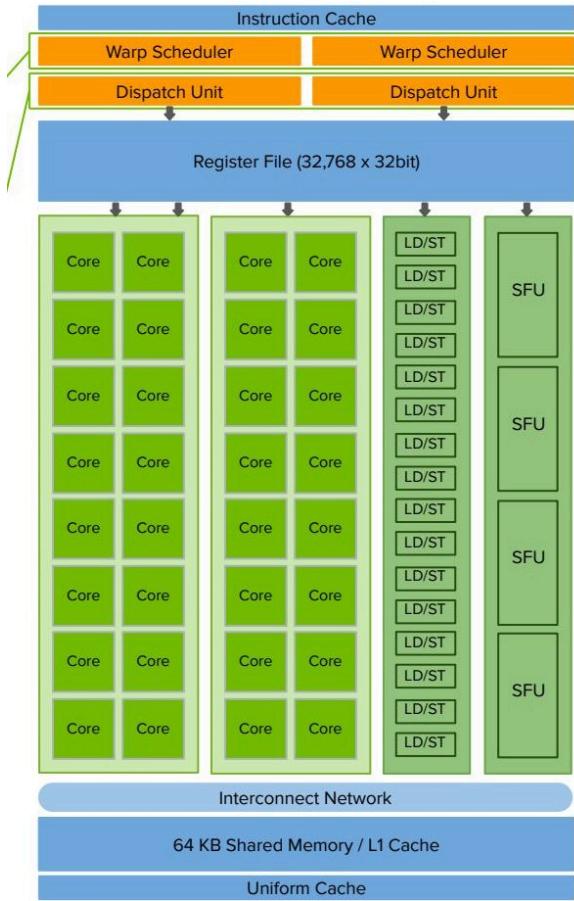
### 3.8.7.1 Warp Scheduler e Dispatch Unit

#### Warp Scheduler

- È il “**cervello strategico**” che decide **quali** warp mandare in esecuzione.
- **Monitora** continuamente lo **stato dei warp** per identificare quelli eleggibili.
- Gestisce la **priorità** e l'**ordine** di esecuzione dei warp, cercando di minimizzare le latenze (*latency hiding*).

#### Dispatch Unit

- È il “**braccio esecutivo**” che si occupa di **come** eseguire i warp selezionati.
- Si occupa di:
  - **Decodificare le istruzioni** del warp.
  - **Distribuire i thread** del warp alle unità di calcolo appropriate (es. FP, INT, Tensor Cores).
  - **Recuperare i dati** dai registri e dalla memoria necessaria per l'esecuzione.
  - **Assegnare fisicamente le risorse** hardware (registri, unità di calcolo) ai thread.



### 3.8.7.2 Scheduling dei Warp: TLP e ILP

#### Thread-Level Parallelism (TLP)

- **Definizione:** Esecuzione simultanea di più warp per sfruttare il parallelismo tra thread.
- **Funzionamento:** Quando un warp è in attesa (ad esempio, per completare un'istruzione), un altro warp viene selezionato ed eseguito, aumentando l'occupazione delle unità di calcolo.

#### Instruction-Level Parallelism (ILP)

- **Definizione:** Esecuzione di istruzioni indipendenti all'interno dello stesso warp.
- **Funzionamento:** Se ci sono più istruzioni pronte da eseguire in un warp, il warp scheduler può emettere queste istruzioni in parallelo alle unità di esecuzione, massimizzando l'utilizzo delle risorse (pipelining).

#### Importanza di TLP e ILP

- **Massimizzazione delle Risorse:** TLP e ILP contribuiscono a mantenere le unità di calcolo attive e occupate, riducendo i tempi morti durante l'esecuzione.
- **Nascondere la Latenza:** TLP e ILP insieme aiutano a nascondere la latenza delle operazioni di memoria e di calcolo, migliorando le prestazioni complessive del sistema (vedi *latency hiding*).

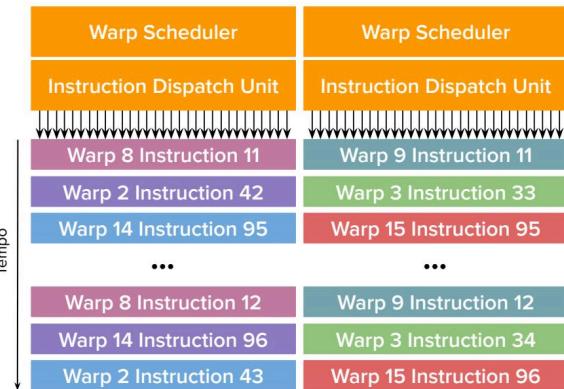
### 3.8.8 Esecuzione Parallelia dei Warp - Esempio con Fermi SM

#### Componenti Chiave per il Parallelismo

- Due Scheduler di Warp:** Selezionano due warp pronti da eseguire dai thread block assegnati all'SM.
- Due Unità di Dispatch delle Istruzioni:** Inviano le istruzioni dei warp selezionati alle unità di esecuzione.

#### Flusso di Esecuzione

- I blocchi vengono assegnati all'SM e **divisi in warp**.
- Due scheduler selezionano warp **pronti** per l'esecuzione.
- Ogni dispatch unit invia un'istruzione per warp a 16 CUDA Core, 16 unità di caricamento/memorizzazione (LD/ST), 4 unità di funzioni speciali (SFU).
- Questo processo **si ripete ciclicamente**, consentendo l'esecuzione parallela di più warp da più blocchi.



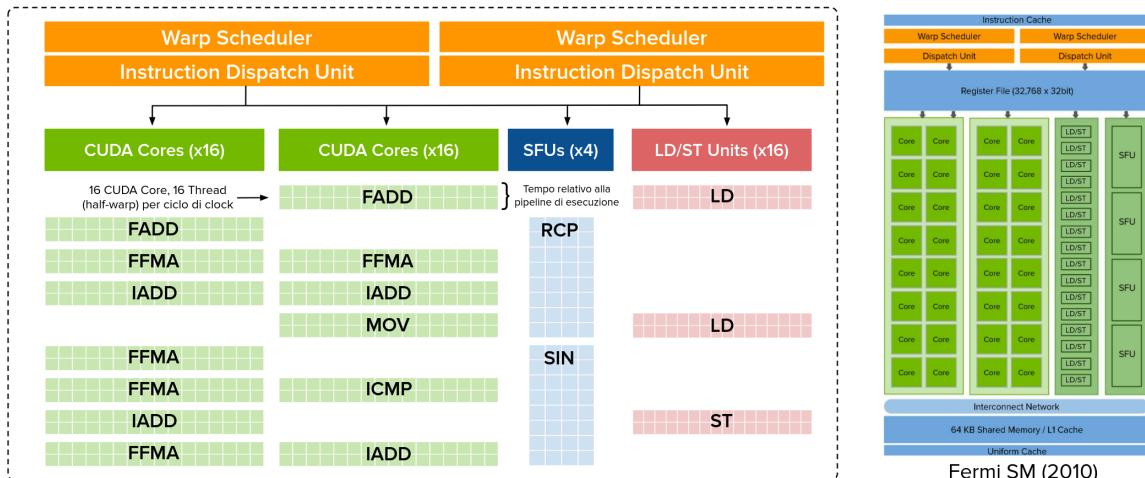
#### Capacità

Fermi (compute capability 2.x) può gestire simultaneamente **48 warp** per SM, per un totale di **1.536 thread residenti** in un singolo SM. Ad ogni ciclo, al più **2 selected warps**.

Poiché le risorse di calcolo sono partizionate tra i warp e mantenute **\*on-chip\*** durante l'intero ciclo di vita del warp, il cambio di contesto tra warp è immediato.

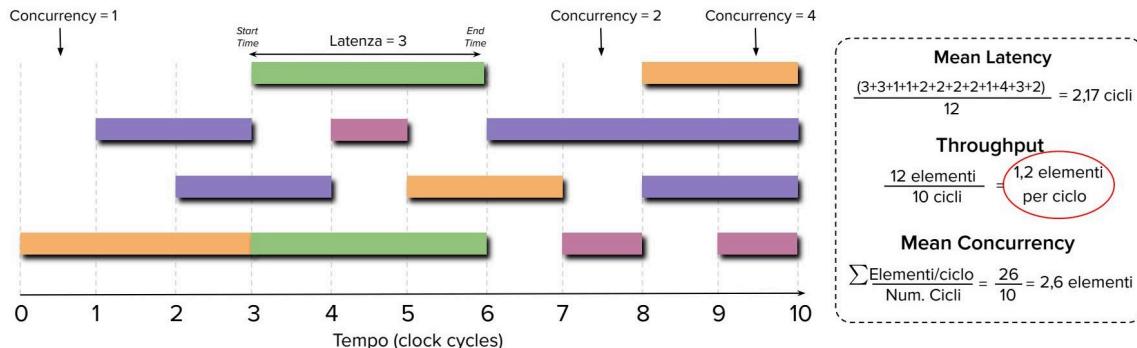
### 3.8.9 Scheduling Dinamico delle Istruzioni - Fermi SM

- Ad ogni ciclo di clock, un warp scheduler **emette un'istruzione** pronta per l'esecuzione.
- L'istruzione può provenire **dallo stesso warp** (ILP, se indipendente, o più spesso da un warp diverso (TLP).
- Se le risorse sono occupate, lo scheduler passa a un altro warp pronto (*latency hiding*).



### 3.8.10 Latency, Throughput e Concurrency

- Mean Latency:** La latenza media è la **media delle latenze** degli elementi individuali. La latenza di un singolo elemento è la differenza tra il suo tempo di inizio e il suo tempo di fine.
- Throughput:** Il throughput rappresenta la velocità di elaborazione. È definito come il **numero di elementi completati entro un dato intervallo di tempo** diviso per la durata dell'intervallo stesso.
- Concurrency:** La concurrency misura **quanti elementi vengono processati contemporaneamente** in un determinato momento. Si può definire sia istantaneamente che come media su un intervallo di tempo.



### 3.8.11 Latency Hiding nelle GPU

#### Cosa è il Latency Hiding?

- Una tecnica che permette di **mascherare i tempi di attesa** dovuti ad operazioni ad alta latenza (come gli accessi alla memoria globale) attraverso l'esecuzione concorrente di più warp all'interno di un SM.
- Si ottiene **intercambiando la computazione tra warp**, per massimizzare l'uso delle unità di calcolo di ogni SM.

#### Funzionamento

- Ogni SM può gestire decine di warp concorrentemente da più blocchi (vedi compute capability della GPU).
- Quando un warp è in stallo (es. accesso memoria), l'SM passa immediatamente all'esecuzione di altri warp pronti.
- I Warp Scheduler dell'SM selezionano costantemente (ad ogni ciclo di clock) i warp pronti all'esecuzione (occorre che abbiano sempre warp eleggibili ad ogni ciclo).

#### Vantaggi del Latency Hiding

- Migliore Utilizzo delle Risorse:** Le unità di elaborazione della GPU sono mantenute costantemente attive.
- Maggiore Throughput:** Completamento di un maggior numero di operazioni nello stesso unità di tempo.
- Minore Latenza Effettiva:** Minimizza l'impatto delle operazioni ad alta latenza.

#### Tipi di Latenza

(variano a seconda dell'architettura e dalla tipologia di operazione)

- Latenza Aritmetica:** Tempo di completamento di operazioni matematiche (bassa, es. 4-20 cicli).
- Latenza di Memoria:** Tempo di accesso ai dati in memoria (alta, es. 400-800 cicli per la memoria globale).

### Meccanismo dei Warp Scheduler

- L'immagine mostra **due warp scheduler** che gestiscono l'esecuzione di diversi warp nel tempo.
- Warp Scheduler 0 e 1 **alternano l'esecuzione di warp diversi** per mantenere le unità di elaborazione occupate.
- Quando un warp è in attesa (es. Warp 0 all'inizio), **altri warp vengono eseguiti** per nascondere la latenza.
- I periodi di inattività (es. 'nessun eligible warp da eseguire') sono **minimizzati**.
- Questo approccio permette di **mascherare i tempi di latenza** e aumentare l'efficienza complessiva.
- Risorse pienamente utilizzate quando ogni scheduler ha un warp eleggibile ad **ogni ciclo di clock**.



### 3.8.12 Legge di Little

#### Cos'è la Legge di Little?

- La Legge di Little (dalla teoria delle code) ci aiuta a calcolare **quanti warp (approssimativamente) devono essere in esecuzione concorrente** per ottimizzare il latency hiding e mantenere le unità di elaborazione della GPU occupate.

$$\text{Warp Richiesti} = \text{Latenza} \times \text{Throughput}$$

- Latenza:** Tempo di completamento di un'istruzione (in cicli di clock).
- Throughput:** Numero di warp (e, quindi, di operazioni) eseguiti per ciclo di clock.
- Warp Richiesti:** Numero di warp attivi necessari per nascondere la latenza.
- Indica che per nascondere la latenza, è necessario avere un **numero sufficiente di warp in esecuzione o pronti per l'esecuzione**, in modo che mentre uno è in attesa, altri possano essere eseguiti.

