



# Modello di Memoria CUDA

Sistemi Digitali, Modulo 2

A.A. 2024/2025

Fabio Tosi, Università di Bologna

# Panoramica del Modello di Memoria CUDA

## ➤ Modelli di Performance

- Memory-Bound vs Compute-Bound
- Intensità Aritmetica e Roofline Model

## ➤ Gerarchia di Memoria CUDA

- Organizzazione Gerarchica Completa
- Scope e Programmabilità

## ➤ Gestione della Memoria Host-Device

- Allocazione e Trasferimenti
- Pinned Memory
- Zero-Copy Memory
- UVA (Unified Virtual Addressing)
- Unified Memory (UM)

## ➤ Global Memory

- Pattern di Accesso
- Lettura Cached vs Uncached
- Scrittura

## ➤ Shared Memory

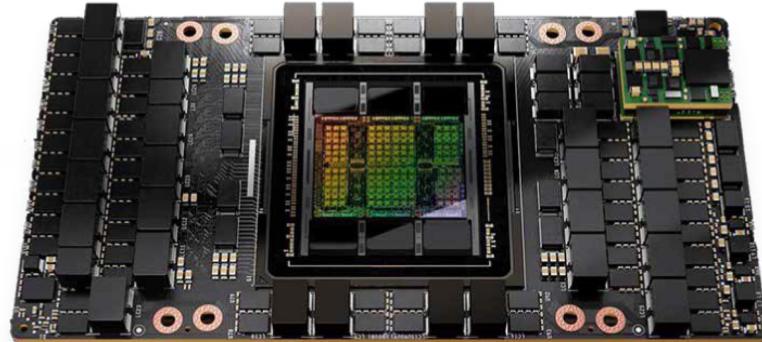
- Memory Banks
- Modalità di Accesso e Bank Conflicts

*Le prestazioni effettive di un kernel non possono essere spiegate unicamente attraverso l'esecuzione dei warp.*

# Introduzione al Modello di Memoria CUDA

## NVIDIA H100 SXM5 GPU

SMs	132
GPU Boost Clock	1980 Mhz
FP32 Cores / SM	128
FP64 Cores / SM (excl. Tensor)	64
INT32 Cores / SM	64
Tensor Cores / SM	4
Peak FP64 TFLOPS	33.5 TFLOPS
Peak FP32 TFLOPS	66.9 TFLOPS
Peak FP64 Tensor	66.9 TFLOPS
Memory Interface	5120-bit HBM3
GPU Memory Bandwidth	3.352 TB/s
GPU Memory	80 Gb



- Il calcolo dei **FLOPS teoricamente raggiungibili (Theoretical Peak FLOPS)** è dato da:

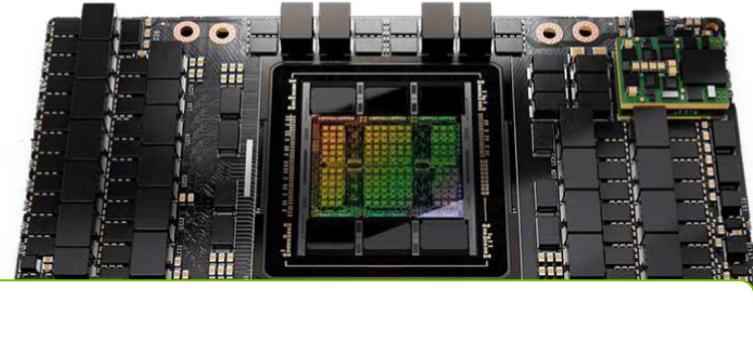
$$(1.980 \text{ Ghz}) \cdot (132 \text{ SM}) \cdot (64 \text{ FP64 Core}) \cdot (2 \text{ Fused Multiply Add}) = 33.5 \text{ TFLOPS (FP64)}$$

$$(1.980 \text{ Ghz}) \cdot (132 \text{ SM}) \cdot (128 \text{ FP32 Core}) \cdot (2 \text{ Fused Multiply Add}) = 66.9 \text{ TFLOPS (FP32)}$$

# Introduzione al Modello di Memoria CUDA

## NVIDIA H100 SXM5 GPU

SMs	132
GPU Boost Clock	1980 Mhz
FP32 Cores / SM	128



Spesso, la potenza di calcolo teorica delle GPU moderne supera di gran lunga quella effettivamente sfruttabile nelle applicazioni reali.

Peak FP32 TFLOPS	66.9 TFLOPS
Peak FP64 Tensor	66.9 TFLOPS
Memory Interface	5120-bit HBM3
GPU Memory Bandwidth	3.352 TB/s
GPU Memory	80 Gb

$$(1.980 \text{ Ghz}) \cdot (132 \text{ SM}) \cdot (64 \text{ FP64 Core}) \cdot (2 \text{ Fused Multiply Add}) = 33.5 \text{ TFLOPS (FP64)}$$

$$(1.980 \text{ Ghz}) \cdot (132 \text{ SM}) \cdot (128 \text{ FP32 Core}) \cdot (2 \text{ Fused Multiply Add}) = 66.9 \text{ TFLOPS (FP32)}$$

# Introduzione al Modello di Memoria CUDA

## NVIDIA H100 SXM5 GPU

SMs	132
GPU Boost Clock	1980 Mhz
FP32 Cores / SM	128
FP64 Cores / SM (excl. Tensor)	64
INT32 Cores / SM	64
Tensor Cores / SM	4
Peak FP64 TFLOPS	33.5 TFLOPS
Peak FP32 TFLOPS	66.9 TFLOPS
Peak FP64 Tensor	66.9 TFLOPS
Memory Interface	5120-bit HBM3
GPU Memory Bandwidth	3.352 TB/s
GPU Memory	80 Gb

