



Modello di Esecuzione CUDA

Sistemi Digitali, Modulo 2

A.A. 2024/2025

Fabio Tosi, Università di Bologna

Panoramica del Modello di Esecuzione CUDA

➤ Architettura Hardware GPU

- Introduzione al Modello di Esecuzione CUDA
- Organizzazione degli Streaming Multiprocessors (SM)
- Panoramica delle Architetture GPU NVIDIA

➤ Organizzazione e Gestione dei Thread

- Mappatura tra Vista Logica e Hardware
- Distribuzione e Schedulazione dei Blocchi sui SM

➤ Modello di Esecuzione SIMT e Warp

- Confronto tra SIMD e SIMT
- Warp e Gestione dei Warp
- Latency Hiding e Legge di Little
- Warp Divergence e Thread Independent Scheduling

➤ Sincronizzazione e Comunicazione

- Meccanismi di Sincronizzazione
- Operazioni Atomiche

➤ Ottimizzazione delle Risorse

- Resource Partitioning
- Occupancy

➤ Parallelismo Avanzato

- CUDA Dynamic Parallelism

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

- Astrazione dell'architettura GPU NVIDIA.
- Conservazione dei **concetti fondamentali** tra le generazioni.
- Esposizione delle **funzionalità architetturali** chiave per la programmazione CUDA.
- 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.

Streaming Multiprocessor (SM)

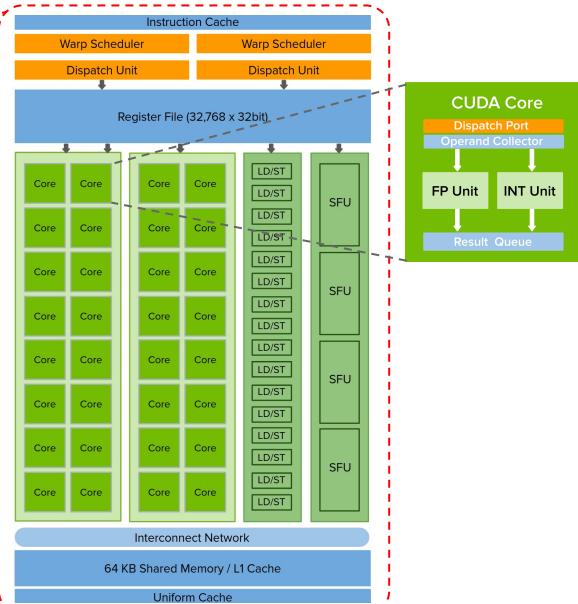
Cosa sono?

- 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 architetturale.

GPU (GeForce GTX 480)



Fermi SM (2010)



Streaming Multiprocessor (SM)

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.

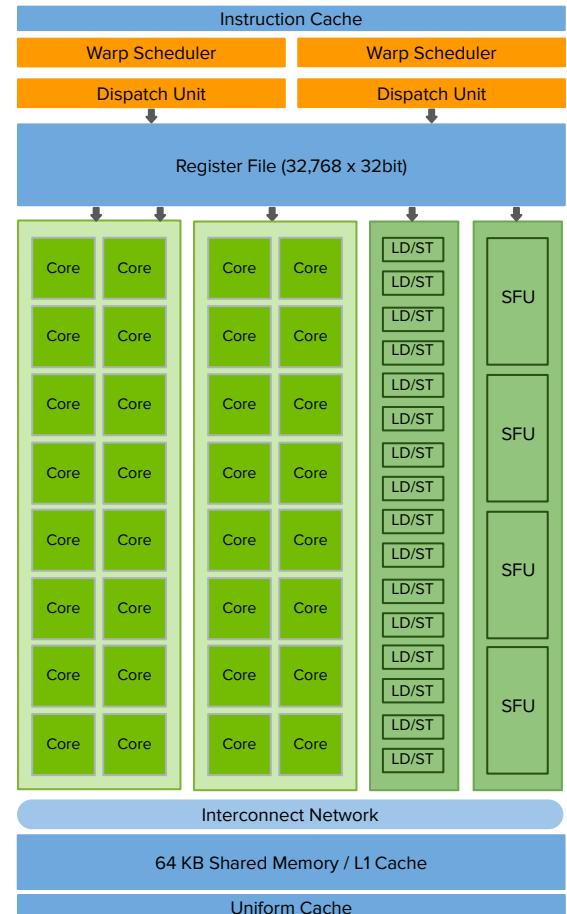
7. Dispatch Unit

- Assegna i thread selezionati alle unità di esecuzione.

8. Instruction Cache

- Memorizza temporaneamente le istruzioni usate di frequente.

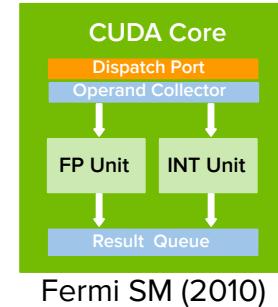
Fermi SM (2010)



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.



Fermi SM (2010)

Composizione e Funzionamento (Architettura Fermi e Precedenti)

- Inizialmente, i CUDA Core erano unità di elaborazione relativamente semplici, in grado di eseguire sia operazioni intere (INT) che in virgola mobile (FP) in un ciclo di clock (fully pipelined).
 - **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.

CUDA Core - Unità di Elaborazione CUDA

Evoluzione dell'Architettura (Kepler e successive)

- Dall'architettura Kepler, NVIDIA ha introdotto la **specializzazione delle unità di calcolo** all'interno di uno SM:
 - Unità FP64:** dedicate alle operazioni in virgola mobile a *doppia precisione*.
 - Unità FP32:** dedicate alle operazioni in virgola mobile a *singola precisione*.
 - Unità INT:** dedicate alle *operazioni intere*.
- General**
- AI**
- Grafica**
 - Tensor Core - TC** (Architettura Volta e successive): Unità specializzate particolarmente ottimizzate per moltiplicazioni fra matrici in *precisione ridotta/mista* (FP32, FP16, TF32, INT8, etc.).
 - Ray Tracing Core - RT** (Ampere e successive): Unità dedicate per l'accelerazione del *ray tracing*.
 - Unità di Texture:** ottimizzate per gestire *texture* e *operazioni di filtraggio*.
 - Unità di Rasterizzazione:** utilizzate per la *rasterizzazione* delle immagini durante il rendering.

Ruolo nel Modello CUDA

- Esecuzione Parallelia:** 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**, il **rendering grafico**.

Architettura Fermi (2010)

Caratteristiche Principali - GPU

- Prima architettura GPU completa per applicazioni HPC ad alte prestazioni.
- Fino a **512 CUDA cores** organizzati in **16 SM**.
- Ogni SM contiene:
 - **32 CUDA Cores**.
 - **2 Unità di Scheduling e Dispatch**.
 - **64KB di Shared Memory/Cache L1**
 - **32,768 Registri da 32 bit** (Max 63 per thread)
- **768 KB di cache L2** condivisa tra tutti gli SM.
- Interfaccia di memoria **GDDR5 a 384-bit**, supporto fino a **6 GB di memoria globale**.
- **GigaThread Engine**: Scheduler globale per la distribuzione dei thread blocks.
- **Interfaccia Host** per connessione CPU via PCI Express.

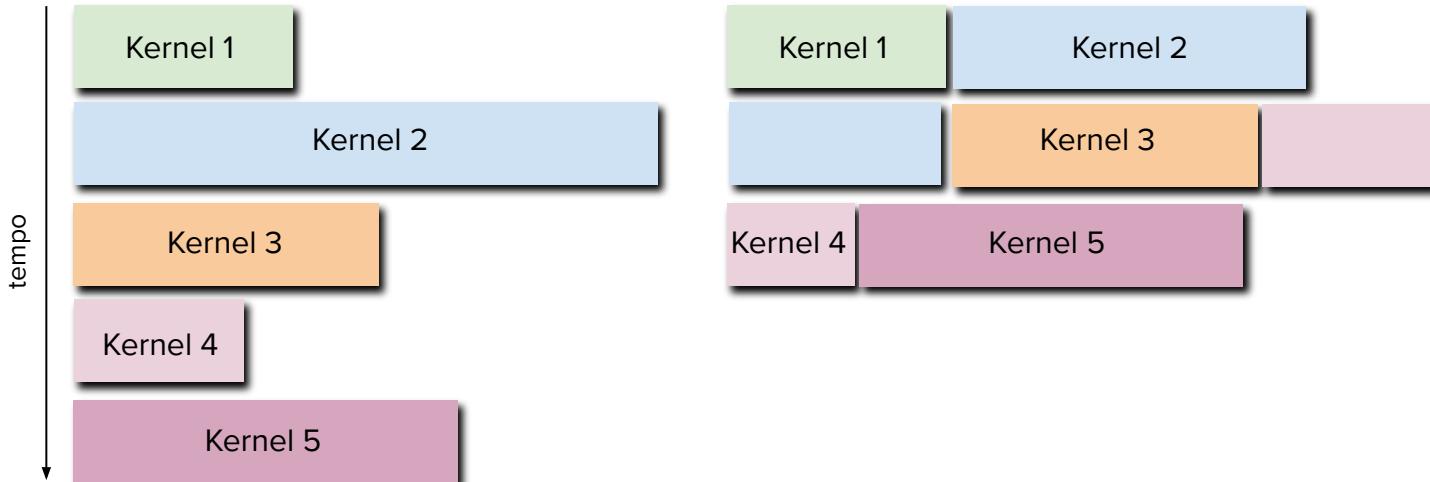
Fermi GPU (2010)



Architettura Fermi (2010)

Esecuzione Concorrente dei Kernel

- L'architettura Fermi permette l'**esecuzione simultanea di più kernel** sulla stessa GPU.
- Supporta fino a **16 kernel eseguiti contemporaneamente**.
- **Maggiore Utilizzo:** Ottimizza l'uso della GPU, soprattutto con kernel piccoli.
- **Appare come un'architettura MIMD (Multiple Instruction, Multiple Data)** dal punto di vista del programmatore.
- **Nota:** Le generazioni successive a Fermi supportano un numero ancora maggiore di kernel simultanei e introducono una gestione più avanzata delle risorse.



Architettura Kepler (2012)

Caratteristiche Principali - GPU

- L'architettura Kepler include 3 importanti novità:
 - Streaming Multiprocessors Potenziati (**SMX**)
 - **Dynamic Parallelism**: Permette ai kernel di lanciare altri kernel (lo vedremo).
 - **Hyper-Q**: Permette a più thread CPU di inviare simultaneamente lavori alla GPU tramite molteplici code hardware.
 - **GPU Boost**: Tecnologia che regola dinamicamente la frequenza di clock.
- **2688 Cuda Core** organizzati in **15 SMX**
- **6 Controller di Memoria a 64 bit**.
- **6 GB** di Memoria Globale DDR5.
- **Larghezza di Banda della Memoria**: 250 GB/s.
- **1536 KB** di **Cache L2**.
- **Interfaccia Host** di tipo **PCIe 3.0 x16**.

Kepler GPU (2012)

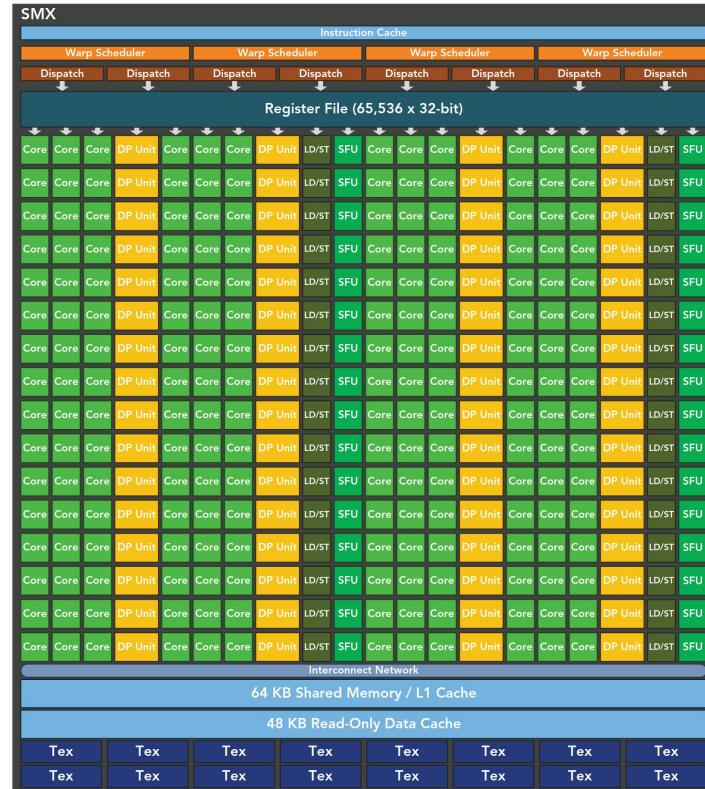


Architettura Kepler (2012) - GK100X SMX

Caratteristiche Principali - Singolo SMX

- Ogni SMX contiene **192 CUDA cores**, per un totale di 2.880 CUDA Core nel chip.
- Unità di Precisione:**
 - Unità di precisione singola (FP32): 192 CUDA cores.
 - Unità di precisione doppia (FP64): 64 unità ($\frac{1}{3}$ della velocità rispetto FP32)
- 32 Unità di Funzione Speciale (SFU).**
- 32 Unità di Load/Store (LD/ST).**
- 64 KB di Shared Memory/L1 Cache.**
- 48 KB di Read-Only Data Cache**
- 65,536 Registri da 32 bit** (Max 255 per thread).
- 4 Warp Scheduler.**
- 8 Instruction Dispatcher.**

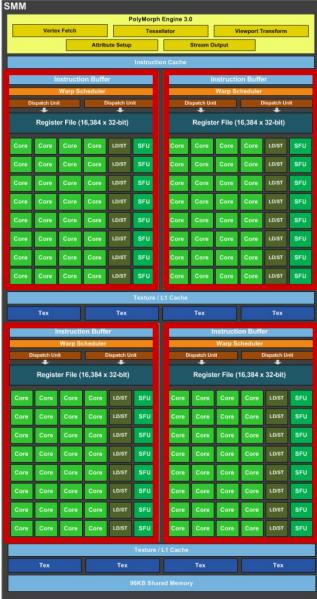
Kepler GK100X SMX (2012)



Streaming Multiprocessor - Evoluzione

Streaming Multiprocessor Sub-Partition (SMSP)

Maxwell SM (2014)



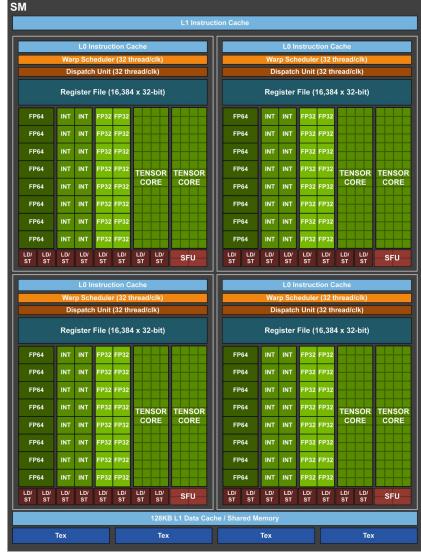
- Maggiore **efficienza energetica** rispetto a Kepler.
- Delta Color Compression** per ridurre la larghezza di banda.
- Divisione degli SM in 4** per ottimizzare l'uso delle risorse.

Pascal SM (2016)



- Introduzione di **NVLink** per connessioni ad alta velocità.
- Supporto per **HMB2** (High Bandwidth Memory).
- Miglioramento** della tecnologia **GPU Boost** per l'overclocking dinamico.

Volta SM (2017)



- Introduzione delle **Tensor Cores** per accelerare l'AI.
- Architettura CUDA più flessibile con prestazioni elevate.
- Multi-Process Service (MPS)** per gestire più carichi di lavoro in parallelo.

Streaming Multiprocessor - Evoluzione

Turing (2018)



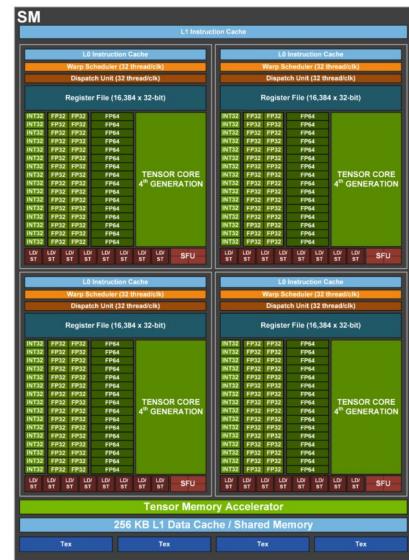
Ampere (2020)



Ada (2022)



Hopper (2022)



- Introduzione dei **RT Cores**: Ray-tracing in tempo reale.
- **DLSS** (Deep Learning Super Sampling)
- **Variable Rate Shading**

- Aumento del throughput FP32 per i CUDA Cores.
- Miglioramenti per **Tensor Cores e RT Cores**.
- **GPU Scheduler** ottimizzato per la gestione del carico di lavoro.

- Incremento delle prestazioni FP32 con **più CUDA Cores**.
- Introduzione di **DLSS 3.0** con frame generation.
- **RT Cores migliorati** per ray tracing avanzato.

- Tensor Cores di quarta generazione per AI.
- **Multi-Instance GPU (MIG)**.
- **Thread Block Clusters**
- **FP8 Precision** per migliorare il throughput.

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) × (CUDA Core per SM)

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

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

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.
(2 operazioni FP32/ciclo, e.g. Fused Multiply-Add)
- **Calcolo Throughput Teorico:** Throughput FP32=Totale CUDA Cores FP32 × 2 × Frequenza di clock.