#### Note

- La **latenza** e il **throughput** possono variare a seconda dell'architettura della GPU e del tipo di istruzioni.
- Questa è una **stima**, il numero effettivo di warp necessari potrebbe essere leggermente diverso.

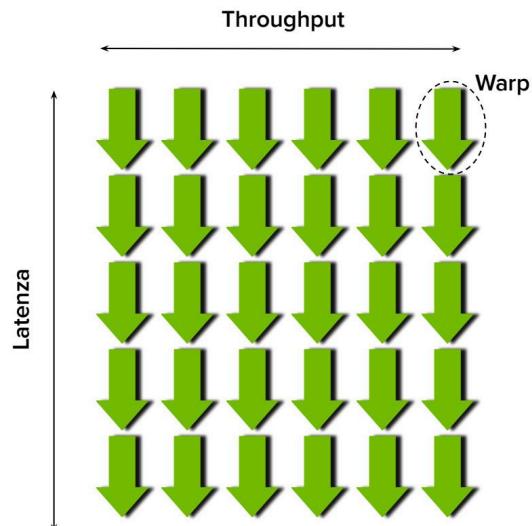
### Esempio della Legge di Little

- Latenza:** 5 Cicli
- Throughput desiderato:** 6 warp/ciclo

Numero di Warp Richiesti =  $5 \times 6 = 30$  warp in flight.

In questo caso, per mantenere un throughput di 6 warp/ciclo con una latenza di 5 cicli, avremmo bisogno di almeno 30 warp in esecuzione o pronti per l'esecuzione.

**Nota:** Un warp (32 thread) che esegue un'istruzione corrisponde a **32 operazioni** (1 operazione per thread)



### 3.8.13 Massimizzare il Parallelismo per Operazioni Aritmetiche

| Architettura | Latenza Istruzione<br>(Cicli) | Throughput<br>(Operazioni/Ciclo) | Parallelismo Necessario<br>(Operazioni) |
|--------------|-------------------------------|----------------------------------|-----------------------------------------|
| Fermi        | 20                            | 32 (1 warp/ciclo)                | 640 (20 warp)                           |
| Kepler       | 20                            | 192 (6 warp/ciclo)               | 3,840 (120 warp)                        |

### Esempio: Operazione Multiply-Add a 32-bit Floating-Point ( $a + b \times c$ )

Limite Warp/SM in Kepler è 64

Consideriamo una GPU con architettura Fermi:

- Throughput:** 32 operazioni/ciclo/SM
  - Un singolo SM può eseguire 32 operazioni di multiply-add a 32-bit floating-point per ciclo di clock.
- Warp Richiesti per SM:**  $640 \div 32$  (operazioni per warp) = 20 warp/SM
  - Per raggiungere il throughput massimo e per mantenere il pieno utilizzo delle risorse computazionali, sono necessari 20 warp attivi contemporaneamente su ogni SM.

Esistono due modi principali per aumentare il parallelismo:

- ILP (Instruction-Level Parallelism):** Aumentare il numero di istruzioni indipendenti all'interno di un singolo thread.
- TLP (Thread-Level Parallelism):** Aumentare il numero di thread (e quindi di warp) che possono essere eseguiti contemporaneamente.

### 3.8.14 Massimizzare il Parallelismo per Operazioni di Memoria

| Architettura | Latenza<br>(Cicli) | Bandwidth<br>(GB/s) | Bandwidth<br>(B/ciclo) | Parallelismo<br>(KB) |
|--------------|--------------------|---------------------|------------------------|----------------------|
| Fermi        | 800                | 144                 | 92                     | 74                   |
| Kepler       | 800                | 250                 | 96                     | 77                   |

### Esempio: Operazione di Memoria

Consideriamo sempre una GPU con **architettura Fermi**:

- **Calcolo del Bandwidth in Bytes/Ciclo:**  $144 \text{ GB/s} \div 1.566 \text{ GHz} \approx 92 \text{ Bytes/Ciclo}$  (Frequenza di memoria Fermi -Tesla C2070 = 1.566 GHz)
- **Calcolo del Parallelismo Richiesto:**
  - **Parallelismo** = **Bandwidth** (B/ciclo) × **Latenza Memoria** (cicli)
  - Fermi:  $92 \text{ B/ciclo} \times 800 \text{ cicli} \approx 74 \text{ KB}$  di I/O in-flight per saturare il bus di memoria.
- Memory Bandwidth è relativo all'intero device
- **Interpretazione:**
  - 74 KB di operazioni di memoria necessarie per nascondere la latenza (per l'intero device, non per SM).
    - Memory Bandwidth è relativo all'intero device

### Recuperare la Memory Frequency di una GPU NVIDIA (da terminale)

```
$ nvidia-smi -a -q -d CLOCK | grep -A 3 "Max Clocks" | grep "Memory"
```

 Shell

### Esempio: Operazione di Memoria

- Il legame tra questi valori e il numero di warp/thread **varia a seconda della specifica applicazione**.
- **Conversione in Thread/Warp** (Supponendo 4 bytes ad esempio, FP32 per thread):
  - $74 \text{ KB} \div 4 \text{ bytes/thread} \approx 18,500 \text{ thread}$
  - $18,500 \text{ thread} \div 32 \text{ thread/warp} \approx 579 \text{ warp}$
  - Per 16 SM:  $579 \text{ warp} \div 16 \text{ SM} = 36 \text{ warp per SM}$
- Ovviamente, se ogni thread eseguisse più di un caricamento indipendente da 4 byte o un tipo di dato più grande (es. FP64), sarebbero necessari meno thread per mascherare la latenza di memoria.

Esistono due modi principali per aumentare il parallelismo di memoria:

- **Maggiore Granularità:** Spostare più dati per thread (ad esempio, caricare più float per thread).
- **Più Thread Attivi:** Aumentare il numero di thread concorrenti per aumentare il numero di warp attivi.

### 3.8.15 Warp Divergence

#### Cosa è la Warp Divergence?

- In un warp, idealmente tutti i thread eseguono la **stessa istruzione contemporaneamente** per massimizzare il parallelismo SIMT (condividono un unico **Program Counter** [Architetture Pre-Volta]).
- Tuttavia, se un'**istruzione condizionale** (come un if-else o switch) porta thread diversi a percorrere **rami diversi** del codice, si verifica la **Warp Divergence**.
- In questo caso, il warp esegue **serialmente ogni ramo**, utilizzando una **maschera di attività** (calcolata automaticamente in hardware) per abilitare/disabilitare i thread.
- La divergenza termina quando i thread **riconvergono** alla fine del costrutto condizionale.
- La Warp Divergence **può significativamente degradare le prestazioni** perché i thread non vengono eseguiti in parallelo durante la divergenza (le risorse non vengono pienamente utilizzate).
- Notare che il fenomeno della divergenza occorre **solo all'interno di un warp**.

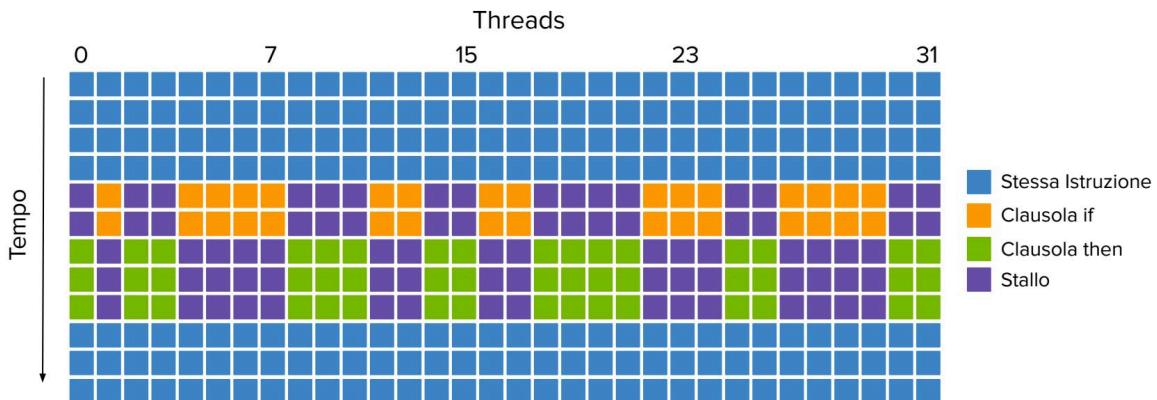
### Esempio

```
if (threadIdx.x % 2 == 0) {
 // Istruzioni per thread con indice pari
} else {
 // Istruzioni per thread con indice dispari
}
```

### 3.8.16 CPU vs GPU: Gestione del Branching e della Warp Divergence

|                                  | CPU                                                                                  | GPU                                                                                                                          |
|----------------------------------|--------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------|
| <b>Esecuzione</b>                | Singoli thread o piccoli gruppi, <b>in-dipendenti</b> tra loro.                      | Warp che eseguono le <b>stesse istruzioni</b> concorrentemente.                                                              |
| <b>Branch Prediction</b>         | <b>Hardware dedicato</b> , con algoritmi di predizione complessi.                    | <b>Non supportata</b>                                                                                                        |
| <b>Esecuzione Speculativa</b>    | Esegue istruzioni <b>in anticipo</b> basandosi sulla <b>branch prediction</b> .      | <b>Non supportata</b>                                                                                                        |
| <b>Impatto della Divergenza</b>  | Mitigato dalla <b>branch prediction</b> e dall' <b>esecuzione speculativa</b> .      | Causa la <b>warp divergence</b> , riducendo il parallelismo e le prestazioni.                                                |
| <b>Gestione della Divergenza</b> | <b>Predizione</b> del ramo più probabile e <b>esecuzione speculativa</b> del codice. | <b>Esecuzione seriale</b> dei rami divergenti nel warp, <b>disabilitando</b> i thread inattivi.                              |
| <b>Ottimizzazioni</b>            | Meno critiche, gestite in parte <b>dall'hardware</b> .                               | <b>Branch predication</b> (non lo vedremo) e <b>riorganizzazione del codice</b> essenziali.                                  |
| <b>Considerazioni</b>            | Il costo della predizione errata è <b>relativamente basso</b> .                      | Il costo della warp divergence è <b>elevato</b> a causa della perdita di parallelismo e dell'overhead di esecuzione seriale. |

### 3.8.16.1 Warp Divergence: Analisi del Flusso di Esecuzione



### Flusso

- All'inizio, tutti i thread eseguono lo stesso codice (**blocchi blu**).
- Quando si incontra un'**istruzione condizionale** (**blocchi arancioni**), il warp si **divide**.
- Alcuni thread eseguono la clausola “**then**” (**blocchi verdi**), mentre altri sono in **stallo** (**blocchi viola**).
- Nei momenti di divergenza, l'efficienza può scendere al 50% (in questo caso, 16 thread attivi su 32).

### 3.8.16.2 Serializzazione nella Warp Divergence

#### Divergenza

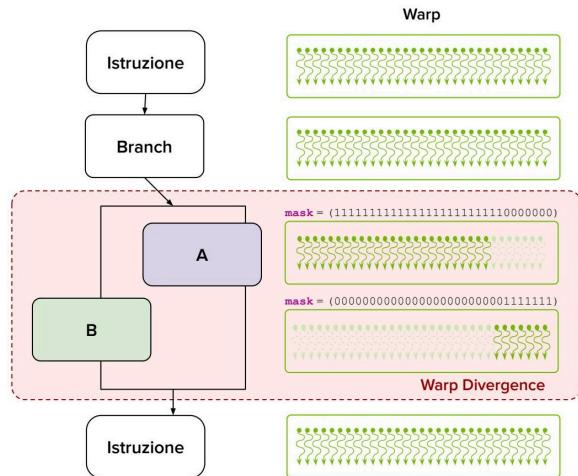
Quando i thread di un warp seguono percorsi diversi a causa di istruzioni condizionali (es. **if**), il warp esegue ogni ramo in serie, **disabilitando i thread inattivi**.

#### Località

- La divergenza si verifica solo all'interno di un **singolo warp**.
- Warp diversi operano **indipendentemente**.
- I passi condizionali in **differenti warp** non causano divergenza.

#### Impatto

- La divergenza può ridurre il parallelismo **fino a 32 volte**.



#### Caso Peggiorre

CUDA

```
__global__ void WorstDivergence(int* x) {
 int i = threadIdx.x + blockDim.x * blockIdx.x;
 switch (i % 32) {
 case 0 :
 x[i] = a(x[i]);
 break;
 case 1 :
 x[i] = b(x[i]);
 break;
 .
 .
 case 31:
 x[i] = v(x[i]);
 break;
 }
}
```

Le prestazioni diminuiscono con l'aumento della divergenza nei warp.