## Codice CUDA C

```
__global__ void kernel(double* x, double* y, int n)
{
    int idx = threadIdx.x + blockDim.x * blockIdx.x;
    if (idx < n) y[idx] = 3.0 * x[idx];
}

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

- Il kernel richiede **una lettura** dalla memoria (8 byte) e **una scrittura** (8 byte) in memoria per ogni operazione floating point.
- Bandwidth richiesta** per supportare l'esecuzione del kernel alla massima capacità computazionale  
FP64 è data da:  $FLOPS \times (\text{bytes\_read} + \text{bytes\_write}) = 33.5 \text{ TFLOPS} \times 16 \text{ byte} = 536 \text{ TB/s}$ .
- L'hardware allo stato dell'arte (in questo caso, H100) raggiunge però al massimo **3.352 TB/s**.

# Introduzione al Modello di Memoria CUDA

## Analisi Bandwidth H100 SXM5 GPU

- 132 SM, frequenza **1980 MHz** (boost clock).
- Ogni SM può richiedere 64 byte per ciclo di clock.
- Tasso massimo di richieste di memoria (**Peak Memory Request Rate**):

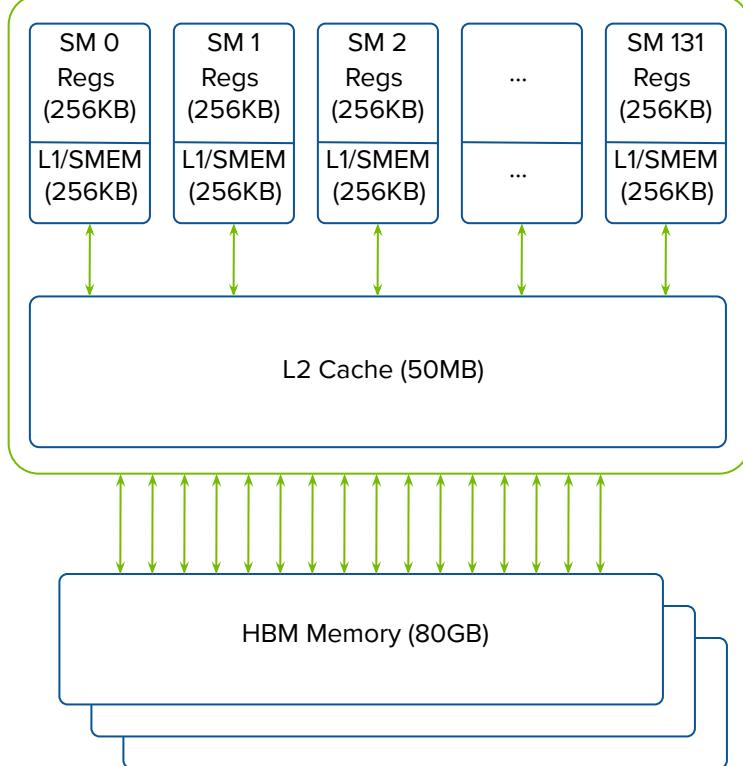
$$64B \times 132 \text{ SM} \times 1980 \text{ MHz} = 16.727 \text{ TB/s}$$

- Memoria **HBM3** con bandwidth **3.36 TB/s**.
- Rapporto tra bandwidth richiesta e fornita:

$$\text{Ratio} = \frac{16.727 \text{ TB/s}}{3.352 \text{ TB/s}} = 4.99x$$

- **Nota:** La GPU può richiedere dati quasi 5 volte più velocemente di quanto la memoria possa fornirli.

## NVIDIA H100 SXM5 GPU



# Differenze tra Memory Bound e Compute Bound

## Limiti Prestazionali

- Per ottimizzare un kernel CUDA è cruciale comprendere se il collo di bottiglia risiede negli accessi alla memoria o nella **capacità computazionale** della GPU. Questa distinzione determina le **strategie di ottimizzazione** da adottare.

## Memory Bound

- Un kernel è *memory bound* quando il tempo di esecuzione è limitato dalla velocità di accesso alla memoria piuttosto che dalla capacità di elaborazione dei core.
- La GPU trascorre più tempo in **attesa dei dati** rispetto a eseguire calcoli (poche operazioni per byte letto/scritto)
- Cause comuni:**
  - Accessi frequenti** alla memoria (lettura/scrittura) con latenza elevata.
  - Banda di memoria insufficiente** rispetto ai rispetto ai requisiti del kernel.

## Compute Bound

- Un'operazione è *compute bound* quando il tempo di esecuzione è limitato dalla capacità di calcolo della GPU, con sufficiente larghezza di banda per i dati.
- La GPU trascorre più tempo a eseguire calcoli rispetto all'attesa dei dati (molte operazioni per byte letto/scritto).
- Cause comuni:**
  - Operazioni aritmetiche intensive, come moltiplicazioni di matrici dense o convoluzioni, che richiedono elevati FLOP rispetto agli accessi in memoria.

# Kernel Performance

Quale metrica utilizzare per misurare le **performance**?



## FLOPS

*Floating Point Operations per Second*

$$\text{FLOPS} = \frac{\text{N}_{\text{Floating Point Operations}} (\text{FLOP})}{\text{Tempo Trascorso (s)}}$$

- Utilizzata per valutare **compute-bound kernel**, dove il tempo è dominato dai calcoli.
- Unità di misura: **MFLOPs, GFLOPs, TFLOPs**.
- La **Peak Performance** della GPU rappresenta il limite teorico massimo (es. H100: 66.5 TFLOPs FP32).

## Bandwidth

*Quantità di dati trasferiti al secondo*

$$\text{Bandwidth} = \frac{\text{Dimensione Dati Trasferiti (Byte)}}{\text{Tempo Trascorso (s)}}$$

- Utilizzata per valutare **memory-bound kernel**, dove il tempo è dominato dagli accessi in memoria.
- Unità di misura: **GB/s, TB/s**
- La **Peak Bandwidth** dell'hardware rappresenta il limite teorico massimo raggiungibile (es. H100: 3.35 TB/s).

# Memory Bandwidth: Teorica vs. Effettiva

## Prestazione del Kernel

- **Memory Latency:** Tempo richiesto per soddisfare una richiesta di dati dalla memoria della GPU, inclusi i ritardi di trasferimento fino ai core.
- **Memory Bandwidth:** La quantità massima di dati che può essere trasferita tra la memoria della GPU e gli altri componenti (ad esempio, gli SM) in un'unità di tempo.
- **Kernel Memory Bound:** Un kernel è vincolato dalla memoria (memory bound) quando le sue prestazioni sono limitate dalla velocità di trasferimento dei dati piuttosto che dalla capacità di calcolo.

## Tipologie di Larghezze di Banda

- **Banda Teorica**
  - Massima larghezza di banda raggiungibile con l'hardware disponibile.
  - **Esempio:** Fermi M2090 è pari a 177.6 GB/s, Ampere A100 è pari a 1.6 TB/s, Hopper H100 pari a 3.35 TB/s.
- **Banda Effettiva**
  - Larghezza di banda realmente raggiunta da un kernel in esecuzione:

$$\text{Bandwidth Effettiva (GB/s)} = \frac{(\text{byte letti} + \text{byte scritti}) \times 10^{-9}}{\text{tempo trascorso (ns)}}$$

- **Esempio:** Copia di una matrice  $2048 \times 2048$  contenente interi da 4 byte da e verso il dispositivo:

$$\text{Bandwidth Effettiva (GB/s)} = \frac{(2048 \times 2048 \times 2 \times 4) \times 10^{-9}}{\text{tempo trascorso (ns)}}$$

# Il Modello di Performance Roofline

## Modello Roofline

- Il **modello Roofline** è un metodo grafico utilizzato per rappresentare le prestazioni di un algoritmo (o di un kernel CUDA) in relazione alle capacità di calcolo e memoria di un sistema (nel nostro caso, GPU).
- Utile per capire se un algoritmo viene limitato da **problemi di calcolo** o da **problemi di accesso alla memoria**.

## Intensità Aritmetica (AI)

- L'**intensità aritmetica** misura il rapporto tra la **quantità di operazioni di calcolo** e il **volume di dati trasferiti** dalla/verso la memoria di un algoritmo/kernel:

$$AI = \frac{\text{FLOPs}}{\text{Bytes Trasferiti}}$$

- **FLOPs**: Numero di operazioni in virgola mobile o operazioni aritmetiche in generale.
- **Bytes trasferiti**: Quantità di dati letti o scritti dalla memoria DRAM.

## Soglia di Intensità Aritmetica

- La soglia dipende dall'hardware specifico (es. GPU), ed è definito dal seguente rapporto:

$$\text{Soglia (AI)} = \frac{\text{Theoretical Computational Peak Performance (FLOPs/s)}}{\text{Bandwidth Peak Performance (Bytes/s)}}$$

- **Computational Peak Performance**: Massima capacità teorica di calcolo di un dispositivo, misurata in FLOPS.
- **Bandwidth Peak Performance**: Velocità massima con cui i dati possono essere trasferiti tra GPU e memoria principale.

# Il Modello di Performance Roofline

## Modello Roofline

- Il **modello Roofline** è un metodo grafico utilizzato per rappresentare le prestazioni di un algoritmo (o di un kernel CUDA) in relazione alle capacità di calcolo e memoria di un sistema (nel nostro caso, GPU).
- Utile per capire se un algoritmo viene limitato da **problemi di calcolo** o da **problemi di accesso alla memoria**.

## Intensità Aritmetica (AI)

- L'**intensità aritmetica** misura il rapporto tra la **quantità di operazioni di calcolo** e il **volume di dati trasferiti** dalla/verso la memoria di un algoritmo/kernel:

$$AI = \frac{\text{FLOPs}}{\text{Bytes Trasferiti}}$$

- FLOPs: Numero di operazioni di calcolo.
- Bytes trasferiti: Quantità di dati trasferiti.

### Interpretazione dell'Intensità Aritmetica

- Bassa intensità aritmetica ( $AI < \text{Soglia}$ ): *Memory Bound*, poiché richiede più accesso alla memoria rispetto al calcolo.
- Alta intensità aritmetica ( $AI > \text{Soglia}$ ): *Compute Bound*, poiché esegue molti calcoli rispetto ai dati trasferiti.

## Soglia di Intensità Aritmetica

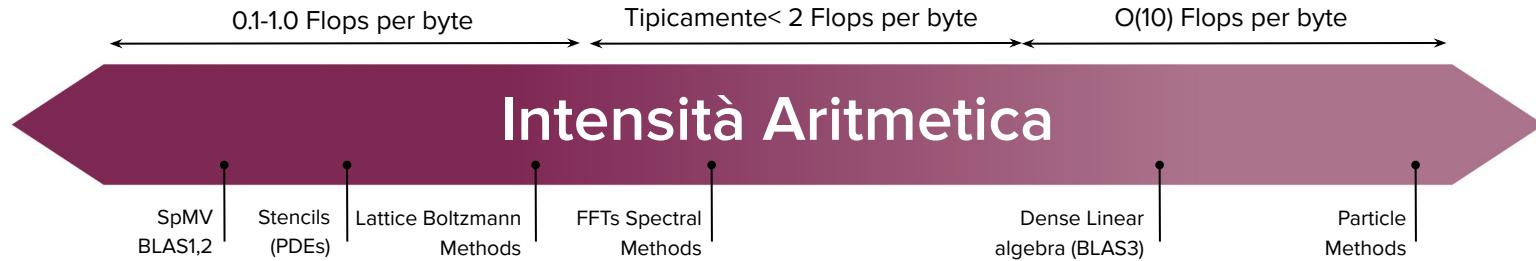
- La soglia dipende dall'hardware.

S

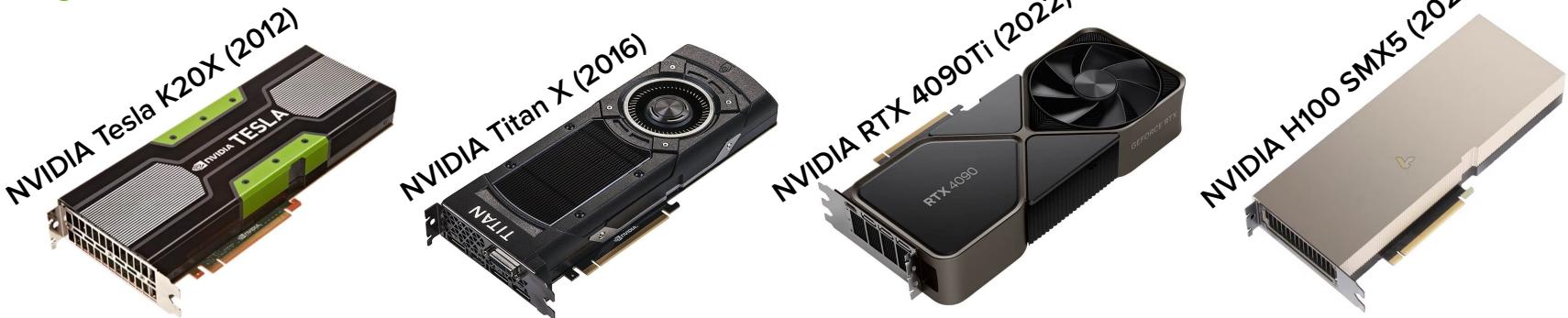
- Computational Peak Performance: Massima capacità teorica di calcolo di un dispositivo, misurata in FLOPS.
- Bandwidth Peak Performance: Velocità massima con cui i dati possono essere trasferiti tra GPU e memoria principale.

# Intensità Aritmetica: Algoritmi e Hardware

## Intensità Aritmetica degli Algoritmi HPC



## Soglie AI nell'Hardware GPU



Peak FP64: 1,312 GFLOPS

Bandwidth (DRAM): 249.6 GB/s

**Soglia (AI):** ≈ 0,00526 Flop/byte

Peak FP64: 342.9 GFLOPS

Bandwidth (DRAM): 480.4 GB/s

**Soglia (AI):** ≈ 0,714 Flop/byte

Peak FP64: 1,457 TFLOPS

Bandwidth (DRAM): 1.15 TB/s

**Soglia (AI):** ≈ 1,27 Flop/byte

Peak FP64: 33,5 TFLOPS

Bandwidth (DRAM): 3.352 TB/s

**Soglia (AI):** ≈ 10 Flop/byte

# Esempi di Memory Bound e Compute Bound in CUDA

## Esempio: Somma di Vettori in CUDA

- Somma di due vettori **a** e **b** di dimensione **N** in **double** (8 byte) per ottenere un vettore **c**.
- **Operazioni Richieste:** **N** somme (1 FLOP per elemento) usando **double**.
- **Bytes Trasferiti:**
  - Accessi per input: **2N x 8** (**a** e **b** in **double**).
  - Accesso per output: **N x 8** (**c** in **double**).
  - Totale: **3N x 8 bytes**
- **Intensità Aritmetica (AI):**

$$AI = \frac{N \text{ FLOPs}}{3N \times 8 \text{ bytes}} = \frac{1}{24} \text{ FLOPs / byte} = 0,041 \text{ FLOPs / byte}$$

- **Calcolo della Soglia:** La soglia per una determinata GPU è data dal **rapporto** tra la potenza di calcolo teorica (Peak FP64) e la bandwidth di memoria (Bandwidth Peak):

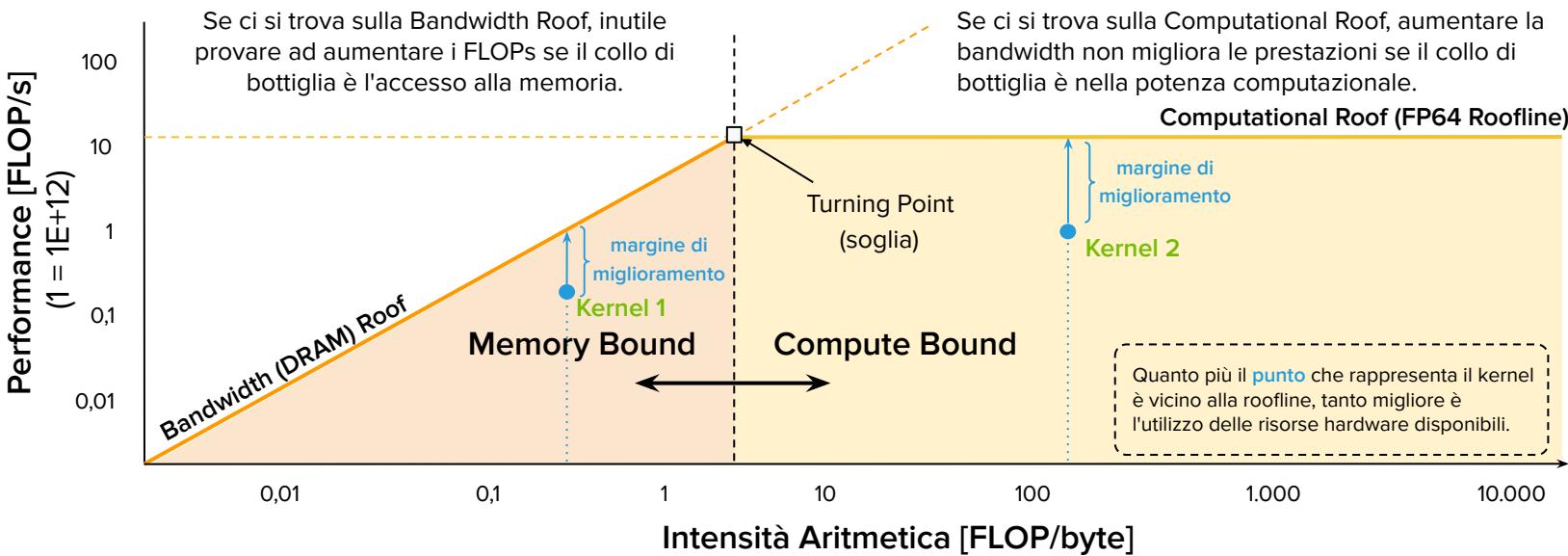
$$\text{Soglia (AI)} = \frac{\text{Theoretical Computational Peak FP64 Performance (FLOPs/s)}}{\text{Bandwidth Peak Performance (Bytes/s)}}$$

- **AI < Soglia** → **Memory Bound** (es. NVIDIA Titan X, NVIDIA RTX 4090Ti, NVIDIA H100 SMX5, etc)
- **AI > Soglia** → **Compute Bound** (es. NVIDIA Tesla K20X)

# Diagramma Roofline

## Curve nel Diagramma

- Bandwidth Roof:** Una linea retta inclinata che rappresenta il limite imposto dalla banda di memoria. La pendenza di questa retta è pari alla bandwidth della memoria del device (DRAM).
- Computational Roof:** Una linea orizzontale che rappresenta il limite massimo di prestazioni computazionali in doppia precisione (FP64 Roofline). Questa è la massima velocità a cui la GPU può eseguire operazioni in FP64.



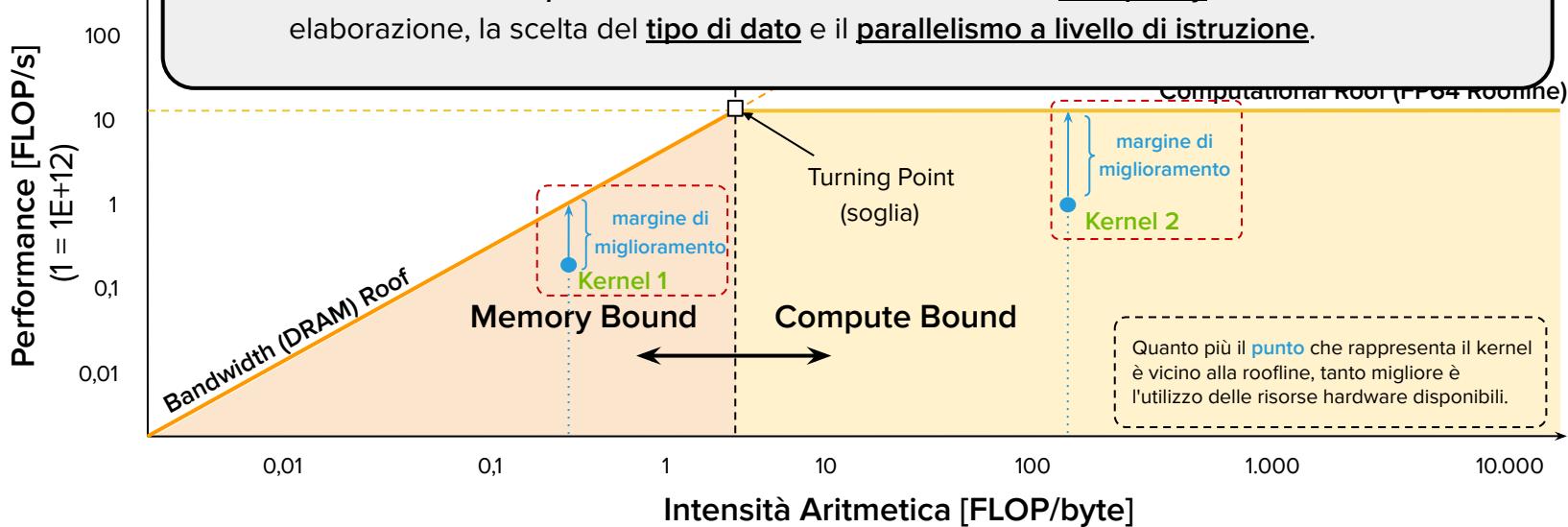
# Diagramma

## Curve nel Diagramma

- Bandwidth Roof: rappresenta la capacità massima di trasferimento dati.
- Computational Roof: rappresenta il limite massimo di performance complessive.
- Compute Bound: rappresenta il limite di performance imposto dalla capacità di elaborazione.
- Memory Bound: rappresenta il limite di performance imposto dal tempo di accesso alla memoria.

## Margine di Miglioramento

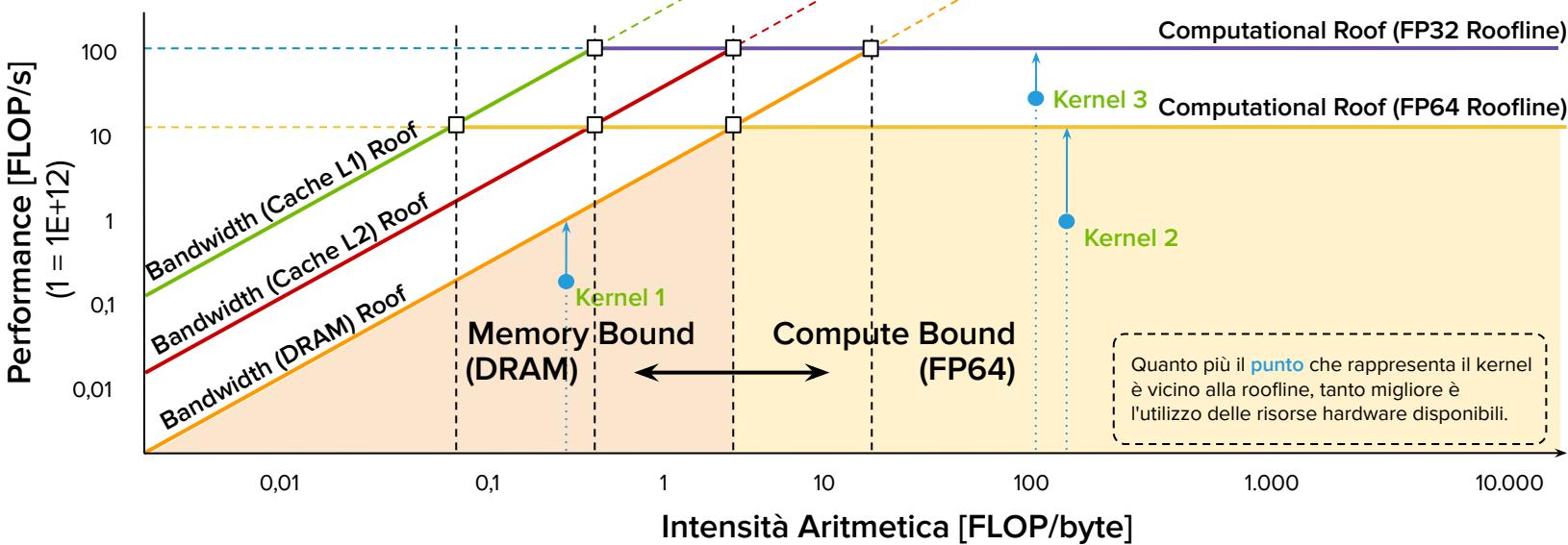
- Quando un kernel si colloca al di sotto della Roofline (sia Bandwidth Roof che Computational Roof), è indicativo di **inefficienze potenziali**.
- È essenziale ottimizzare lo **sfruttamento delle risorse computazionali** e gli **accessi ai dati in memoria**, poiché in entrambe le situazioni c'è margine di miglioramento per le prestazioni complessive.
- In un contesto ***memory bound***, è di notevole importanza ottimizzare gli accessi alla memoria considerando la **gerarchia delle memorie** (località dei dati) e i **pattern di accesso**.
- In un contesto ***compute bound***, è cruciale massimizzare l'**occupancy** delle unità di elaborazione, la scelta del **tipo di dato** e il **parallelismo a livello di istruzione**.



# Diagramma Roofline

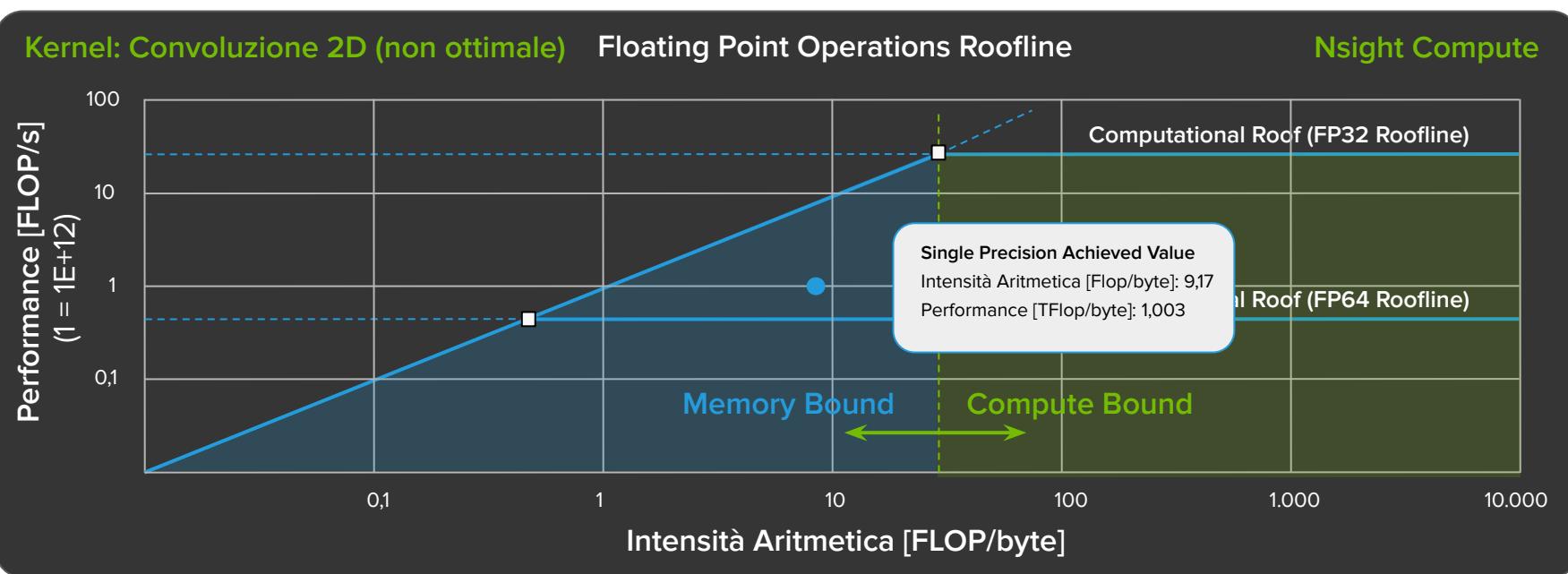
## Multiple Roofline

- Roofline di Memoria GPU:** Diversi limiti di bandwidth tra le memorie della gerarchia (DRAM, Cache, Shared Memory). Migliori prestazioni spostando i dati nelle memorie più veloci.
- Roofline di Calcolo GPU:** Diversi limiti prestazionali tra FP16/BF16 (più veloci), FP32, FP64 e INT8/INT4 per inferenza, con performance che dipendono dal tipo di unità di calcolo utilizzata (CUDA cores, Tensor cores).



# Diagramma Roofline in Nsight Compute

- NVIDIA Nsight Compute integra uno strumento di analisi Roofline che visualizza automaticamente il posizionamento di ogni kernel rispetto ai limiti hardware (memoria e computazionale) della GPU specifica in uso.
- Il grafico interattivo mostra **intensità aritmetica vs performance**, permettendo di identificare rapidamente se i kernel sono *memory-bound* o *compute-bound* e guidare le strategie di ottimizzazione.



# Panoramica del Modello di Memoria CUDA

## ➤ Modelli di Performance

- Memory-Bound vs Compute-Bound
- Intensità Aritmetica e Roofline Model

## ➤ Gerarchia di Memoria CUDA

- Organizzazione Gerarchica Completa
- Scope e Programmabilità

## ➤ Gestione della Memoria Host-Device

- Allocazione e Trasferimenti
- Pinned Memory
- Zero-Copy Memory
- UVA (Unified Virtual Addressing)
- Unified Memory (UM)

## ➤ Global Memory

- Pattern di Accesso
- Lettura Cached vs Uncached
- Scrittura

## ➤ Shared Memory

- Memory Banks
- Modalità di Accesso e Bank Conflicts

# Introduzione al Modello di Memoria CUDA

## 1. Gestione Memoria

- La gestione efficiente della memoria è **cruciale** per le prestazioni.

## 2. Collo di Bottiglia: Accesso ai Dati

- Le prestazioni sono molto spesso **limitate dalla velocità** di caricamento e archiviazione dei dati.

## 3. Compromesso Costo-Prestazioni

- Memoria ad alta capacità e prestazioni elevate spesso comporta **costi proibitivi**.

## 4. CUDA: Ponte tra Host e Dispositivo

- Il modello di memoria CUDA **unifica** i sistemi di memoria dell'host e del device.

## 5. Controllo Granulare dei Dati

- CUDA offre **controllo esplicito** sul posizionamento dei dati nell'intera gerarchia di memoria.

## 6. Ottimizzazione Hardware-Consapevole

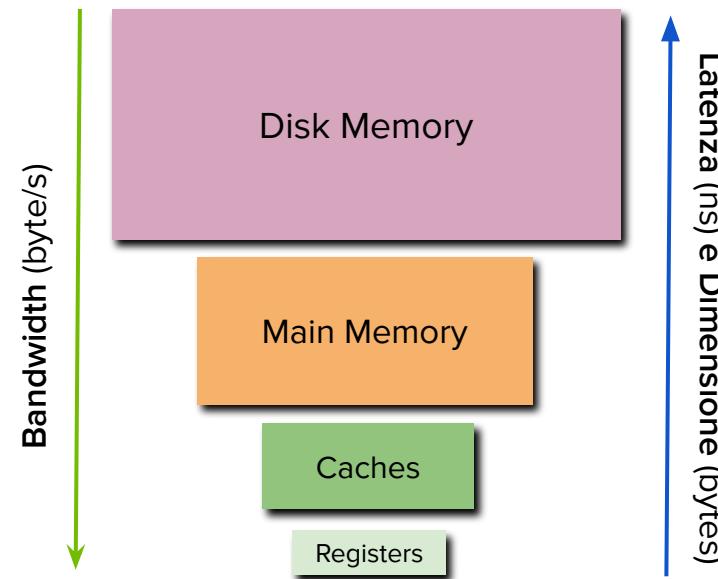
- Sfruttamento ottimale dell'hardware per **minimizzare latenza** e **massimizzare larghezza di banda**.

# I Vantaggi di una Gerarchia di Memoria

- Le applicazioni spesso seguono il **principio di località**, accedendo a una porzione relativamente piccola e localizzata del loro spazio di indirizzamento in un dato momento:
  - Temporale**: Dati usati di recente hanno più probabilità di essere riutilizzati a breve.
  - Spaziale**: Dati vicini a quelli usati di recente hanno più probabilità di essere necessari.

## Gerarchia di Memoria

- La **gerarchia di memoria** offre livelli di memoria con differenti latenze, larghezza di banda, e capacità:
  - Livelli Bassi (Registri, Cache)**: Bassa latenza, bassa capacità, costo elevato per bit, accesso frequente.
  - Livelli Alti (Disco)**: Alta latenza, alta capacità, costo ridotto per bit, accessi meno frequenti.
- CPU e GPU usano **DRAM** per la memoria principale, **SRAM** per registri/cache e **Dischi/Flash** per la memoria più lenta e capiente.
- CUDA espone **più livelli** della gerarchia rispetto ai modelli CPU, offrendo **un controllo più esplicito** per ottimizzare le prestazioni.



*"Illusione di una memoria grande ma rapida"*

# Confronto tra SRAM e DRAM - Memorie Volatili

## SRAM (Static Random Access Memory)

### Tecnologia

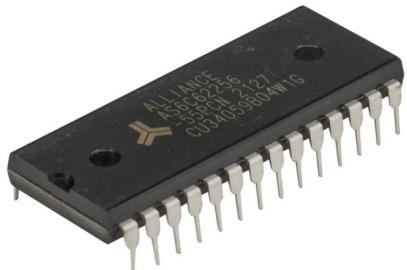
- Utilizza **latch** o **flip-flop** costruiti da transistor CMOS per memorizzare i dati, eliminando la necessità di refresh.
- Più **complessa, meno densa** ma più **costosa**.

### Prestazioni

- Tempi di accesso più rapidi** e latenza inferiore.
- Perfetta per applicazioni che richiedono alta velocità e prestazioni, come le **cache** o **registri**.

### Consumo Energetico

- Consumo di energia statica **più elevato** a causa della circuiteria continuamente attiva che mantiene i dati.



## DRAM (Dynamic Random Access Memory)

### Tecnologia

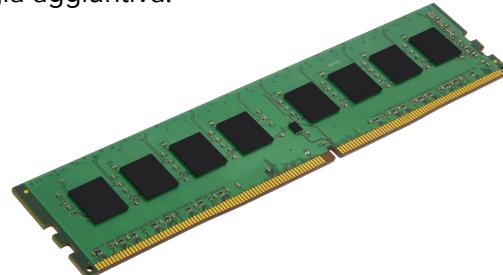
- Utilizza un condensatore e un transistor per bit, richiedendo un **refresh periodico**.
- Più **semplice e densa** ma più **lenta**.

### Prestazioni

- Offre una maggiore densità e un costo inferiore per bit, ideale per grandi sistemi di **memoria principale**.
- Tempi di accesso più lenti a causa del refresh.

### Consumo Energetico

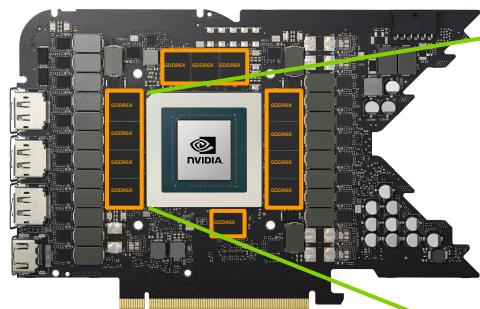
- Consumo di energia statica **inferiore** grazie alla circuiteria semplice. Tuttavia, il refresh consuma energia aggiuntiva.



# Confronto tra Memoria DDR e GDDR

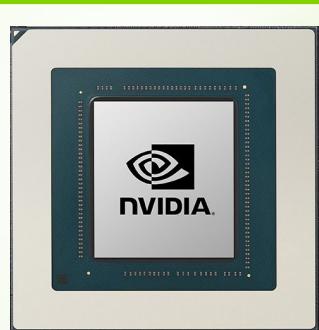
	DDR (Double Data Rate)	GDDR (Graphics Double Data Rate)
<b>Target</b>	CPU	GPU
<b>Utilizzo</b>	Sistemi operativi, applicazioni multi-tasking, database	Gaming, rendering 3D, intelligenza artificiale
<b>Architettura</b>	Ottimizzata per bassa latenza con bus a 64 bit, progettata per accessi rapidi nelle operazioni di sistema.	<b>Memoria ottimizzata per massimo throughput</b> con bus dati ampi (es. 384 bit) per garantire alta banda.
<b>Memory Clock</b>	Fino a 3600 MHz (DDR5)	Fino a 1500 MHz (GDDR6X)
<b>Larghezza di banda</b>	Fino a 100 GB/s (DDR5)	Fino a 1 TB/s (GDDR6X)
<b>Consumo energetico</b>	Basso consumo in idle, efficiente per carichi di lavoro variabili	Consumo elevato anche in idle, ottimizzato per prestazioni costanti
<b>Costo per GB</b>	Circa \$10-\$20 (DDR5)	Circa \$30-\$60 (GDDR6X)
<b>Capacità massima per modulo</b>	Fino a 128-256GB (DDR5)	Fino a 24-48GB ( GDDR6/GDDR6X)

# Confronto tra Memoria DDR e GDDR

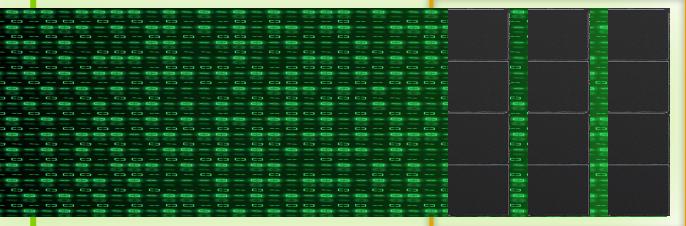


NVIDIA 4090 PCB

GPU



GDDR6X



Bus Width - 512 Bits  
Bandwidth - 1.15 TB/s

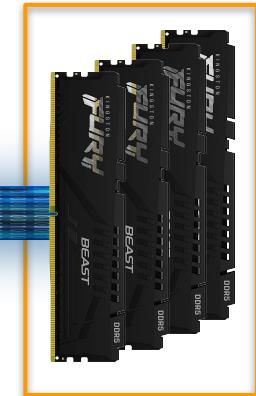


MSI Z790

CPU



DDR5

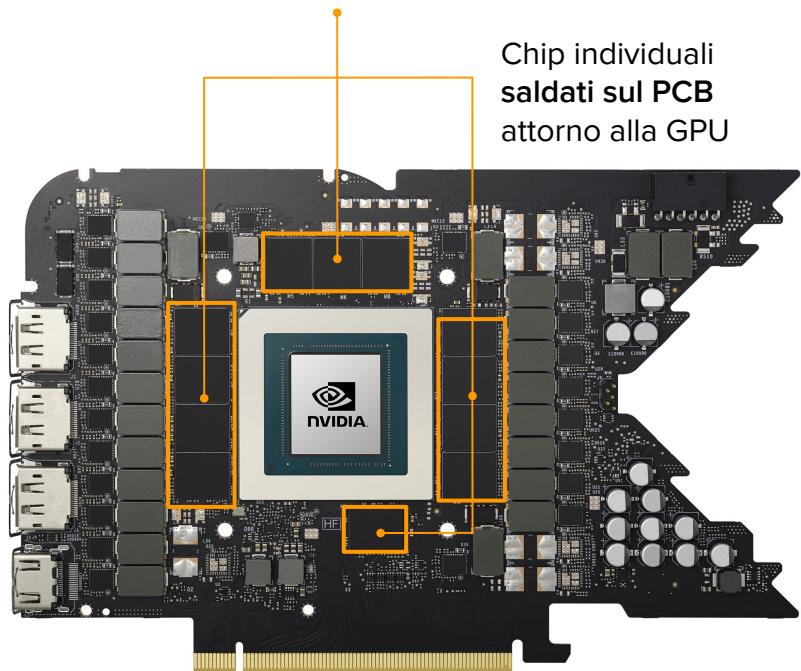


Bus Width - 64 Bits  
Bandwidth - 100 GB/s

# Confronto tra Memoria GDDR e HBM nelle GPU NVIDIA

**GDDR6X**

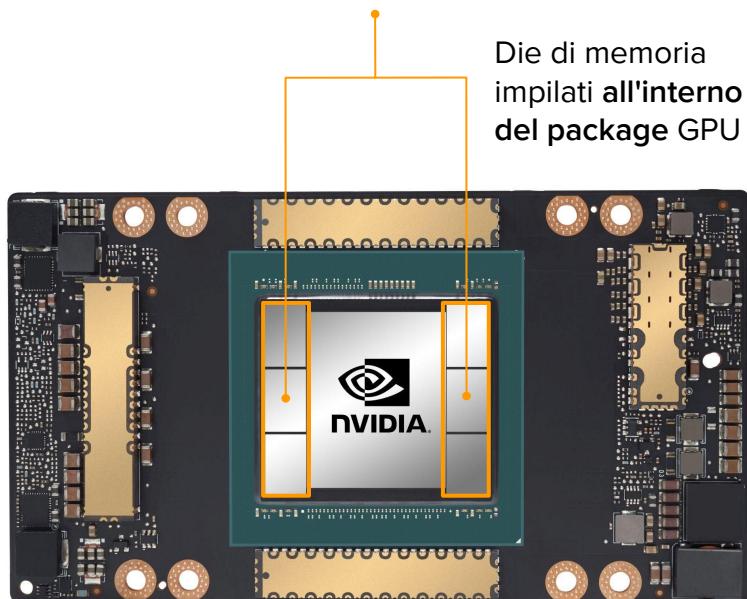
(Graphics Double Data Rate)



NVIDIA 4090 PCB

**HBM3**

(High Bandwidth Memory)



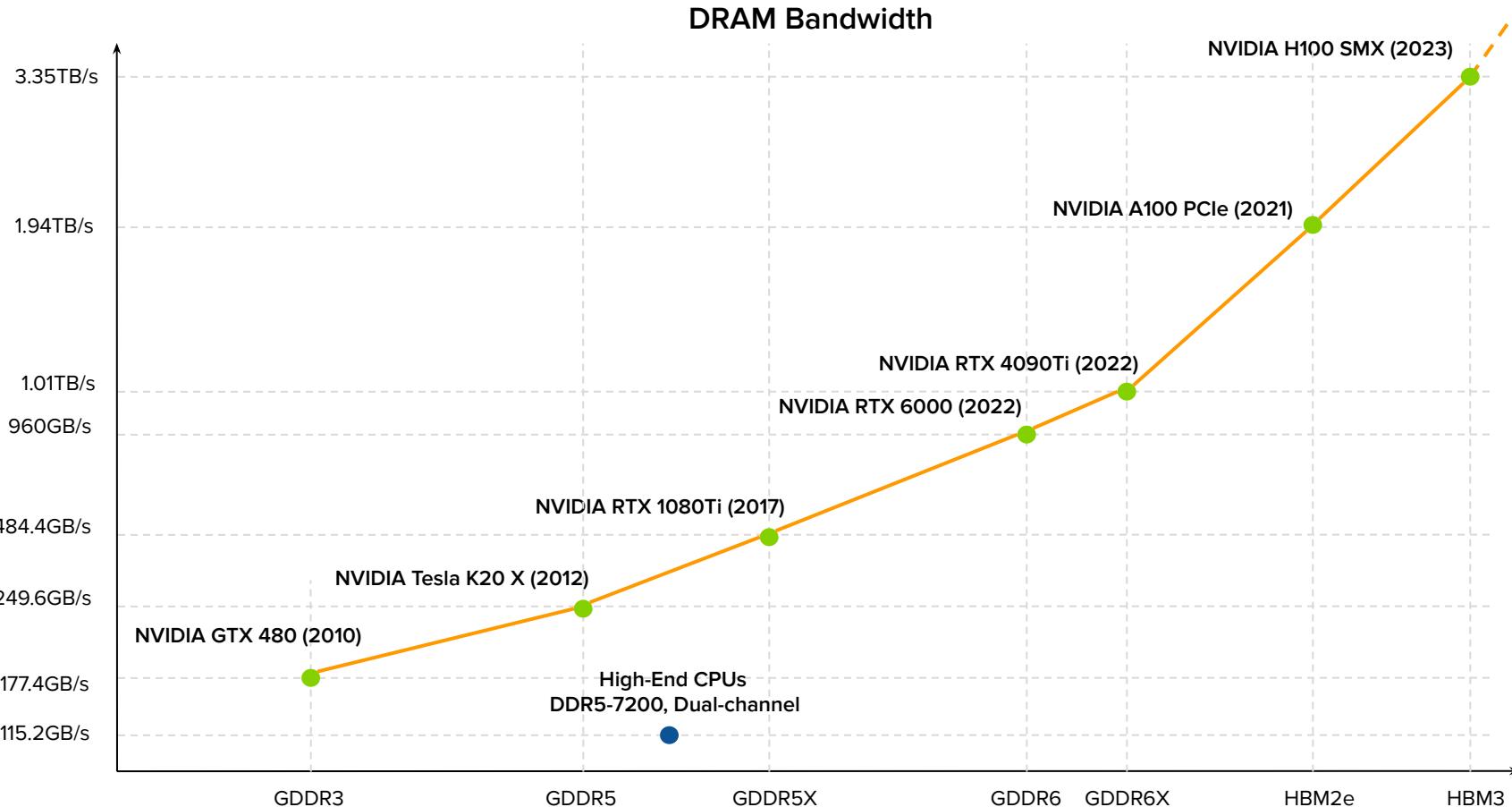
NVIDIA H100 PCB

Die di memoria  
impilati all'interno  
del package GPU

# Confronto tra Memoria GDDR e HBM nelle GPU NVIDIA

	GDDR (Graphics Double Data Rate)	HBM (High Bandwidth Memory)
Esempio GPU	NVIDIA RTX 4090 Ada (GDDR6X)	NVIDIA H100, GH200 (HBM3, HBM3e)
Utilizzo	Grafica avanzata, AI su piccola scala, rendering	HPC, training AI intensivo, inferenza AI in tempo reale
Architettura	Chip singoli saldati al PCB	Moduli impilati sul die della GPU (più DRAM in uno spazio ridotto)
Bus Width	Fino a 384-bit (GDDR6X)	Fino a 5120-bit (HBM3, NVIDIA H100)
Banda Passante	~1 TB/s (GDDR6X, RTX 6000 Ada)	~2 TB/s (HBM3, NVIDIA H100), ~3 TB/s (HBM3e, NVIDIA GH200)
Latenza	Più bassa rispetto a HBM	Più alta rispetto a GDDR
Capacità Massima	Fino a 24GB (modulo GDDR6/GDDR6X)	Fino a 80-144 GB (NVIDIA H100, GH200)
Efficienza Energetica	Più alto consumo rispetto a HBM	Più efficiente, ottimizzato per HPC
Costo	Più accessibile, adatta a workstation e GPU mainstream	Costosa, ideale per acceleratori di fascia alta

# Confronto tra Memoria GDDR e HBM nelle GPU NVIDIA



# Gerarchia di Memoria CUDA

## 1. Registri

- Memoria più veloce, **privata per ogni thread**, usata per variabili temporanee.

## 2. Shared Memory

- Memoria veloce **condivisa tra i thread di un blocco**, per la comunicazione e la cooperazione.

## 3. Caches (L1, L2, Texture, Constant, Instructions)

- Memoria intermedia automatica che riduce i tempi di accesso ai dati frequentemente utilizzati.

## 4. Memoria Locale

- Privata per ogni thread, usata per variabili grandi o **register spill**.

## 5. Memoria Costante

- Memoria read-only per dati che non cambiano durante l'esecuzione del kernel.

## 6. Memoria Texture

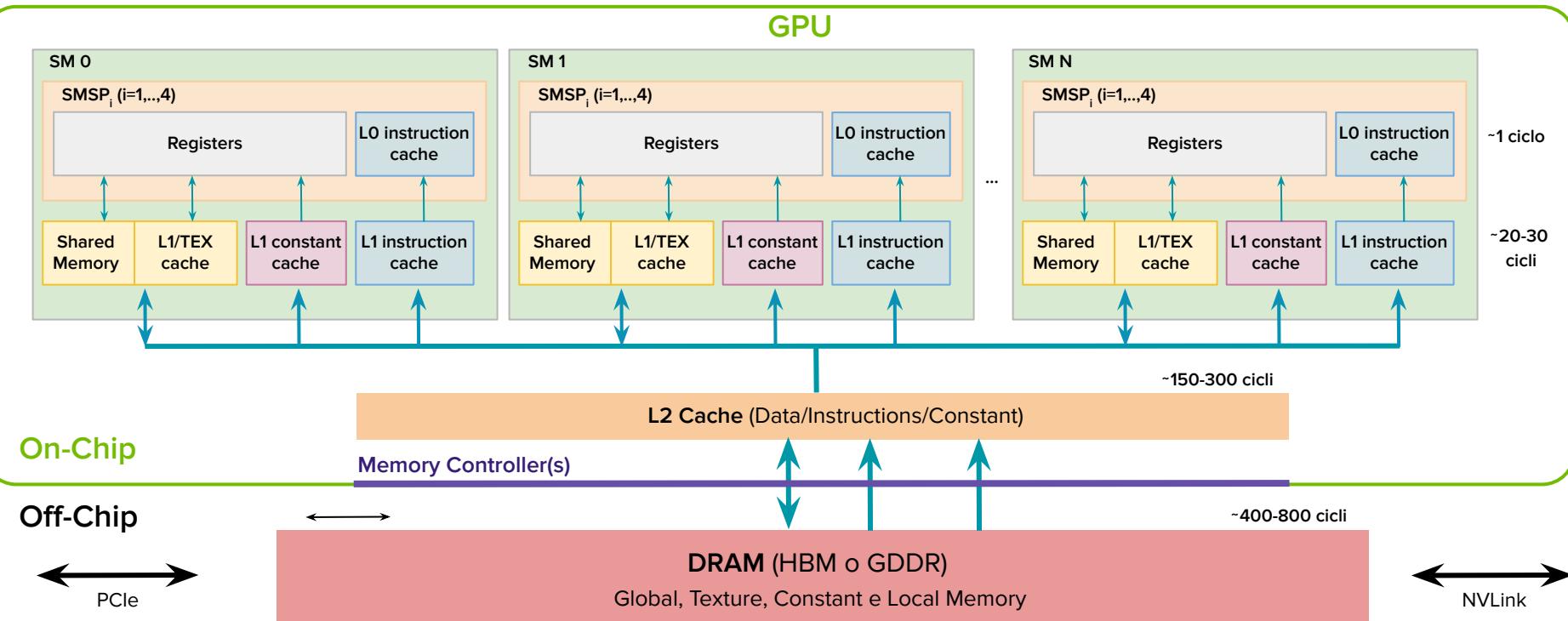
- Memoria read-only, ottimizzata per **accessi spazialmente coerenti** (es. immagini).

## 7. Memoria Globale

- Memoria più grande e lenta, **accessibile da tutti i thread** e dalla CPU.

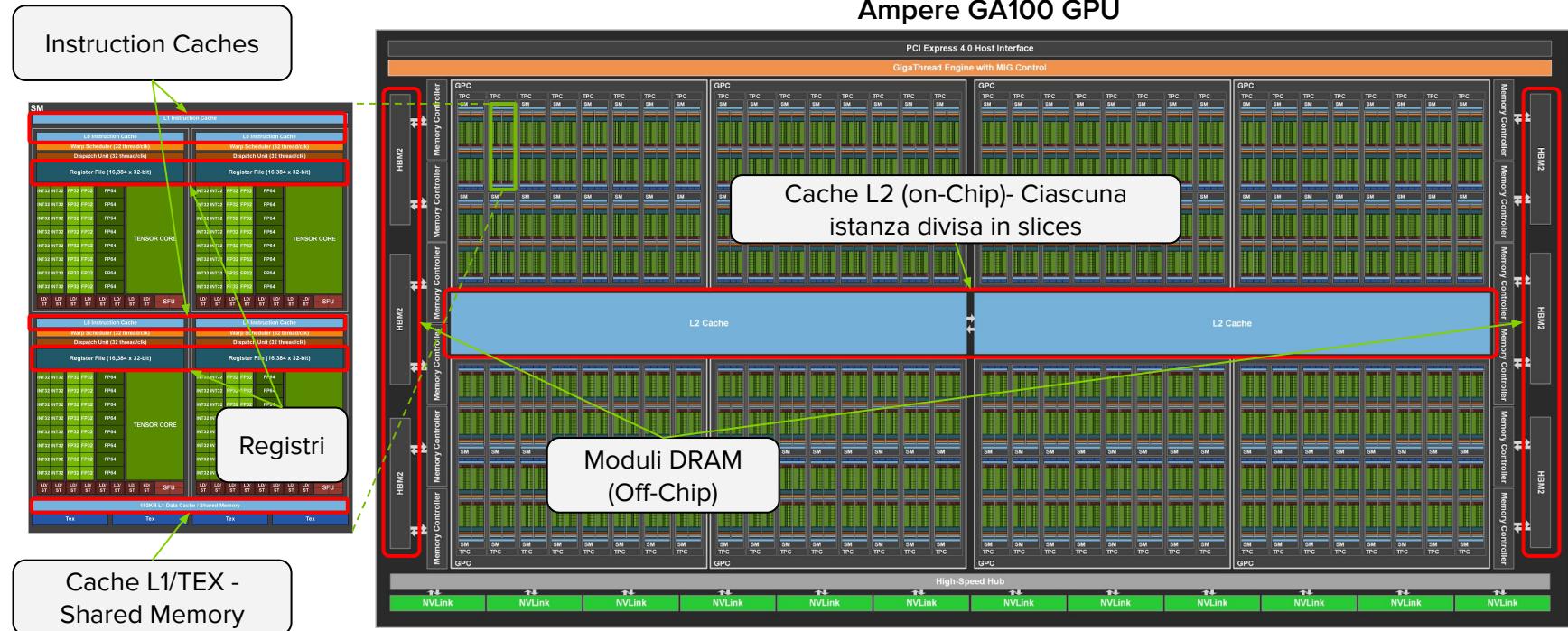
# Modelli di Memoria CUDA

- Agli occhi dei programmati, esistono due tipi di memoria:
  - Programmabile:** Controllo esplicito del posizionamento dati. CUDA ne espone diverse tipologie.
  - Non Programmabile:** Nessun controllo, gestione automatica (es. cache L1/L2 nelle CPU).



# Modelli di Memoria CUDA

- Agli occhi dei programmati, esistono due tipi di memoria:
  - Programmabile:** Controllo esplicito del posizionamento dati. CUDA ne espone diverse tipologie.
  - Non Programmabile:** Nessun controllo, gestione automatica (es. cache L1/L2 nelle CPU).



# Registri GPU

## Definizione e Caratteristiche

- Memoria **on-chip** più veloce (**massima larghezza di banda e minima latenza**) sulla GPU (accesso ~1 ciclo di clock).
- Tipicamente **32-bit** per registro.
- In un kernel, le variabili automatiche **senza altri qualificatori di tipo** vengono generalmente allocate nei registri.
- Allocati automaticamente per **variabili locali** e **array con indici costanti** nei kernel (determinabili a tempo di compilazione).
- **Strettamente privati per thread** (non condivisi) con durata **limitata all'esecuzione del kernel**.
- Una volta che il kernel ha completato l'esecuzione, non è più possibile accedere a una variabile di registro.

## Limiti e Considerazioni

- **Limite** massimo di 63 (architettura Fermi) o 255 (Kepler e successive) per thread - vedere compute capability.
- **Allocati dinamicamente tra warp attivi** in un SM, influenzando l'occupancy.
- Minor uso di registri permette di avere **più blocchi concorrenti per SM** (maggior occupancy).
- **Register Spilling:** Eccedere il limite hardware **sposta automaticamente** le variabili dai registri alla memoria locale (100-300 cicli), riducendo le prestazioni.

## Ottimizzazione

- **Euristiche del compilatore:** nvcc utilizza euristiche per minimizzare l'utilizzo dei registri ed evitare il register spilling.
- **Launch Bounds:** `_launch_bounds_ (maxThreadsPerBlock, minBlocksPerMultiprocessor)` aiuta il compilatore nell'allocazione efficiente per ciascun kernel se inserito prima della chiamata.
- **Direttive compilatore per analisi e controllo:**
  - **-Xptxas -v, -abi=no:** Mostra l'utilizzo delle risorse hardware (numero di registri, bytes di shared memory, etc.).
  - **-maxrregcount=32:** Limita il numero massimo di registri per unità di compilazione (ignorato specificati i launch bounds).

# Memoria Locale

## Definizione e Caratteristiche

- Memoria **off-chip (DRAM)**. Nome ambiguo, fisicamente collocata nella stessa posizione della **memoria globale**.
- **Privata per thread, non condivisa** tra thread.
- Utilizzata per variabili che non possono essere allocate nei registri a causa di **limiti di spazio** (array locali, grandi strutture).
- **Alta latenza** (tipicamente centinaia di cicli) e **bassa larghezza di banda**, stessa della memoria del device (DRAM).
- Per GPU con compute capability 2.0 e oltre, i dati sono posti in cache in **L1** a livello di SM e **L2** a livello di device.

## Variabili poste in Memoria Locale:

- Array locali referenziati con indici il cui valore **non può essere determinato a tempo di compilazione**.
- **Grandi strutture o array locali** che consumerebbero troppo spazio nei registri.
- **Variabili** che eccedono il limite di registri del kernel.
  - "Register Spill" automatico da parte del compilatore quando i registri sono esauriti.

## Considerazioni sulle prestazioni

- Preferire l'**uso di registri** dove possibile.
- Ristrutturare il codice per **ridurre variabili locali** di grandi dimensioni.
- Utilizzare **shared memory** per dati frequentemente acceduti.
- **Nota:** Nonostante il nome "locale", questa memoria non è veloce come i registri o la shared memory (stessa posizione fisica della memoria globale). Il suo utilizzo eccessivo può portare a un significativo calo delle prestazioni.

# Shared Memory (SMEM) e Cache L1

## Definizione e Caratteristiche

- Ogni SM ha memoria on-chip limitata (es. 48-228 KB), condivisa tra shared memory e cache L1 (in alcune GPU, separate).
- Partizionata fra i thread block residenti in un SM.
- Questa memoria è ad alta velocità, con **elevata bandwidth e bassa latenza** rispetto a memoria locale e globale.
- La **shared memory** è organizzata in **memory banks** di uguale dimensione che permettono l'accesso simultaneo a più dati, a condizione che i thread leggano da indirizzi diversi su banchi distinti (evitando le **bank conflicts**).
- La **shared memory** è **programmabile**, con controllo esplicito da parte del programmatore, mentre la **cache L1** è **gestita automaticamente dall'hardware** per ridurre la latenza degli accessi alla memoria globale.
- Shared memory è condivisa tra **thread di un blocco**; cache L1 serve **tutti i thread di un SM**.
- La **quantità di memoria** assegnata alla cache L1 e alla memoria condivisa è **configurabile** per ogni chiamata al kernel.

## Utilizzo

- **Shared Memory:** Per variabili dichiarate con **\_\_shared\_\_** in un kernel. Ottimizza la **condivisione e comunicazione** tra thread di un blocco. **Ciclo di vita legato al blocco di thread**, rilasciata al completamento del blocco.
- **Cache L1:** Ottimizza l'accesso alla memoria globale, migliorando le prestazioni senza richiedere intervento manuale.

## Sincronizzazione

- **Shared Memory:** Richiede sincronizzazione esplicita (**\_\_syncthreads**) per prevenire data hazard, importante per garantire l'integrità dei dati. **L'uso eccessivo di barriere può impattare negativamente le prestazioni** (SM in idle frequentemente).
- **Cache L1:** Gestisce automaticamente la coerenza dei dati, senza bisogno di sincronizzazione esplicita.

# Memoria Costante

## Definizione e Caratteristiche

- Spazio di memoria di sola lettura off-chip (**DRAM**), accessibile a tutti i **thread** di un kernel.
- Dimensione totale limitata a **64 KB** per tutte le compute capabilities (potrebbe comunque mutare in futuro).
- Una **porzione** della constant memory (tipicamente 8 KB) è **cachata on chip** per ogni SM, offrendo un accesso a bassa latenza.
- Dichiarata con **scope globale**, visibile a tutti i kernel nella stessa unità di compilazione.
- Inizializzata dall'**host** (readable and writable) e non modificabile dai kernel (read-only).

## Dichiarazione e Inizializzazione

- Dichiarata con l'attributo **constant**
- Inizializzata dall'**host** usando:

```
cudaError_t cudaMemcpyToSymbol(const void* symbol, const void* src, size_t count);
```

- L'operazione di copia è generalmente **sincrona**.

## Prestazioni e Utilizzo Ottimale

- Ideale per dati letti frequentemente e condivisi tra tutti i thread, come **coefficienti**, **costanti matematiche** o **parametri di kernel** usati uniformemente.
- Offre **prestazioni elevate** quando tutti i thread in un warp leggono dallo **stesso indirizzo** (broadcast).
- La constant memory è **meno efficiente** quando i thread di un warp leggono da **indirizzi diversi**, poiché gli accessi vengono serializzati e ogni lettura viene comunque trasmessa a tutti i thread del warp, potenzialmente sprecando larghezza di banda.

# Memoria Costante

## Definizione e Caratteristiche

- Spazio di memoria di scrittura sola
- Dimensione totale limitata
- Una **porzione** della memoria
- Dichiarata con **scope global**
- Inizializzata dall'**host** e non dal **kernel**

## Dichiarazione e Inizializzazione

- Dichiarata con l'attributo `__constant__`
- Inizializzata dall'**host** utilizzando la funzione `cudaMemcpyToSymbol`
- L'operazione di copia è sincrona

## Prestazioni e Utilizzo Ottimale

- Ideale per dati letti frequentemente e usati uniformemente.
- Offre **prestazioni elevate**.
- La constant memory è serializzata e ogni lettura

## Esempio