Architettura	Totale CUDA FP32 Cores	Frequenza di Clock (Ghz)	FP32 Throughput (TFLOPS)
Tesla (GTX 200 series)	240	1,4	~0,67
Fermi (GTX 400/500 series)	512	1,4	~1,44
Kepler (GTX 600/700 series)	2880	0,9	~5,18
Maxwell (GTX 900 series)	2048	1,1	~4,51
Pascal (GTX 10 series)	2560	1,6	~8,19
Volta (Tesla V100)	5120	1,53	~15,62
Turing (RTX 20 series)	4608	1,65	~15,23
Ampere (RTX 30 series)	10752	1,8	~38,78
Ada Lovelace (RTX 40 series)	16384	2,5	~81,92
Hopper (GH series)	18432	1,8	~66,56

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

Tensor Core: Acceleratori di 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 **FP16**, **FP32**, **FP64**, **INT8**, **INT4**, **BF16** e nuovi formati come **TF32 (TensorFloat-32)**.
- Offrono un significativo **speedup** nel calcolo senza compromettere l'accuratezza.

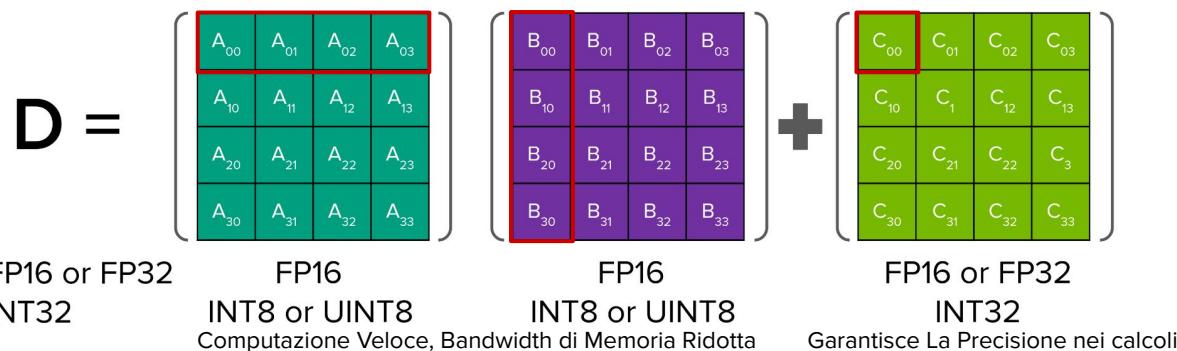
Evoluzione

- Miglioramenti:** Volta → Turing → Ampere → Hopper
- Integrazione con CUDA, cuDNN, TensorRT



Tensor Core: Acceleratori di Intelligenza Artificiale

- **Fused Multiply-Add (FMA)**: Un'operazione che combina una moltiplicazione e un'addizione di scalari in un unico passo, eseguendo $d = a \times b + c$. 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 $D = A \times B + C$
- Una **MMA** di dimensione $m \times n$ richiede $m \times n \times k$ operazioni FMA, dove k è il numero di colonne di A (o di righe di B).

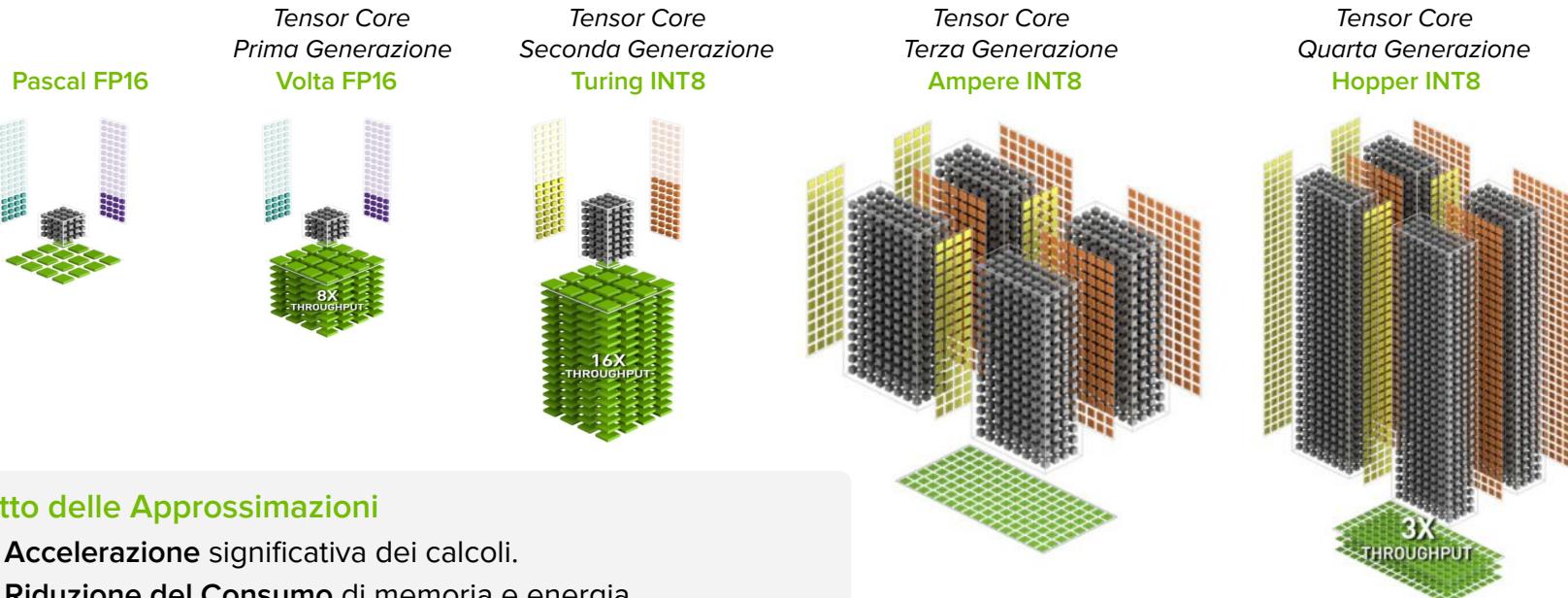


Esecuzione Parallelia

- 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**.

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 e energia.
- Perdita di Precisione:** Si è dimostrato che ha un impatto minimo sull'accuratezza finale dei modelli di deep learning.

Panoramica del Modello di Esecuzione CUDA

➤ Architettura Hardware GPU

- Introduzione al Modello di Esecuzione CUDA
- Organizzazione degli Streaming Multiprocessors (SM)
- Panoramica delle Architetture GPU NVIDIA

➤ Organizzazione e Gestione dei Thread

- Mappatura tra Vista Logica e Hardware
- Distribuzione e Schedulazione dei Blocchi sui SM

➤ Modello di Esecuzione SIMT e Warp

- Confronto tra SIMD e SIMT
- Warp e Gestione dei Warp
- Latency Hiding e Legge di Little
- Warp Divergence e Thread Independent Scheduling

➤ Sincronizzazione e Comunicazione

- Meccanismi di Sincronizzazione
- Operazioni Atomiche

➤ Ottimizzazione delle Risorse

- Resource Partitioning
- Occupancy

➤ Parallelismo Avanzato

- CUDA Dynamic Parallelism

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.
- Le variabili di identificazione e dimensione **gridDim**, **blockIdx**, **blockDim**, e **threadIdx** sono rese disponibili ad ogni thread e condivise nello stesso SM.
- Una volta assegnati a un SM, i thread di un blocco eseguono **esclusivamente** su quell'SM.

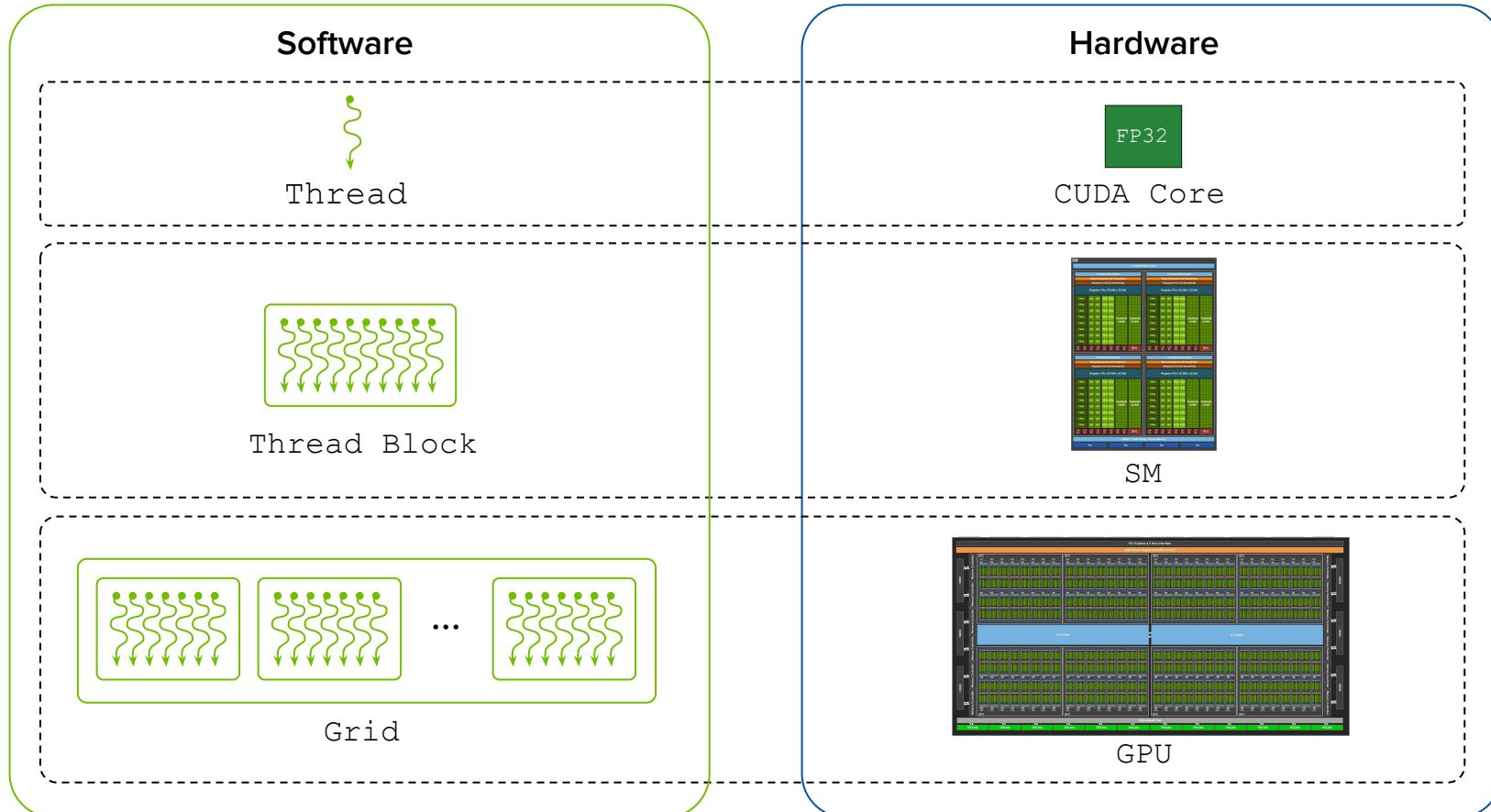
Gestione delle Risorse

- **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.).

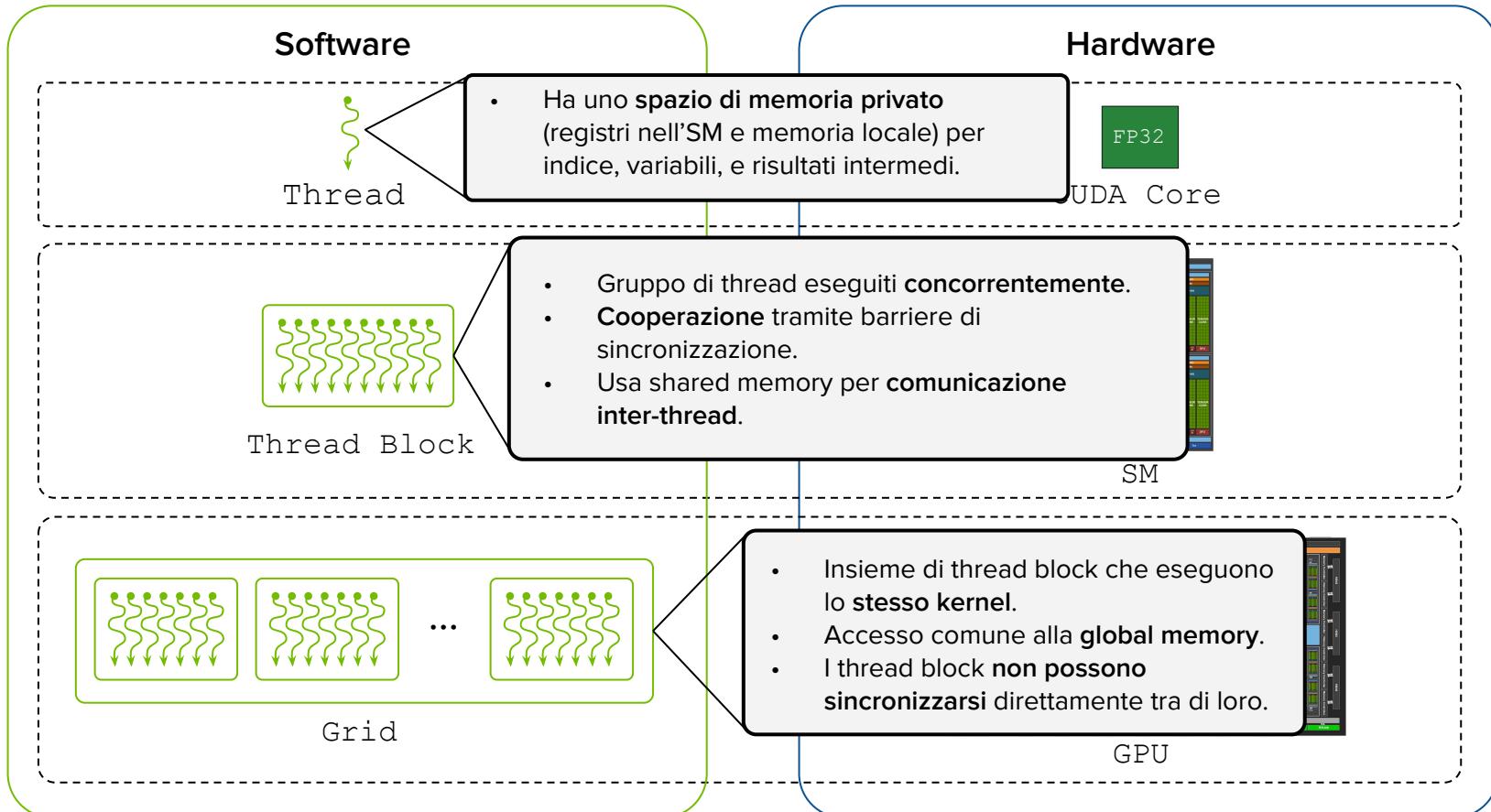
Parallelismo Multi-Livello

- **Parallelismo a Livello di Istruzione:** Le istruzioni all'interno di un singolo thread sono eseguite in pipeline.
- **Parallelismo a Livello di Thread:** Esecuzione concorrente di gruppi di thread (*warps*) sugli SM.

Corrispondenza tra Vista Logica e Vista Hardware



Corrispondenza tra Vista Logica e Vista Hardware



Corrispondenza tra Vista Logica e Vista Hardware

Software

- Un blocco di thread viene assegnato esclusivamente ad **un solo SM**.
- Una volta che un blocco di thread è stato assegnato ad un SM, vi rimane fino al completamento dell'esecuzione.



Thread Block



Grid

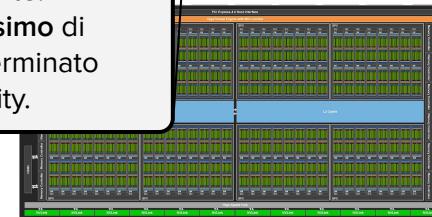
Hardware

FP32

CUDA Core



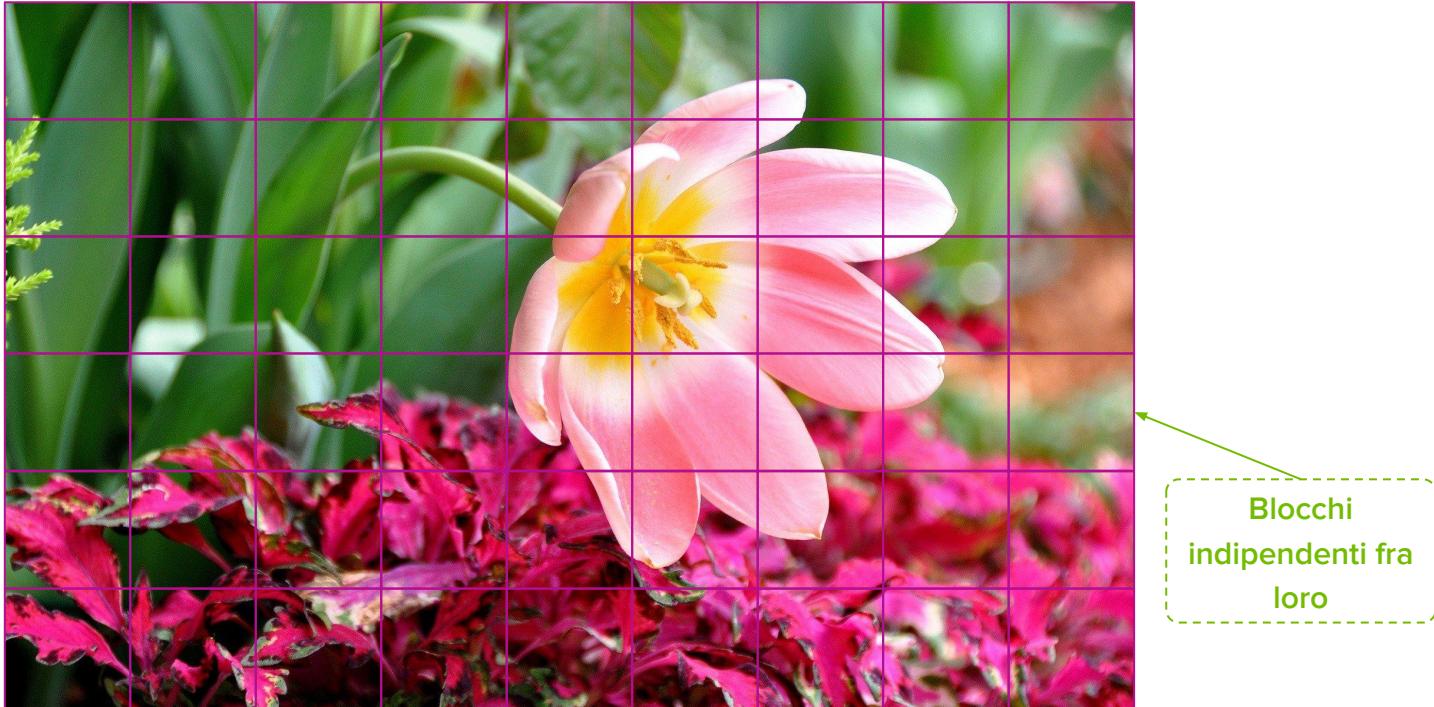
SM



GPU

- Un SM può contenere **più blocchi** di thread contemporaneamente.
- Ogni SM ha un **limite massimo** di thread block gestibili, determinato dalla sua compute capability.