### 3.8.16.3 Confronto delle Condizioni di Branch

**Kernel 1** CUDA

```
__global__ void mathKernel1(float *sum) {
 int tid = blockIdx.x * blockDim.x +
 threadIdx.x;
 float a = 0.0f, b = 0.0f;
 if (tid % 2 == 0) a = 100.0f;
 else b = 200.0f;
 *sum += a + b;}// Race condition
(risolveremo con atomicAdd dopo)
```

**Kernel 2** CUDA

```
__global__ void mathKernel2(float *sum) {
 int tid = blockIdx.x * blockDim.x +
 threadIdx.x;
 float a = 0.0f, b = 0.0f;
 if ((tid / warpSize) % 2 == 0) a = 100.0f;
 else b = 200.0f;
 *sum += a + b;}// Race condition
(risolveremo atomicAdd dopo)
```

#### Funzionamento

Valuta la parità dell'**ID** di ogni singolo thread.

#### Effetto sui thread

- **Thread pari** (ID 0, 2, 4, ...): eseguono il ramo **if**.
- **Thread dispari** (ID 1, 3,...): eseguono il ramo **else**.

#### Impatto sul warp

In ogni warp (32 thread), 16 thread eseguono **if** e 16 eseguono **else**.

#### Risultato

**Warp divergence**, con esecuzione serializzata dei due percorsi all'interno del warp.

#### Funzionamento

- **tid / warpSize**: Identifica l'**ID del warp** a cui appartiene il thread.
- (...) % 2: Determina la parità del numero del warp.

#### Effetto sui warp

- **Warp pari**: tutti i 32 thread eseguono il ramo **if**.
- **Warp dispari**: tutti i 32 thread eseguono il ramo **else**.

#### Impatto sul warp

Tutti i thread in un warp eseguono lo **stesso percorso**.

#### Risultato

**Eliminazione del warp divergence**, con esecuzione parallela all'interno di ogni warp (nessun overhead).

#### Branch Efficiency (calcolata in Nsight Compute)

La **Branch Efficiency** misura la percentuale di branch non divergenti rispetto al totale dei branch eseguiti da un warp.

$$\text{Branch Efficiency} = 100 \times \left( \frac{\# \text{ Branches} - \# \text{ DivergentBranches}}{\# \text{ Branches}} \right)$$

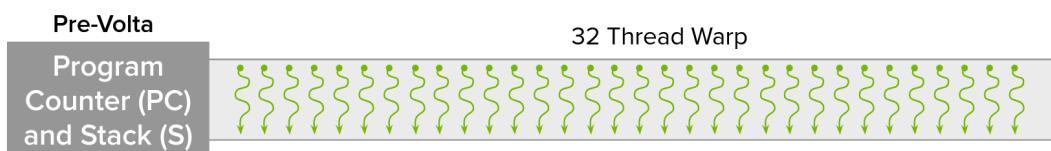
- Un **valore elevato** indica che la maggior parte dei branch eseguiti dal warp non causa divergenza.
- Un **valore basso** indica un'elevata divergenza, con conseguente perdita di prestazioni.

mathKernel1: Branch Efficiency 80.00%

mathKernel2: Branch Efficiency 100.00%

Nonostante la warp divergence, il compilatore CUDA applica ottimizzazioni anche con **-G** abilitato, risultando in una branch efficiency di **mathKernel1** superiore al 50% teorico.

### 3.8.17 Architetture Pre-Volta (< CC 7.0)



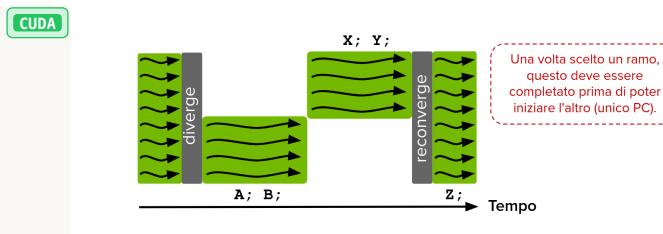
- **Singolo Program Counter** e **Call Stack** condiviso per tutti i 32 thread del warp (puntano alla stessa istruzione).
- Il warp agisce come una unità di esecuzione coesa/sincrona (stato dei thread è tracciato a livello di warp intero).
- **Maschera di Attività (Active Mask)** per specificare i thread attivi nel warp in ciascun istante.
- La maschera viene **salvata** fino alla riconvergenza del warp, poi **ripristinata** per riesecuzione sincrona.

#### Limitazioni

- Quando c'è divergenza, i thread che prendono branch diverse **perdonano concorrenza** fino alla riconvergenza.
- Possibili **deadlock** tra thread in un warp, se i thread dipendono l'uno dall'altro in modo circolare.

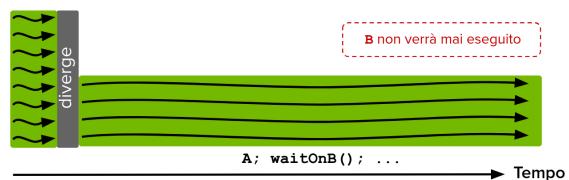
#### Esempio di Divergenza (Pseudo-Code)

```
if (threadIdx.x < 4) {
 A;
 B;
} else {
 X;
 Y;
}
Z;
```



#### Esempio di Potenziale Deadlock

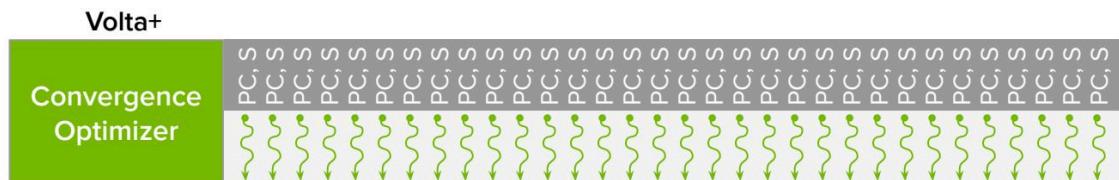
```
if (threadIdx.x < 4) {
 A;
 waitOnB();
} else {
 B;
 waitOnA();
}
```



### 3.8.17.1 Architettura Volta (CC 7.0+) e Independent Thread Scheduling

#### Concetto chiave

L'**Independent Thread Scheduling (ITS)** consente piena concorrenza tra i thread, indipendentemente dal warp.



### Stato di Esecuzione per Thread

- Ogni thread mantiene il **proprio stato di esecuzione**, inclusi program counter e stack di chiamate.
- Permette di cedere l'esecuzione a livello di **singolo thread** (non sono più obbligati a eseguire in lockstep).

### Attesa per Dati

- Un thread può attendere che un altro thread produca dati, **facilitando la comunicazione e la sincronizzazione** tra di essi.

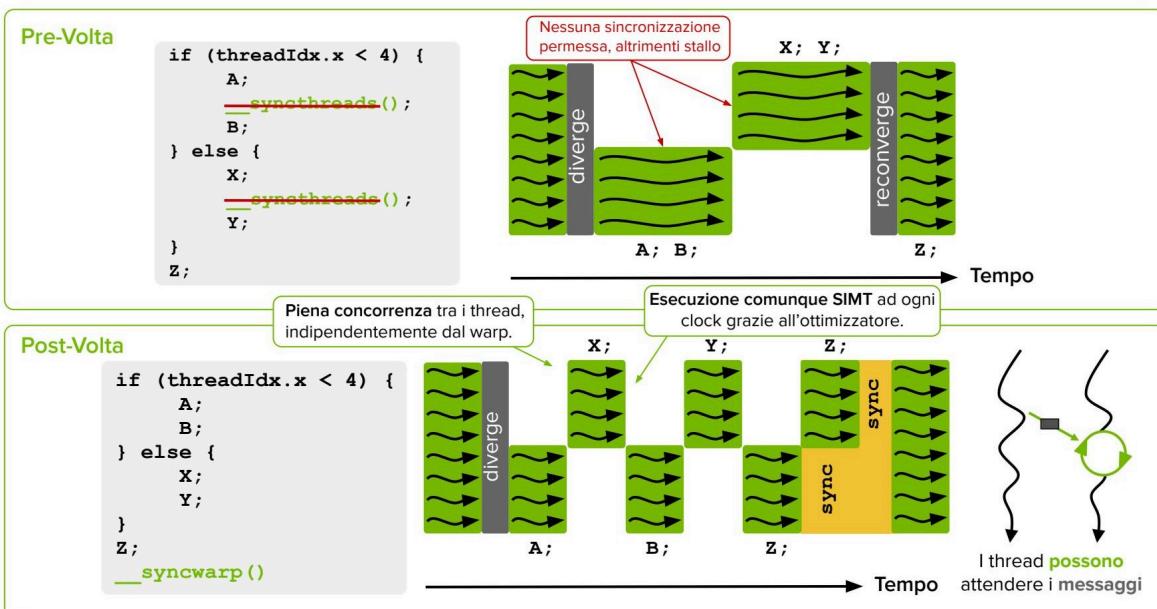
### Ottimizzazione della Pianificazione

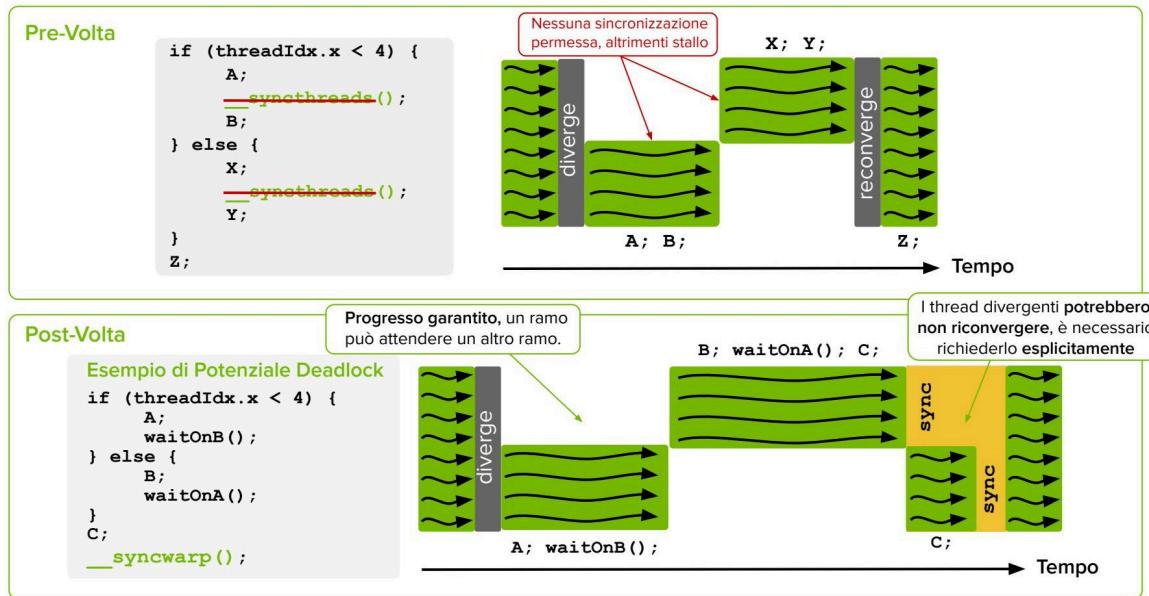
- Un **ottimizzatore di scheduling** raggruppa i thread attivi dello stesso warp in unità SIMT.
- Così facendo, si mantiene l'alto throughput dell'esecuzione SIMT, come nelle GPU NVIDIA precedenti.

### Flessibilità Maggiore

- I thread possono ora divergere e riconvergere indipendentemente con **granularità sub-warp**.
- Apre a pattern di programmazione che erano impossibili o problematici nelle architetture precedenti.

### 3.8.17.2 Confronto Pre-Volta vs Post-Volta





### 3.8.18 Introduzione di `__syncwarp` in Volta

#### Scopo

- Introdotta dall'architettura Volta per supportare l'ITS e migliorare la gestione della divergenza dei thread.
- Permette di sincronizzare esplicitamente e riconvergere i thread all'interno di un warp.
- Blocca l'esecuzione del thread corrente finché tutti i thread specificati nella maschera non hanno raggiunto il punto di sincronizzazione.

```
void __syncwarp(unsigned mask=0xffffffff);
```

CUDA

#### Vantaggi

- Evita comportamenti non deterministici dovuti alla divergenza intra-warp.
- Garantisce che tutti i thread del warp specificato siano allineati prima di comunicare o accedere a dati condivisi.
- Abilita l'esecuzione sicura di algoritmi a grana fine (riduzioni, scambi in shared memory, operazioni cooperative).

#### Esempio di Utilizzo

```

if (threadIdx.x < 16) {
 // Codice per i primi 16 thread
} else {
 // Codice per gli ultimi 16 thread
}
__syncwarp(); // Sincronizza tutti i thread del warp qui

```

CUDA

#### Quando è davvero necessaria `__syncwarp`?

Dopo una divergenza:

- È superflua se ogni thread lavora su dati privati, senza comunicare.
- È necessaria se c'è comunicazione o dipendenza tra thread del warp.