```
// Dichiarazione di costanti individuali
constant int dA;
constant int dB;
__constant__ int dC;

// Dichiarazione di un array di costanti
__constant__ float dABC[3];

global void useConstants(int *output){
    int idx = threadIdx.x;
    if (idx == 0) output[0] = dA + dB + dC;
    if (idx == 1) output[1] = dABC[0] + dABC[1] + dABC[2];
}

int main(void){
    // Inizializzazione
    int hA = 1, hB = 2, hC = 3;
    cudaMemcpyToSymbol(dA, &hA, sizeof(int));
    cudaMemcpyToSymbol(dB, &hB, sizeof(int));
    cudaMemcpyToSymbol(dC, &hC, sizeof(int));

    float hABC[] = {1.0f, 2.0f, 3.0f};
    cudaMemcpyToSymbol(dABC, hABC, 3 * sizeof(float));

    // Allocazione e inizializzazione dell'output
    int *d_output, h_output[2];
    cudaMalloc((void**)&d_output, 2 * sizeof(int));

    useConstants <<<1, 2>>>(d_output);
}
```

accesso a bassa latenza.

`_t count);`

che o parametri di kernel

è gli accessi vengono  
ndo larghezza di banda.

# Memoria Texture

## Definizione e Caratteristiche

- Spazio di memoria di **sola lettura** nel device, accessibile a **tutti i thread** di un kernel.
- La memoria delle texture è **off-chip (DRAM)**, ma è supportata da una cache on-chip per migliorare le prestazioni.
- **Supporto hardware** per filtraggio e interpolazione in virgola mobile nel processo di lettura dei dati.
- Ottimizzata per **località spaziale 2D**, ideale per accessi con pattern regolari (dati espressi sotto forma di **matrici**).
- I thread in un warp che usano la texture memory per accedere a dati 2D hanno **migliori prestazioni** rispetto a quelle standard.
- Dimensione della cache texture: tipicamente 8-12 KB per SM (varia per generazione).

## Prestazioni e Utilizzo

- Vantaggiosa per applicazioni con **pattern spaziali prevedibili** (es. elaborazione **immagini/video**).
- Per altre applicazioni l'uso della texture memory potrebbe essere più lento della global memory.

## Considerazioni

- Accesso in sola lettura dai kernel, limitando la flessibilità per operazioni di scrittura.
- Ideale per applicazioni di **computer graphics, elaborazione immagini e simulazioni spaziali**.
- Richiede valutazione del **trade-off** tra benefici della cache, overhead di setup e flessibilità.
- È necessario dichiarare variabili di tipo **texture** e ristrutturare il codice per sfruttarne i vantaggi.
- Possono essere necessarie funzioni come **tex1Dfetch, tex2D**, e altre, a seconda del tipo di texture utilizzata.

# Memoria Globale

## Definizione e Caratteristiche

- Memoria più grande (alcuni GB a decine di GB), con latenza più alta (400-800 cicli), e più comunemente usata sulla GPU.
- Memoria principale off-chip (**DRAM**) della GPU, accessibile tramite transazioni da 32, 64, o 128 byte.
- Scope e lifetime globale (da qui *global memory*): Accessibile da ogni thread in ogni SM per tutta la durata dell'applicazione.

## Dichiarazione e Allocazione

- **Statica:** Usando il qualificatore `_device_` nel codice device.
- **Dinamica:** Allocata dall'host con `cudaMalloc` e liberata con `cudaFree`.
  - Puntatori passati ai kernel come parametri.
  - Le allocazioni persistono per l'intera applicazione ed sono accessibili ai **thread** di tutti i kernel.

## Prestazioni e Ottimizzazione

- Fattori chiave per l'efficienza:
  - **Coalescenza:** Raggruppare accessi di thread adiacenti a indirizzi contigui.
  - **Allineamento:** Indirizzi di memoria allineati a 32, 64, o 128 byte.

## Considerazioni sull'Uso

- Accessibile da **tutti i thread di tutti i kernel**, ma richiede attenzione per la sincronizzazione (no sincronizzazione fra blocchi).
- Potenziali problemi di **coerenza** con **accessi concorrenti** da blocchi diversi (hazards).
- L'efficienza dipende dalla **compute capability** del device.
- I dispositivi beneficiano di **caching** delle transazioni, sfruttando la località dei dati.

# Memoria Globale Statica - Lettura/Scrittura dall'Host

## Manipolazione di Variabili Globali Statiche nel Dispositivo e nell'Host

```
__device__ float devData; // Variabile globale statica sul device

// Kernel CUDA che legge e modifica la variabile globale
__global__ void checkGlobalVariable() {
    printf("Dispositivo: il valore della variabile globale è %f\n", devData);
    devData += 2.0f; // Modifica il valore della variabile globale
}

int main(void) {
    float value = 3.14f;
    // Inizializza la variabile globale con un valore specifico
    cudaMemcpyToSymbol(devData, &value, sizeof(float)); // devData passato come simbolo, non indirizzo
    printf("Host: copiato %f nella variabile globale\n", value);

    checkGlobalVariable<<<1, 1>>>(); // Avvia il kernel
    cudaDeviceSynchronize(); // Assicura che l'esecuzione del kernel sia completata

    // Copia il valore della variabile globale dal dispositivo all'host
    cudaMemcpyFromSymbol(&value, devData, sizeof(float));
    printf("Host: il valore cambiato dal kernel è %f\n", value);

    cudaDeviceReset(); // Ripristina il dispositivo CUDA
    return EXIT_SUCCESS;
}
```

# Memoria Globale Statica

## Host e Device: Due Mondi Separati

- Nonostante il codice sorgente sia lo stesso, host e device operano in spazi di memoria distinti.

## Impossibile l'accesso diretto

- Il codice host **non può** accedere direttamente alle variabili del device, anche se dichiarate nello stesso file.
- Analogamente, il codice device **non può** accedere direttamente alle variabili dell'host.

## `cudaMemcpyToSymbol`: Un'illusione di accesso diretto

- Sembra permettere l'accesso diretto al codice host a **devData**, ma...
- Utilizza l'**API CUDA runtime** e l'**hardware GPU** per eseguire l'accesso.
- devData** è passato come **simbolo**, non come indirizzo fisico.
- Non è possibile utilizzare `cudaMemcpy` per passare il dato a **devData**, perché il codice host non ha accesso diretto all'indirizzo fisico delle variabili del device (**&devData**).

Tuttavia, è possibile ottenere l'indirizzo fisico con: `cudaGetSymbolAddress ( (void**) &dptr, devData) ;`

- Permette di ottenere l'**indirizzo fisico** di una variabile globale del device.
- Rende possibile l'utilizzo di `cudaMemcpy` con l'indirizzo ottenuto.

## Eccezione

- La memoria pinned CUDA permette l'accesso diretto sia da host che da device (verrà trattata in seguito).

# Memoria Globale Statica

## Compilazione con nvcc

```
nvcc globalVariable.cu -o globalVariable
```

## Esecuzione e Risultato

```
./globalVariable
Host: copiato 3.140000 nella variabile globale
Dispositivo: il valore della variabile globale è 3.140000
Host: il valore cambiato dal kernel è 5.140000
```

# Cache GPU: Struttura e Funzionamento

## Definizione e Caratteristiche

- Le cache GPU, come quelle CPU, sono memorie on chip non programmabili cruciali per accelerare l'accesso ai dati.
- La cache viene utilizzata per **memorizzare temporaneamente porzioni della memoria principale** per accessi più veloci.

## Tipi di Cache

- **Cache L1**
  - La cache L1 è la più veloce e ogni SM ne ha una propria, garantendo un accesso rapido ai dati.
  - Memorizza dati sia dalla memoria locale che globale, inclusi i dati che non trovano spazio nei registri (*register spills*).
- **Cache L2**
  - Unica e condivisa tra SM. Funge da ponte tra le cache L1 più veloci e la memoria principale più lenta.
  - Memorizza dati provenienti sia dalla memoria locale che globale, inclusi i dati derivanti da *register spills*.
- **Constant Cache** (sola lettura, per SM)
  - Presente in ogni SM, memorizza dati che non cambiano durante l'esecuzione del kernel.
  - Ottimizzata per l'accesso rapido a dati immutabili, come tabelle di lookup o parametri costanti.
- **Texture Cache** (sola lettura, per SM)
  - Specializzata per dati di texture; cruciale per rendering e accessi 2D/3D.
  - Supporta funzionalità hardware come interpolazione e filtraggio.
  - Nelle ultime architetture NVIDIA, unificata con cache L1 (L1/TEX Cache).

## Particolarità delle Cache GPU

- Su alcune GPU, è possibile configurare se i dati vengono cachati solo sia in L1 che L2, o solo in L2.

# Caratteristiche Principali della Gerarchia di Memoria

- Caratteristiche principali dei vari tipi di memoria.

Memoria	On/Off Chip	Cached	Accesso	Scope	Durata
<b>Registro</b>	On	n/a	R/W	<b>Thread</b>	<b>Thread</b>
<b>Condivisa</b>	On	n/a	R/W	<b>Thread nel Blocco</b>	<b>Blocco</b>
<hr/>					
<b>Locale</b>	Off	†	R/W	<b>Thread</b>	<b>Thread</b>
<b>Globale</b>	Off	†	R/W	<b>Tutti i Thread + Host</b>	<b>Allocazione Host</b>
<b>Costante</b>	Off	Sì	R	<b>Tutti i Thread + Host</b>	<b>Allocazione Host</b>
<b>Texture</b>	Off	Sì	R	<b>Tutti i Thread + Host</b>	<b>Allocazione Host</b>

† In cache solo su dispositivi con compute capability 2.x+

# Qualificatore di Variabili e Tipi CUDA

- Dichiarazioni di variabili CUDA e relative posizioni di memoria, scope, durata e qualificatore.

Qualificatore	Nome Variabile	Memoria	Scope	Durata
	<code>float</code> LocalVar	Registro	Thread	Thread
	<code>float</code> LocalVar[100]	Locale	Thread	Thread
<u><code>__shared__</code></u>	<code>float</code> SharedVar †	Condivisa	Blocco	Blocco
<u><code>__device__</code></u>	<code>float</code> GlobalVar †	Globale	Globale	Applicazione
<u><code>__constant__</code></u>	<code>float</code> ConstantVar †	Costante	Globale	Applicazione

† Può essere una variabile scalare o una variabile array

# Qualificatore di Variabili e Tipi CUDA

## Esempi di utilizzo dei diversi tipi di memoria in CUDA

```
__constant__ float constVal; // Variabile costante, memorizzata nella constant memory
__device__ float globalFactor = 5.0; // Variabile globale statica in memoria globale

__global__ void memoryExample(float *input, float *output) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x; // Variabile locale, memorizzata nei registri

    __shared__ float sharedData[256]; // Shared memory, condivisa tra i thread di un blocco

    float globalValue = input[idx]; // `globalValue` è nei registri, `input` è nella memoria globale

    float constValue = constVal; // `constValue` è nei registri, `constVal` è in memoria costante

    float factor = globalFactor; // `factor` è nei registri, `globalFactor` è in memoria globale

    float localValue[256]; // localValue è in memoria locale a causa della struttura array grande
    localValue[0] = globalValue * constValue * factor;

    sharedData[threadIdx.x] = localValue[0]
    __syncthreads();

    output[idx] = sharedData[threadIdx.x]; // Scrittura del risultato in memoria globale
}
```

# Qualificatore di Variabili e Tipi CUDA

## Esempi di utilizzo dei diversi tipi di memoria in CUDA

```
_int main() {
    const int arraySize = 256;
    float h_input[arraySize], h_output[arraySize];
    float *d_input, *d_output;

    // Inizializzazione dei dati e allocazione della memoria
    for (int i = 0; i < arraySize; ++i) h_input[i] = 1.0f * i;
    cudaMalloc(&d_input, arraySize * sizeof(float));
    cudaMalloc(&d_output, arraySize * sizeof(float));
    cudaMemcpy(d_input, h_input, arraySize* sizeof(float), cudaMemcpyHostToDevice);

    float newGlobalFactor = 3.0f; // Modifica della variabile globale `globalFactor` in memoria device
    cudaMemcpyToSymbol(globalFactor, &newGlobalFactor, sizeof(float));

    float hostConstVal = 2.0f; // Inizializzazione della memoria costante dall'host
    cudaMemcpyToSymbol(constVal, &hostConstVal, sizeof(float));

    memoryExample<<<1, 256>>>(d_input, d_output); // Esecuzione del kernel

    // Recupero dei risultati
    cudaMemcpy(h_output, d_output, arraySize* sizeof(float), cudaMemcpyDeviceToHost);