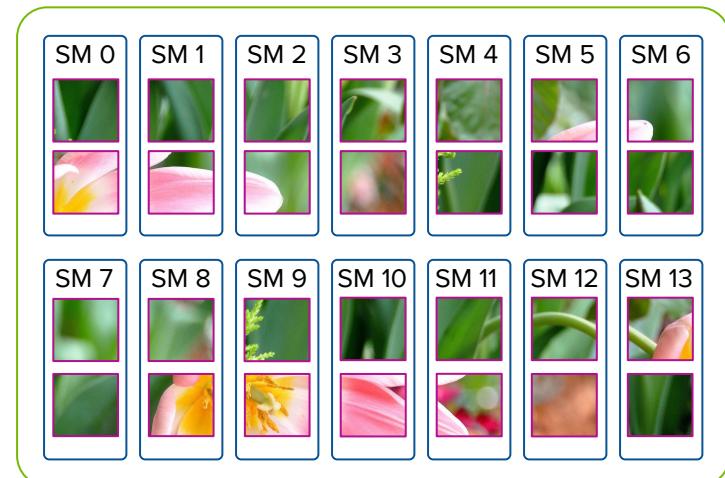
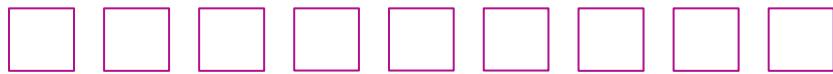
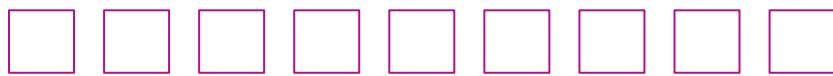
Distribuzione dei Blocchi su Streaming Multiprocessors



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

Distribuzione dei Blocchi su Streaming Multiprocessors

- 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**.



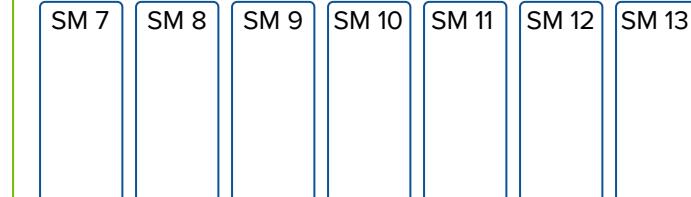
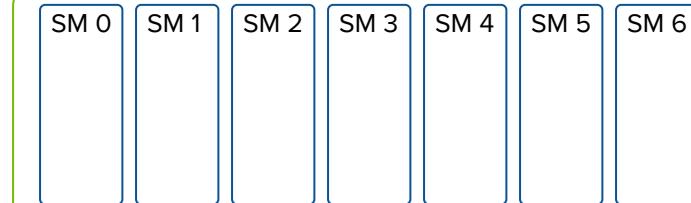
Capacità Massima Raggiunta

Distribuzione dei Blocchi su Streaming Multiprocessors

- 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.



GPU

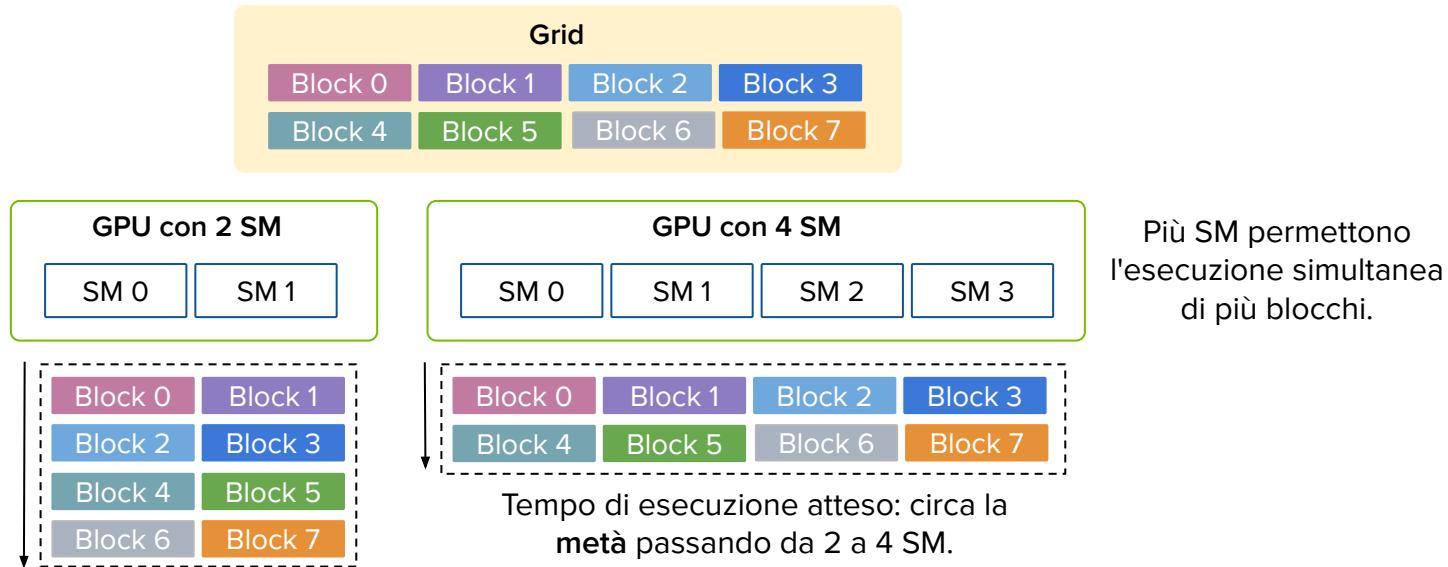


Spazio di Esecuzione Disponibile

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.



Panoramica del Modello di Esecuzione CUDA

➤ Architettura Hardware GPU

- Introduzione al Modello di Esecuzione CUDA
- Organizzazione degli Streaming Multiprocessors (SM)
- Panoramica delle Architetture GPU NVIDIA

➤ Organizzazione e Gestione dei Thread

- Mappatura tra Vista Logica e Hardware
- Distribuzione e Schedulazione dei Blocchi sui SM

➤ Modello di Esecuzione SIMT e Warp

- Confronto tra SIMD e SIMT
- Warp e Gestione dei Warp
- Latency Hiding e Legge di Little
- Warp Divergence e Thread Independent Scheduling

➤ Sincronizzazione e Comunicazione

- Meccanismi di Sincronizzazione
- Operazioni Atomiche

➤ Ottimizzazione delle Risorse

- Resource Partitioning
- Occupancy

➤ Parallelismo Avanzato

- CUDA Dynamic Parallelism

Modello di Esecuzione: SIMD

Approfondimenti nel Modulo 1

SIMD (Single Instruction, Multiple Data)

- È un **modello di esecuzione parallela** comunemente utilizzato nelle CPU dove una singola istruzione opera simultaneamente su più elementi usando **unità di elaborazione vettoriale**.
- Utilizza **registri vettoriali** che possono contenere più elementi (es. 4 float, 8 int16, 16 byte).
- Il programma segue un **flusso di controllo centralizzato** (un unico thread).
- **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 eseguiti insieme in un **gruppo sincrono unificato**.
 - **Divergenza non è ammessa** in SIMD; se sono richieste condizioni (**if-else**), si usano **maschere esplicite** che indicano su quali elementi il calcolo deve essere eseguito.

Somma di Due Array (SIMD)

```
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);  
    } }  
    // I dati vengono suddivisi in vettori di  
    // dimensione fissa e il loop elabora  
    // questi vettori utilizzando istruzioni  
    // intrinsics con nomenclatura specifica  
    // dell'architettura.
```

Modello di Esecuzione: SIMT

SIMT (Single Instruction, Multiple Thread)

- **Modello ibrido** adottato in CUDA che combina parallelismo a livello di più thread con esecuzione tipo SIMD.
- **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).
- **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];  
}
```

Questo riferisce all'essenza del SIMT

Modello di Esecuzione: SIMT

SIMT (Single Instruction, Multiple Thread)

- M
- C

Perché 32 Thread in un Warp CUDA?

- **Efficienza Hardware:** Massimizza l'utilizzo delle risorse hardware.
 - 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 (lo vedremo).
- **Flessibilità Software:** Offre una granularità gestibile per il parallelismo e la divergenza dei thread (lo vedremo).
- **Adattabilità:** Questa dimensione si è dimostrata efficace per varie generazioni di GPU NVIDIA, pur rimanendo aperta a future evoluzioni.

Somm

_ glo

_ in

_ if

}

Questo rigo rappresenta l'essenza del SIMT

Modello di Esecuzione: SIMD vs. SIMT

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

*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 (verrà affrontato nelle prossime slide).

Modello di Esecuzione Gerarchico di CUDA

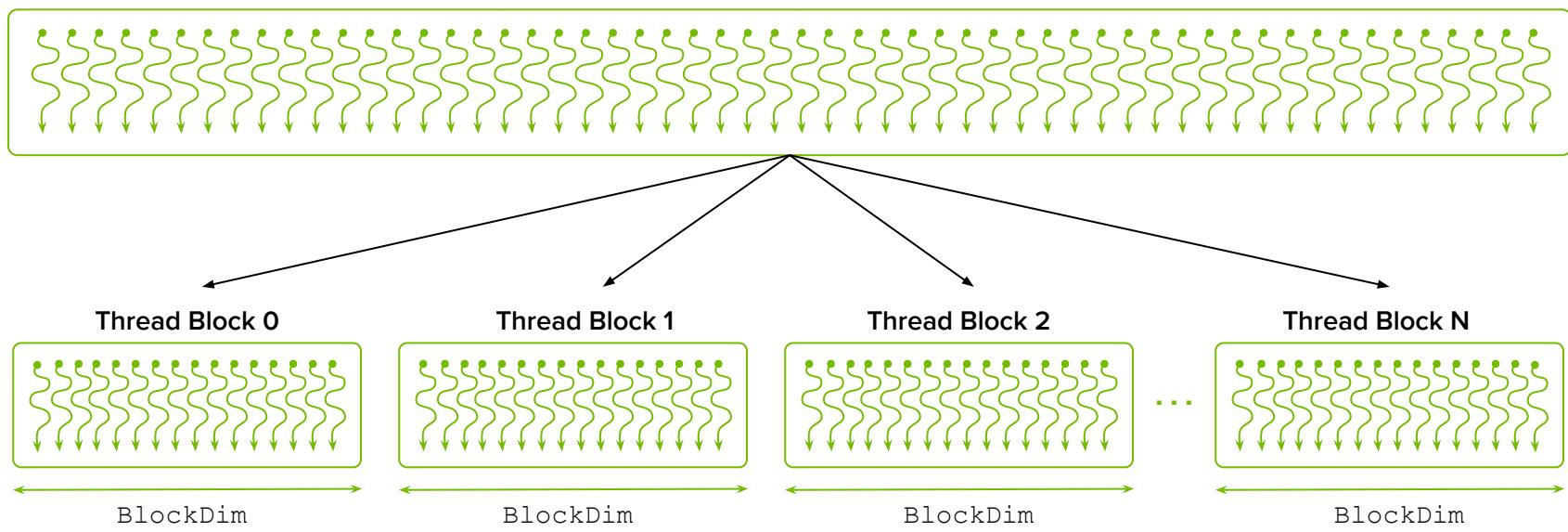
Livello di Programmazione