### 3.8.19 Confronto Pre-Volta vs Post-Volta

|                                   | Pre-Volta                                                      | Post-Volta                                                              |
|-----------------------------------|----------------------------------------------------------------|-------------------------------------------------------------------------|
| <b>Program Counter</b>            | Singolo per warp                                               | Individuale per thread                                                  |
| <b>Scheduling</b>                 | <u>Lockstep</u> : tutti i thread del warp eseguono insieme     | <u>Indipendente</u> : ogni thread può progredire autonomamente          |
| <b>Sincronizzazione</b>           | <u>Implicita</u> : i thread sono automaticamente sincronizzati | <u>Explicita</u> : richiede <u>__syncwarp</u>                           |
| <b>Divergenza</b>                 | Serializzazione dei rami divergenti                            | Esecuzione interlacciata dei rami possibile                             |
| <b>Deadlock Intra-Warp</b>        | Possibili in certi scenari                                     | Largamente mitigati                                                     |
| <b>Prestazioni con Divergenza</b> | Penalità per serializzazione                                   | Penalità simile, nessun miglioramento intrinseco                        |
| <b>Complessità del Codice</b>     | Workaround necessari per certi algoritmi                       | Implementazioni più naturali possibili (ma richiede gestione esplicita) |

### ITS: Limitazioni

- ITS non può esonerare gli sviluppatori da una programmazione parallela impropria. Nessuno scheduling hardware può salvare dal **livelock** (ovvero thread che sono tecnicamente in esecuzione ma non fanno progressi reali).
- Il progresso è garantito **solo per i warp residenti** al momento. I thread rimarranno in Sinc attesa infinita se il loro progresso dipende da un warp che non lo è.
- Non garantisce la riconvergenza, quindi le assunzioni relative alla programmazione a Diversibile livello di warp potrebbero non essere valide (usare esplicitamente \_\_syncwarp).
- Bisogna prestare più attenzione per garantire il comportamento SIMD dei warp.
- ITS introduce **overhead hardware** per la gestione indipendente di program counter e Pres call stack per ogni thread, aumentando la flessibilità ma richiedendo più risorse.

## 3.9 Sincronizzazione e Comunicazione

### 3.9.1 Sincronizzazione in CUDA - Motivazioni

#### 1. Asincronia tra Host e Device

- **Comportamento di Base**: L'host e il device operano in modo **asincrono**.
- Senza sincronizzazione, l'host potrebbe tentare di utilizzare risultati **non ancora pronti** o **modificare dati ancora in uso** dalla GPU.

#### 2. Sincronizzazione tra Thread all'interno di un Blocco

- **Comportamento di Base**: I thread all'interno di un blocco possono eseguire in ordine arbitrario e a velocità diverse.
- Quando i thread dello stesso blocco necessitano di condividere dati (utilizzando, ad esempio, la shared memory) o coordinare le loro azioni, è necessaria una sincronizzazione esplicita.

### 3. Coordinazione all'Interno dei Warp

- **Comportamento di Base:**
  - **Pre-Volta:** I thread all'interno di un warp eseguivano sempre la stessa istruzione contemporaneamente (modello SIMD).
  - **Post-Volta** (CUDA 9.0+): Introdotta l'esecuzione indipendente dei thread (ITS) nel warp.
- Con l'esecuzione indipendente dei thread, la sincronizzazione esplicita diventa necessaria per garantire la **coerenza nelle operazioni intra-warp**.

### 3.9.2 Race Condition (Hazard)

#### Cos'è?

Una **race condition** si verifica quando più thread accedono **concorrentemente (almeno uno in scrittura)** e in modo **non sincronizzato** alla stessa locazione di memoria, causando **risultati imprevedibili ed errori**.

#### Tipi di Race Condition (Noti anche nella pipeline dei processori)

- **Read-After-Write (RAW):** Un thread legge prima che un altro finisca di scrivere.
- **Write-After-Read (WAR):** Un thread scrive dopo che un altro ha letto, invalidando il valore.
- **Write-After-Write (WAW):** Più thread scrivono nella stessa locazione, rendendo il valore indeterminato.

#### Perché si verificano?

- I thread in un blocco sono logicamente paralleli ma non sempre fisicamente simultanei.
- Sono eseguiti in warp che possono trovarsi in punti diversi del codice.
- **Senza sincronizzazione**, l'ordine di esecuzione tra thread è **imprevedibile**.

#### Prevenzione delle Race Condition

- **All'interno di un Thread Block**
  - Utilizzare `__syncthreads()` per sincronizzare i thread e garantire la visibilità dei dati condivisi.
  - `__syncthreads()` garantisce che il thread A legga dopo che il thread B ha scritto.
- **Tra Thread Block diversi:**
  - Non esiste sincronizzazione diretta. L'unico modo sicuro è terminare il kernel e avviare uno nuovo.

### 3.9.3 Deadlock in CUDA

#### Cos'è?

- Un **deadlock** (o stallo) in CUDA si verifica quando i thread di un blocco si bloccano reciprocamente in attesa di sincronizzazioni o risorse non raggiungibili, causando il **blocco permanente** dell'esecuzione del kernel.
- Può insorgere in presenza di **sincronizzazioni condizionali** o **dipendenze** non gestite correttamente.

#### Condizioni per il Deadlock

- **Sincronizzazione Condizionale:** Uso di `__syncthreads()` all'interno di condizioni (`if, else`), dove solo una parte dei thread del blocco raggiunge il punto di sincronizzazione.
- **Dipendenze Circolari:** Situazioni in cui gruppi di thread attendono reciprocamente il completamento di operazioni, creando un ciclo di dipendenze irrisolvibile.
- **Risorse Condivise:** Gestione non corretta dell'accesso alla memoria condivisa o ad altre risorse comuni.

### Prevenzione/Gestione del Deadlock

- **Sincronizzazione Completa:** Evitare `__syncthreads()` nei rami condizionali divergenti; assicurarsi che tutti i thread del blocco raggiungano i punti di sincronizzazione.
- **Ristrutturazione del Codice:** Rimuovere le dipendenze condizionali organizzando le operazioni in modo che tutti i thread completino una fase prima di passare alla successiva.
- **Independent Thread Scheduling:** Con architetture Volta e successive, i thread di un warp possono avanzare in modo più indipendente, grazie all'Independent Thread Scheduling ed alleviare il problema.

### 3.9.4 Sincronizzazione in CUDA

La sincronizzazione è il meccanismo che permette di **coordinare** l'esecuzione di thread paralleli e garantire la **correttezza** dei risultati, evitando **race condition deadlock** e **accessi concorrenti non sicuri** alla memoria.

### Livelli di Sincronizzazione in CUDA

- **Livello di Sistema (Host-Device):**
  - Blocca l'applicazione host finché tutte le operazioni sul device non sono completate.
  - Garantisce che il device abbia terminato l'esecuzione (copie, kernels, etc) prima che l'host proceda.
  - **Firma:** `cudaError_t cudaDeviceSynchronize(void);` // può causare overhead bloccando l'host
- **Livello di Blocco (Thread Block):**
  - Sincronizza tutti i thread all'interno di un singolo thread block.
  - Ogni thread attende che tutti gli altri thread nel blocco raggiungano il punto di sincronizzazione.
  - Garantisce la visibilità delle modifiche alla shared memory tra i thread del blocco.
  - **Firma:** `__device__ void __syncthreads(void);` // riduce le prestazioni se usato troppo
- **Livello di Warp** (Disponibile con ITS a partire da CUDA 9.0 e architetture Volta+)
  - Sincronizza i thread all'interno di un singolo warp.
  - Garantisce la **riconvergenza** dei thread in presenza di divergenza.
  - **Ottimizza la cooperazione** tra thread dello stesso warp.
  - **Firma:** `__device__ void __syncwarp(unsigned mask=0xffffffff);` // minimo overhead

La sincronizzazione è il meccanismo che permette di **coordinare** l'esecuzione di thread paralleli e garantire la **correttezza** dei risultati, evitando **race condition\*deadlock** e **accessi concorrenti non sicuri** alla memoria.

### Esempi

| Livello di Sistema                                                                                                                                                                                                                            | Livello di Blocco                                                                                                                                                                                                                                                                      | Livello di Warp                                                                                                                                                                                                                                                                                     |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre><code>__global__ void simpleKernel() {     // Operazioni del kernel }  int main() {     ...     simpleKernel&lt;&lt;&lt;g, b&gt;&gt;&gt;();     cudaDeviceSynchronize();     printf("Kernel completato\n");     return 0; }</code></pre> | <pre><code>__global__ void blockSyncKernel() {     __shared__ int sharedData;      if (threadIdx.x == 0) {         sharedData = 42;     }     __syncthreads();      if (threadIdx.x == 1) {         printf("Valore condiviso: %d\n",                 sharedData);     } }</code></pre> | <pre><code>__global__ void warpSyncKernel() {     __shared__ int sharedData;      if (threadIdx.x == 0)         sharedData = 99;     __syncwarp();      if (threadIdx.x &lt; 32)         printf("Thread %d, valore: %d\n",                 threadIdx.x,                 sharedData); }</code></pre> |

### 3.9.5 Operazioni Atomiche in CUDA

v

### Perché sono Necessarie le Operazioni Atomiche?

- **Problema:** Race condition in operazioni **Read-Modify-Write**
  - ▶ Più thread **accedono e modificano** la stessa locazione di memoria contemporaneamente.
  - ▶ Risultati **imprevedibili e inconsistenti**.

### Scenario Tipico

```
__global__ void increment(int *counter) {
 int old = *counter; // Legge il valore attuale dalla memoria
 old = old + 1; // Incrementa il valore letto
 *counter = old; // Scrive il nuovo valore nella stessa locazione
}
```

CUDA

### Conseguenze

- **Conteggi Errati:** Il valore finale potrebbe non riflettere correttamente il numero di incrementi eseguiti.
- **Aggiornamenti di Dati Persi:** Le modifiche apportate da alcuni thread potrebbero essere sovrascritte da altri.
- **Comportamento Non Deterministico:** L'applicazione potrebbe dare risultati diversi ad ogni esecuzione.

### Soluzione

Operazioni atomiche **garantiscono l'integrità** delle operazioni Read-Modify-Write in ambiente concorrente.

### Cosa sono le Operazioni Atomiche? ([Documentazione Online](#))

- Operazioni **Read-Modify-Write** eseguite (solo su funzioni device) come **singola istruzione hardware invisibile**.

### Caratteristiche

- **Esclusività dell'accesso alla memoria:** L'hardware assicura che solo un thread alla volta può eseguire l'operazione sulla stessa locazione di memoria. I thread che eseguono operazioni atomiche sulla stessa posizione vengono messi in coda ed eseguiti in serie (correttezza garantita).
- **Prevenzione delle interferenze:** Evitano che i thread interferiscano tra loro durante la modifica dei dati.
- **Compatibilità con memoria globale e condivisa:** Operano su word di 32, 64 bit o 128 bit.
- **Riduzione del parallelismo effettivo**, poiché i thread devono aspettare il proprio turno per accedere alla memoria.

### Tipiche Operazioni Atomiche:

- **Matematiche:** Addizione, sottrazione, massimo, minimo, incremento, decremento.
- **Bitwise:** Operazioni bit a bit come AND, OR, XOR.
- **Swap:** Scambio del valore in memoria con un nuovo valore.

### Utilizzo di Base

```
__global__ void safeIncrement(int *counter) {
 atomicAdd(counter, 1); // Incrementa il valore atomico, evitando condizioni di gara
}
```

CUDA

### 3.9.6 Operazioni Atomiche in CUDA - Esempi d'Uso

#### Operazioni Atomiche

```
// Operazioni atomiche aritmetiche
int atomicAdd(int* addr, int val); // Somma val a *addr
int atomicSub(int* addr, int val); // Sottrae val da *addr
int atomicMax(int* addr, int val); // Aggiorna *addr al max tra *addr e val
```

CUDA

```

int atomicMin(int* addr, int val); // Aggiorna *addr al min tra *addr e val
unsigned int atomicInc(unsigned int* addr, unsigned int val); // Incrementa *addr, ciclato tra 0 e val
unsigned int atomicDec(unsigned int* addr, unsigned int val); // Decrementa *addr, ciclato tra 0 e val
// Operazioni atomiche di confronto
int atomicExch(int* addr, int val); // Scambia *addr con val
int atomicCAS(int* addr, int cmp, int val); // Confronta *addr con cmp, aggiorna *addr a val se uguali
// Operazioni atomiche bitwise
int atomicAnd(int* addr, int val); // AND tra *addr e val, aggiorna addr
int atomicOr(int* addr, int val); // OR tra *addr e val, aggiorna addr
int atomicXor(int* addr, int val); // XOR tra *addr e val, aggiorna addr

```

- Leggono il valore originale dalla memoria, eseguono l'operazione e salvano il risultato **nello stesso indirizzo, restituendo il valore originale pre-modifica.**
- **Supporto a Tipi Estesi:** Esistono anche varianti atomiche per operazioni su tipi a 64 bit (**long long int**) e floating point (**float** e **double**), supportate su architetture recenti.