    // Liberazione della memoria
    cudaFree(d_input);
    cudaFree(d_output);
    return 0;
}
```

# Panoramica del Modello di Memoria CUDA

## ➤ Modelli di Performance

- Memory-Bound vs Compute-Bound
- Intensità Aritmetica e Roofline Model

## ➤ Gerarchia di Memoria CUDA

- Organizzazione Gerarchica Completa
- Scope e Programmabilità

## ➤ Gestione della Memoria Host-Device

- Allocazione e Trasferimenti
- Pinned Memory
- Zero-Copy Memory
- UVA (Unified Virtual Addressing)
- Unified Memory (UM)

## ➤ Global Memory

- Pattern di Accesso
- Lettura Cached vs Uncached
- Scrittura

## ➤ Shared Memory

- Memory Banks
- Modalità di Accesso e Bank Conflicts

# Gestione della Memoria in CUDA

## Somiglianze con il C, ma con una Responsabilità Aggiuntiva

- Come in C, il programmatore deve **allocare e deallocare memoria** manualmente.
- In più, è necessario **gestire esplicitamente il trasferimento dei dati tra host e device**, operazione cruciale per il corretto funzionamento delle applicazioni CUDA.

## Operazioni Chiave per la Gestione della Memoria

- CUDA offre strumenti per preparare la memoria del device nel codice host, gestendo le risorse necessarie al kernel.
- **Allocazione/Deallocazione sul Device:** Richiede funzioni specifiche come `cudaMalloc()` e `cudaFree()`.
- **Trasferimento Dati:** Movimentazione esplicita dei dati tramite il **bus PCIe**, utilizzando funzioni come `cudaMemcpy()`.

## Limiti della Gestione Manuale

- **Overhead nei Trasferimenti:** La comunicazione tra host e device via PCIe può essere lenta e introduce latenza.
- **Codice Complesso:** Necessità di gestire manualmente ogni fase, aumentando la complessità e il rischio di errori.
- **Sincronizzazione:** Garantire la coerenza tra le memorie può essere non banale.

## Evoluzione verso la Memoria Unificata

- NVIDIA ha gradualmente unificato nel tempo gli spazi di memoria di host e device.
- Tuttavia, per la maggior parte delle applicazioni, il trasferimento manuale dei dati rimane ancora un requisito.
- Le ultime novità in questo ambito (es., **Unified Memory**) saranno trattate nelle prossime slide.

# Allocazione della Memoria sul Device

## Ruolo della Funzione

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

## Firma della Funzione ([Documentazione Online](#))

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

## Parametri

- **devPtr**: Puntatore doppio che conterrà l'indirizzo della memoria allocata sulla GPU.
- **size**: Dimensione in byte della memoria da allocare.

## Valore di Ritorno

- **cudaError\_t**: Codice di errore (**cudaSuccess** se l'allocazione ha successo).

## Note Importanti

- **Allocazione**: Riserva memoria lineare contigua sulla GPU a **runtime**.
- **Puntatore**: Aggiorna puntatore CPU con indirizzo memoria GPU.
- **Stato iniziale**: La memoria allocata non è inizializzata.

# Allocazione della Memoria sul Device

## Esempio di Allocazione di Memoria sulla GPU

- Mostra come allocare memoria sulla GPU utilizzando **cudaMalloc**.

```
float* d_array; // Dichiarazione di un puntatore per la memoria sul device (GPU)

size_t size = 10 * sizeof(float); // Calcola la dimensione della memoria da allocare (10 float)

// Allocazione della memoria sul device
cudaError_t err = cudaMalloc((void**) &d_array, size);

// Controlla se l'allocazione della memoria ha avuto successo
if (err != cudaSuccess) {
    // Se c'è un errore, stampa un messaggio di errore con la descrizione dell'errore
    printf("Errore nell'allocazione della memoria: %s\n", cudaGetErrorString(err));
} else {
    // Se l'allocazione ha successo, stampa un messaggio di conferma
    printf("Memoria allocata con successo sulla GPU.\n");
}
```

# Allocazione della Memoria sul Device

## Ruolo della Funzione

- **cudaMemset** è una funzione CUDA utilizzata per impostare un valore specifico in un blocco di memoria allocato sulla GPU (device).

## Firma della Funzione ([Documentazione Online](#))

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

## Parametri

- **devPtr**: Puntatore alla memoria allocata sulla GPU.
- **value**: Valore da impostare in ogni byte della memoria.
- **count**: Numero di byte della memoria da impostare al valore specificato.

## Valore di Ritorno

- **cudaError\_t**: Codice di errore (**cudaSuccess** se l'inizializzazione ha successo).

## Note Importanti

- **Utilizzo**: Comunemente utilizzata per azzerare la memoria (impostando **value** a 0).
- **Gestione**: L'inizializzazione deve avvenire dopo l'allocazione della memoria tramite **cudaMalloc**.
- **Efficienza**: È preferibile usare **cudaMemset** per grandi blocchi di memoria per ridurre l'overhead.

# Trasferimento Dati

## Ruolo della Funzione

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

## Firma della Funzione ([Documentazione Online](#))

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

## Parametri

- **dst**: Puntatore alla memoria di destinazione.
- **src**: Puntatore alla memoria sorgente.
- **count**: Numero di byte da copiare.
- **kind**: Direzione della copia.

## Tipi di Trasferimento (kind)

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

## Valore di Ritorno

- **cudaError\_t**: Codice di errore (**cudaSuccess** se il trasferimento ha successo).

## Note importanti

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

# Trasferimento Dati

## Ruolo della Funzione

- `cudaMemcpy`: trasferisce dati dal device all'host.

## Firma della Funzione

`cudaError_t cudaMemcpy(...)`

## Parametri

- `dst`: destinazione (host o device).
- `src`: sorgente (host o device).
- `count`: numero di byte da trasferire.
- `kind`: tipo di memoria (host, device, unified).

## Valore di Ritorno

- `cudaError_t`: codice di errore.

## Note importanti

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

## Spazi di Memoria Differenti

- **Attenzione:** I puntatori del device non devono essere dereferenziati nel codice host (spazi di memoria CPU e GPU differenti).
- **Esempio:** Assegnazione errata come:

`host_array = dev_ptr`

invece di

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

- **Conseguenza dell'errore:** L'applicazione potrebbe bloccarsi durante l'esecuzione a causa del tentativo di accesso a uno spazio di memoria non valido.
- **Soluzione:** CUDA 6 ha introdotto la Memoria Unificata (Unified Memory), che consente di accedere sia alla memoria CPU che GPU utilizzando un unico puntatore (prossime slide).

e, o

vice

# Deallocazione della Memoria sul Device

## Ruolo della Funzione

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

## Firma della Funzione ([Documentazione Online](#))

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

## Parametri

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

## Valore di Ritorno

- **cudaError\_t**: Codice di errore (**cudaSuccess** se la deallocazione ha successo).

## Note Importanti

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

# Allocazione e Trasferimento Dati sul Device

## Esempio di Allocazione e Trasferimento Dati (1/2)

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

```
size_t size = 10 * sizeof(float); // Calcola la dimensione della memoria da allocare (10 float)
float* h_data = (float*)malloc(size); // Alloca memoria sull'host (CPU) per memorizzare i dati
for (int i = 0; i < 10; ++i) h_data[i] = (float)i; // Inizializza ogni elemento di h_data

float* d_data; // Dichiarazione di un puntatore per la memoria sulla GPU (device)
cudaMalloc((void**)&d_data, size); // Allocazione della memoria sulla GPU

// Copia dei dati dalla memoria dell'host (CPU) alla memoria del device (GPU)
cudaError_t err = cudaMemcpy(d_data, h_data, size, cudaMemcpyHostToDevice);

// Controlla se la copia è avvenuta con successo
if (err != cudaSuccess) {
    // Se c'è un errore, stampa un messaggio di errore e termina il programma
    fprintf(stderr, "Errore nella copia H2D: %s\n", cudaGetErrorString(err));
    exit(EXIT_FAILURE);
}
// continua
```

# Allocazione e Trasferimento Dati sul Device

## Esempio di Allocazione e Trasferimento Dati (2/2)

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

```
// Esegui operazioni sulla memoria della GPU (d_data)
// (Le operazioni specifiche da eseguire non sono mostrate in questo esempio)

// Copia dei risultati dalla memoria della GPU (device) alla memoria dell'host (CPU)
err = cudaMemcpy(h_data, d_data, size, cudaMemcpyDeviceToHost);

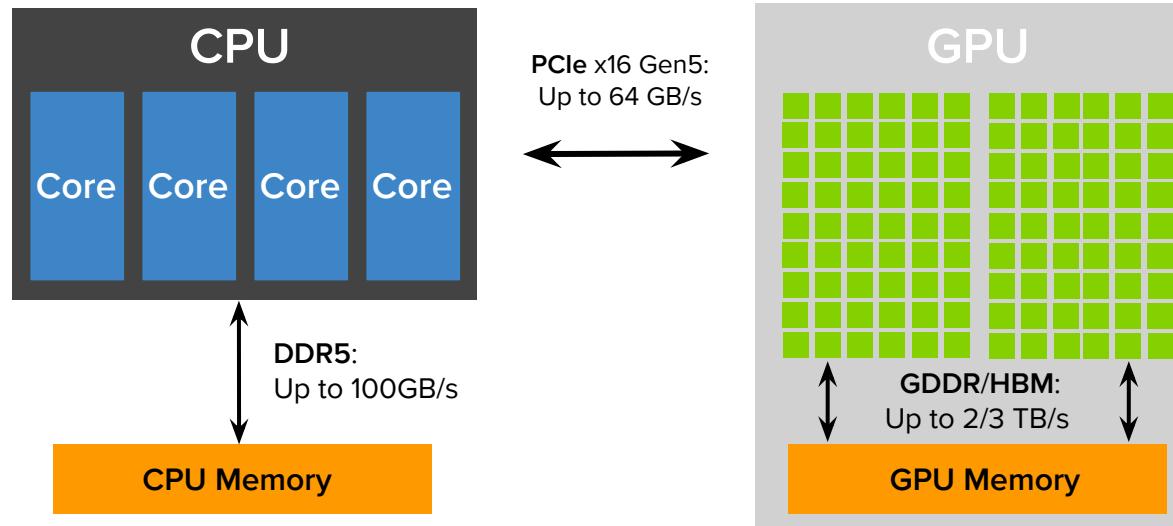
// Controlla se la copia è avvenuta con successo
if (err != cudaSuccess) {
    fprintf(stderr, "Errore nella copia D2H: %s\n", cudaGetErrorString(err));
    exit(EXIT_FAILURE);
}

free(h_data); // Libera la memoria allocata sull'host
cudaFree(d_data); // Libera la memoria allocata sulla GPU
```

# Connettività Host-Device e Throughput di Memoria

## Punti chiave

- La memoria GDDR della GPU offre una larghezza di banda teorica più alta (fino a 2-3 TB/s per HBM).
- Il collegamento PCIe ha una larghezza di banda teorica massima di 64 GB/s (per PCIe x16 Gen5).
- Significativa differenza tra la larghezza di banda della memoria GPU e quella del PCIe.
- I trasferimenti di dati tra host e dispositivo possono rappresentare un **collo di bottiglia**.
- Essenziale ridurre al minimo i trasferimenti di dati tra host e dispositivo.



# Connettività Host-Device e Throughput di Memoria

- Analisi dell'impatto della frammentazione dei trasferimenti di memoria CPU-GPU (100MB totali).
- Hardware per l'esperimento: **NVIDIA RTX 3090Ti GPU**, PCIe x 16 3.0 (**X299 PRO MS-7B94 Motherboard**).

```
// 1: Trasferimento unico grande (100 MB)
cudaMemcpy(d_data, h_data, total_size, cudaMemcpyHostToDevice);
```

**9,47 GB/s (1 x 100MB)**

```
// 2: N trasferimenti da 100MB/N ciascuno
int num_transfers = 10; // 100, 1000, 10000, 100000
size_t chunk_size = total_size / num_transfers;
for(int i = 0; i < num_transfers; i++) {
    cudaMemcpy(d_data + i*(chunk_size/sizeof(double)),
               h_data + i*(chunk_size/sizeof(double)),
               chunk_size,
               cudaMemcpyHostToDevice);
}
```

**8,20 GB/s (10 x 10MB)**

**4,47 GB/s (100 x 1MB)**

**2,35 GB/s (1K x 100KB)**

**900 MB/s (10K x 10KB)**

**100 MB/s (100K x 1KB)**

## Raccomandazione

- Aggregare i piccoli trasferimenti di dati in un **unico trasferimento di dimensioni maggiori** per ridurre l'impatto della latenza associata a ciascuna operazione e massimizzare l'efficienza della larghezza di banda.

# Memoria Pinned in CUDA

## Memoria Pageable:

- La memoria allocata dall'host di default è **pageable** (soggetta a *page fault*).
- Il **sistema operativo** può spostare i dati della memoria virtuale host in diverse locazioni fisiche.
- La GPU non può accedere in modo sicuro alla memoria host pageable (mancanza di controllo sui page fault).

## Come avviene allora il trasferimento da Memoria Pageable?

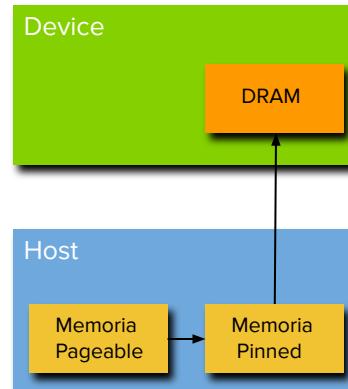
- Il **driver CUDA** alloca temporaneamente memoria host pinned (*page-locked* o *pinned*, bloccata in RAM).
- **Copia** i dati dalla memoria host sorgente alla memoria pinned.
- **Trasferisce** i dati dalla memoria pinned alla memoria del device.

Può dare problemi di inefficienza

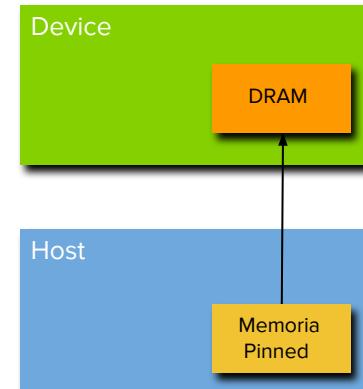
## Soluzione: Memoria Pinned

- **`cudaMallocHost()`** alloca direttamente memoria **host page-locked**, accessibile al device.
- Lettura/scrittura con **larghezza di banda** più elevata rispetto alla memoria pageable.
- **Elimina la necessità di copie intermedie**, migliorando la velocità dei trasferimenti dati.
- **Attenzione:** Allocare troppa memoria pinned può degradare le prestazioni del sistema host.

Pagable Data Transfer



Pinned Data Transfer



# Memoria Pinned in CUDA

## Memoria Pageable:

- La memoria allocata dall'host di default è **pageable** (soggetta a *page fault*).
- Il **sistema operativo** può spostare i dati della memoria virtuale host in diverse locazioni fisiche.
- La **GPU non può accedere in modo sicuro alla memoria host pageable** (mancanza di controllo sul *page fault*).

## Cor

## Memoria Paginabile

### Significa che la memoria allocata per l'host:

- Non è necessariamente residente fisicamente nella **RAM** in ogni momento.
- Può essere **spostata** su disco dal sistema operativo tramite il meccanismo di **paging**.
- Un accesso a una pagina non presente in RAM causa un **page fault**.

## Solu

### Page Fault:

- Un page fault è **un'interruzione** che si verifica quando un programma tenta di accedere a una pagina di memoria che non è attualmente caricata nella RAM. Il sistema operativo deve allora **recuperare** la pagina dal disco rigido, causando un ritardo significativo.

### Conseguenza:

- La GPU non può accedere direttamente alla memoria soggetta a page fault, perché il suo indirizzo fisico potrebbe **cambiare** in qualsiasi momento.

- **Attenzione:** Allocare troppa memoria pinned può degradare le prestazioni del sistema host.



# Memoria Pinned in CUDA

## Allocazione di Memoria Pinned

- `cudaMallocHost` alloca memoria host pinned (page-locked), che non può essere spostata dal sistema operativo e permette trasferimenti dati ed elaborazione **asincrona** tra host e device.

```
cudaError_t cudaMallocHost(void **devPtr, size_t count);
```

- **dstPtr**: Puntatore alla memoria di destinazione.
- **count**: Numero di byte da copiare.

## Vantaggi

- Trasferimenti dati ad **alta velocità** tra host e device.
- **Evita la necessità di copiare i dati** in una regione di memoria intermedia prima del trasferimento.

## Svantaggi

- Allocazione eccessiva riduce la memoria disponibile, **peggiorando le prestazioni del sistema** in caso di alta pressione sulla RAM.

## Esempio di Allocazione Memoria Pinned

```
cudaError_t status = cudaMallocHost((void**) &h_aPinned, bytes);
if (status != cudaSuccess) {
    fprintf(stderr, "Errore durante l'allocazione della memoria host pinned \n");
    exit(1);
// ... Utilizzo di h_aPinned per i trasferimenti di dati ...
cudaFreeHost(h_aPinned); // Libera la memoria allocata
```

# Memoria Pinned in CUDA

## Senza Memoria Pinned

```
// alocca la memoria sull'host  
h_a = (float *)malloc(nbytes);  
  
// alocca la memoria sul device  
CHECK(cudaMalloc((float **)&p_a, nbytes));  
  
// trasferisce i dati dall'host al device  
CHECK(cudaMemcpy(p_a, h_a, nbytes,  
cudaMemcpyHostToDevice));  
  
// trasferisce i dati dal device all'host  
CHECK(cudaMemcpy(h_a, p_a, nbytes,  
cudaMemcpyDeviceToHost));
```

## Con Memoria Pinned

```
// alocca memoria pinned sull'host  
CHECK(cudaMallocHost((float **)&h_a, nbytes));  
  
// alocca la memoria sul device  
CHECK(cudaMalloc((float **)&p_a, nbytes));  
  
// trasferisce i dati dall'host al device  
CHECK(cudaMemcpy(p_a, h_a, nbytes,  
cudaMemcpyHostToDevice));  
  
// trasferisce i dati dal device all'host  
CHECK(cudaMemcpy(h_a, p_a, nbytes,  
cudaMemcpyDeviceToHost));
```

### Velocità di Trasferimento

- Generalmente più lenta a causa della copia intermedia.

### Trasferimenti di Memoria

- Richiede una copia intermedia in un buffer di sistema prima del trasferimento al dispositivo via PCIe.

### Possibilità di Trasferimenti Asincroni

- Non supporta nativamente trasferimenti asincroni.

### Impatto sulle Risorse di Sistema

- Memoria non bloccata, più flessibile per il sistema.

### Velocità di Trasferimento

- Più veloce, specialmente per grandi trasferimenti di dati.

### Trasferimenti di Memoria

- Permette trasferimenti diretti tra host e device tramite **DMA (Direct Memory Access)** su bus PCIe.

### Possibilità di Trasferimenti Asincroni

- Supporta trasferimenti asincroni.

### Impatto sulle Risorse di Sistema

- Può ridurre la memoria disponibile per altre applicazioni.

# Dati Nsight System: Memoria Pinned vs Non Pinned

## Obiettivo

- Analizzare l'impatto della memoria pinned sulle **prestazioni di trasferimento** dati in CUDA.
- Confronto diretto** tra trasferimenti con memoria pinned e non pinned.
- Misurazione accurata dei tempi di trasferimento con **Nsight Systems**.
- Device: **NVIDIA GeForce RTX 3090**, Memoria Globale: 23.69 GB, Dati da Trasferire: 10 Gb

Metrica	Allocazione Non Pinned (ms)	Allocazione Pinned (ms)	Miglioramento Pinned
<code>cudaMallocHost</code> time	N/A	2211.533 ms	
<code>cudaMalloc</code> time	69.807 ms	0.412 ms	
<code>cudaMemcpy</code> H2D time	1128.165 ms	870.038 ms	↑ 22.88%
<code>cudaMemcpy</code> D2H time	952.819 ms	814.713 ms	↑ 14.49%
<code>cudaFree</code> time	3.088 ms	3.177 ms	↓ -2.88%
<code>cudaFreeHost</code> time	N/A	857.630 ms	-
Tempo totale <code>cudaMemcpy</code>	2080.984 ms	1684.751 ms	↑ 19.04%

- La memoria pinned è **più costosa da allocare/deallocare** rispetto a quella paginabile, ma **accelera i trasferimenti di grandi volumi di dati**, soprattutto se ripetuti dallo stesso buffer, ammortizzando il costo iniziale.

# Memoria Zero-Copy

## Di cosa si tratta?

- La memoria "**Zero-Copy**" è una tecnica che consente al device di **accedere direttamente alla memoria dell'host senza la necessità di copiare esplicitamente i dati** tra le due memorie.
- È un'**eccezione** alla regola che l'host non può accedere direttamente alle variabili del dispositivo e viceversa.

## Accesso alla Memoria Zero-Copy

- Sia l'**host** che il **device** possono accedere alla memoria zero-copy.
- Gli accessi alla memoria zero-copy dal device **avvengono direttamente tramite PCIe**, con trasferimenti dati eseguiti implicitamente **quando richiesti dal kernel**, senza necessità di trasferimenti esplicativi tra host e device.

## Vantaggi

- **Sfruttamento della memoria host:** Consente di usare la memoria dell'host quando quella del device è insufficiente.
- **Eliminazione trasferimenti esplicativi:** Evita la necessità di trasferire esplicitamente i dati tra host e device, semplificando il codice e riducendo l'overhead di gestione della memoria.
- **Accesso diretto:** Utile per dati a cui si accede raramente o una sola volta, evitando copie non necessarie in memoria device e riducendo l'occupazione della memoria GPU.

## Sincronizzazione

- Gli accessi alla memoria **devono essere sincronizzati** tra host e device per evitare comportamenti indefiniti.

# Memoria Zero-Copy

## Allocazione

- La memoria zero-copy è **memoria pinned** dell'host che è mappata nello spazio degli indirizzi del device.
- Per creare una regione di memoria zero-copy:

```
cudaError_t cudaHostAlloc(void **pHost, size_t count, unsigned int flags);
```

- La funzione alloca **count** byte di memoria host che è page-locked e accessibile dal device.
- La memoria allocata con **cudaHostAlloc()** deve essere liberata utilizzando **cudaFreeHost()**.

## Flag

- cudaHostAllocDefault**: Comportamento identico a **cudaMallocHost()**.
- cudaHostAllocPortable**: Ritorna memoria pinned utilizzabile da tutti i contesti CUDA.
- cudaHostAllocWriteCombined**: Memoria write-combined per trasferimenti PCIe più rapidi (dati non cached).
- cudaHostAllocMapped**: Memoria dell'host mappata nello spazio di indirizzo del device (memoria zero-copy).

## Come ottenere il puntatore device per la memoria pinned?