```
__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];  
}
```

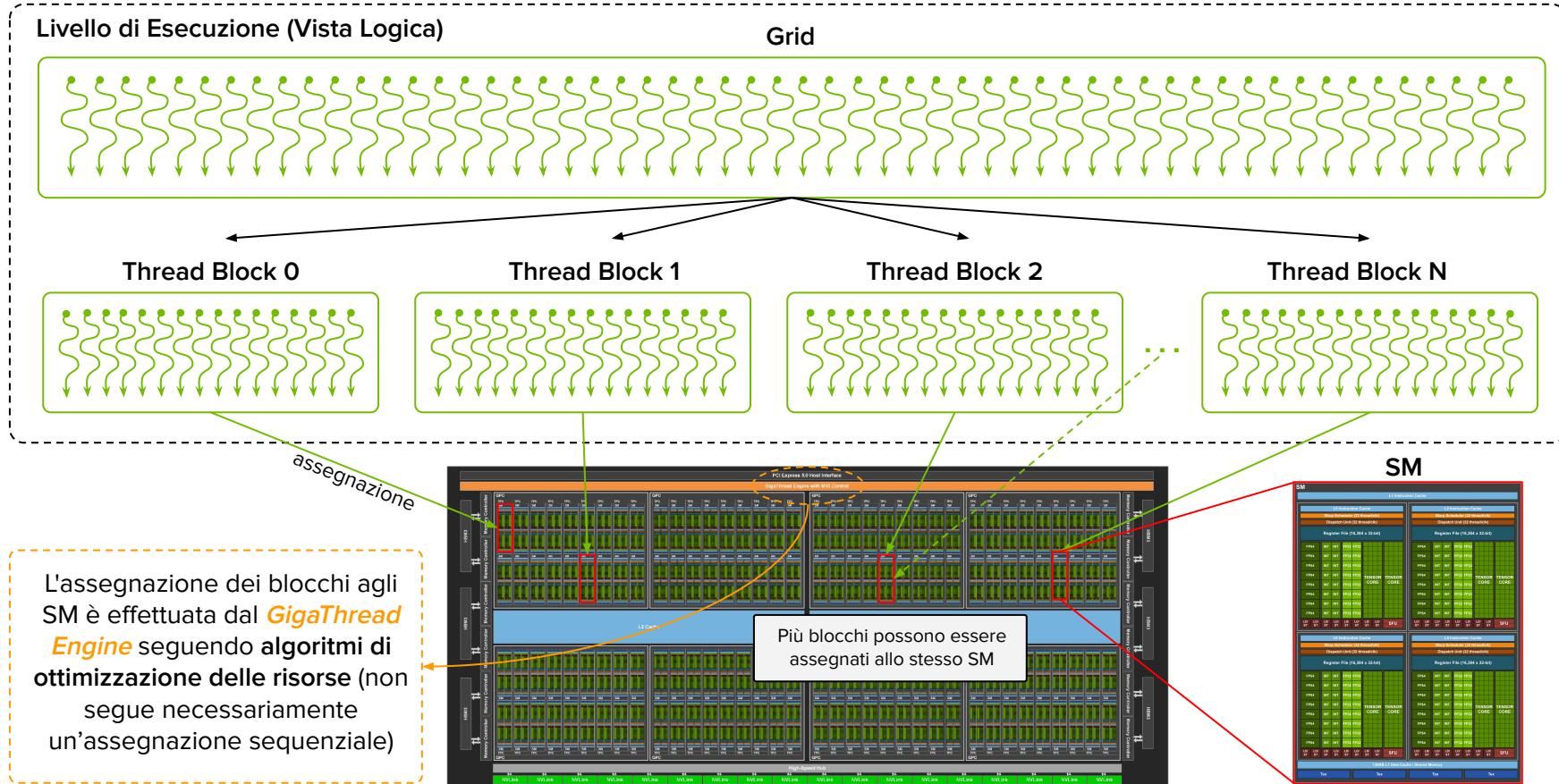
```
int main(int argc, char **argv) {  
    // ...  
    // Chiamata del kernel  
    array_sum<<<gridDim,blockDim>>>(args)  
};  
}
```

Livello di Esecuzione (Vista Logica)

Grid



Modello di Esecuzione Gerarchico di CUDA



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

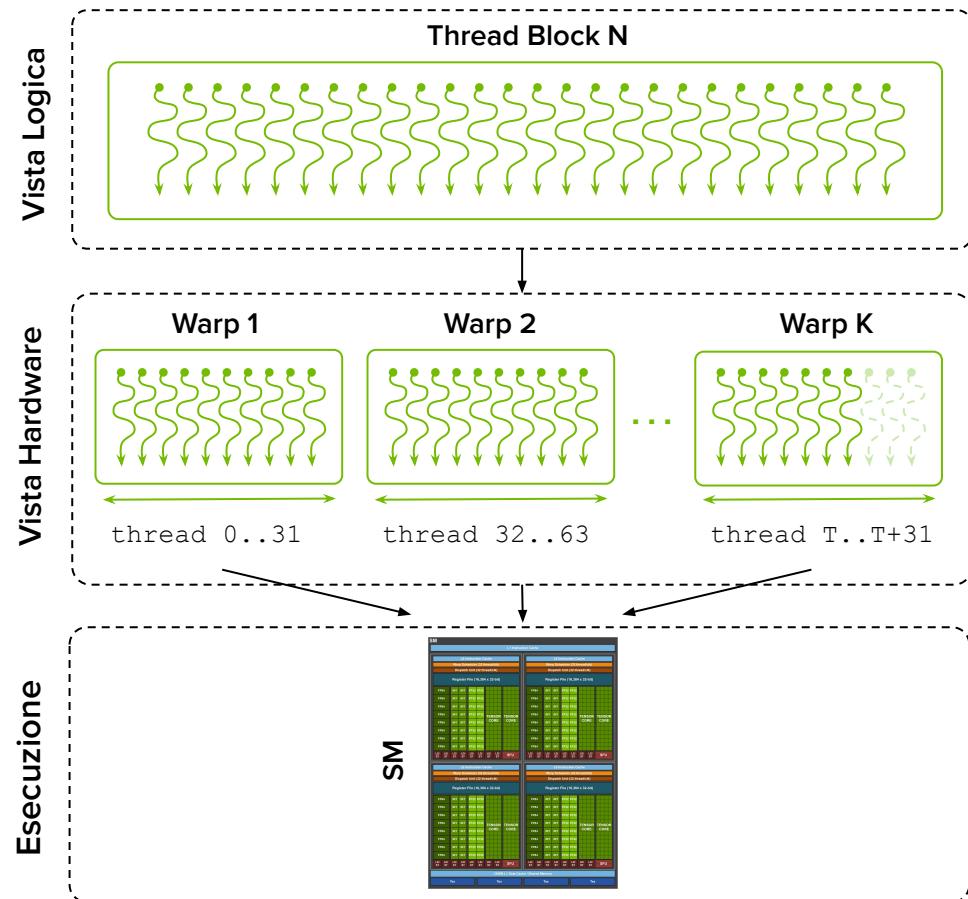
- Thread eseguiti in parallelo **logicamente**, ma non sempre fisicamente.

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.



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

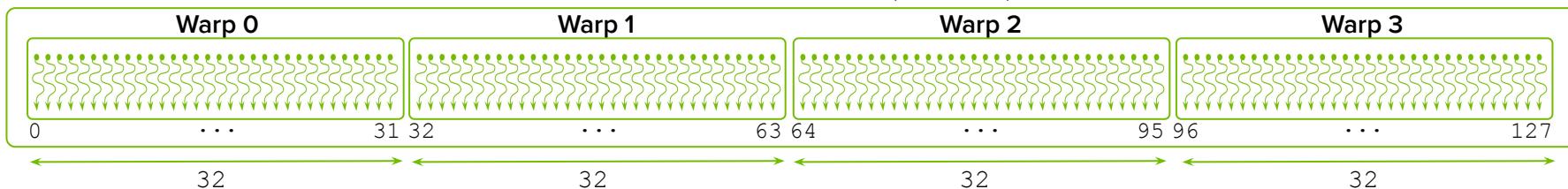
↓
`threadIdx.x`

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)



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.

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, 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: `threadIdx.y * blockDim.x + threadIdx.x`
 - Caso 3D: `threadIdx.z * blockDim.y * blockDim.x + threadIdx.y * blockDim.x + threadIdx.x`
 - Calcolo del Numero di Warp: `ceil(ThreadsPerBlock/warpSize)`
- L'hardware alloca sempre un numero **discreto** di warp.

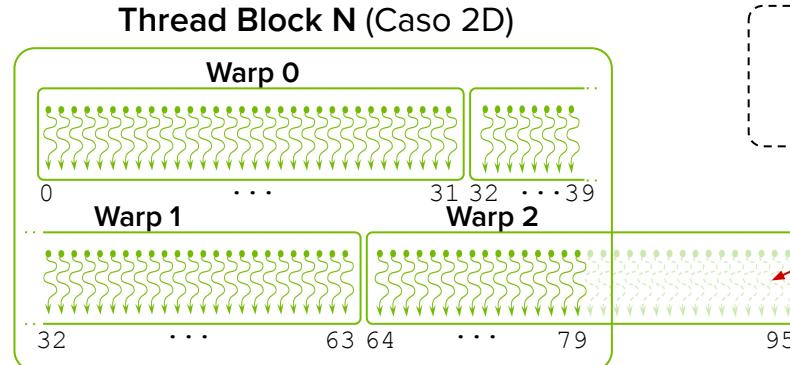
Organizzazione dei Thread e Warp

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.

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.



Thread ID nel Blocco

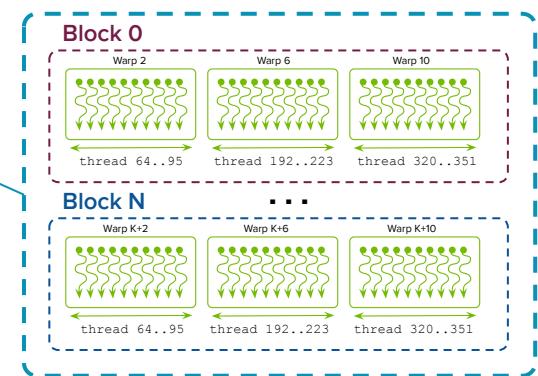
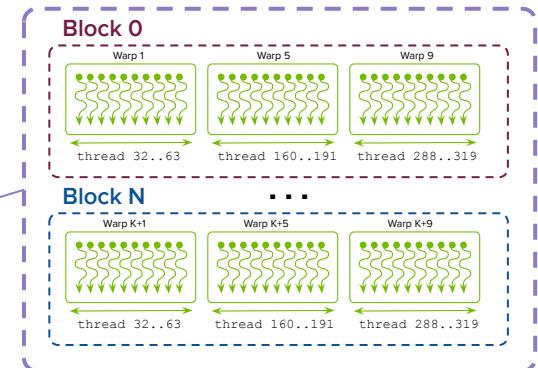
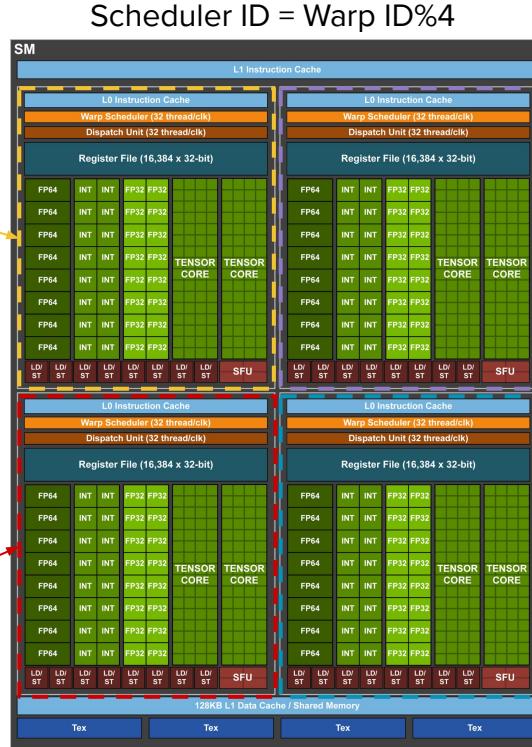
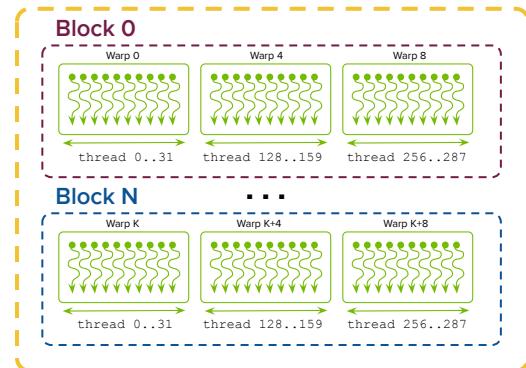
$$\text{threadIdx.y} * \text{blockDim.x} + \text{threadIdx.x}$$

16 Thread nel warp inattivo
(Soluzione da evitare)

Best Practice: Scegliere blocchi multipli di 32 per evitare warp parziali e migliorare l'uso delle 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).



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.

*Valori concorrenti per singolo SM

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

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

Warp: Contesto di Esecuzione

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



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];}
```

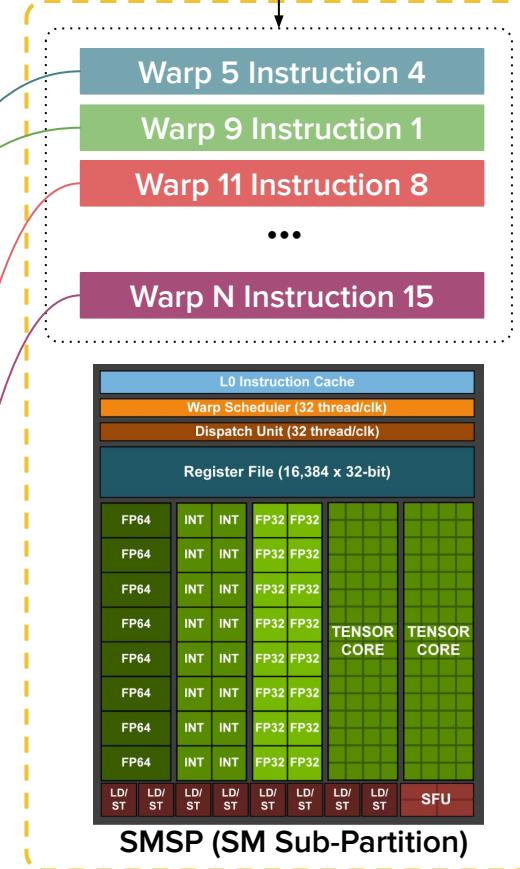
Compilazione (Codice comune a tutti i thread della grid)

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] • PC*  
EXIT // Termina il kernel
```

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

I warp possono appartenere anche a blocchi differenti



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.

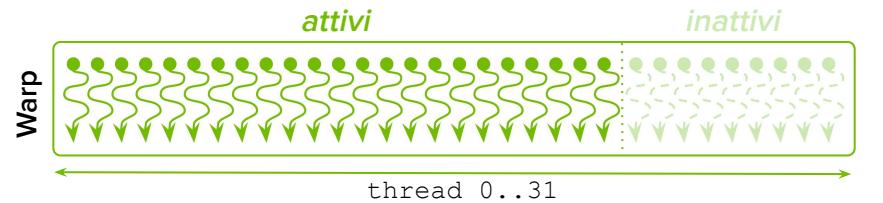
Classificazione degli Stati dei Thread

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).

Stati dei Thread

1. **Thread Attivo (Active Thread)**
 - Esegue l'istruzione corrente del warp.
 - 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.



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.

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).

Warp Scheduler e Dispatch Unit

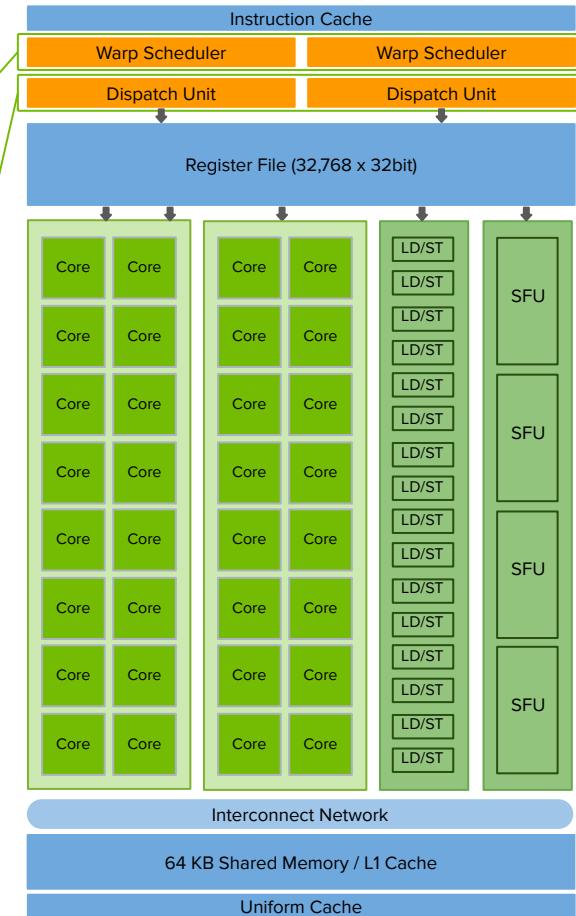
Warp Scheduler

- 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.

Fermi SM (2010)



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*).

Esecuzione Parallelia dei Warp - Esempio con Fermi SM

Componenti Chiave per il Parallelismo

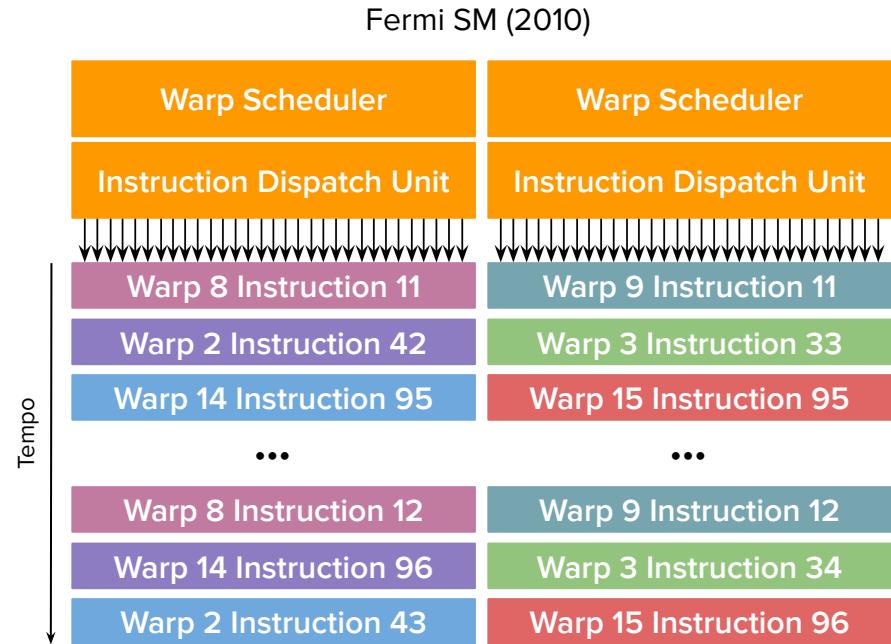
- **Due Scheduler di Warp:** Selezionano due warp pronti da eseguire dai thread block assegnati all'SM.
- **Due Unità di Dispatch delle Istruzioni:** Inviano le istruzioni dei warp selezionati alle unità di esecuzione.

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

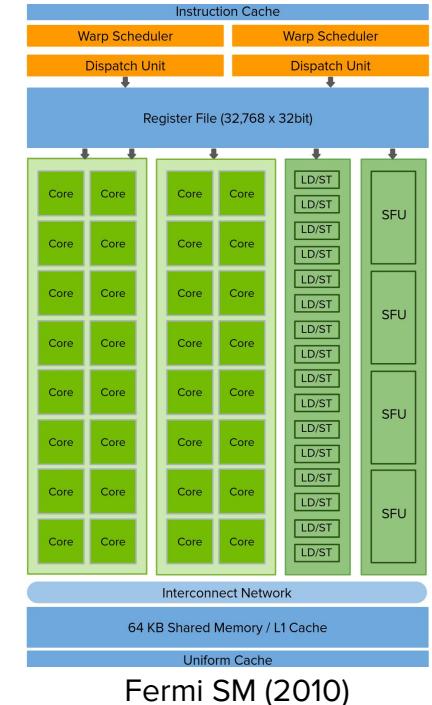
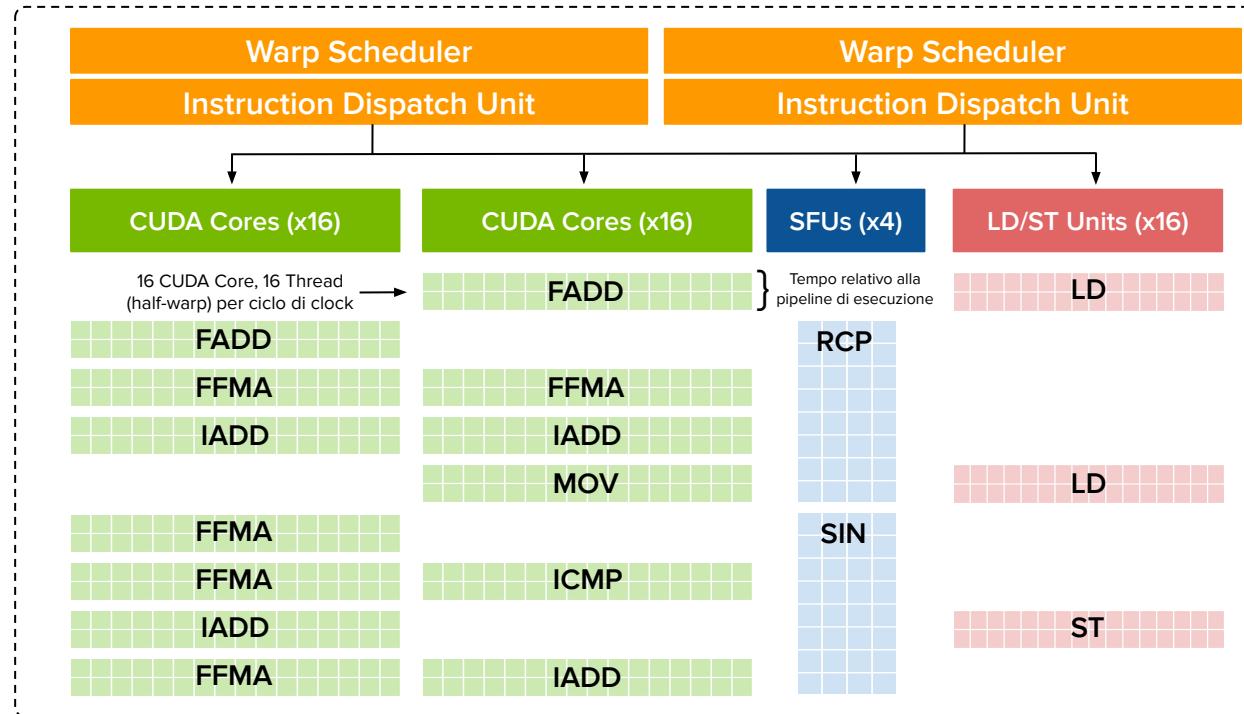
- Fermi (compute capability 2.x) può gestire simultaneamente 48 warp per SM, per un totale di **1.536 thread residenti** in un singolo SM. Ad ogni ciclo, al più **2 selected warps**.



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.

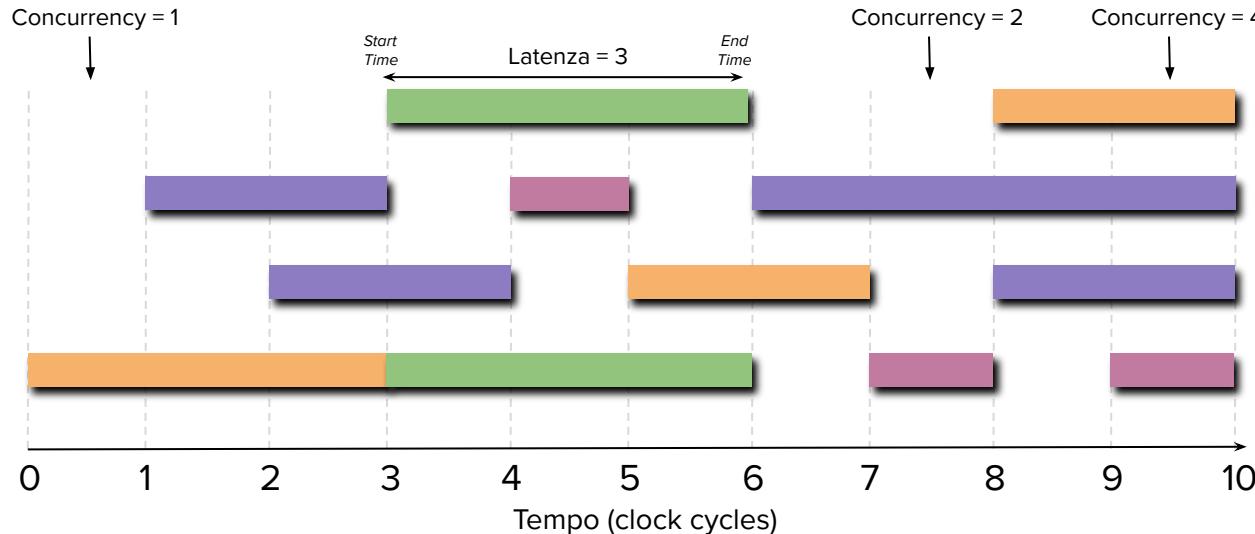
Scheduling Dinamico dell'Istruzioni - Fermi SM

- 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).



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.



$$\text{Mean Latency} = \frac{(3+3+1+1+2+2+2+2+1+4+3+2)}{12} = 2,17 \text{ cicli}$$
$$\text{Throughput} = \frac{12 \text{ elementi}}{10 \text{ cicli}} = 1,2 \text{ elementi per ciclo}$$
$$\text{Mean Concurrency} = \frac{\sum \text{Elementi/ciclo}}{\text{Num. Cicli}} = \frac{26}{10} = 2,6 \text{ elementi}$$

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).

Latency Hiding nelle GPU

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**.



Esempio di Esecuzione di Blocchi e Warp su un SM

Contesto

- **Esempio:** Esecuzione di un kernel con **1000 blocchi**, ciascuno composto da **128 thread**.

Fasi Principali

- **Allocazione ai SM**
 - Ogni SM può gestire simultaneamente un numero limitato di blocchi attivi (esempio: 8-12 blocchi)
 - All'interno di ogni blocco attivo, i thread sono suddivisi in warp (128 thread / 32 thread per warp = 4 warp attivi per blocco).
- **Warp Attivi per SM**
 - Se un SM ad esempio gestisce 12 blocchi, vuol dire che saranno presenti 48 warp attivi in esecuzione (12 blocchi x 4 warp per blocco) smistati poi fra i vari SM Sub-Partition (SMSP).
- **Funzionamento del Warp Scheduler**
 - I Warp Scheduler selezionano un warp eleggibile ad ogni ciclo di clock per l'esecuzione.
 - In presenza di un warp in stallo, gli scheduler assegnano un warp alternativo pronto all'esecuzione, garantendo la continuità operativa.

Nota

- **Dettagli trasparenti al programmatore:** La gestione dei warp e dei warp scheduler è **automatica**. Il programmatore deve solo garantire un elevato numero di warp in esecuzione per massimizzare l'efficienza.

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.

Legge di Little

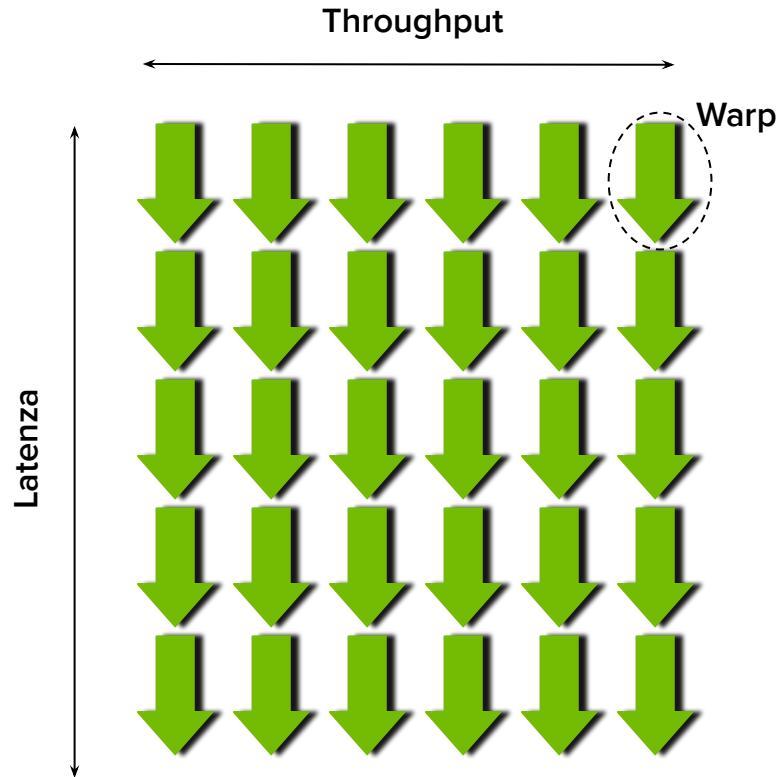
Esempio della Legge di Little

- 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)



Massimizzare il Parallelismo per Operazioni Aritmetiche

Architettura	Latenza Istruzione (Cicli)	Throughput (Operazioni/Ciclo)	Parallelismo Necessario (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 ($a + b \times c$)

Consideriamo una GPU con architettura Fermi:

Limite Warp/SM in Kepler è 64

- **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.

Massimizzare il Parallelismo per Operazioni di Memoria

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

Esempio: Operazione di Memoria

1/2

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: $92 \text{ B/ciclo} \times 800 \text{ cicli} \approx 74 \text{ KB di I/O in-flight per il pieno utilizzo delle risorse.}$
- **Interpretazione:**
 - 74 KB di operazioni di memoria necessarie per nascondere la latenza (per l'intero dispositivo, 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"
```