## 3.10 Ottimizzazione delle Risorse

### 3.10.1 Resource Partitioning in CUDA

#### Cos'è il Resource Partitioning?

- Come abbiamo visto, assegnare molti warp a un SM aiuta a nascondere la latenza, ma i limiti delle risorse possono impedire di raggiungere il massimo supportato.
- Il **Resource Partitioning** riguarda la **suddivisione e la gestione delle risorse hardware** limitate all'interno di una GPU, in particolare all'interno di ogni SM.
- L'obiettivo è **trovare un equilibrio** nella distribuzione di registri e memoria condivisa tra thread e blocchi, **ottimizzando l'efficienza complessiva dell'esecuzione** dei kernel CUDA.

#### Partizionamento delle Risorse nell'SM

- Ogni SM ha una quantità limitata di registri e memoria condivisa:
  - **Register File:** Un insieme di registri a 32 bit, partizionati tra i thread attivi.
  - **Memoria Condivisa:** Una quantità fissa di memoria condivisa, partizionata tra i blocchi di thread attivi.
- Il numero di thread block e warp che possono risiedere simultaneamente su un SM dipende dalla:
  - **Disponibilità di Risorse:** Quantità di registri e memoria condivisa disponibili sull'SM.
  - **Richiesta del Kernel:** Quantità di registri e memoria condivisa richiesti dal kernel per l'esecuzione.
- Se le risorse di uno SM non permettono di eseguire almeno un blocco di thread, il kernel **fallisce**.

### 3.10.2 Anatomia di un Thread Block

#### Requisiti di Risorse per SM

Tutti i blocchi in una griglia eseguono lo stesso programma usando lo stesso numero di thread, portando a **3 requisiti di risorse fondamentali:**

##### 1. Dimensione del Blocco

Il numero di thread che devono essere concorrenti.

##### 2. Memoria Condivisa

È comune a tutti i thread dello stesso blocco.

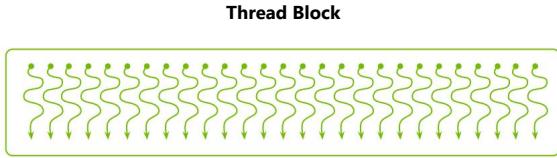
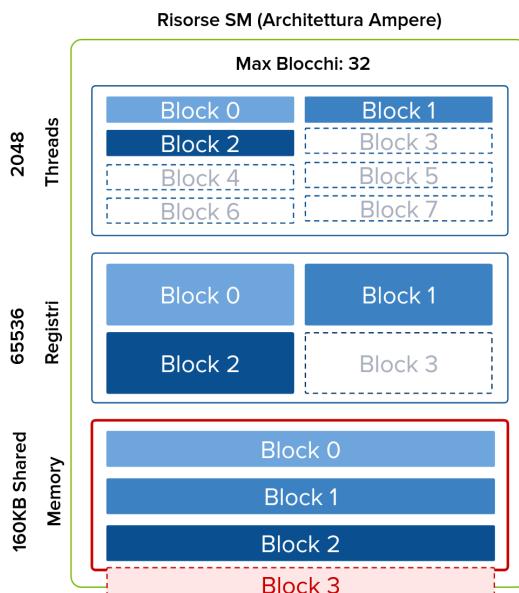
##### 3. Registri

Dipendono dalla complessità del programma.  
**(thread-per-blocco × registri-per-thread)**



Un blocco mantiene un numero costante di thread ed esegue unicamente su un singolo SM

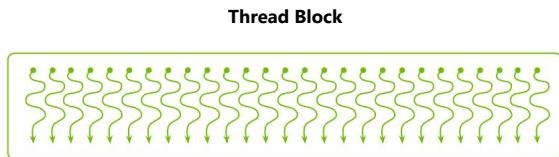
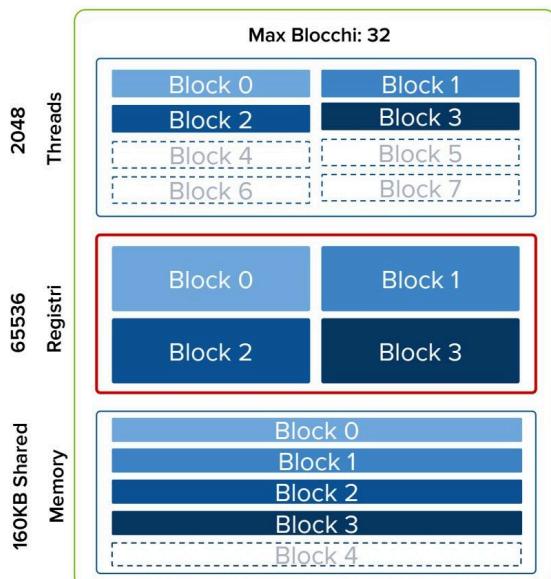
```
__global__ void simpleKernel(float* out) {
 CUDA
 int tid = threadIdx.x;
 float myValue = 3.14;
 __shared__ float sharedData[64];
 sharedData[tid] = myValue;
 __syncthreads();
 out[tid] = sharedData[tid] * myValue;
}
```



Un blocco contiene un numero fisso di thread ed esegue unicamente su un singolo SM

#### Esempio di Requisiti di Risorse per i Blocchi

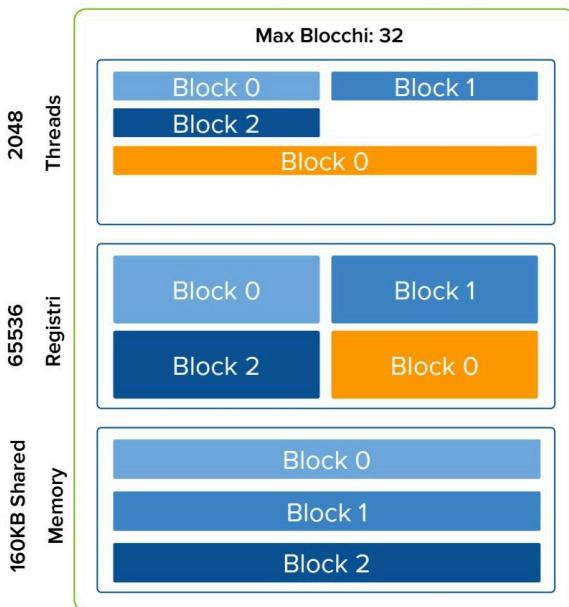
|                          |                |
|--------------------------|----------------|
| Thread per Blocco        | 256            |
| Registri per Thread      | 64             |
| Registri per Blocco      | (256*64)=16384 |
| Shared Memory per Blocco | 48Kb           |



Un blocco contiene un numero fisso di thread ed esegue unicamente su un singolo SM

#### Esempio di Requisiti di Risorse per i Blocchi

|                          |                  |
|--------------------------|------------------|
| Thread per Blocco        | 256              |
| Registri per Thread      | 64               |
| Registri per Blocco      | $(256*64)=16384$ |
| Shared Memory per Blocco | 32Kb             |

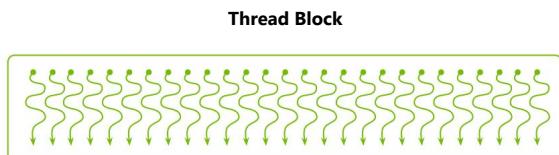
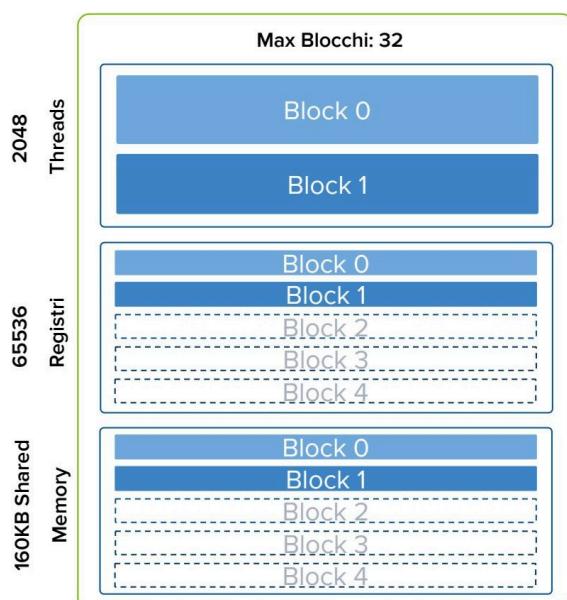


#### Esempio di requisiti (Griglia Blu)

|                          |                  |
|--------------------------|------------------|
| Thread per Blocco        | 256              |
| Registri per Thread      | 64               |
| Registri per Blocco      | $(256*64)=16384$ |
| Shared Memory per Blocco | 48Kb             |

#### Esempio di requisiti (Griglia Arancione)

|                          |                  |
|--------------------------|------------------|
| Thread per Blocco        | 512              |
| Registri per Thread      | 32               |
| Registri per Blocco      | $(512*32)=16384$ |
| Shared Memory per Blocco | 0                |



Un blocco contiene un numero fisso di thread ed esegue unicamente su un singolo SM

#### Esempio di Requisiti di Risorse per i Blocchi

Thread per Blocco

768

Registri per Thread

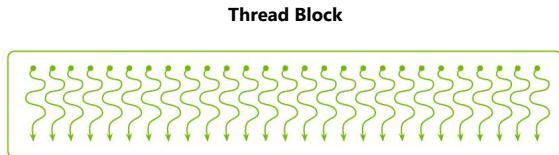
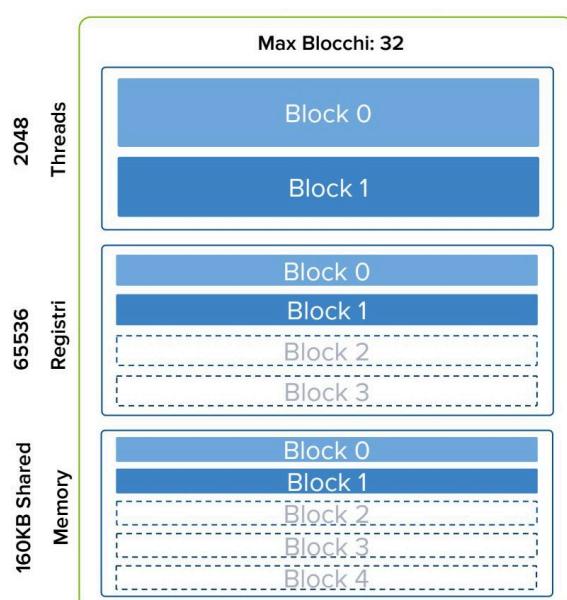
16

Registri per Blocco

$(768 * 16) = 12288$

Shared Memory per Blocco

32Kb



Un blocco contiene un numero fisso di thread ed esegue unicamente su un singolo SM

#### Esempio di Requisiti di Risorse per i Blocchi

Thread per Blocco

1024

Registri per Thread

16

Registri per Blocco

$(1024 * 16) = 12288$

Shared Memory per Blocco

32Kb

### 3.10.2.1 Compute Capability (CC) - Limiti SM

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

| Compute Capabil-<br>ity | Architet-<br>tura | Max Thread<br>per Blocco | Max Thread<br>per SM* | Max Warps<br>per SM* | Max Blocchi<br>per SM* | Max Reg-<br>istri per<br>Thread | Memoria Condivisa<br>per SM |
|-------------------------|-------------------|--------------------------|-----------------------|----------------------|------------------------|---------------------------------|-----------------------------|
| 1.x                     | Tesla             | 512                      | 768                   | 24/32                | 8                      | 124                             | 16KB                        |
| 2.x                     | Fermi             | 1024                     | 1536                  | 48                   | 8                      | 63                              | 48KB                        |
| 3.x                     | Kepler            | 1024                     | 2048                  | 64                   | 16                     | 255                             | 48KB                        |
| 5.x                     | Maxwell           | 1024                     | 2048                  | 64                   | 32                     | 255                             | 64KB                        |
| 6.x                     | Pascal            | 1024                     | 2048                  | 64                   | 32                     | 255                             | 64KB                        |
| 7.x                     | Volta/<br>Turing  | 1024                     | 1024/2048             | 32/64                | 16/32                  | 255                             | 96KB                        |
| 8.x                     | Ampere/<br>Ada    | 1024                     | 1536/2048             | 48/64                | 16/24                  | 255                             | 164KB                       |
| 9.x                     | Hopper            | 1024                     | 2048                  | 64                   | 32                     | 255                             | 228KB                       |
| 10.x/12.x               | Black-<br>well    | 1024                     | 2048/1536             | 64/48                | 32                     | 255                             | 128KB                       |

\*Valori concorrenti per singolo SM

[https://en.wikipedia.org/wiki/UDA=Version\\\_features\\\_and\\\_specifications](https://en.wikipedia.org/wiki/UDA=Version\_features\_and\Specifications)

### 3.10.3 Occupancy

#### Cosa è l'Occupancy?

- L'occupancy rappresenta il **grado di utilizzo delle risorse** di calcolo dell'SM.
- L'occupancy è il **rappporto** tra i warp attivi e il numero massimo di warp supportati per SM (vedi compute capability):

$$\text{Occupancy [%]} = \text{Active Warps} / \text{Maximum Warps}$$

#### Punti Chiave

- L'occupancy misura l'efficacia nell'uso delle risorse dell'SM:
  - **Occupancy Ottimale:** Quando raggiunge un livello sufficiente per nascondere la latenza. Un ulteriore aumento potrebbe degradare le prestazioni a causa della riduzione delle risorse disponibili per thread.
  - **Occupancy Bassa:** Risulta in una scarsa efficienza nell'emissione delle istruzioni, poiché non ci sono abbastanza warp eleggibili per nascondere la latenza tra istruzioni dipendenti.
- **Un'occupancy elevata non garantisce sempre prestazioni migliori:** Oltre certa soglia, fattori come i pattern di accesso alla memoria e il parallelismo delle istruzioni possono diventare più rilevanti per l'ottimizzazione.