```
cudaError_t cudaHostGetDevicePointer(void **pDevice, void *pHost, unsigned int flags);
```

## Note

- Utilizzare la memoria zero-copy per operazioni di lettura e scrittura frequenti o con grandi blocchi di dati può **rallentare significativamente le prestazioni** perché ogni transazione alla memoria mappata passa per il bus PCIe.

# Memoria Zero-Copy

1/2

## Somma di Due Array

```
int main(int argc, char **argv) {
    // Configurazione del dispositivo
    int dev = 0;
    cudaSetDevice(dev);

    // Verifica del supporto per la memoria mappata
    cudaDeviceProp deviceProp;
    cudaGetDeviceProperties(&deviceProp, dev);
    if (!deviceProp.canMapHostMemory) {
        printf("Il device %d non supporta la mappatura della memoria host\n", dev);
        return EXIT_FAILURE;
    }

    // Impostazione delle dimensioni
    int nElem = 1<<20;
    size_t nBytes = nElem * sizeof(float);

    // Allocazione della memoria zero-copy
    float *h_A, *h_B, *h_C;
    cudaHostAlloc((void **) &h_A, nBytes, cudaHostAllocMapped);
    cudaHostAlloc((void **) &h_B, nBytes, cudaHostAllocMapped);
    cudaHostAlloc((void **) &h_C, nBytes, cudaHostAllocMapped);
```

# Memoria Zero-Copy

## Somma di Due Array

2/2

```
// Inizializzazione dei dati
initialData(h_A, nElem);
initialData(h_B, nElem);

// Passaggio dei puntatori al device
float *d_A, *d_B, *d_C;
cudaHostGetDevicePointer((void **) &d_A, (void *) h_A, 0);
cudaHostGetDevicePointer((void **) &d_B, (void *) h_B, 0);
cudaHostGetDevicePointer((void **) &d_C, (void *) h_C, 0);

// Non è necessario trasferire i dati verso il device

// Esecuzione del kernel (nessuna modifica) dopo l'inizializzazione
dim3 block(256);
dim3 grid((nElem + block.x - 1) / block.x);
sumArraysZeroCopy<<<grid, block>>>(d_A, d_B, d_C, nElem);

cudaDeviceSynchronize(); // Sincronizzazione del dispositivo

cudaFreeHost(h_A); cudaFreeHost(h_B); cudaFreeHost(h_C);

return EXIT_SUCCESS;
}
```

# Dati Nsight Systems: Memoria Zero-Copy

- Confronto per la somma di array ( $2^{20}$  elementi): Utilizzo della memoria del device vs memoria zero-copy.
- Analisi: Tempi di allocazione memoria e trasferimento dati per cudaMalloc + cudaMemcpy vs cudaHostAlloc (zero-copy).
- Performance: Tempi di kernel per la somma degli array, con focus su memoria host (zero-copy) vs memoria GPU.

Dimensione Vettore:  $2^{20}$  elementi   Dimensione Blocco: 256   Device: RTX 3090 (GPU)

Funzione	Tempo Totale (ms)	Chiamate	Tempo Medio (ms)
cudaHostAlloc	56,103	3	18,701
cudaFreeHost	1,355	3	0,451
cudaDeviceSynchronize	0,715	1	0,715
sumArraysZeroCopy (kernel)	0,712	1	0,712

Funzione	Tempo Totale (ms)	Chiamate	Tempo Medio (ms)
cudaMalloc	51,437	3	17,145
cudaMemcpy	1,520	3	0,506
cudaFree	0,374	3	0,124
cudaDeviceSynchronize	0,075	1	0,075
sumArrays (kernel)	0,015	1	0,015

# Dati Nsight Systems: Memoria Zero-Copy

- Confronto per la somma di array: Utilizzo della *memoria del device* vs *memoria zero-copy*.
- Prestazioni del kernel per array di diverse dimensioni.

Dimensione Blocco: 256   Device: RTX 3090 (GPU)

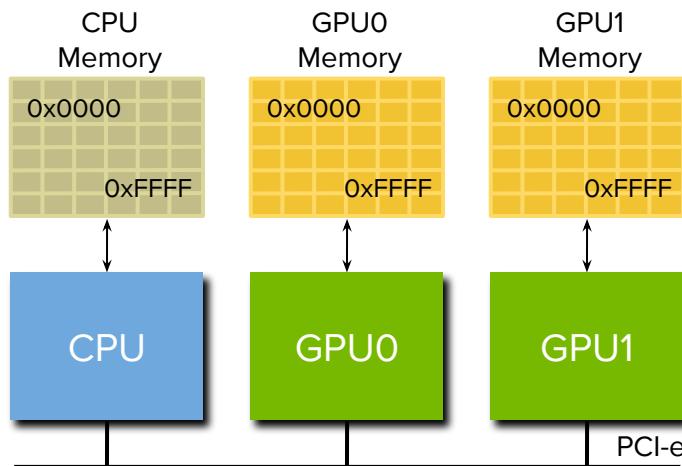
Dimensione Array	Device Memory (ns)	Zero-Copy Memory (ns)	Slowdown
1K	1.728	4.448	2,574
4K	1.664	6.880	4,134
16K	1.792	14.880	8,304
64K	2.048	67.584	33,00
256K	3.232	203.744	63,04
1M	15.360	712.446	46,38
4M	60.544	2.751.033	45,44
16M	237.023	10.913.415	46,04
64M	943.101	43.529.977	46,16
256M	3.753.496	173.990.111	46,35

- Per **piccoli dati** condivisi, la memoria zero-copy semplifica la programmazione e offre prestazioni ragionevoli.
- Per **dataset grandi** su GPU discrete via PCIe, la zero-copy causa un forte degrado delle prestazioni.

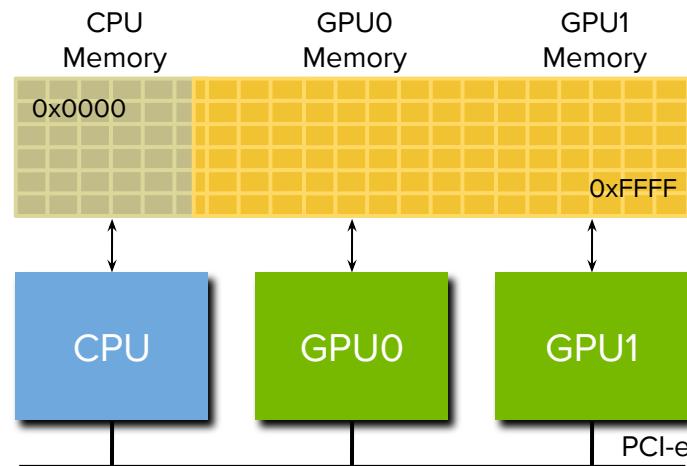
# Unified Virtual Addressing (UVA)

## Cosa è?

- La Unified Virtual Addressing (UVA), o Indirizzamento Virtuale Unificato, è una tecnica che permette alla CPU e alla GPU di condividere lo stesso spazio di indirizzamento virtuale (la memoria fisica rimane distinta).
- Introdotta in CUDA 4.0 per dispositivi con compute capability 2.0+ e sistemi Linux e Windows a 64-bit.
- Non vi è distinzione** tra un puntatore virtuale host e uno device.
- Il sistema di runtime di CUDA **gestisce automaticamente la mappatura degli indirizzi virtuali agli indirizzi fisici** nella memoria della CPU o della GPU, a seconda delle necessità.



No UVA: spazio di memoria multiplo

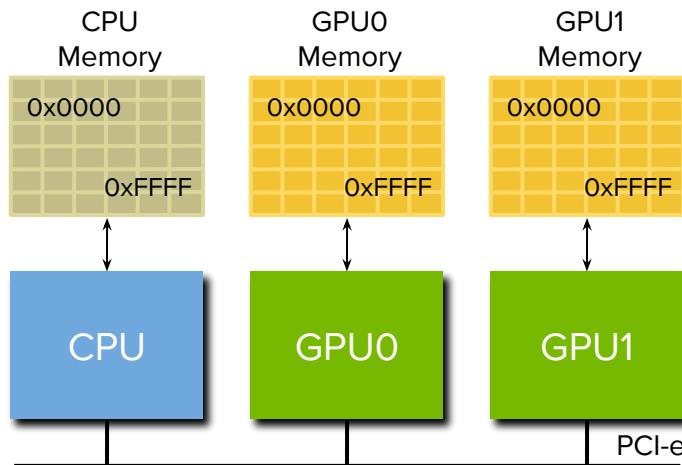


UVA: spazio di memoria singolo

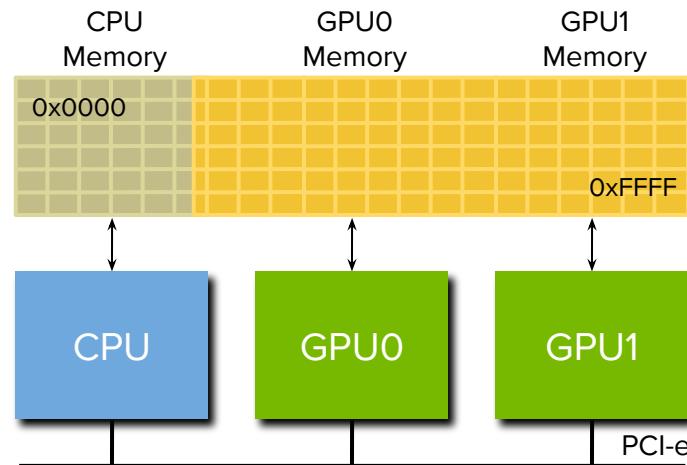
# Unified Virtual Addressing (UVA)

## Caratteristiche Principali

- Il runtime gestisce le mappature per `cudaMalloc` (device) e `cudaHostAlloc` (host) in uno spazio unificato.
- Non è ancora possibile dereferenziare un puntatore host sul dispositivo o viceversa (Eccezione: memoria zero-copy).**
- Il parametro `cudaMemcpyKind` di `cudaMemcpy` diventa obsoleto e può essere impostato su `cudaMemcpyDefault`, poiché il runtime gestisce automaticamente il tipo di memoria (host o device) a cui il puntatore fa riferimento.
- Con UVA, la memoria host zero-copy allocata con `cudaHostAlloc` ha puntatori host e device **identici**, consentendo di passare direttamente il puntatore al kernel senza bisogno di usare `cudaHostGetDevicePointer`.



No UVA: spazio di memoria multiplo



UVA: spazio di memoria singolo

# Unified Virtual Addressing (UVA)

- L'UVA semplificherebbe ulteriormente l'esempio della somma di array visto in precedenza con memoria zero-copy.

## Somma di due array con UVA e Memoria Zero-Copy (Pinned)

```
// Allocazione della memoria zero-copy (page-locked)
float *h_A, *h_B, *h_C;
cudaHostAlloc((void **) &h_A, nBytes, cudaHostAllocMapped);
cudaHostAlloc((void **) &h_B, nBytes, cudaHostAllocMapped);
cudaHostAlloc((void **) &h_C, nBytes, cudaHostAllocMapped);

// Inizializzazione dei dati
initialData(h_A, nElem);
initialData(h_B, nElem);

// Ottenimento dei puntatori del dispositivo per la memoria mappata
float *d_A, *d_B, *d_C;
cudaHostGetDevicePointer((void **) &d_A, (void *) h_A, 0);
cudaHostGetDevicePointer((void **) &d_B, (void *) h_B, 0);
cudaHostGetDevicePointer((void **) &d_C, (void *) h_C, 0);

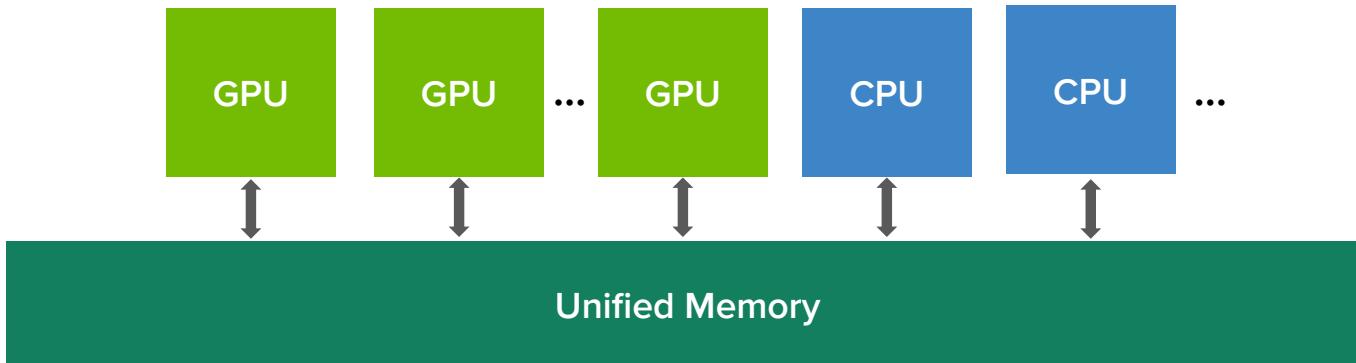
// Esecuzione del kernel
sumArraysZeroCopy<<<grid, block>>>(h_A, h_B, h_C, nElem); ✓
// continue...
```

Con l'UVA, non è necessario ottenere il puntatore del dispositivo o gestire due puntatori per quello che è fisicamente lo stesso dato.

# Unified Memory (UM)

## Cosa è?

- La **Unified Memory** (introdotta in CUDA 6.0) fornisce uno spazio di memoria virtuale unificato a 49 bit che permette di accedere agli stessi dati da tutti i processori del sistema usando un unico puntatore (**Single-pointer-to-data**).
- La memoria è gestita **automaticamente e dinamicamente dal CUDA Runtime** tramite il **Page Migration Engine**, che trasferisce i dati tra host e device tramite PCIe o NVLink quando necessario (**migrazione automatica dei dati**).
- Quando le GPU o CPU accedono a dati non residenti localmente, un **page fault** avvia il trasferimento automatico dei dati, gestito in modo **trasparente** dall'applicazione.
- L'uso della combinazione di **cudaHostAlloc** e **cudaMemcpy** non è più un requisito.
- **Utilizza la Managed Memory**, semplificando notevolmente il codice dell'applicazione e la gestione della memoria.



*Possibilità di allocare oltre le dimensioni della memoria della GPU (da CUDA 8.0+)*

# Memoria Gestita in Unified Memory

## Cosa è?

- La Memoria Gestita (**Managed Memory**) si riferisce alle **allocazioni di Unified Memory** che sono gestite automaticamente dal sistema sottostante e sono interoperabili con le allocazioni specifiche del device.

## Caratteristiche

- **Gestione Automatica:** Il sistema migra automaticamente i dati tra host e device, semplificando il codice.
- **Interoperabilità:** Completamente compatibile con le allocazioni specifiche del device (es. `cudaMalloc`), consentendo di utilizzare entrambi i tipi di memoria all'interno dello stesso kernel (*managed* e *unmanaged*).
- **Accesso Unificato:** Accessibile tramite lo stesso puntatore sia dal codice host che device, eliminando la necessità di trasferimenti di memoria espliciti.
- **Supporto Completo:** Le operazioni CUDA valide per la memoria del device funzionano anche con la Memoria Gestita.

## Metodi di Allocazione della Managed Memory

- **Statica:** Dichiarando variabili device con l'annotazione `__managed__` a livello di file o globale:  
`__device__ __managed__ int var;`
- **Dinamica:** Utilizzando la funzione runtime `cudaMallocManaged()`:  
`cudaError_t cudaMallocManaged(void **devPtr, size_t size, unsigned int flags=0);`
  - Il puntatore `devPtr` è valido su tutti i device e sull'host.

# Memoria Gestita in Unified Memory

## Cosa è?

- CUDA 6.0 non permette di chiamare `cudaMallocManaged` dal codice device.
- Deve essere chiamato dall'host oppure dichiarato staticamente nell'ambito globale.

Operazioni di Unified Memory che sono gestite bili con le allocazioni specifiche del device.

Accesisti tra host e device, semplificando il codice. Operazioni specifiche del device (es. `cudaMalloc`),

- consentendo di utilizzare entrambi i tipi di memoria all'interno dello stesso kernel (*managed* e *unmanaged*).
- **Accesso Unificato:** Accessibile tramite lo stesso puntatore per i trasferimenti di memoria espliciti.
  - **Supporto Completo:** Le operazioni CUDA valide per la memoria

Il flag indica chi condivide il puntatore con il device:

- `cudaMemAttachHost`: Solo la CPU.
- `cudaMemAttachGlobal`: Qualsiasi altra GPU.

## Metodi di Allocazione della Managed Memory

- **Statica:** Dichiarando variabili device con l'annotazione `_managed_` a livello di file o globale:

```
_device_ _managed_ int var;
```

- **Dinamica:** Utilizzando la funzione runtime `cudaMallocManaged()`:

```
cudaError_t cudaMallocManaged(void **devPtr, size_t size, unsigned int flags=0);
```

- Il puntatore `devPtr` è valido su tutti i device e sull'host.

# Allocazione e Migrazione in Unified Memory

## Allocazione su Richiesta

- L'allocazione fisica della memoria tramite `cudaMallocManaged` avviene in modo **lazy**: le pagine vengono allocate solo al primo utilizzo da parte di uno dei processori (CPU o GPU) nel sistema.
- Questo approccio ottimizza l'utilizzo della memoria evitando **allocazioni non necessarie a priori**.

## Migrazione Automatica delle Pagine

- Le pagine di memoria possono migrare dinamicamente tra CPU e GPU in base alle necessità.
- Il driver CUDA utilizza **euristiche intelligenti** per:
  - Mantenere la **località dei dati**.
  - **Minimizzare i page fault**.
  - **Ottimizzare le prestazioni complessive**.

## Controllo Programmabile ([Documentazione Online](#))

- Gli sviluppatori possono opzionalmente **guidare il comportamento del driver** usando:
  - `cudaMemAdvise` () fornisce al runtime CUDA suggerimenti su come accedere ai dati (lettura/scrittura), sulla loro posizione preferenziale e sul dispositivo principale che li utilizzerà. Non innescano il trasferimento.
  - `cudaMemPrefetchAsync` () consente di **migrare proattivamente i dati nella memoria del dispositivo target**, riducendo i page fault e preparando i dati prima dell'elaborazione.

# Semplificazione del Codice di Gestione della Memoria

Codice CPU

```
void sortfile(FILE *fp, int N) {  
    char *data;  
    data = (char *)malloc(N);  
  
    fread(data, 1, N, fp);  
  
    qsort(data, N, 1, compare);  
  
    use_data(data);  
  
    free(data);  
}
```

Codice CUDA Standard

```
void sortfile(FILE *fp, int N) {  
    char *data, *d_data;  
    data = (char *)malloc(N);  
    cudaMalloc(&d_data, N);  
  
    fread(data, 1, N, fp);  
    cudaMemcpy(d_data, data, N, ...); // 1  
    qsort<<<...>>>(data, N, 1, compare); // 2  
    cudaMemcpy(data, d_data, N, ...); // 3  
  
    use_data(data);  
    cudaFree(d_data);  
    free(data);  
}
```

# Semplificazione del Codice di Gestione della Memoria

Codice CPU

```
void sortfile(FILE *fp, int N) {  
    char *data;  
    data = (char *)malloc(N);  
  
    fread(data, 1, N, fp);  
  
    qsort(data, N, 1, compare);  
  
    use_data(data);  
  
    free(data);  
}
```

Codice CUDA con Unified Memory

```
void sortfile(FILE *fp, int N) {  
    char *data;  
    cudaMallocManaged(&data, N);  
  
    fread(data, 1, N, fp);  
  
    qsort<<<...>>>(data, N, 1, compare);  
  
    use_data(data);  
  
    cudaFree(data);  
}
```

# Semplificazione del Codice di Gestione della Memoria

Codice CPU

```
void sortfile(FILE *fp, int N) {  
    char *data;  
    data = (char *)malloc(N);  
  
    fread(data, 1, N, fp);  
  
    qsort(data, N, 1, compare);  
  
    use_data(data);  
  
    free(data);  
}
```

Codice CUDA con Unified Memory (CUDA 8+)

```
void sortfile(FILE *fp, int N) {  
    char *data;  
    data = (char *)malloc(N);  
  
    fread(data, 1, N, fp);  
  
    qsort<<<...>>>(data, N, 1, compare);  
  
    use_data(data);  
  
    free(data);  
}
```

\*Con supporto del Sistema Operativo

**Nota:** (CUDA 8+) Su piattaforme compatibili, la memoria allocata con l'allocatore del sistema operativo (`malloc` o `new`) è accessibile sia dal codice GPU che CPU usando lo stesso puntatore.

# Gestione dell'Accesso Concorrente alla Unified Memory

## Mutua Esclusione:

- Durante l'esecuzione di un kernel, la GPU ha accesso **esclusivo** alla memoria unificata.
- La CPU non può accedere alla memoria unificata fino a che la GPU non ha terminato il suo lavoro.

## `cudaDeviceSynchronize()`:

- Forza la CPU ad attendere la fine di tutti i compiti in esecuzione sulla GPU.
- Essenziale per **evitare conflitti di accesso** tra CPU e GPU.

## Errore di Accesso Concorrente alla Memoria Unificata

```
__device__ __managed__ int x, y = 2;

__global__ void mykernel() {
    // Modifica da parte della GPU
    x = 10; }

int main() {

    mykernel <<<1,1>>> ();

    // ERRORE: Accesso CPU durante l'esecuzione GPU
    y = 20;      return 0;
}
```

## Soluzione - Sincronizzazione Necessaria

```
__device__ __managed__ int x, y = 2;