Massimizzare il Parallelismo per Operazioni di Memoria

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

Esempio: Operazione di Memoria

2/2

- 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 KB ÷ 4 bytes/thread = 18,500 thread
 - 18,500 thread ÷ 32 thread/warp = 579 warp
 - Per 16 SM: 579 warp ÷ 16 SM = 36 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.

Warp Divergence

Cosa è la Warp Divergence?

- In un warp, idealmente tutti i thread eseguono la **stessa istruzione contemporaneamente** per massimizzare il parallelismo SIMD (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**.

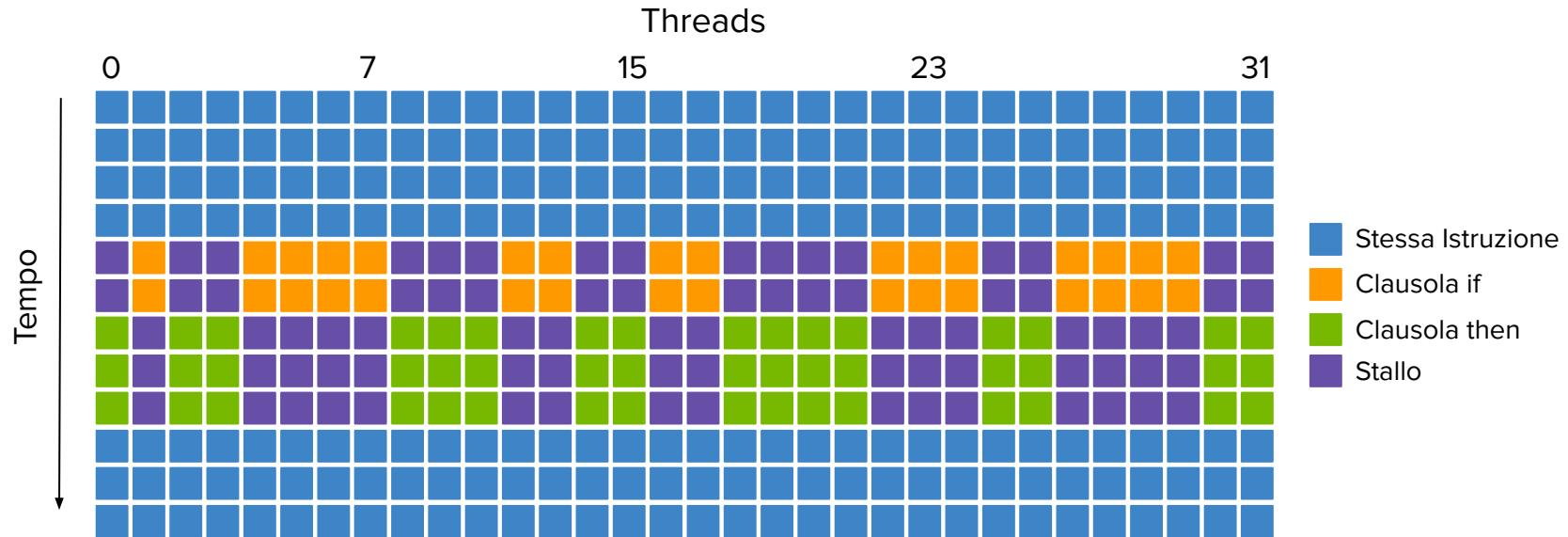
Esempio

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

CPU vs GPU: Gestione del Branching e della Warp Divergence

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

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).

Serializzazione nella Warp Divergence

Divergenza

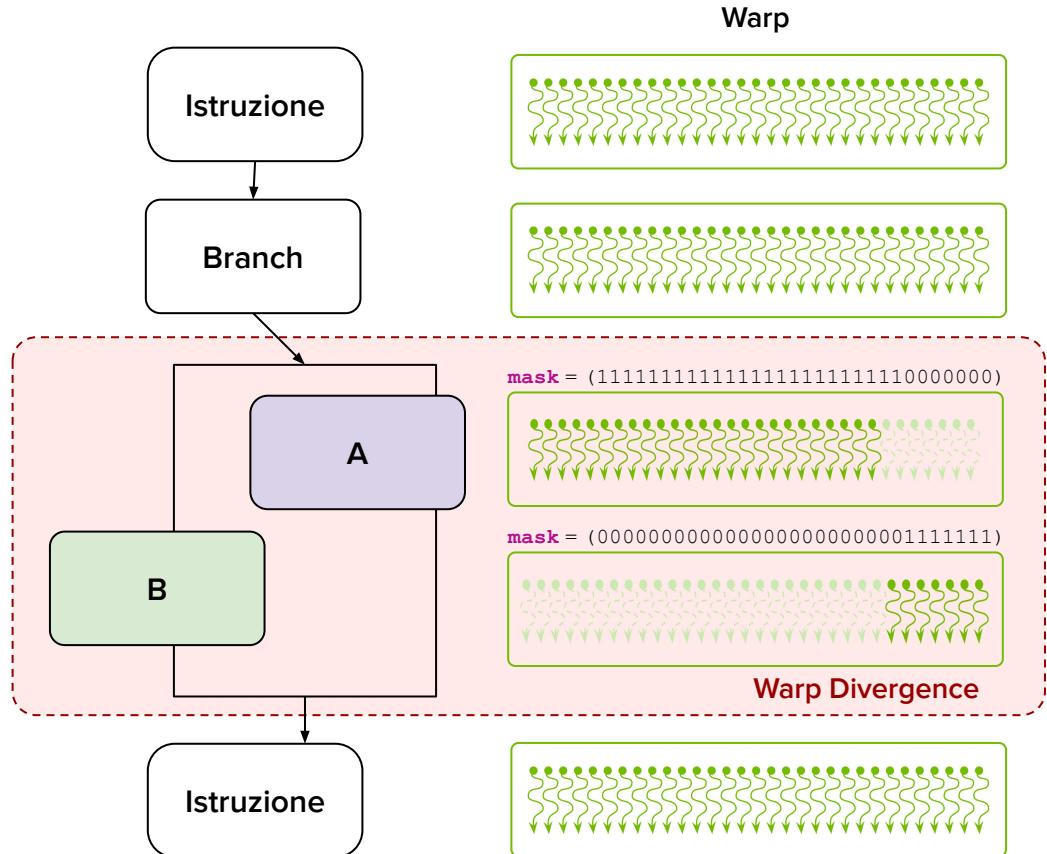
- Quando i thread di un warp seguono percorsi diversi a causa di istruzioni condizionali (es. `if`), il warp esegue ogni ramo **in serie**, disabilitando i thread inattivi.

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.



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 Peggior

```
__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.

Confronto delle Condizioni di Branch

Kernel 1

```
__global__ void mathKernel1(float *c) {
    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;
    c[tid] = a + b;}
```

Kernel 2

```
__global__ void mathKernel2(float *c) {
    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;
    c[tid] = a + b;}
```

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.

Confronto delle Condizioni di Branch

Kernel 1

```
__global__ void mathKernel1(float *c) {
    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;
    c[tid] = a + b;}
```

Kernel 2