#### Strumenti per l'Ottimizzazione

- **Strumenti di Profiling:** Nsight Compute consente di recuperare facilmente l'occupancy, offrendo dettagli sul numero di warp attivi per SM e sull'efficienza delle risorse di calcolo (tuttavia, l'occupancy non deve mai essere guardata in isolamento. Diventa utile se combinata con altre metriche del profiler).
- **Suggerimento:** Osservare gli effetti sul tempo di esecuzione del kernel a diversi livelli di occupancy.

### 3.10.3.1 Occupancy Teorica vs Effettiva

#### Misure di Occupancy

L'occupancy di un kernel CUDA si divide in **teorica**, basata sui limiti hardware, ed **effettiva**, misurata a runtime.

#### Occupancy Teorica (Theoretical)

- L'occupancy teorica è **determinata dalla configurazione di lancio** (numero di blocchi/thread, quantità di memoria condivisa, numero di registri per thread) e i limiti dell'SM (compute capability).
- **Limite massimo warp attivi per SM = (Limite massimo blocchi attivi) × (Warp per blocco)**
- È possibile aumentare il limite incrementando il numero di warp per blocco (dimensioni del blocco) o modificando i fattori limitanti (registri e/o shared memory) per aumentare i blocchi attivi per SM.

#### Occupancy Effettiva (Achieved)

- Misura il **numero reale di warp attivi** durante l'esecuzione del kernel.
- Il numero reale di warp attivi varia durante l'esecuzione del kernel, man mano che i warp iniziano e terminano.
- **Calcolo dell'occupazione effettiva** (vedere Nsight Compute):
  - L'occupazione ottenuta è misurata su ciascun scheduler di warp utilizzando **contatori di prestazioni hardware** che registrano i warp attivi ad ogni ciclo di clock.
  - I conteggi vengono sommati su tutti i warp scheduler di ogni SM (1 per SMSP) e divisi per i cicli di clock attivi dell'SM per calcolare la **media dei warp attivi**.
  - Dividendo per il numero massimo di warp attivi supportati dall'SM (Maximum Warps), si ottiene l'**occupazione effettiva media** per SM durante l'esecuzione del kernel.

#### Obiettivi di Ottimizzazione

- L'occupazione effettiva **non può** superare l'occupazione teorica (rappresenta il limite superiore).
- Pertanto, il primo passo per aumentare l'occupazione è **incrementare quella teorica**, modificando i fattori limitanti.
- Successivamente, è necessario verificare se il valore ottenuto è vicino a quello teorico per ridurre il gap.

#### Cause di Bassa Occupazione Effettiva

- L'occupancy effettiva sarà inferiore a quella teorica quando il numero teorico di warp attivi non viene mantenuto durante l'attività dello SM (il problema forse non è il resource partitioning). Ciò può accadere quando si ha:
  - **Carico di lavoro sbilanciato nei blocchi**: Quando i warp in un blocco hanno tempi di esecuzione diversi (es. warp divergence), si crea un "tail effect" che riduce l'occupazione. Soluzione: bilanciare il carico tra i warp.
  - **Carico di lavoro sbilanciato tra blocchi**: Se i blocchi della grid hanno durate diverse, si può lanciare un maggior numero di blocchi o kernel concorrenti per ridurre l'effetto coda.
  - **Numero insufficiente di blocchi lanciati**: Se la grid ha meno blocchi del numero di SM del device, alcuni SM rimarranno inattivi. Ad esempio, lanciare 60 blocchi su un dispositivo con 80 SM lascia 20 SM sempre inattivi, riducendo l'utilizzo complessivo della GPU.
  - **Wave Parziale**: L'ultima ondata di blocchi potrebbe non saturare tutti gli SM. Ad esempio, con 80 SM che supportano 2 blocchi ciascuno e una grid di 250 blocchi: le prime due wave eseguono 160 blocchi (80 SM × 2), ma la terza wave ha solo 90 blocchi, lasciando alcuni SM parzialmente utilizzati o inattivi.

### 3.10.3.2 Nota Importante sull'Occupancy

#### Ricorda

L'obiettivo finale **non è massimizzare l'occupancy**, ma **minimizzare il tempo di esecuzione del kernel**.

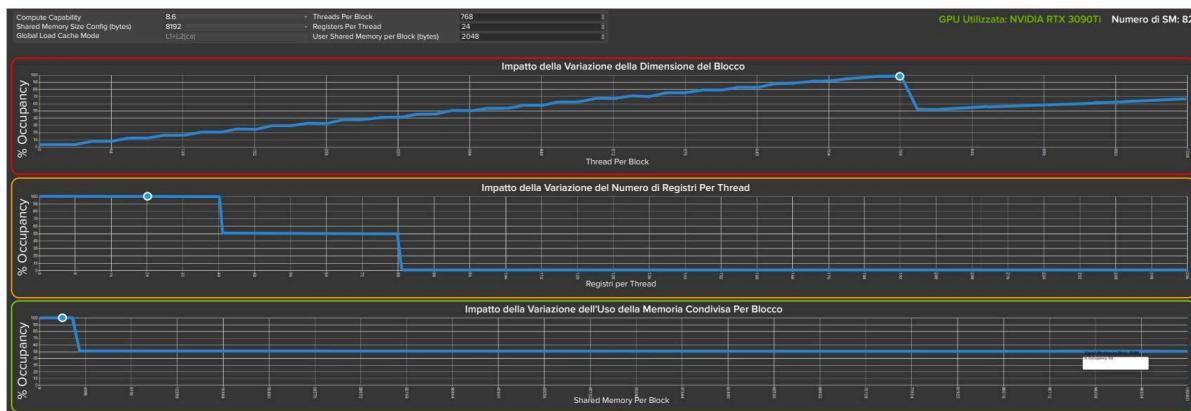
### Linee guida pratiche

- **✗ Occupancy < 25-30 % → problema serio**
  - Non ci sono abbastanza warp per nascondere le latenze (critico soprattutto per kernel memory-bound).
  - La GPU risulta sottoutilizzata, con SM spesso inattivi
  - **Azione:** ridurre l'uso di registri / shared memory per thread oppure aumentare il numero di thread per blocco, se possibile.
- **✓ Occupancy 50-80 % → generalmente buono**
  - Warp sufficienti per un buon **latency hiding** nella maggior parte dei casi
  - Risorse adeguate per ciascun thread
  - **Focus:** ottimizzare coalescing, divergenza, e accessi alla memoria
- **⚠ Occupancy > 90 % → non sempre vantaggiosa** (kernel memory bound compute bound)
  - Latency hiding ottimo ma attenzione: se limita troppo le risorse il guadagno si perde.
  - Oltre una certa soglia, più occupancy potrebbe non migliorare le performance.
- **Occupancy** = strumento per trovare il **giusto equilibrio tra latency hiding e risorse per thread**.

### 3.10.3.3 Nsight Compute: Occupancy Calculator

Nsight Compute offre uno strumento utile chiamato "Occupancy Calculator" (Documentazione</u>) che consente di:

- **Stimare l'Occupancy:** Calcola l'occupancy di un kernel CUDA su una determinata GPU.
- **Ottimizzare le Risorse:** Mostra l'impatto di registri e memoria condivisa sull'occupancy.
- **Migliorare le Prestazioni:** Fornisce suggerimenti per massimizzare l'uso delle risorse dell'SM e migliorare le prestazioni complessive.



### Nsight Compute: Occupancy Calculator

#### Linee Guida per le Dimensioni di Griglia e Blocchi

- Mantenere il numero di thread per block **multiplo** della dimensione del warp (32).
- **Evitare dimensioni di block piccole:** Iniziare con almeno 128 o 256 thread per block.
- Regolare la dimensione del blocco **in base ai requisiti di risorse** del kernel.
- Antenere il **numero di blocchi molto maggiore del numero di SM** per esporre sufficiente parallelismo al dispositivo (latency hiding).
- **Condurre esperimenti** per scoprire la migliore configurazione di esecuzione e utilizzo delle risorse.

## 3.11 Parallelismo Avanzato

### 3.11.1 Introduzione al CUDA Dynamic Parallelism

#### Il Problema:

- Algoritmi complessi (altamente dinamici) possono richiedere **strutture di parallelismo più flessibili**.
- La suddivisione dei problemi in kernel separati da lanciare in sequenza **dalla CPU** creano un collo di bottiglia.

#### La Soluzione: Dynamic Parallelism

- Introdotto in CUDA 5.0 nel 2012 (Architettura Kepler), il CUDA Dynamic Parallelism (CDP) è disponibile su device con una Compute Capability 3.5 o superiore.
- Permette la **creazione e sincronizzazione** dinamica (on the fly) di nuovi kernel direttamente dalla GPU.
- È possibile posticipare a **runtime** la decisione su quanti blocchi e griglie creare sul device (utile quando la **quantità di lavoro nidificato è sconosciuta**)
- Supporta un approccio **gerarchico e ricorsivo** al parallelismo **evitando** continui passaggi fra CPU e GPU.

#### Possibili Applicazioni

- **Algoritmi ricorsivi** (es: Quick Sort, Merge Sort) → [Ricorsione con profondità sconosciuta]
- **Strutture dati ad albero** (es: Alberi di ricerca, Alberi decisionali) → [Elaborazione parallela nidificata irregolare]
- **Elaborazione di immagini e segnali** (es. Region growing) → [Decomposizione dinamica delle aree di elaborazione]

#### Vantaggi

- **Flessibilità**: Adattamento dinamico del parallelismo in base ai dati elaborati, senza dover prevedere tutto a priori.
- **Scalabilità**: Sfruttamento ottimale delle risorse GPU, creando nuovi blocchi e griglie solo quando necessario.
- **Efficienza**: Riduzione del collo di bottiglia CPU-GPU, spostando parte del controllo dell'esecuzione sulla GPU.

#### 3.11.1.1 Dynamic Parallelism: Eliminare il Round-trip CPU-GPU

##### Lancio da CPU (Approccio Tradizionale)

```
__global__ void kernelA() {
}

__global__ void kernelB() {
}

int main() {
 ...
 kernelA<<<1,1>>>();
 ... // ottieni i risultati
 if(condition) {
 kernelB <<<1, 1>>>();
 }
}
```

C

##### Lancio dalla GPU (Dynamic Parallelism)

```
__global__ void kernelB() {
}

__global__ void kernelA() {
 if(condition)
```

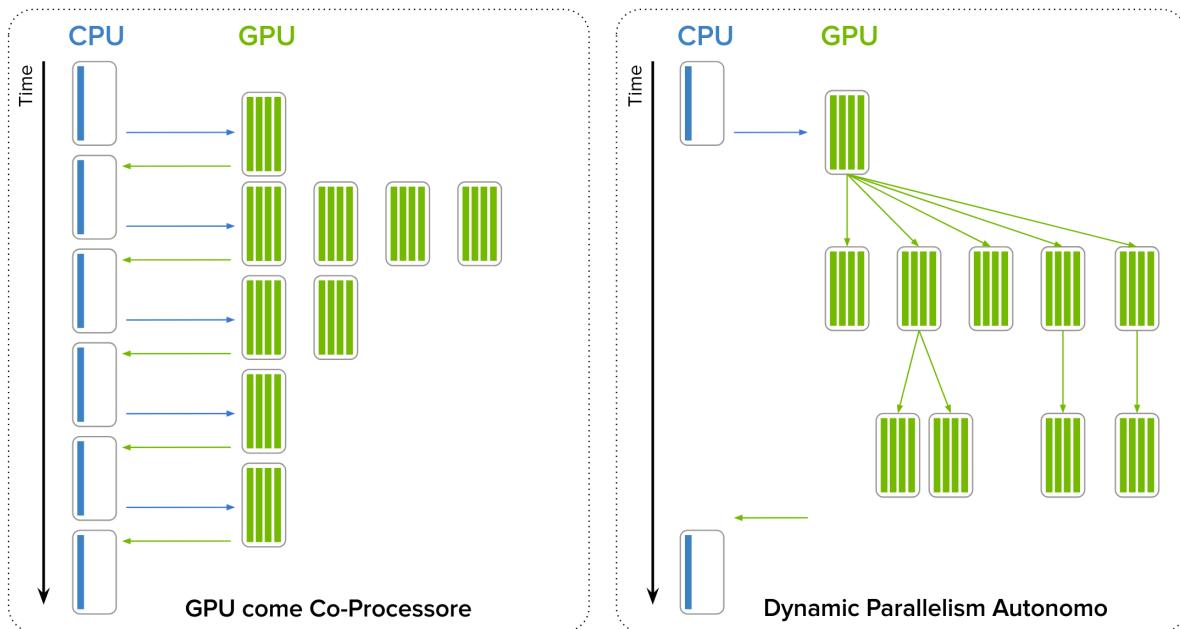
CUDA

```

 kernelB <<<1, 1>>>();
}

int main() {
 ...
 kernelA<<<1,1>>>();
}

```



### 3.11.2 Esecuzione Nidificata con CUDA Dynamic Parallelism

#### Come Funziona:

- Un thread, un blocco di thread o una griglia (**parent**) lancia una nuova griglia (**child grid**).
- Una child grid lanciata con dynamic parallelism **eredita** dal kernel padre certi attributi e limiti come, ad esempio, la configurazione della **cache L1/memoria condivisa** e **dimensione dello stack**.
- I blocchi della griglia child possono essere eseguiti in parallelo e in modo indipendente rispetto al kernel padre.
- Il kernel/griglia parent continua immediatamente dopo il lancio del kernel child (**asincronicità**).
- Il **child deve sempre completare prima che il thread/blocco/griglia parent sia considerato completo**.
- Un parent si considera **completato** solo quando tutte le griglie child create dai suoi thread (tutti) hanno terminato l'esecuzione.

#### Visibilità e Sincronizzazione:

- Ogni child grid lanciata da un thread è **visibile a tutti i thread dello stesso blocco**.
- Se i thread di un blocco terminano prima che tutte le loro griglie child abbiano completato, il sistema attiva automaticamente una **sincronizzazione implicita** per attendere il completamento di queste griglie.
- Un thread può **sincronizzarsi esplicitamente** con le proprie griglie child e con quelle lanciate da altri thread **nel suo blocco** utilizzando **primitive di sincronizzazione** (`cudaDeviceSynchronize`).
- Quando un thread parent lancia una child grid, **l'esecuzione della griglia figlio non è garantita immediatamente**, a meno che il blocco di thread genitore non esegua una **sincronizzazione esplicita**.

### 3.11.3 Esempio di CUDA Dynamic Parallelism

```

// Kernel Figlio
__global__ void childKernel(void* data){
 // Operazioni sui dati
}

```

CUDA

```
// Kernel Genitore
__global__ parentKernel(void *data){
 childKernel<<<16, 16>>>(data);
}

// Chiamata del Parent Kernel dall'Host
parentKernel<<<256, 64>>>(data);

// Kernel ricorsivo supportato
__global__ recursiveKernel(void* data){
 if(continueRecursion == true)
 recursiveKernel<<<64, 16>>>(data);
}
```

```
// Kernel Figlio
__global__ childKernel(void* data) {
 // Operazioni sui dati
}

// Kernel Genitore
__global__ parentKernel(void *data) {
 childKernel<<<16, 16>>>(data);
}

// Chiamata del Parent Kernel dall'Host
parentKernel<<<256, 64>>>(data);

// Kernel ricorsivo supportato
__global__ recursiveKernel(void* data) {
 if(continueRecursion == true)
 recursiveKernel<<<64, 16>>>(data);
}
```

**Struttura del Codice**

- Stessa sintassi usata nel codice host.
- Si noti che ogni thread che incontra un lancio di kernel lo esegue.
- Quanti thread vengono lanciati in totale per l'esecuzione di `childKernel`?

Nel caso in cui si desidera solo una griglia child per blocco parent usare:

```
if (threadIdx.x == 0)
 childKernel<<<16,16>>>(data);
```

**Configurazione Griglia/Blocco:** I kernel lanciati dinamicamente possono avere una configurazione di griglia e blocco indipendente dal kernel genitore.

### 3.11.4 Memoria in CUDA Dynamic Parallelism

#### Memoria Globale e Costante:

- Le griglie parent e child **condividono lo stesso spazio di memoria globale** (accesso **concorrente**) e **memoria costante**. Tuttavia, la **memoria locale e condivisa** (shared memory) sono **distinte** fra parent e child.
- La coerenza della memoria globale non è garantita tra parent e child (be careful), tranne che:
  - **All'avvio della griglia child.**
  - **Quando la griglia child completa.**

#### Visibilità della Memoria:

- Tutte le operazioni sulla memoria globale eseguite dal thread parent **prima** di lanciare una griglia child sono garantite essere **visibili e accessibili** ai thread della griglia child.
- Tutte le operazioni di memoria eseguite dalla griglia child sono garantite essere visibili al thread genitore **dopo che il genitore si è sincronizzato** con il completamento della griglia child.

#### Memoria Locale e Condivisa (Shared Memory): X

- La memoria locale e condivisa sono **private** per un thread o un blocco di thread, rispettivamente.
- La memoria locale e condivisa **non sono visibili o coerenti** tra parent e child.
- La memoria locale è uno spazio di archiviazione privato per un thread e **non è visibile al di fuori di quel thread**.

### Limitazioni

- Non è valido passare un puntatore a memoria locale o shared come argomento quando si lancia una griglia child.
- È possibile passare variabili **per copia** (by value).

### 3.11.5 Memoria in CUDA Dynamic Parallelism

#### Memoria Globale e Costante:

#### Passaggio dei Puntatori alle Child Grid

##### Possono Essere Passati ✓

- Memoria Globale (sia variabili `__device__` sia memoria allocata con `cudaMalloc`)
- Memoria Zero-Host Copy
- Memoria Costante (ereditata dal parent e non può essere modificata)

##### Non Possono Essere Passati ✗

- Memoria Condivisa (variabili `__shared__`)
- Local Memory (incluse variabili dello stack)

\* Analizzeremo meglio queste memorie in seguito ("2.3 Modello di Memoria in CUDA")

### Limitazioni

- Non è valido passare un puntatore a memoria locale o shared come argomento quando si lancia una griglia child.
- È possibile passare variabili per copia (by value).

### 3.11.6 Gestione dello Scambio Dati nel Parallelismo Dinamico

#### Quindi, Come Restituire un Valore da un Child Kernel?

```
__global__ void childKernel(void* p) { ... }
__global__ void parentKernel(void) {
 int v = 0; // Variabile nei registri/memoria locale del padre
 childKernel<<<16, 16>>>(&v); // Passa indirizzo non accessibile
 ...
}
```

Versione Errata

#### Versione Corretta

```
__device__ int v = 0; // Variabile in memoria globale
__global__ void childKernel(void* p) { ... }
__global__ void parentKernel(void) {
 childKernel<<<16, 16>>>(&v); // Passa indirizzo accessibile della memoria globale
 ...
}
```

### 3.11.7 Consistenza della Memoria nel Parallelismo Dinamico

#### Scenario Sicuro:

- Quando il thread parent scrive in memoria globale **prima** di lanciare la griglia child.
- Il thread figlio vedrà **correttamente** il valore scritto dal padre.

#### Scenario Problematico:

- **Scrittura da parte del child:**
  - Il thread parent potrebbe **non** vedere i valori scritti dal child.
- **Scrittura del parent dopo il lancio:**
  - Se il padre scrive dopo aver lanciato il figlio, si crea una “**race condition**”.
  - Non si può sapere quale valore verrà letto.

```
__global__ void parentKernel(void) {
 v = 1; // OK
 childKernel<<<16,16>>>();
}

__device__ int v = 0; // Variabile globale
__global__ void childKernel(void) {
 printf("v = %d\n", v);
}
```

```
__global__ void parentKernel(void) {
 v = 1; // OK
 childKernel<<<16,16>>>();
 v = 2; // Race condition!
}
```

Non c'è sincronizzazione esplicita

### 3.11.8 Dipendenze Annidate in CUDA

CPU

|   |  |
|---|--|
| A |  |
| B |  |
| C |  |

GPU

|   |  |
|---|--|
| X |  |
| Y |  |
| Z |  |

```
__global__ void B(float *data)
{
 do_stuff(data);
 X <<< ... >>> (data);
 Y <<< ... >>> (data);
 Z <<< ... >>> (data);
 cudaDeviceSynchronize();
 do_more_stuff(data);
}

void main()
{
 float *data;
 do_stuff(data);
 A <<< ... >>> (data);
 B <<< ... >>> (data);
 C <<< ... >>> (data);
 cudaDeviceSynchronize();
 do_more_stuff(data);
}
```

Stessa Sintassi

### 3.11.9 Sincronizzazione con `cudaDeviceSynchronize()`

#### Funzione Principale

- **cudaDeviceSynchronize()** attende il **completamento** di tutte le griglie (kernel) precedentemente lanciate da qualsiasi thread del blocco corrente, includendo tutti i kernel discendenti (child, nipoti, ecc.) nella gerarchia.
- Se chiamata da un singolo thread, gli altri thread del blocco **continueranno** l'esecuzione.

## Sincronizzazione a Livello di Blocco

- **Attenzione:** `cudaDeviceSynchronize()` non implica una sincronizzazione fra thread del blocco.
- Il blocco di tutti i thread può essere ottenuto sia chiamando `cudaDeviceSynchronize()` da tutti i thread, sia facendo seguire la chiamata di `cudaDeviceSynchronize()` da parte di un singolo thread con `_syncthreads()`.

```
__global__ void parentKernel(float *a, float *b, float *c) {
 createData(a, b); // Tutti i thread generano i dati
 __syncthreads(); // Sincronizzazione dei thread nel blocco per garantire i dati
 if (threadIdx.x == 0) {
 childKernel<<<n, m>>>(a, b, c); // Lancio della griglia child (1 thread call)
 cudaDeviceSynchronize(); // Attesa per il completamento dei kernel discendenti
 }
 __syncthreads(); // Tutti i thread nel blocco attendono prima di utilizzare i dati
 consumeData(c); // I thread nel blocco possono ora usare i dati della griglia child
}
```

## Sincronizzazione con `cudaDeviceSynchronize()`

### 3.11.9.0.0.0.0.0.1 Funzione Principale

`cudaDeviceSynchronize()` attende il completamento di tutte le griglie (kernel) precedentemente lanciate da qualsiasi thread del blocco corrente, includendo tutti i kernel discendenti (child, nipoti, ecc.) nella gerarchia.

### 3.11.9.0.0.0.0.1.0.0.1 Sincre

#### Limiti

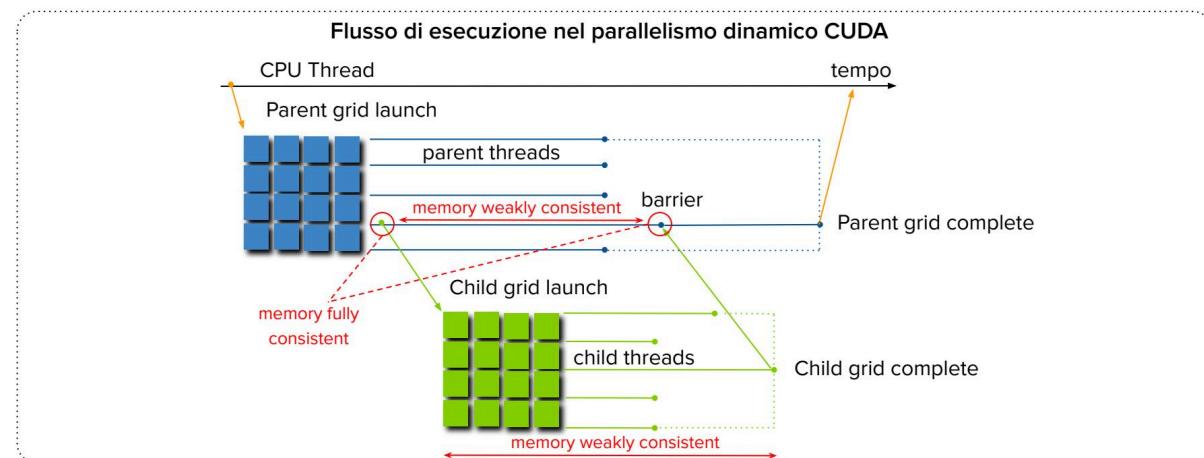
- `cudaDeviceSynchronize()` è un'operazione computazionalmente costosa perché:
  - ▶ Può causare la sospensione (swap-out) del blocco in esecuzione.
  - ▶ In caso di sospensione, richiede il trasferimento dell'intero stato del blocco (registri, memoria condivisa, program counter) nella memoria del device.
  - ▶ Il blocco dovrà poi essere ripristinato (swap-in) quando i kernel child saranno completati.
- Non dovrebbe essere chiamato al termine di un kernel genitore, poiché la **sincronizzazione implicita** viene già eseguita automaticamente.