```
__global__ void mathKernel2(float *c) {
    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;
    c[tid] = a + b;}
```

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%

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

Architetture Pre-Volta (< CC 7.0)

Pre-Volta

Program Counter (PC) and Stack (S)

32 Thread Warp

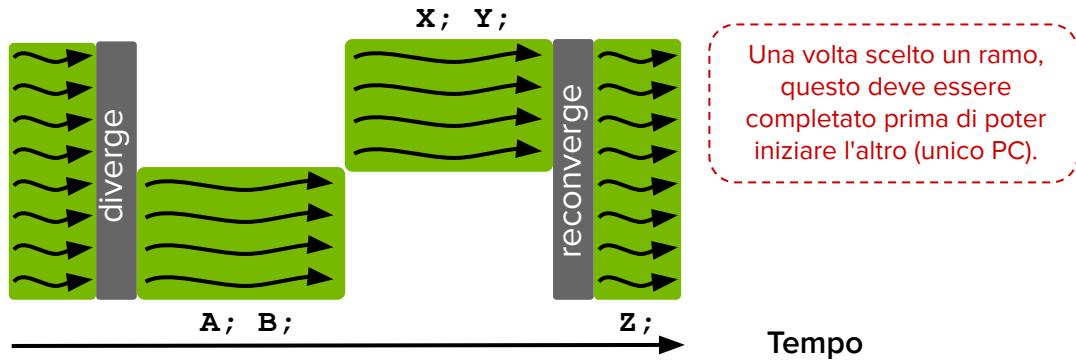
- Singolo Program Counter e Call Stack condiviso per tutti i 32 thread del warp (puntano alla stessa istruzione).
- Il warp agisce come una unità di esecuzione coesa/sincrona (stato dei thread è tracciato a livello di warp intero).
- Maschera di Attività (Active Mask) per specificare i thread attivi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

Limitazioni

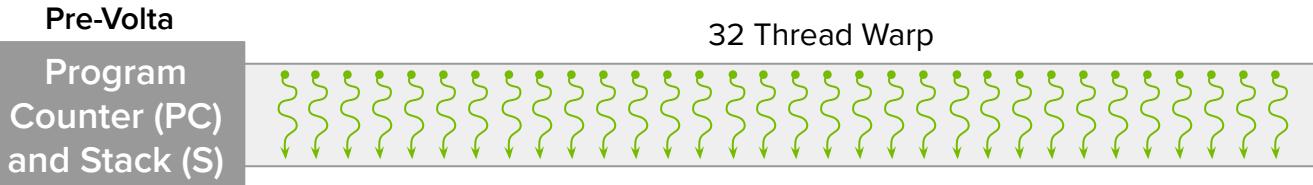
- Quando c'è divergenza, i thread che prendono branch diverse perdono concorrenza fino alla riconvergenza.
- Possibili deadlock tra thread in un warp, se i thread dipendono l'uno dall'altro in modo circolare.

Esempio di Divergenza (Pseudo-Code)

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



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.
- La maschera viene **salvata** fino alla riconvergenza del warp, poi **ripristinata** per riesecuzione sincrona.

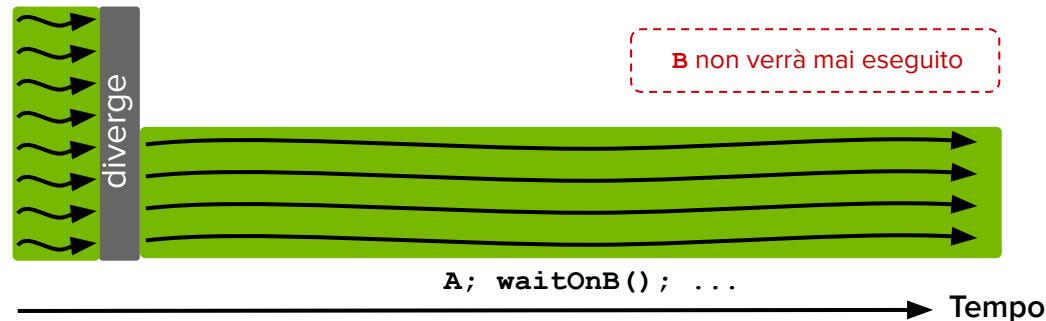
Limitazioni

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

Esempio di Potenziale Deadlock

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

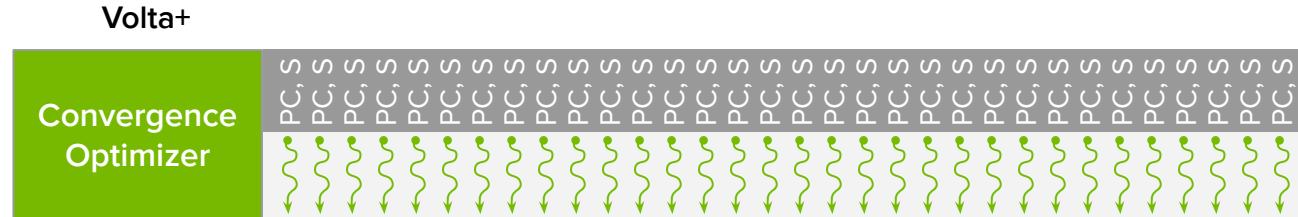
Thread divergenti dello stesso warp non possono comunicare.



Architettura Volta (CC 7.0+) e Independent Thread Scheduling

Concetto chiave

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

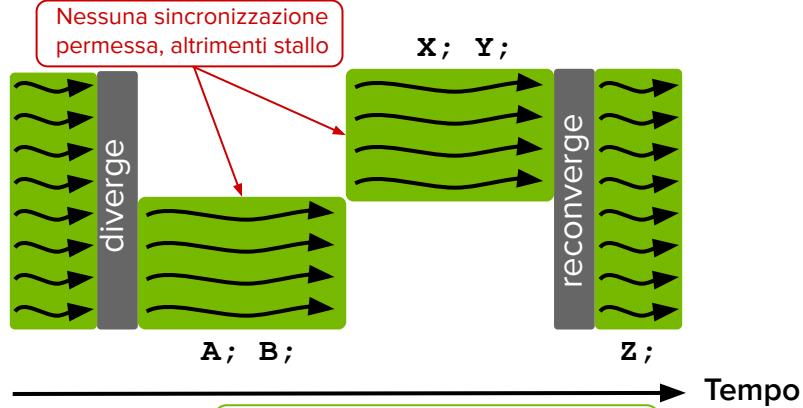


- Stato di Esecuzione per Thread**
 - Ogni thread mantiene il **proprio stato di esecuzione**, inclusi program counter e stack di chiamate.
 - Permette di cedere l'esecuzione a livello di **singolo thread**, migliorando l'uso delle risorse.
- Attesa per Dati**
 - Un thread può attendere che un altro 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 con **granularità sub-warp** (maggiore flessibilità).

Confronto Pre-Volta vs Post-Volta

Pre-Volta

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

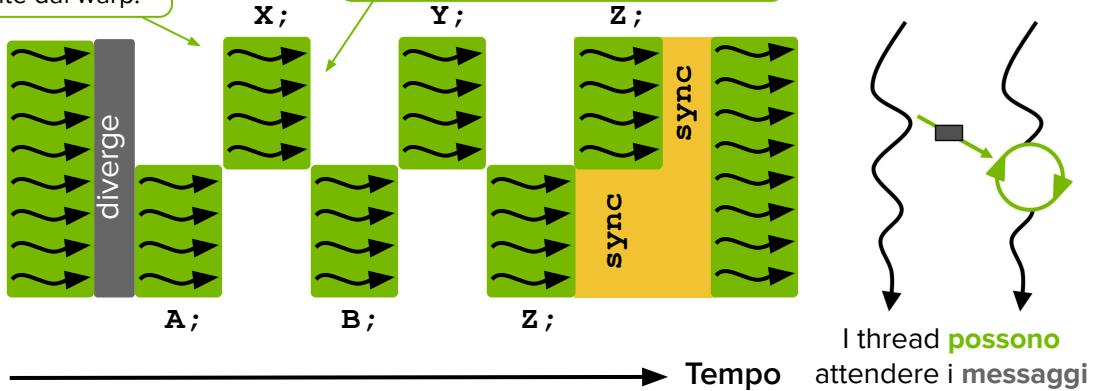


Post-Volta

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

Piena concorrenza tra i thread, indipendentemente dal warp.

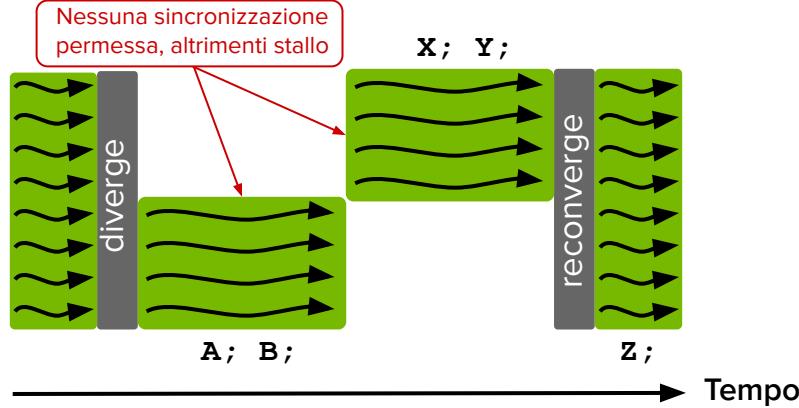
Esecuzione comunque SIMD ad ogni clock grazie all'ottimizzatore.



Confronto Pre-Volta vs Post-Volta

Pre-Volta

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

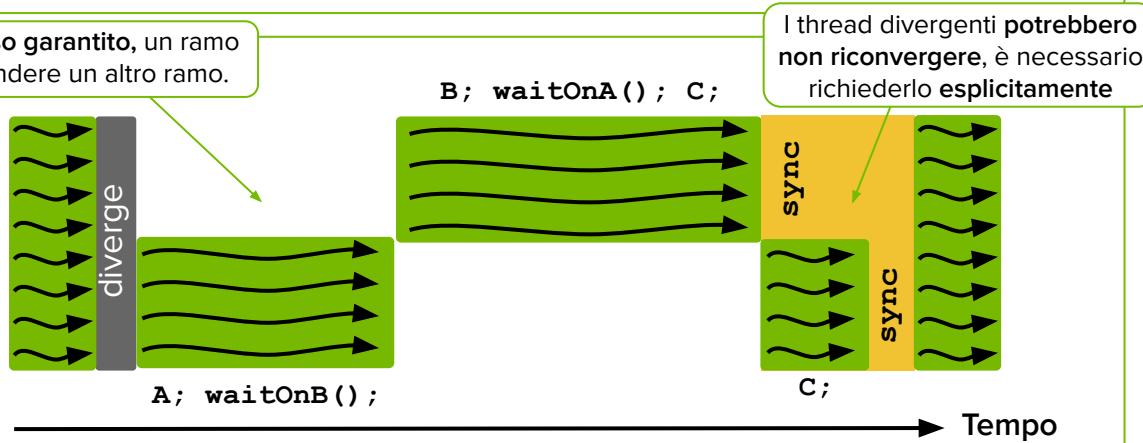


Post-Volta

Esempio di Potenziale Deadlock

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

Progresso garantito, un ramo può attendere un altro ramo.



Introduzione di `__syncwarp` in Volta

Scopo

- Introdotta dall'architettura Volta per supportare l'ITS e migliorare la gestione della divergenza dei thread.
- Sincronizza esplicitamente e riconverge 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);
```

Vantaggi

- Permette a un ramo di attendere un altro ramo all'interno del warp, garantendo che tutti i thread specificati siano sincronizzati prima di proseguire.
- Abilita l'esecuzione di algoritmi paralleli a grana fine in modo più sicuro.

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
```

Confronto Pre-Volta vs Post-Volta

	Pre-Volta	Post-Volta
Gestione dei Thread	Singolo program counter <u>per warp</u>	Program counter individuale <u>per thread</u>
Scheduling	A livello di warp	Indipendente per thread
Sincronizzazione	Implicita all'interno del warp	Richiede uso esplicito di <u><code>__syncwarp</code></u>
Divergenza	Serializzazione dei rami divergenti	Esecuzione interlacciata possibile
Deadlock Intra-Warp	Possibili in certi scenari	Largamente mitigati
Prestazioni con Divergenza	Potenzialmente compromesse	Migliorate grazie a scheduling flessibile
Complessità del Codice	Workaround necessari per certi algoritmi	Implementazioni più naturali possibili

Confronto Pre-Volta vs Post-Volta

	Pre-Volta	Post-Volta
Gestione delle Condizioni Critiche	• 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).	• ITS introduce overhead per la gestione separata di program counter e call stack per ogni thread, aumentando la flessibilità ma richiedendo più risorse.
Schedulazione	• Il progresso è garantito solo per i warp residenti al momento. I thread rimarranno in <u>attesa infinita</u> se il loro progresso dipende da un warp che non lo è.	• Non garantisce la riconvergenza, quindi le assunzioni relative alla programmazione a livello di warp potrebbero non essere valide (usare esplicitamente <u><code>syncwarp</code></u>).
Sincronizzazione	• Bisogna prestare più attenzione per garantire il comportamento SIMD dei warp.	
Dive		
Deadlock		
Preservazione della Progressività		
Complessità del Codice	Workaround necessari per certi algoritmi	Implementazioni più naturali possibili

Panoramica del Modello di Esecuzione CUDA

➤ Architettura Hardware GPU

- Introduzione al Modello di Esecuzione CUDA
- Organizzazione degli Streaming Multiprocessors (SM)
- Panoramica delle Architetture GPU NVIDIA

➤ Organizzazione e Gestione dei Thread

- Mappatura tra Vista Logica e Hardware
- Distribuzione e Schedulazione dei Blocchi sui SM

➤ Modello di Esecuzione SIMT e Warp

- Confronto tra SIMD e SIMT
- Warp e Gestione dei Warp
- Latency Hiding e Legge di Little
- Warp Divergence e Thread Independent Scheduling

➤ Sincronizzazione e Comunicazione

- Meccanismi di Sincronizzazione
- Operazioni Atomiche

➤ Ottimizzazione delle Risorse

- Resource Partitioning
- Occupancy

➤ Parallelismo Avanzato

- CUDA Dynamic Parallelism

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**.

Race Condition (Hazard)

Cos'è?

- Una **race condition** si verifica quando più thread accedono **concorrentemente** e in modo **non sincronizzato** alla stessa locazione di memoria (shared memory/global memory), causando risultati imprevedibili ed errori.

Tipi di Race Condition

- **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.
- **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.

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.

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 memoria globale e condivisa tra i thread del blocco.
 - Firma:** `__device__ void __syncthreads(void); // riduce le prestazioni se usato troppo`
- Livello di Warp (Disponibile a partire da CUDA 9.0 e architetture Volta+)**
 - Sincronizza i thread all'interno di un singolo warp.
 - Ottimizza le operazioni in un warp e permette comunicazioni efficienti tra thread dello stesso warp.
 - Firma:** `__device__ void __syncwarp(unsigned mask=0xffffffff);`

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.

Esempi

Livello di Sistema

```
__global__ void simpleKernel() {
    // Operazioni del kernel
}

int main() {
    ...
    simpleKernel<<<g, b>>>();
    cudaDeviceSynchronize();
    printf("Kernel completato\n");
    return 0;
}
```

Livello di Blocco

```
__global__ void blockSyncKernel()
{
    __shared__ int sharedData;

    if (threadIdx.x == 0) {
        sharedData = 42;
    }
    __syncthreads();
    if (threadIdx.x == 1) {
        printf("Valore condiviso:
               %d\n",
               sharedData);
    }
}
```

Livello di Warp

```
__global__ void warpSyncKernel()
{
    __shared__ int sharedData;

    if (threadIdx.x == 0)
        sharedData = 99;
    __syncwarp();

    if (threadIdx.x < 32)
        printf("Thread %d,
               valore: %d\n",
               threadIdx.x,
               sharedData);
}
```

Operazioni Atomiche in CUDA

Perché sono Necessarie le Operazioni Atomiche?

- **Problema:** Race conditions 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
}
```

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 altamente concorrente.

Operazioni Atomiche in CUDA

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

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

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.
- **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
}
```

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
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.

Panoramica del Modello di Esecuzione CUDA

➤ Architettura Hardware GPU

- Introduzione al Modello di Esecuzione CUDA
- Organizzazione degli Streaming Multiprocessors (SM)
- Panoramica delle Architetture GPU NVIDIA

➤ Organizzazione e Gestione dei Thread

- Mappatura tra Vista Logica e Hardware
- Distribuzione e Schedulazione dei Blocchi sui SM

➤ Modello di Esecuzione SIMT e Warp

- Confronto tra SIMD e SIMT
- Warp e Gestione dei Warp
- Latency Hiding e Legge di Little
- Warp Divergence e Thread Independent Scheduling

➤ Sincronizzazione e Comunicazione

- Meccanismi di Sincronizzazione
- Operazioni Atomiche

➤ Ottimizzazione delle Risorse

- Resource Partitioning
- Occupancy

➤ Parallelismo Avanzato

- CUDA Dynamic Parallelism

Resource Partitioning in CUDA

Cos'è il Resource Partitioning?

- 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 **ottimizzare la distribuzione delle risorse limitate** tra thread e blocchi, massimizzando l'occupazione degli SM e l'efficienza dell'esecuzione delle applicazioni 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 sono insufficienti per elaborare almeno un blocco di thread, il lancio del kernel **fallisce**.

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.

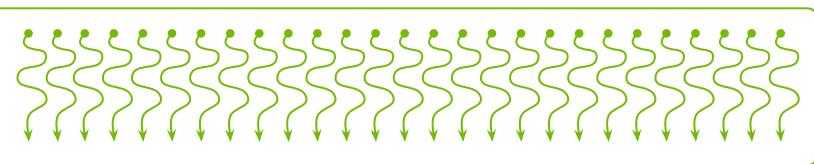
2. Memoria Condivisa

È comune a tutti i thread dello stesso blocco.

3. Registri

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

Thread Block

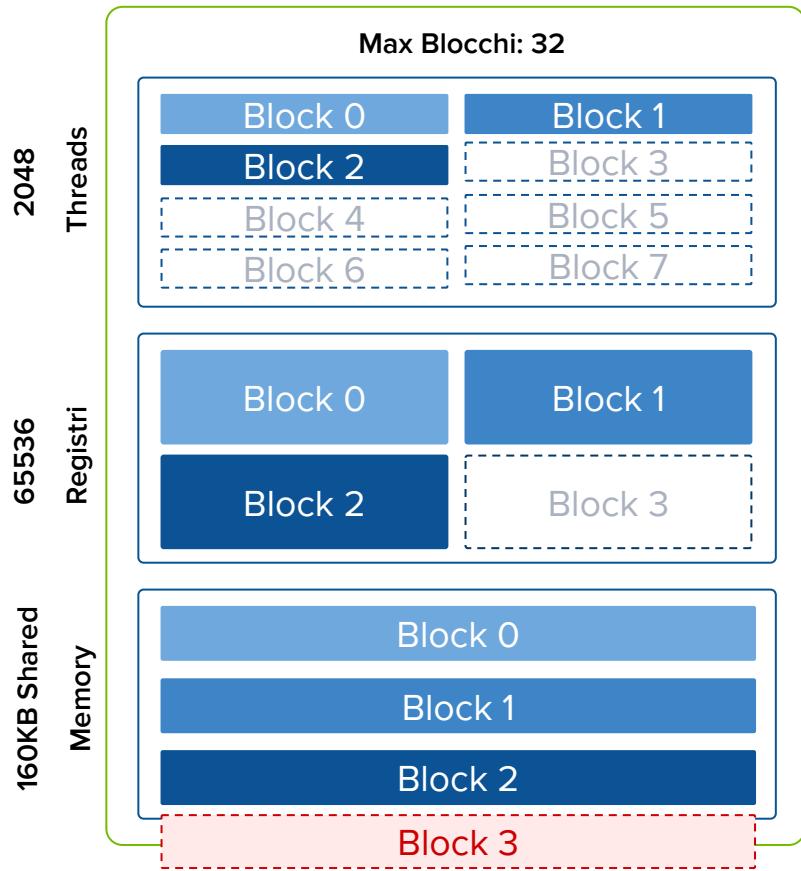


Un blocco mantiene un numero costante di thread ed esegue unicamente su un singolo SM

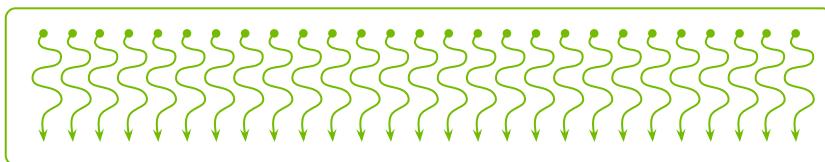
```
_global_ void simpleKernel(float* out) {  
    int tid = threadIdx.x;  
    float myValue = 3.14;  
  