`_syncthreads(); consumeData(c);`

cendo

L

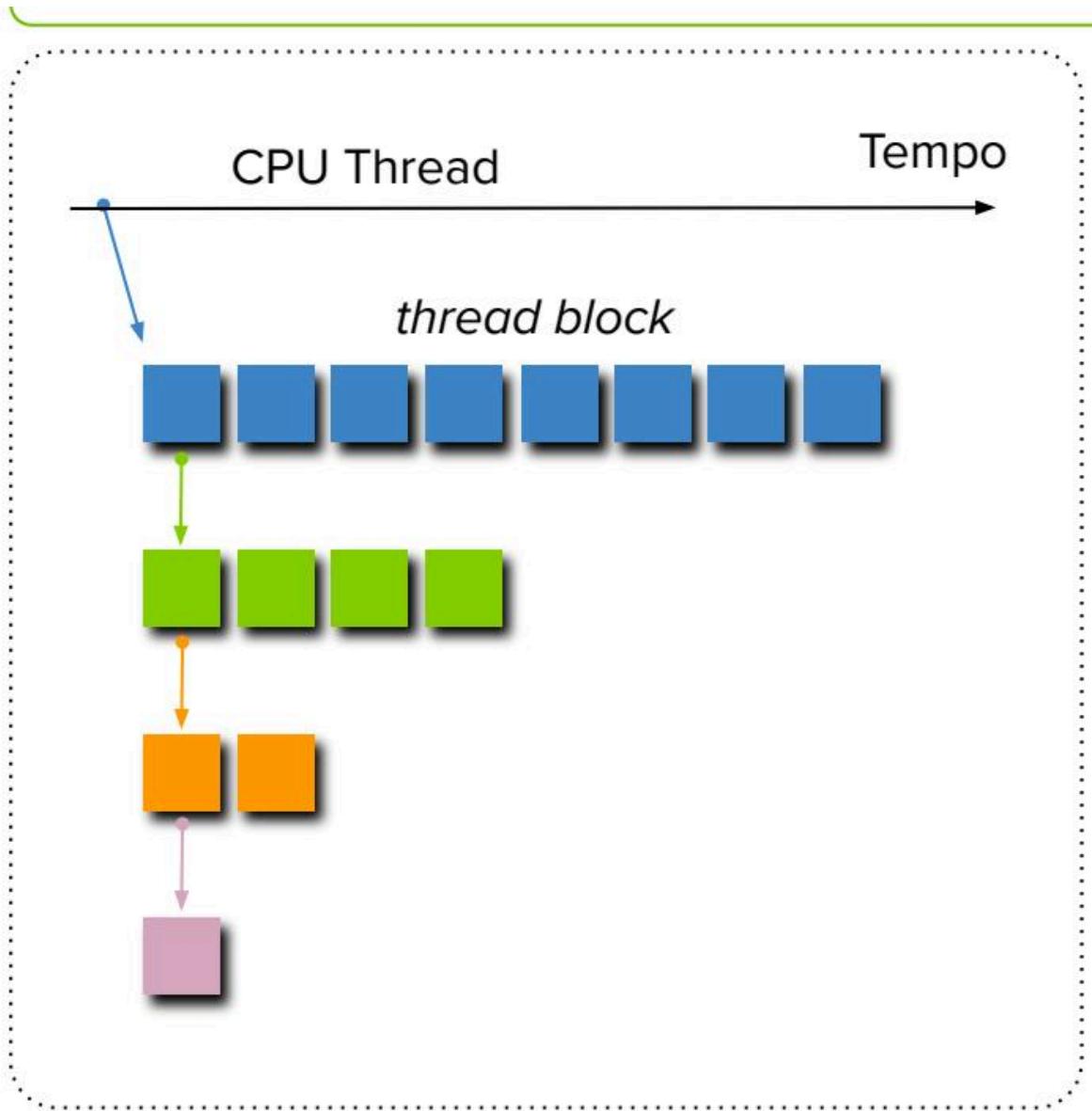
### 3.11.10 Esecuzione Nidificata con CUDA Dynamic Parallelism



- **Esecuzione Nidificata:** Il thread CPU lancia la griglia parent (**blu**), che a sua volta lancia una griglia child (**verde**).
- **Sincronizzazione Esplicita:** La barriera nella griglia parent dimostra una **sincronizzazione esplicita** (`cudaDeviceSynchronize()`) con la griglia child, assicurando che il parent attenda il completamento del child.
- **Completamento Gerarchico:** La griglia parent si considera **completata** solo dopo che la griglia child ha terminato.

### 3.11.11 Parallelismo Dinamico su GPU: Nested Hello World

- Il kernel seguente è un esempio di come utilizzare la **parallelizzazione dinamica** sulla GPU per eseguire un kernel ricorsivo.
- Il kernel viene invocato dalla applicazione **host** con una griglia di 8 thread in un singolo blocco. Il thread 0 di questo grid invoca un **nuovo grid** con la metà dei thread, e così via fino a quando non rimane solo un thread.



```
__global__ void nestedHelloWorld(int const iSize, int iDepth) {
 int tid = threadIdx.x;
 printf("Recursion=%d: Hello World from thread %d block %d\n",
 iDepth, tid, blockIdx.x);
 // Condizione di terminazione:
 // se c'è solo un thread, termina la ricorsione
 if (iSize == 1) return;
 // Calcola il numero di thread per
 // il prossimo livello (dimezza)
 int nthreads = iSize >> 1;
 // Solo il thread 0 lancia ricorsivamente una nuova grid,
 // se ci sono ancora thread da lanciare
 if (tid == 0) {
```

```
// Ricorsione
nestedHelloWorld<<<1, nthreads>>>(nthreads, ++iDepth);

// Stampa la profondità di esecuzione nidificata
printf("-----> nested execution depth: %d\n", iDepth);
}}
```

### 3.11.12 Nested Hello World : Compilazione ed Esecuzione

Per compilare il codice abilitando il parallelismo dinamico:

```
$ nvcc -arch=sm_86 -rdc=true -lcudadevrt nested_hello_world.cu -o nested_hello_world
--rdc=True: Abilita Relocatable Device Code, necessario per il parallelismo dinamico.
-lcudadevrt: Collega la CUDA Device Runtime Library (spesso implicito con -rdc=true).
-arch: Specifica l'architettura di destinazione della GPU (min. Kepler per il parallelismo dinamico.
Ampere in questo caso).
```

Profiling con **Nsight Compute** (tuttavia, il tracciamento dei kernel CDP per le architetture GPU Volta e superiori non è supportato).

#### Output (Terminale)

```
./nestedHelloWorld Configuration: grid 1 block 8
Recursion=0: Hello World from thread 0 block 0
Recursion=0: Hello World from thread 1 block 0
Recursion=0: Hello World from thread 2 block 0
Recursion=0: Hello World from thread 3 block 0
Recursion=0: Hello World from thread 4 block 0
Recursion=0: Hello World from thread 5 block 0
Recursion=0: Hello World from thread 6 block 0
Recursion=0: Hello World from thread 7 block 0
-----> nested execution depth: 1
Recursion=1: Hello World from thread 0 block 0
Recursion=1: Hello World from thread 1 block 0
Recursion=1: Hello World from thread 2 block 0
Recursion=1: Hello World from thread 3 block 0
-----> nested execution depth: 2
Recursion=2: Hello World from thread 0 block 0
Recursion=2: Hello World from thread 1 block 0
-----> nested execution depth: 3
Recursion=3: Hello World from thread 0 block 0
```

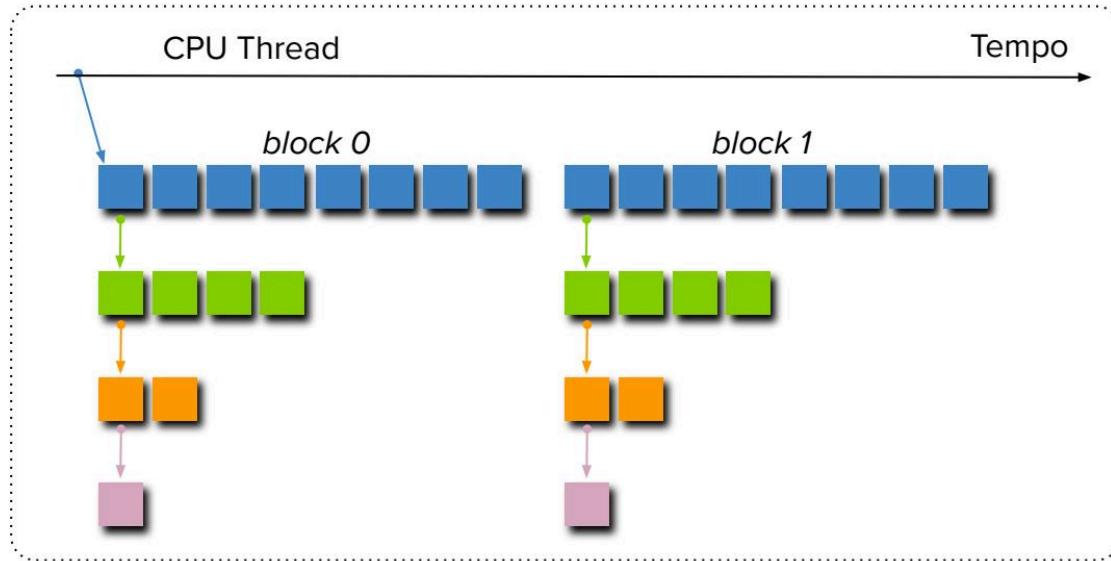
### 3.11.13 Nested Hello World : Compilazione ed Esecuzione

Ora, si provi a invocare la griglia parent con 2 blocchi invece di uno solo:

```
$./nestedHelloWorld 2
```

Perché l'ID dei blocchi per le griglie child è sempre 0 nei messaggi di output? (vedi codice precedente)

```
nestedHelloWorld<<<1, nthreads>>>(nthreads, ++iDepth);
```



### 3.11.13.0.0.0.0.0.0.0.1 Output (Terminale)

```
./nestedHelloWorld Configuration: grid 1 block 8
Recursion=0: Hello World from thread 0 block 1
Recursion=0: Hello World from thread 1 block 1
Recursion=0: Hello World from thread 2 block 1
Recursion=0: Hello World from thread 3 block 1
Recursion=0: Hello World from thread 4 block 1
Recursion=0: Hello World from thread 5 block 1
Recursion=0: Hello World from thread 6 block 1
Recursion=0: Hello World from thread 7 block 1
Recursion=0: Hello World from thread 0 block 0
Recursion=0: Hello World from thread 1 block 0
Recursion=0: Hello World from thread 2 block 0
Recursion=0: Hello World from thread 3 block 0
Recursion=0: Hello World from thread 4 block 0
Recursion=0: Hello World from thread 5 block 0
Recursion=0: Hello World from thread 6 block 0
Recursion=0: Hello World from thread 7 block 0
-----> nested execution depth: 1
-----> nested execution depth: 1
Recursion=1: Hello World from thread 0 block 0
Recursion=1: Hello World from thread 1 block 0
Recursion=1: Hello World from thread 2 block 0
Recursion=1: Hello World from thread 3 block 0
Recursion=1: Hello World from thread 0 block 0
Recursion=1: Hello World from thread 1 block 0
Recursion=1: Hello World from thread 2 block 0
Recursion=1: Hello World from thread 3 block 0
-----> nested execution depth: 2
-----> nested execution depth: 2
Recursion=2: Hello World from thread 0 block 0
Recursion=2: Hello World from thread 1 block 0
Recursion=2: Hello World from thread 0 block 0
Recursion=2: Hello World from thread 1 block 0
-----> nested execution depth: 3
Recursion=3: Hello World from thread 0 block 0
-----> nested execution depth: 3
```

```
Recursion=3: Hello World from thread 0 block 0
```

### 3.11.14 Restrizioni sul Parallelismo Dinamico

#### Compatibilità dei Dispositivi

Supportato solo da device con capacità di calcolo  $\geq 3.5$ .

#### Limitazioni di Lancio

I kernel **non** possono essere lanciati su device **fisicamente separati**.

#### Profondità Massima di Nidificazione

- Nesting depth imitata a **24 livelli**
- Nella pratica, limitata dalla memoria richiesta dal **runtime** del device.
- Runtime riserva **memoria aggiuntiva** per sincronizzazione griglia padre-figlio.

#### Deprecazione

- L'uso di **cudaDeviceSynchronize** nel **codice device** è stato **deprecato** in CUDA 11.6 (la versione host-side rimane supportata). Rimosso per compute capability  $> 9.0$ .
- Per GPU con compute capability  $< 9.0$  (es. Tesla T4 in Google Colab) e versione di CUDA  $\geq 11.6$  è possibile **forzare il supporto** usando il flag di compilazione **-D CUDA\_FORCE\_CDP1\_IF\_SUPPORTED**.

### 3.11.15 Riferimenti Bibliografici

#### Testi Generali

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

#### NVIDIA Docs

## **Appendices**