    __shared__ float sharedData[64];  
  
    sharedData[tid] = myValue;  
    __syncthreads();  
  
    out[tid] = sharedData[tid] * myValue;  
}
```

Resource Partitioning in CUDA - Esempi

Risorse SM (Architettura Ampere)



Thread Block



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

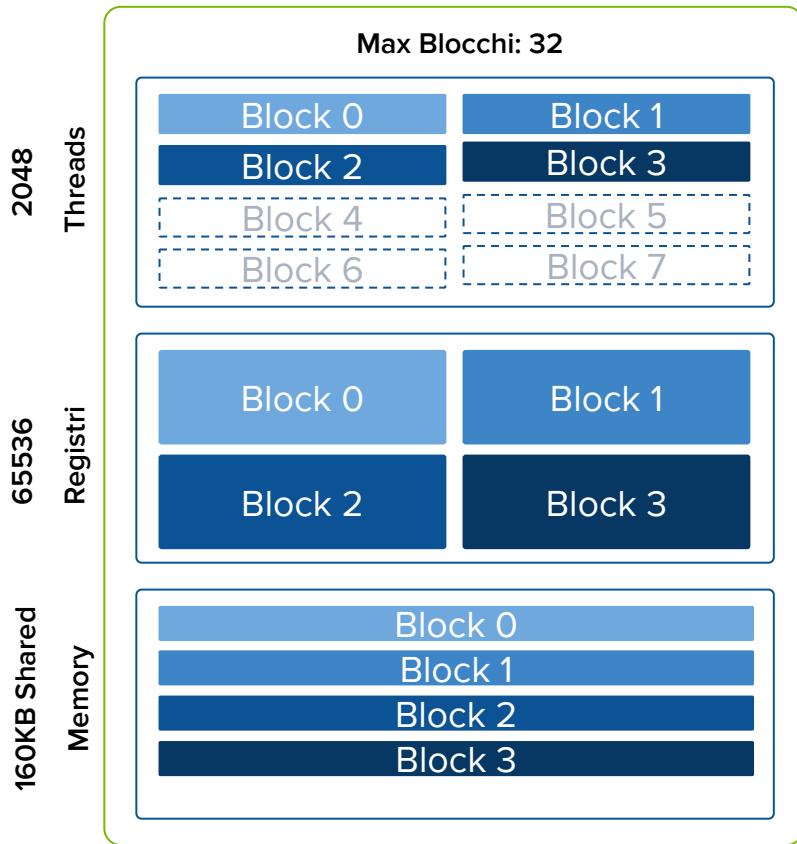
Registri per Thread 64

Registri per Blocco $(256 * 64) = 16384$

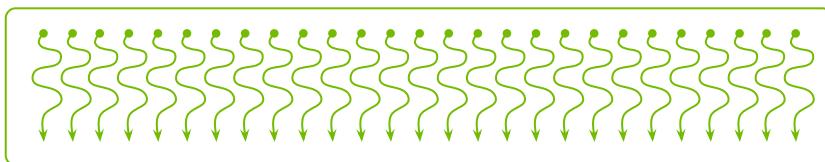
Shared Memory per Blocco 48Kb

Resource Partitioning in CUDA - Esempi

Risorse SM (Architettura Ampere)



Thread Block



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

Esempio di Requisiti di Risorse per i Blocchi

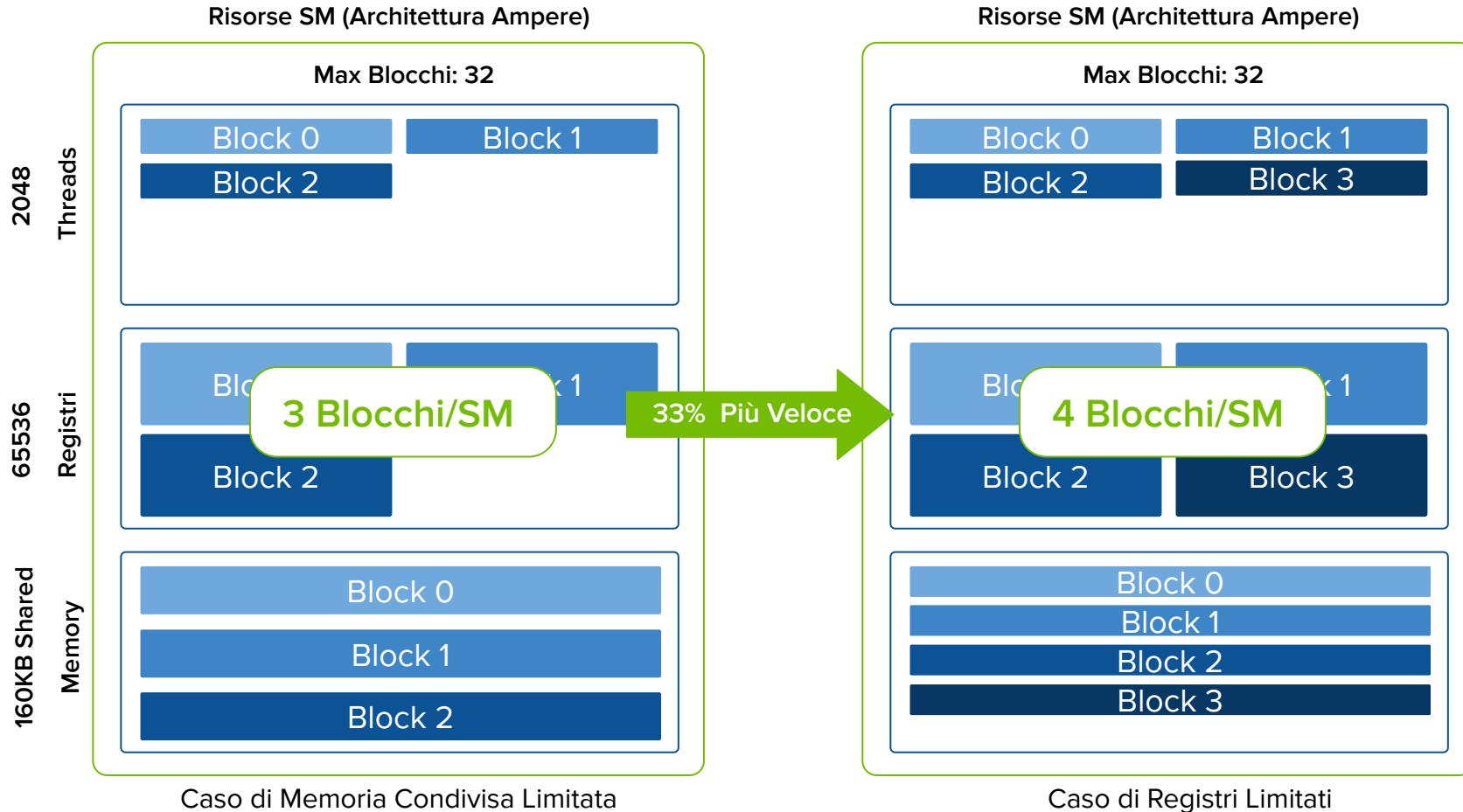
Thread per Blocco 256

Registri per Thread 64

Registri per Blocco $(256 * 64) = 16384$

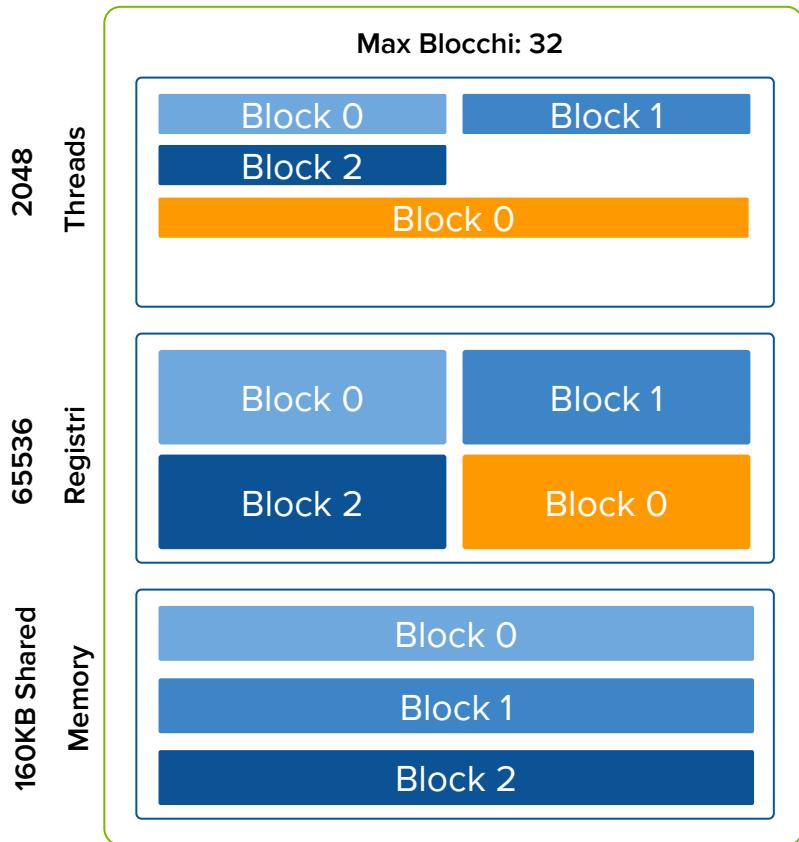
Shared Memory per Blocco 32Kb

Resource Partitioning in CUDA - Esempi



Resource Partitioning in CUDA - Esempi

Risorse SM (Architettura Ampere)



Esempio di Requisiti (Griglia Blu)

Thread per Blocco 256

Registri per Thread 64

Registri per Blocco $(256 * 64) = 16384$

Shared Memory per Blocco 48 Kb

Esempio di Requisiti (Griglia Arancione)

Thread per Blocco 512

Registri per Thread 32

Registri per Blocco $(512 * 32) = 16384$

Shared Memory per Blocco 0

Compute Capability (CC) - Limiti SM

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

Compute Capability	Architettura	Max Thread per Blocco	Max Thread per SM*	Max Warps per SM*	Max Blocchi per SM*	Max Registri per Thread	*Valori concorrenti per singolo SM Memoria Condivisa 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/Turing	1024	1024/2048	32/64	16/32	255	96KB
8.x	Ampere/Ada	1024	1536/2048	48/64	16/24	255	164KB
9.x	Hopper	1024	2048	64	32	255	228KB

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

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.
- **Suggerimento:** Osservare gli effetti sul tempo di esecuzione del kernel a diversi livelli di occupancy.

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 ha un limite superiore **determinato 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 = (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 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.

Occupancy Teorica vs Effettiva

Obiettivi di Ottimizzazione

- L'occupazione effettiva **non può** superare l'occupazione teorica.
- 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. Ciò può accadere nelle seguenti situazioni:
 - **Carico di lavoro sbilanciato nei blocchi:** Quando i warp all'interno di un blocco hanno tempi di esecuzione diversi, si crea un "**tail effect**" (effetto coda) 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:** Lanciare meno blocchi del massimo supportato abbassa l'occupazione. Ad esempio, una configurazione che richiederebbe 16 blocchi per il 100% di occupazione teorica scende al 75% con soli 12 blocchi lanciati.
 - **Wave Parziale:** Si verifica quando l'ultimo insieme di blocchi attivi non riesce a utilizzare completamente le capacità del dispositivo. Ad esempio, se un SM può supportare 64 warp attivi ma l'ultima ondata di esecuzione ha solo 48 warp attivi, l'occupazione effettiva sarà inferiore al massimo possibile.

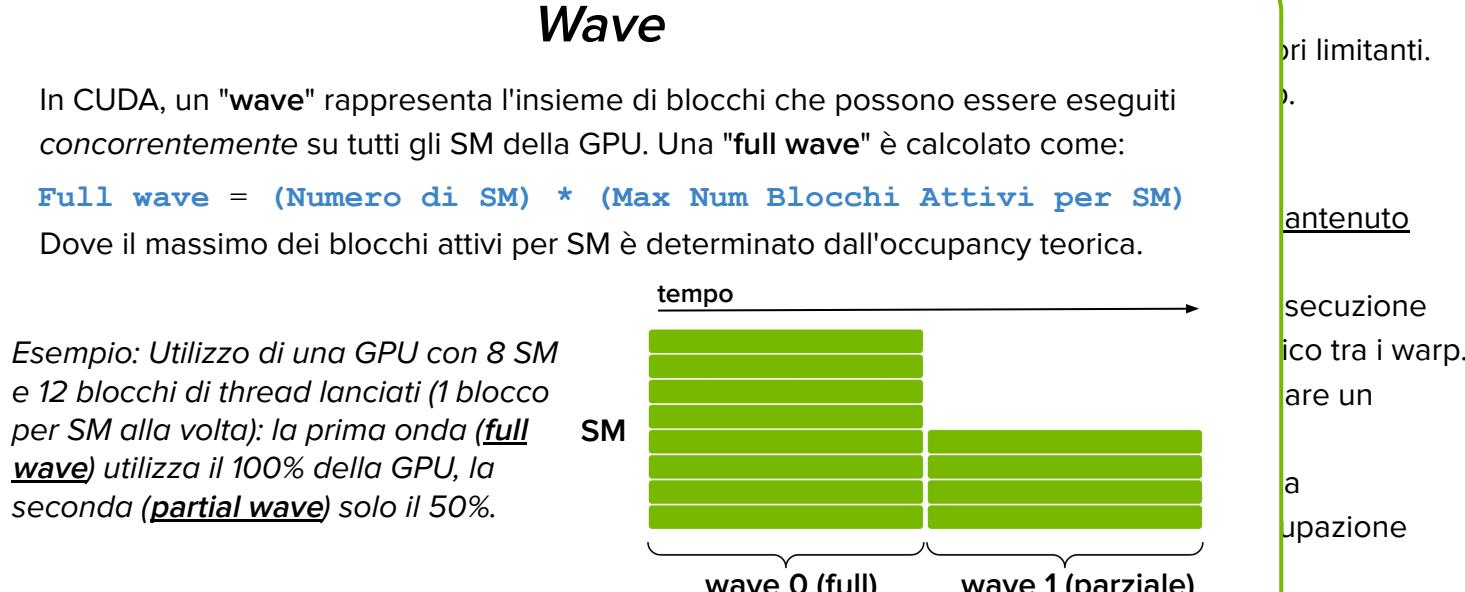
Occupancy Teorica vs Effettiva

Obiettivi di Ottimizzazione

- L'occupazione teorica
- Pertanto, l'occupazione effettiva è minima.
- Successivamente, si calcola la capacità massima di esecuzione.
- L'occupazione effettiva è minima.

Cause di Occupazione Minima

- o Durante l'esecuzione, i warp sono interrotti.

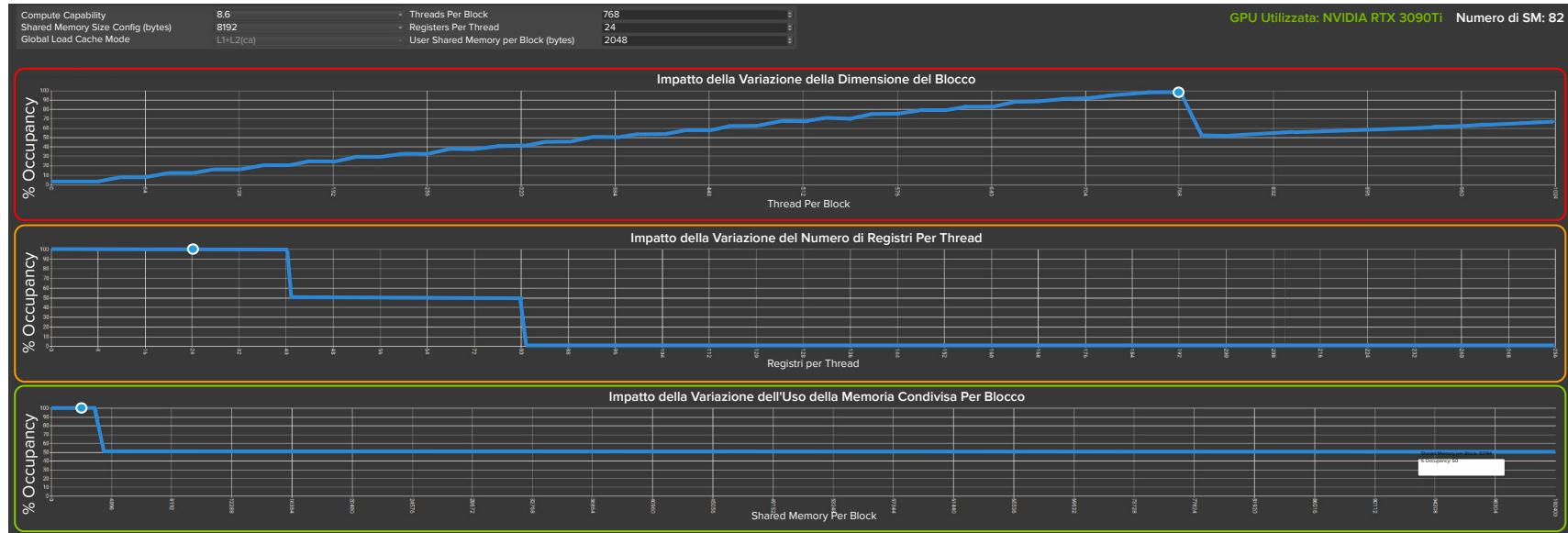


capacità del dispositivo. Ad esempio, se un SM può supportare 64 warp attivi ma l'ultima ondata di esecuzione ha solo 48 warp attivi, l'occupazione effettiva sarà inferiore al massimo possibile.

Nsight Compute: Occupancy Calculator

Nsight Compute offre uno strumento utile chiamato "Occupancy Calculator" ([Documentazione](#)) 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.



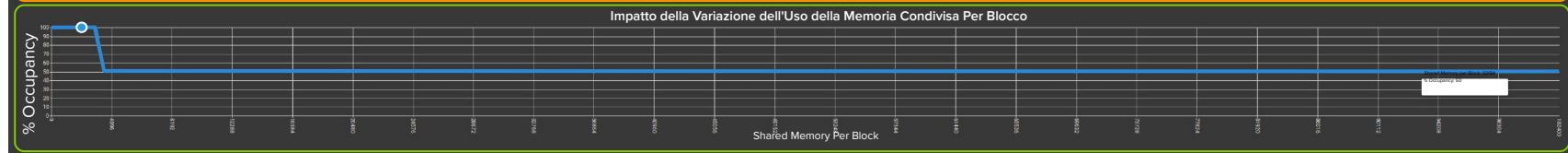
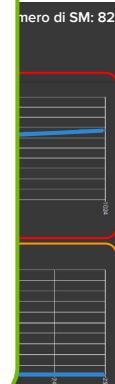
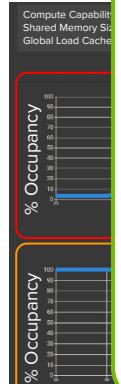
Nsight Compute: Occupancy Calculator

Nsight Compute offre uno strumento utile chiamato "Occupancy Calculator" ([Documentazione](#)) 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.
-

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.
- Mantenere 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.



Panoramica del Modello di Esecuzione CUDA

➤ Architettura Hardware GPU

- Introduzione al Modello di Esecuzione CUDA
- Organizzazione degli Streaming Multiprocessors (SM)
- Panoramica delle Architetture GPU NVIDIA

➤ Organizzazione e Gestione dei Thread

- Mappatura tra Vista Logica e Hardware
- Distribuzione e Schedulazione dei Blocchi sui SM

➤ Modello di Esecuzione SIMT e Warp

- Confronto tra SIMD e SIMT
- Warp e Gestione dei Warp
- Latency Hiding e Legge di Little
- Warp Divergence e Thread Independent Scheduling

➤ Sincronizzazione e Comunicazione

- Meccanismi di Sincronizzazione
- Operazioni Atomiche

➤ Ottimizzazione delle Risorse

- Resource Partitioning
- Occupancy

➤ Parallelismo Avanzato

- CUDA Dynamic Parallelism

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 dispositivi 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.

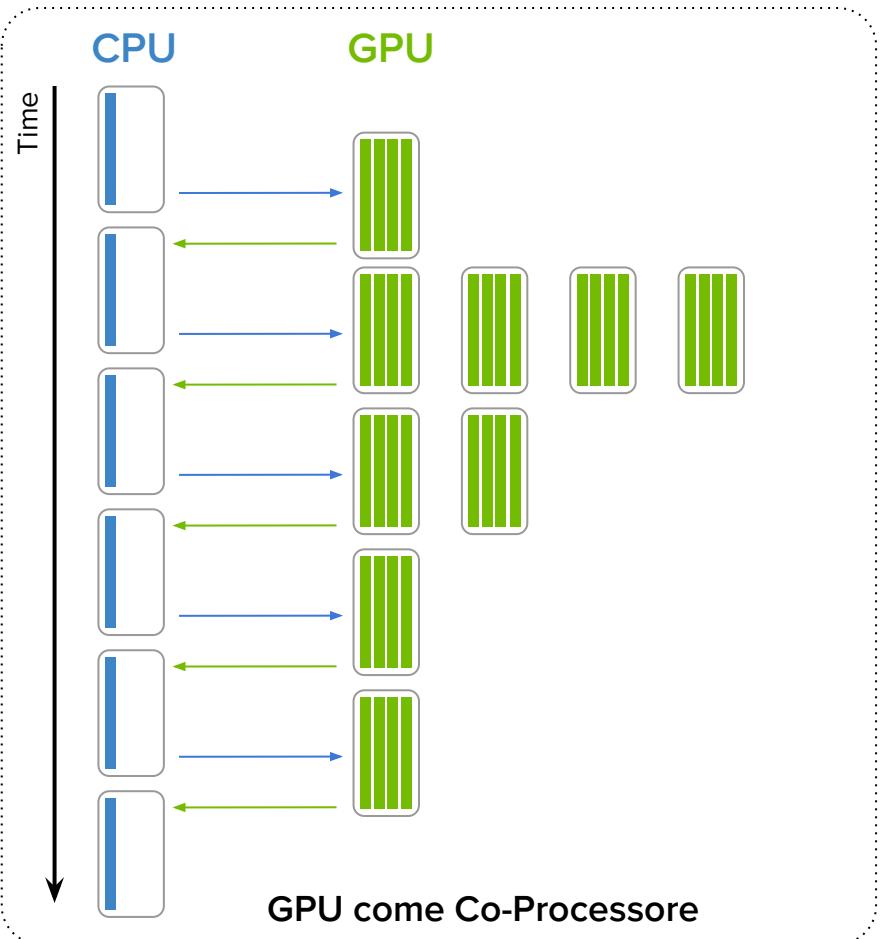
Possibili Applicazioni

- **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]

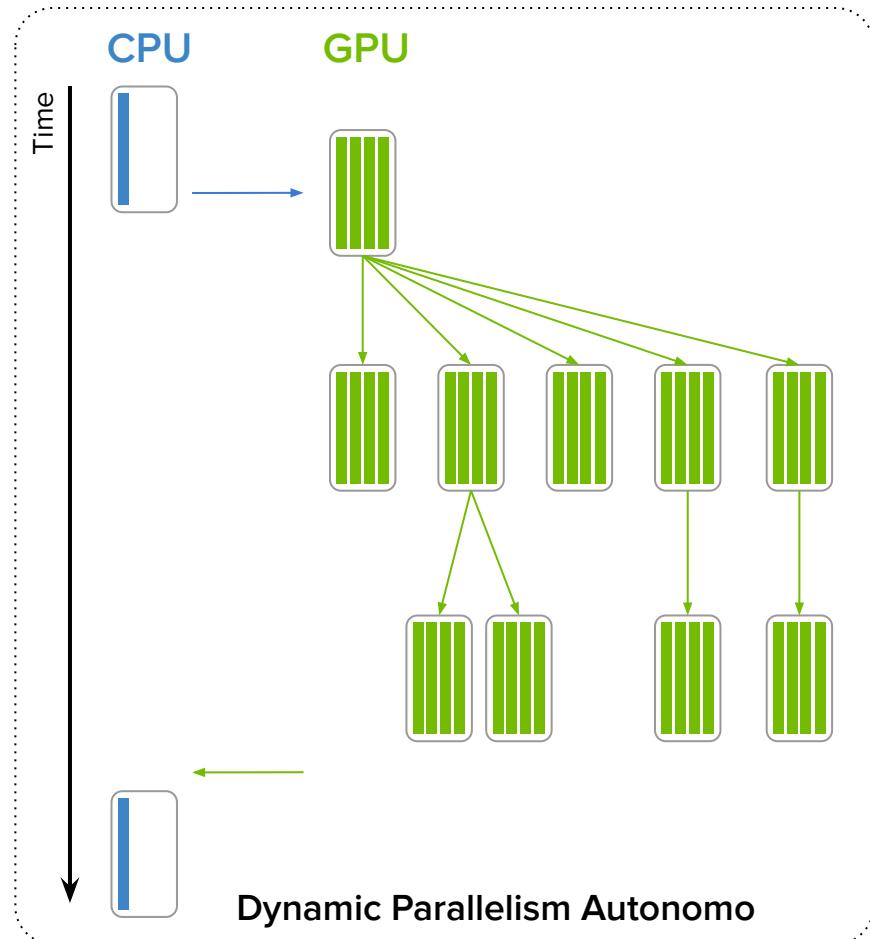
Vantaggi

- **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.

Introduzione al CUDA Dynamic Parallelism



GPU come Co-Processore



Dynamic Parallelism Autonomo

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**.

Introduzione al CUDA Dynamic Parallelism

```
// 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.

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 è 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):

- 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).

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).

Gestione dello Scambio Dati nel Parallelismo Dinamico

Quindi, Come Restituire un Valore da un Child Kernel?

Versione Errata

```
__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 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
    ...
}
```

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: ↓

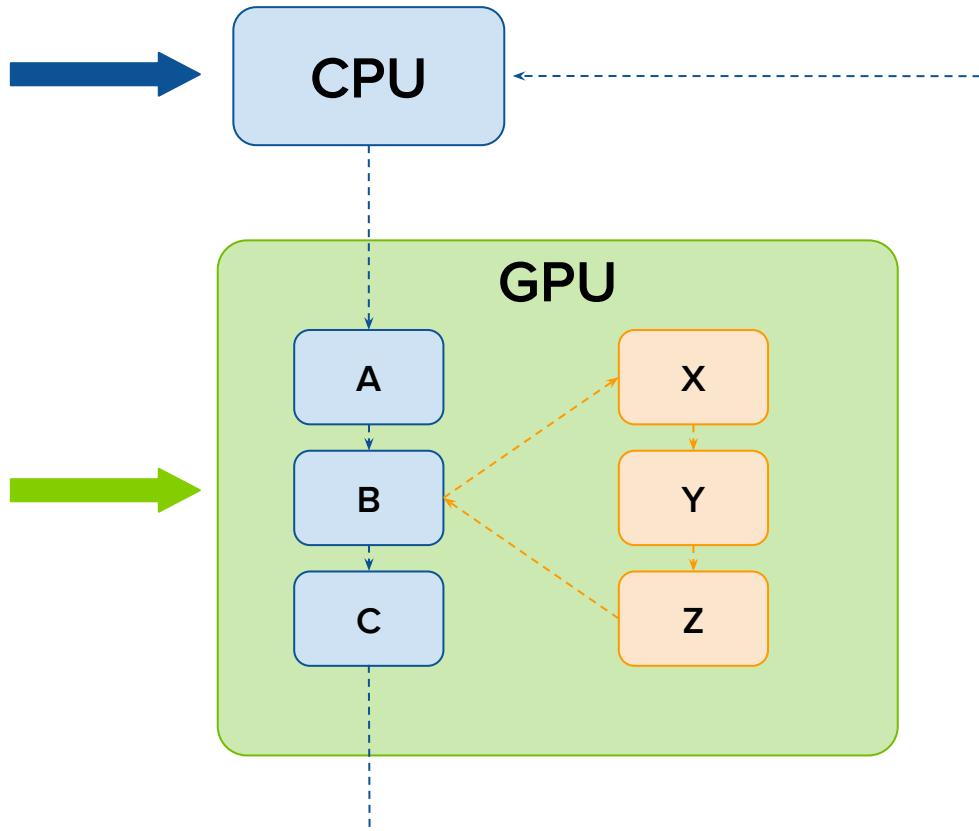
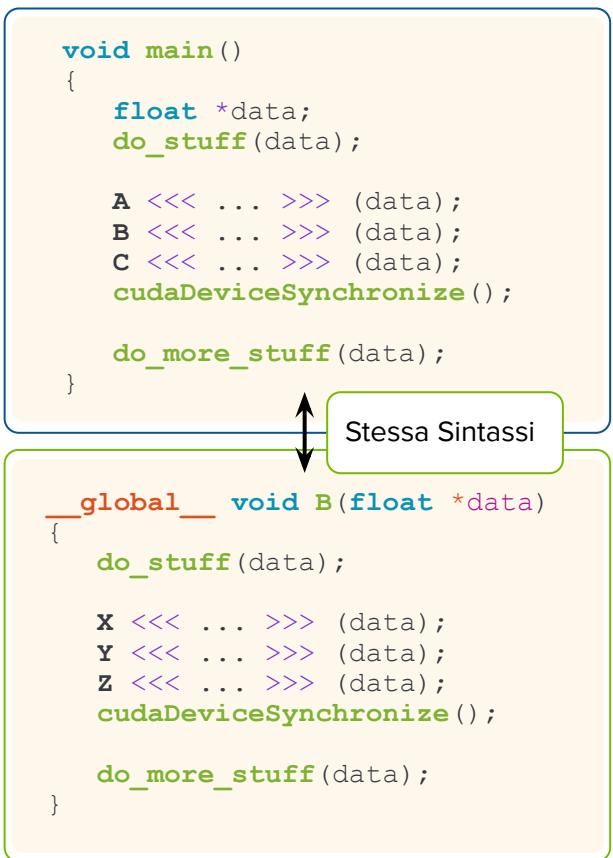
- **Scrittura da parte del child:**
 - Il thread parent potrebbe **non** vedere i valori scritti dal child.
- **Scrittura del parent dopo il lancio:**
 - Se il padre scrive dopo aver lanciato il figlio, si crea una "**race condition**".
 - Non si può sapere quale valore verrà letto.

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

Non c'è sincronizzazione esplicita

```
__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>>();  
}
```

Dipendenze Annidate in CUDA



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()`

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.
-

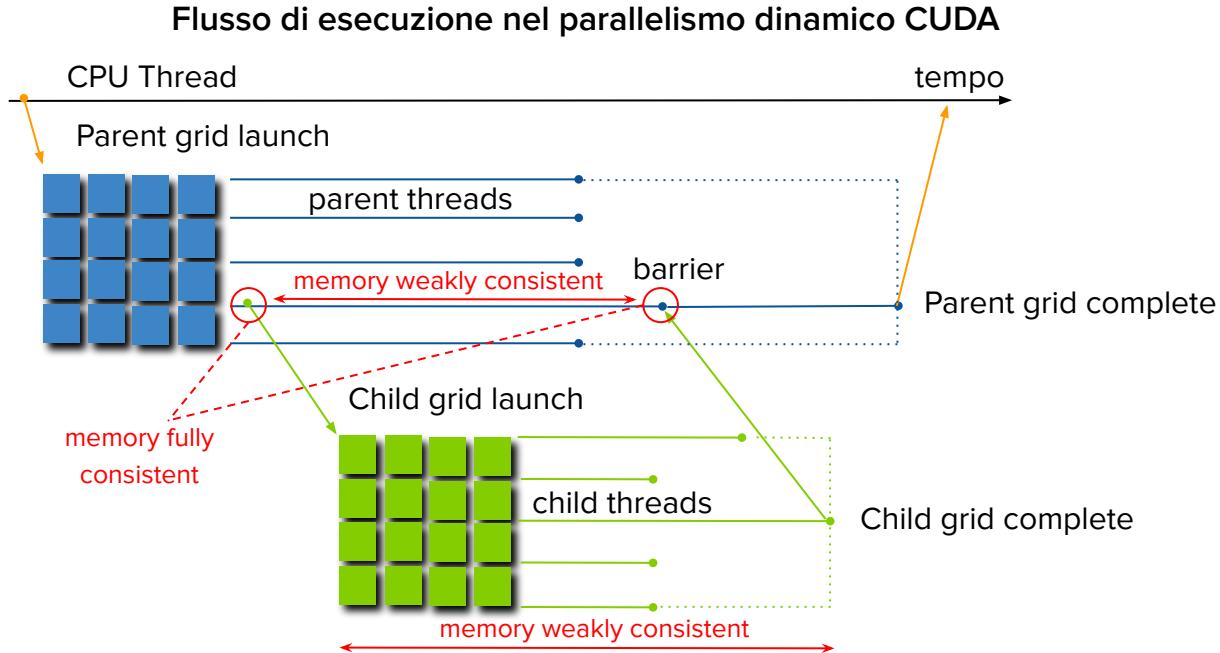
Sincronizzazione

- `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.

Limiti

```
    __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
}
```

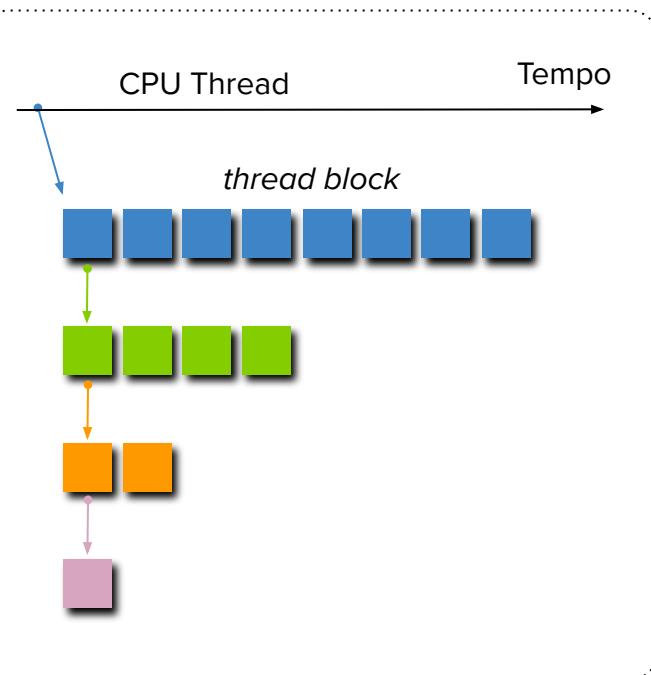
Esecuzione Nidificata con CUDA Dynamic Parallelism



- **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.

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 && nthreads > 0) {
        // Ricorsione
        nestedHelloWorld<<<1, nthreads>>>(nthreads, ++iDepth);
    }

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

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 libreria device runtime (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
```

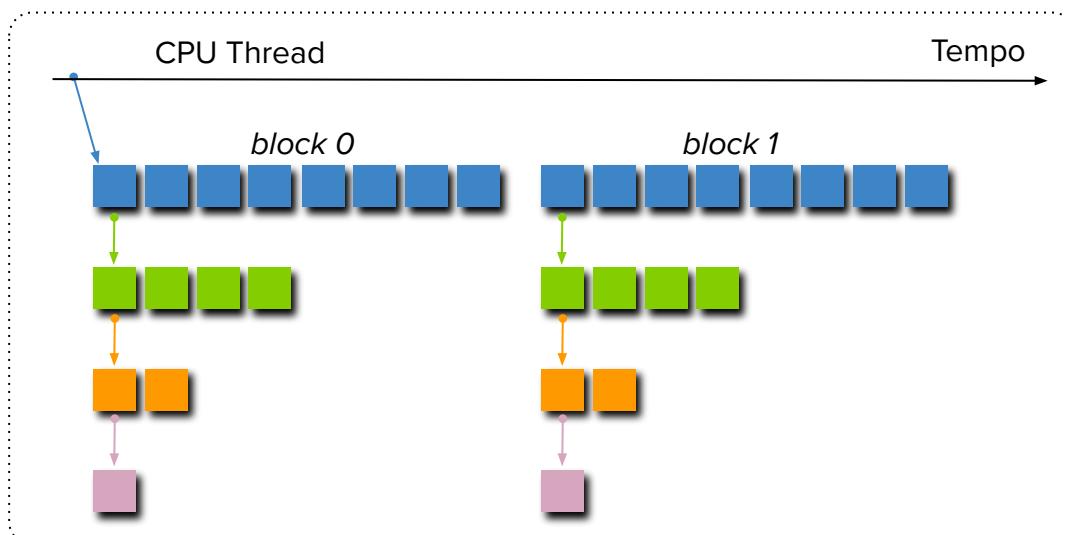
Nested Hello World : Compilazione ed Esecuzione

- 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)
```



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
```

Restrizioni sul Parallelismo Dinamico

Compatibilità dei Dispositivi

- Supportato solo da dispositivi con capacità di calcolo ≥ 3.5 .

Limitazioni di Lancio

- I kernel **non** possono essere lanciati su dispositivi **fisicamente separati**.

Profondità Massima di Nidificazione

- Limitata a **24** livelli
- Nella pratica, limitata dalla memoria richiesta dal **runtime** del dispositivo.
- 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.

Riferimenti Bibliografici

Testi Generali

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

NVIDIA Docs

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

Risorse Online

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