__global__ void mykernel() {
    // Modifica da parte della GPU
    x = 10; }

int main() {

    mykernel <<<1,1>>> ();
    // Sincronizzazione CPU-GPU
    cudaDeviceSynchronize();
    // Ora l'accesso è sicuro
    y = 20;  return 0;
}
```

# Unified Memory (UM)

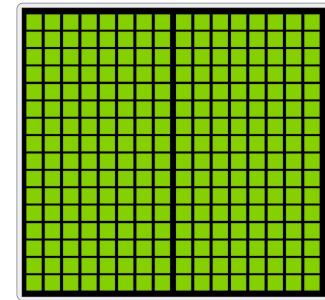
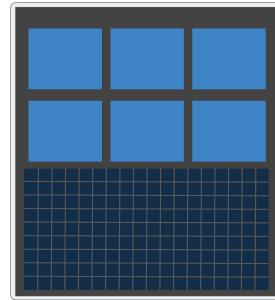
## Vantaggi

- **Allocazione unica, puntatore unico, accessibile ovunque.**
- **Elimina** la necessità di duplicare puntatori e la necessità di copie esplicite fra host e device.
- **Semplifica** la programmazione in CUDA.

## Svantaggi

- **Latenza aggiuntiva** dovuta alla gestione automatica delle migrazioni e page fault.
- **Controllo limitato** sul posizionamento dei dati in memoria.
- La gestione automatica del posizionamento potrebbe **non essere ottimale** per certe applicazioni.

CUDA 6+



Unified Memory

# Zero-Copy Memory, UVA, e Unified Memory

## Zero-Copy Memory

- **Obiettivo:** Evitare copie esplicite dei dati tra CPU e GPU, permettendo accessi diretti alla memoria dell'host.
- **Funzionamento:** Memoria allocata sull'host accessibile direttamente dalla GPU tramite il bus PCIe.
- **Vantaggio:** Riduce la latenza legata alla copia dei dati (utile per piccoli blocchi di dati).
- **Limite:** Prestazioni limitate dalla banda e latenza del PCIe, soprattutto per accessi frequenti a grandi quantità di dati.

## Unified Virtual Addressing (UVA)

- **Obiettivo:** Semplificare la gestione degli indirizzi di memoria nei sistemi eterogenei.
- **Funzionamento:** Crea uno spazio di indirizzamento virtuale unico condiviso da CPU e GPU.
- **Vantaggio:** Elimina la necessità di conversioni manuali dei puntatori.
- **Limite:** Non gestisce la migrazione dei dati, richiedendo trasferimenti manuali.

## Unified Memory (UM)

- **Obiettivo:** Semplificare la programmazione e ottimizzare le prestazioni attraverso una gestione automatica della memoria.
- **Funzionamento:** Basata su UVA, aggiunge la migrazione automatica e trasparente dei dati tra CPU e GPU.
- **Vantaggio:**
  - **Semplicità:** Modello "single-pointer-to-data" per un accesso unificato ai dati.
  - **Trasparenza:** Migrazione automatica dei dati per ottimizzare la località e ridurre i trasferimenti manuali.
- **Limitazioni:** Può causare overhead nella gestione automatica dei trasferimenti, e non è sempre ideale per applicazioni ad alte prestazioni che richiedono un controllo preciso sul posizionamento dei dati.

# Panoramica del Modello di Memoria CUDA

## ➤ Modelli di Performance

- Memory-Bound vs Compute-Bound
- Intensità Aritmetica e Roofline Model

## ➤ Gerarchia di Memoria CUDA

- Organizzazione Gerarchica Completa
- Scope e Programmabilità

## ➤ Gestione della Memoria Host-Device

- Allocazione e Trasferimenti
- Pinned Memory
- Zero-Copy Memory
- UVA (Unified Virtual Addressing)
- Unified Memory (UM)

## ➤ Global Memory

- Pattern di Accesso
- Lettura Cached vs Uncached
- Scrittura

## ➤ Shared Memory

- Memory Banks
- Modalità di Accesso e Bank Conflicts

# Pattern di Accesso alla Memoria

## Importanza della Memoria Globale

- La maggior parte delle applicazioni GPU è **limitata** dalla **larghezza di banda** della memoria DRAM.
- **Ottimizzare** l'uso della memoria globale è fondamentale per le prestazioni del kernel.
- Senza questa ottimizzazione, **altri miglioramenti** potrebbero avere **effetti trascurabili**.

## Modello di Esecuzione CUDA e Accesso alla Memoria

- **Istruzioni ed operazioni di memoria** sono emesse ed eseguite per warp (32 thread).
- Ogni thread fornisce un indirizzo di memoria quando deve leggere/scrivere, e la dimensione della richiesta del warp dipende dal tipo di dato (es.: 32 thread x 4 byte per **int**, 32 thread x 8 byte per **double**).
- La richiesta (lettura o scrittura) è servita da **una o più transazioni di memoria**.
- Una transazione è un'**operazione atomica** di lettura/scrittura tra la memoria globale e gli SM della GPU.

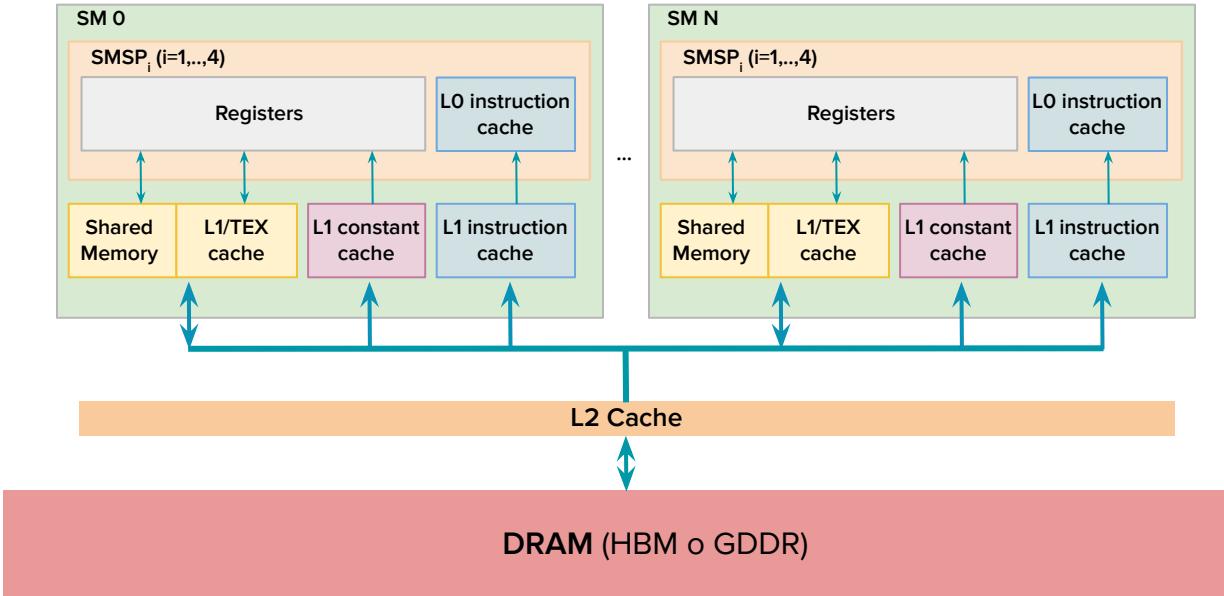
## Pattern di Accesso alla Memoria

- Gli accessi possono essere classificati in **pattern** basati sulla **distribuzione degli indirizzi** in un warp.
- Comprendere questi pattern è **vitale** per **ottimizzare** l'accesso alla memoria globale.
- L'obiettivo è raggiungere le migliori prestazioni nelle operazioni di **lettura e scrittura**.

# Architettura delle Memoria Globale

## Struttura della Memoria Globale

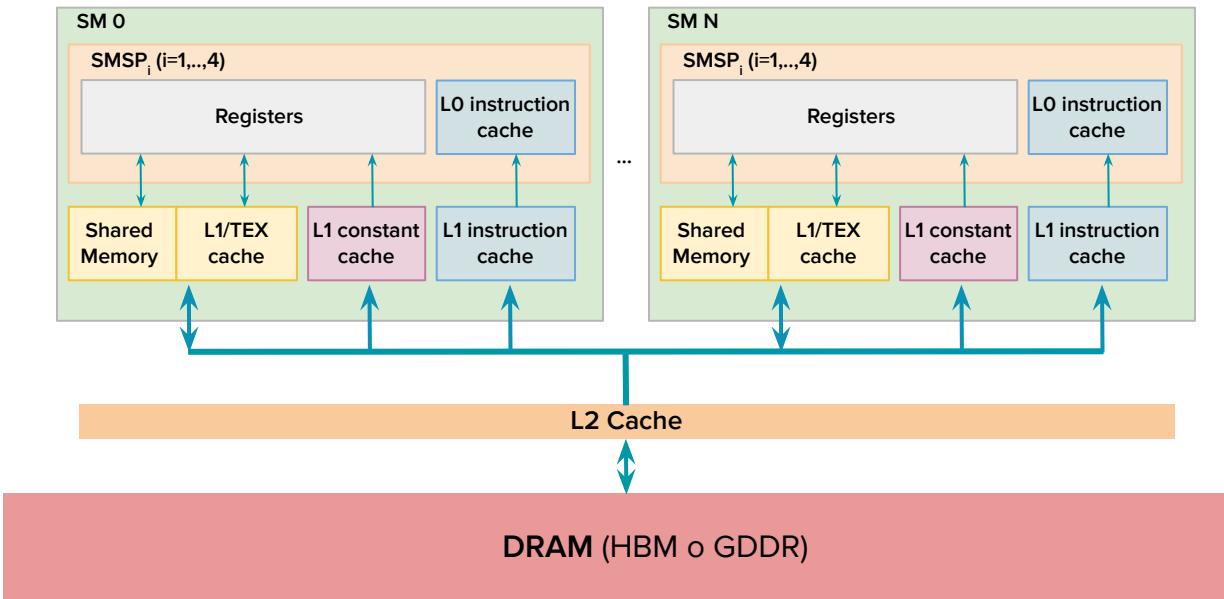
- La memoria globale è uno **spazio logico** accessibile da un kernel.
- I dati dell'applicazione risiedono inizialmente nella DRAM fisica del device, accessibile in lettura e scrittura tramite transazioni di memoria da 32, 64 o 128 byte.
- Le richieste di memoria del kernel sono **gestite tra la DRAM del device e la memoria on-chip SM**.



# Architettura delle Memoria Globale

## Transazioni di Memoria e Cache

- Le transazioni di memoria avvengono in blocchi di dimensioni variabili, come 128 byte o 32 byte (ad esempio).
- Tutti gli accessi alla memoria globale passano attraverso la **cache L2**.
- Molti accessi passano anche attraverso la **cache L1**, a seconda del tipo di accesso e dell'architettura GPU.



# Accessi Allineati e Coalescenti in CUDA

## Caratteristiche Ottimali degli Accessi alla Memoria

- Accessi **allineati** alla memoria.
- Accessi **coalescenti** alla memoria.

**Nota:** La memoria allocata tramite CUDA Runtime API, ad esempio con `cudaMalloc()`, è garantita essere allineata ad almeno 256 byte.

### Accessi Allineati alla Memoria

- L'indirizzo iniziale di una transazione di memoria è un **multiplo della dimensione della transazione stessa**.
- Gli accessi non allineati richiedono più transazioni, sprecando banda di memoria.

### Accessi Coalescenti alla Memoria

- Si verificano quando tutti i 32 thread in un warp accedono a un blocco contiguo di memoria.
- Se gli accessi sono contigui, l'hardware **può combinarli in un numero ridotto di transazioni verso posizioni consecutive nella DRAM**.
- Tuttavia, la coalescenza **da sola non è sufficiente** per ottimizzare l'accesso ai dati.

In algoritmi specifici, la coalescenza può essere **difficile o intrinsecamente impossibile** da ottenere.

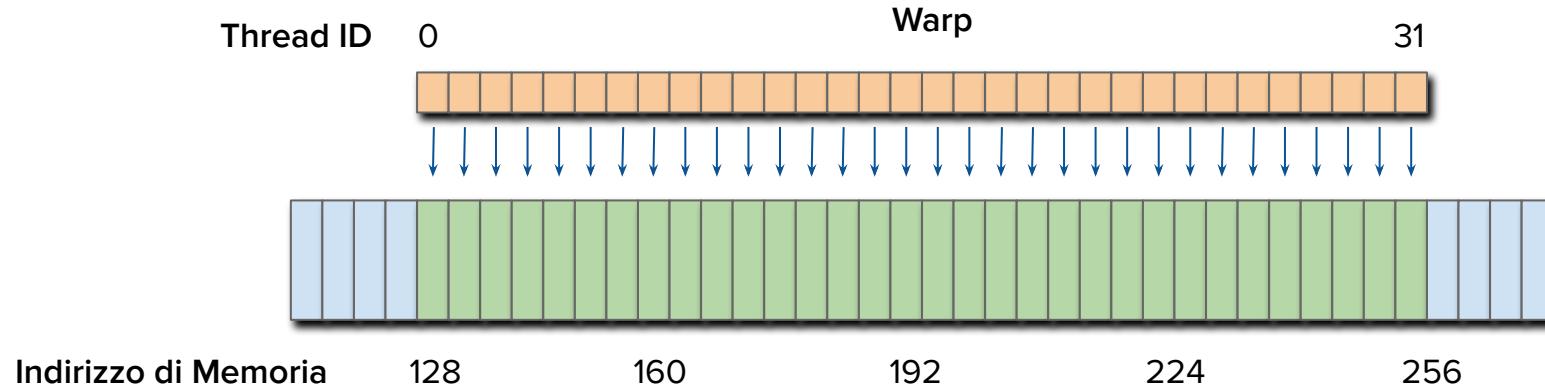
### Accessi Allineati e Coalescenti

- Un warp accede a un **blocco contiguo** di memoria partendo da un indirizzo allineato.
- **Ottimizza il throughput** della memoria globale e migliora le prestazioni complessive del kernel.
- Combinare accessi allineati e coalescenti è **fondamentale** per ottenere kernel dalle massime prestazioni.

# Accessi Allineati e Coalescenti in CUDA

## Esempio: Accesso Allineato e Coalescente ✓

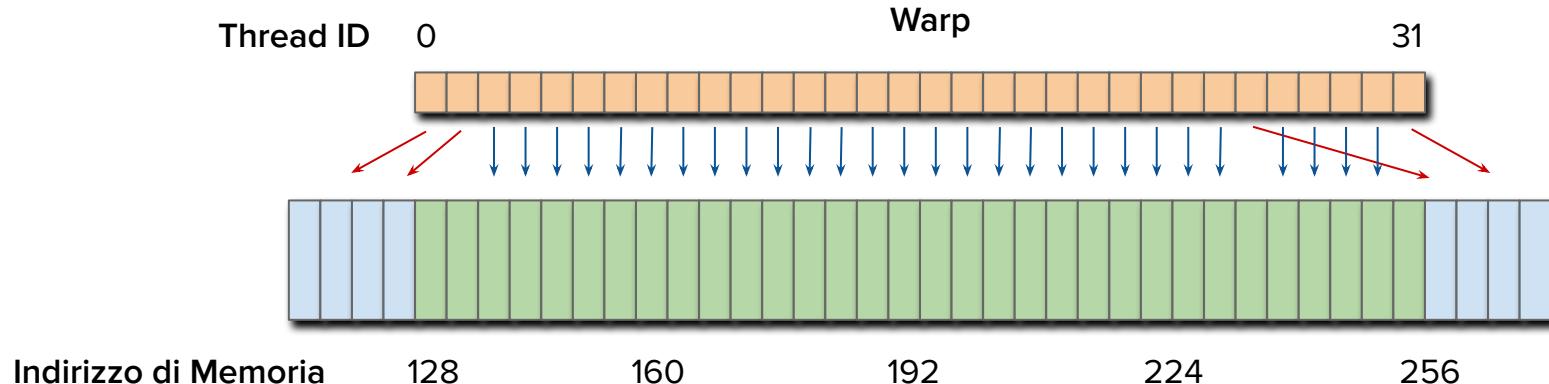
- Una **singola transazione** da 128 byte (4 byte per thread in un warp) recupera tutti i dati necessari.
- Utilizzo **ottimale** della larghezza di banda.
- Riduzione del numero totale di transazioni di memoria.
- **Minimizzazione** della latenza di accesso ai dati.



# Accessi Disallineati e Non Coalescenti in CUDA

## Esempio: Accesso Disallineato e Non Coalescente X

- Richiede tre transazioni da 128 byte per gli stessi dati.
- Causa significativo spreco di larghezza di banda (256 byte aggiuntivi caricati che non vengono poi utilizzati).
- Aumento del traffico di memoria (moltiplica le transazioni necessarie per accedere ai dati richiesti).
- Maggiore latenza (tempi di attesa più lunghi per il completamento di tutte le transazioni).



## Principio di Ottimizzazione

*Utilizzare il **minor numero di transazioni** per servire il **massimo numero di richieste di memoria**.*

# Lettura dalla Memoria Globale - Cached/Uncached

Fattori che influenzano il passaggio dei dati attraverso la Cache L1:

- Compute Capability del device.
- Opzioni del compilatore nvcc.

Comportamento su diverse GPU:

- Compute Capability 2.x+: Cache L1 abilitata di default.
- Compute Capability 3.5-5.2: Cache L1 disabilitata di default (usata solo in caso di *register spills*).
- Compute Capability  $\geq 6.0$ : Cache L1 è abilitata di default.

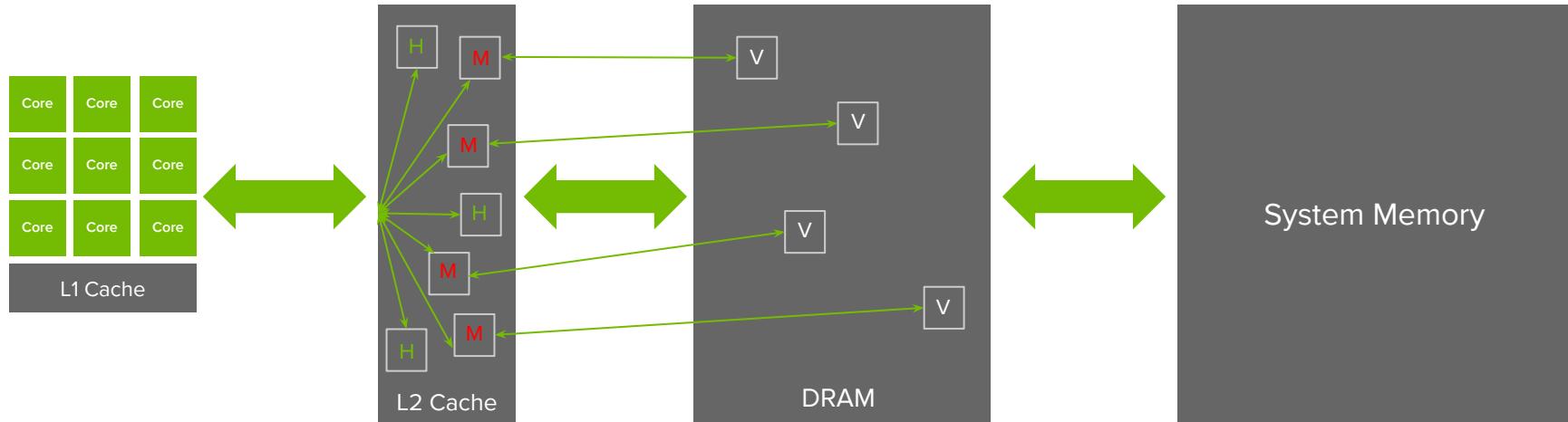
Controllo della cache L1 tramite flag del compilatore nvcc:

- Disabilitazione: **-Xptxas -dlcm=cg**
  - Tutti accessi alla global memory passano attraverso la cache L2; in caso di miss, sono serviti dalla DRAM.
  - Transazioni di memoria da 32 byte.
- Abilitazione: **-Xptxas -dlcm=ca**
  - Le richieste passano prima da L1, poi da L2 e infine dalla DRAM in caso di miss.
  - Transazioni di memoria minime da 128 byte con Compute Capability  $\leq 5.2$ , 32 byte altrimenti.

# Lettura dalla Memoria Globale - Cached/Uncached

## Cache e Accessi Memoria GPU

- Prima si cercano i dati nella cache L1 dell'SM (se abilitata); in caso di "cache miss" prosegue nella cache L2. Un "cache hit" in L2 comporta il trasferimento dei dati in L1 e successivamente ai registri dell'SM.
- Se il dato non è presente in L2 ("cache miss"), si accede alla DRAM. Se non presente neanche qui, il dato viene richiesto alla memoria di sistema.
- **Ogni accesso aggiuntivo** attraverso la gerarchia di memoria **rallenta le prestazioni** (introduce latenza) e aumenta il consumo energetico: migliorare il "cache hit rate" significa aumentare framerate ed efficienza.



# Cache L1: CPU vs GPU

## Cache L1 della CPU

- **Località Spaziale:** La cache della CPU è progettata per memorizzare dati che sono fisicamente vicini nella memoria, poiché i programmi tendono ad accedere a dati contigui.
- **Località Temporale:** La CPU è ottimizzata per riutilizzare i dati che sono stati recentemente usati, mantenendoli nella cache per un accesso rapido.
- **Scopo:** Le CPU sono progettate per eseguire una varietà di compiti diversi, molti dei quali beneficiano di entrambe le località.

## Cache L1 della GPU

- **Località Spaziale:** Le GPU elaborano grandi blocchi di dati in parallelo, e quindi tendono a beneficiare maggiormente dell'accesso a dati contigui.
- **Località Temporale:** Le GPU non si affidano alla località temporale allo stesso modo delle CPU, poiché elaborano molti thread che potrebbero accedere a dati differenti.
- **Scopo:** Le GPU sono ottimizzate per operazioni di elaborazione parallela massiccia dove l'accesso ai dati vicini è più importante che riutilizzare gli stessi dati nel tempo.

# Pattern di Accesso per il Caricamento dalla Memoria

## Tipi di Caricamento dalla Memoria

- **Cached Loads** (cache L1 abilitata)
  - Passa attraverso la cache L1 - le linee di cache sono da 128 byte (composte da 4 settori da 32 byte).
  - Transazioni di memoria a **granularità di 128 byte con compute capability ≤ 5.2**.
- **Uncached Loads** (cache L1 disabilitata)
  - Non passa attraverso la cache L1.
  - Transazioni di memoria a **granularità di 32 byte** (segmento di memoria).
- La granularità dei segmenti di memoria è 32 byte ma il numero di segmenti per transazione può variare, ad esempio: **32 byte su Pascal, 64 byte (2 settori con prefetch) su Volta, e configurabile a 32/64/128 su Ampere.**

## Caratterizzazione dei Pattern di Accesso

Il pattern di accesso ai caricamenti dalla memoria può essere caratterizzato dalle seguenti **combinazioni**:

- **Con Cache vs Senza Cache**
  - Il caricamento è con cache se la cache L1 è abilitata.
- **Allineato vs Disallineato**
  - Il caricamento è allineato se il primo indirizzo di accesso è multiplo della dimensione della transazione.
- **Coalescente vs Non Coalescente**
  - Il caricamento è coalescente se un warp accede a un blocco contiguo di dati.

# Pattern di Accesso per il Caricamento dalla Memoria

## Tipi di Caricamento dalla Memoria

- **Compute capability  $\geq 6.0$  (da Pascal)**
- *“The concurrent accesses of the threads of a warp will coalesce into a number of transactions equal to the number of 32-byte transactions necessary to service all of the threads of the warp.”*
- [\(Cuda-C-Best-Practices-Guide\)](#)

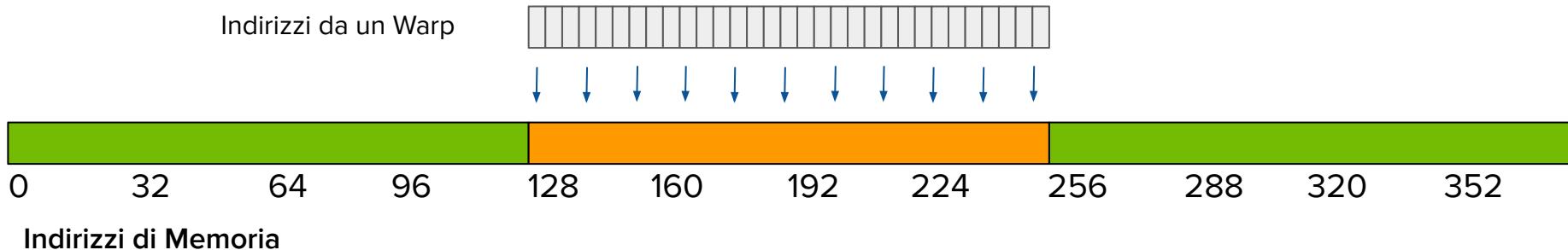
esempio: 32 byte su Pascal, 64 byte (2 settori con prefetch) su Volta, e configurabile a 32/64/128 su Ampere.

## Carico da Global Memory

- Il pattern di accesso alla memoria globale può essere:
  - **Compute capability  $\leq 5.2$  (prima di Pascal)**
    - *“L1-caching of accesses to global memory can be optionally enabled. If L1-caching is enabled on these devices, the number of required transactions is equal to the number of required 128-byte aligned segments.”*
  - **Coalescente vs Non Coalescente**
    - Il caricamento è coalescente se un warp accede a un blocco contiguo di dati.

# Cached Loads: Allineato e Coalescente

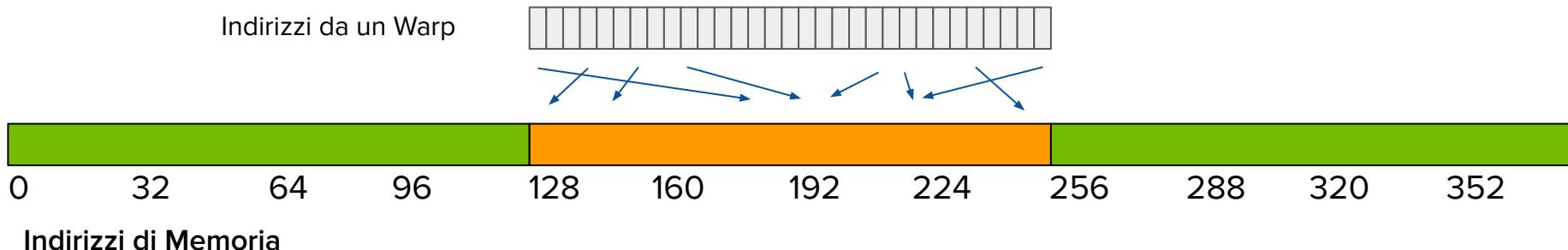
- Caso ideale di accesso alla memoria con **transazioni di dimensioni 128 byte**.
- Tutti gli indirizzi richiesti dai thread in un warp cadono all'interno di una **singola linea di cache** da 128 byte.
- Richiede **una sola transazione** di memoria da 128 byte per completare l'operazione di caricamento.
- **Utilizzo del bus al 100%**.
- **Nessun dato inutilizzato** nella transazione.
- `int c = a[idx]; // Es: idx = blockIdx.x * blockDim.x + threadIdx.x;`



**Nota:** Questo comportamento vale per CC precedenti alla 5.2; per CC 6.0 e successive, le transazioni minime sono da 32 byte.

# Cached Loads: Allineato con Indirizzi Randomizzati

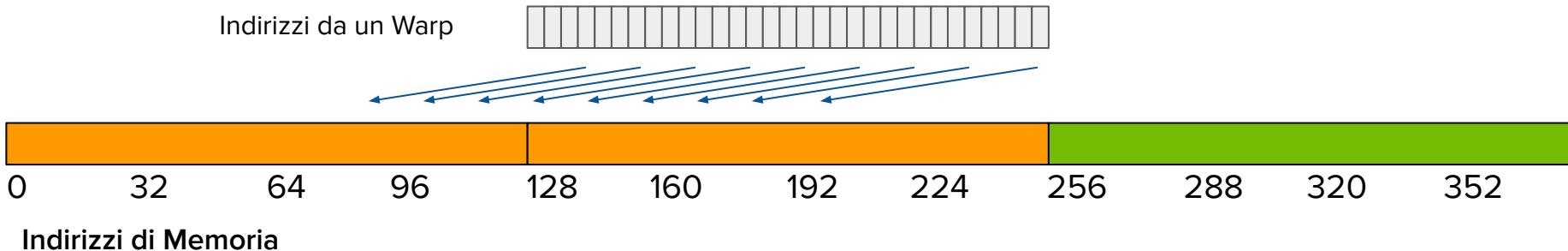
- Gli indirizzi richiesti sono **contigui** in memoria, non consecutivi rispetto al thread ID e distribuiti casualmente.
- Gli indirizzi sono randomizzati ma confinati all'interno di una singola cache line di 128 byte.
- La richiesta genera **una sola transazione** di memoria da 128 byte.
- L'allineamento è mantenuto poiché l'indirizzo iniziale è multiplo di 128 byte.
- **Nessuno spreco** di dati se ogni thread richiede 4 byte distinti.
- **Utilizzo del bus** di memoria al **100%**.
- `int c = a[(threadIdx.x * 17) % warpSize];`



**Nota:** Questo comportamento vale per CC precedenti alla 5.2; per CC 6.0 e successive, le transazioni minime sono da 32 byte.

# Cached Loads: Disallineato

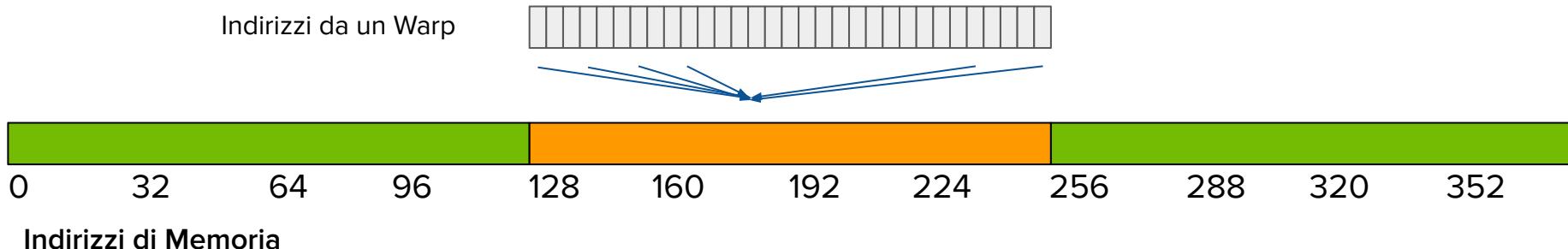
- I thread di un warp richiedono **32 elementi consecutivi di 4 byte** che non sono allineati.
- Gli indirizzi richiesti dai thread si estendono su **due segmenti da 128 byte** in memoria globale.
- Il primo indirizzo **non è multiplo** di 128 byte.
- Sono necessarie **due transazioni** da 128 byte per completare l'operazione di caricamento.
- Utilizzo del bus di memoria ridotto al 50%.**
- Metà dei byte caricati nelle due transazioni non vengono utilizzati** (significativo **spreco di larghezza di banda**).
- `int c = a[idx - 16];`



**Nota:** Questo comportamento vale per CC precedenti alla 5.2; per CC 6.0 e successive, le transazioni minime sono da 32 byte.

# Cached Loads: Accesso allo Stesso Indirizzo

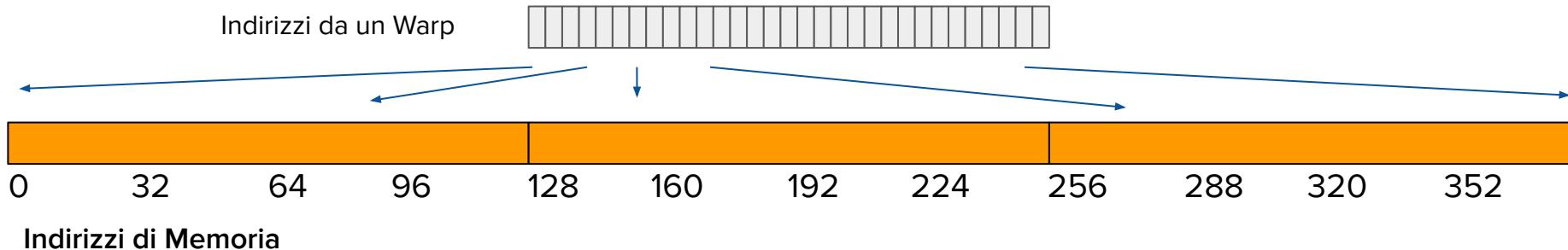
- Tutti i thread in un warp richiedono un dato dallo **stesso indirizzo di memoria**.
- L'indirizzo richiesto cade su una singola linea di cache.
- Richiede **una sola transazione** di memoria da 128 byte.
- **Utilizzo del bus estremamente basso**.
- Per un valore di 4 byte, l'utilizzo effettivo è di **4 byte richiesti su 128 byte** caricati, con un'efficienza di solo il **3,125%**.
- `int c = a[45];`



**Nota:** Questo comportamento vale per CC precedenti alla 5.2; per CC 6.0 e successive, le transazioni minime sono da 32 byte.

# Cached Loads: Accessi Sparsi

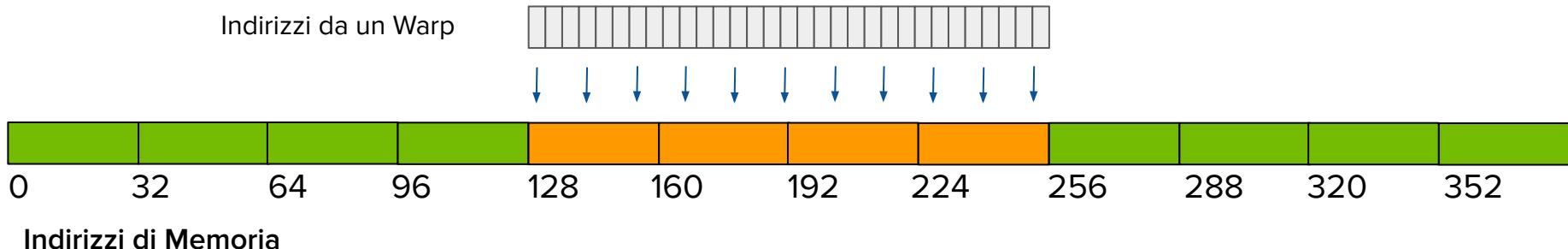
- Gli indirizzi possono estendersi su **N** linee di cache, dove  $0 < N \leq 32$ .
- Sono necessarie **N transazioni di memoria** per completare una **singola operazione di caricamento**.
- **Worst-case Scenario:** I thread di un warp richiedono **32 indirizzi da 4 byte sparsi** (scattered) nella memoria globale.
- Il totale dei byte richiesti dal warp è solamente di 128 byte ( $32 \text{ indirizzi} \times 4 \text{ byte per indirizzo}$ ).
- **Utilizzo del bus:** 128 byte richiesti / ( $N \times 128 \text{ byte caricati}$ ).
- **Massima inefficienza** nell'utilizzo della larghezza di banda per  $N = 32$  (32 transazioni totali richieste).
- `int c = a[rand()];`



**Nota:** Questo comportamento vale per CC precedenti alla 5.2; per CC 6.0 e successive, le transazioni minime sono da 32 byte.

# Uncached Loads: Allineato e Coalescente

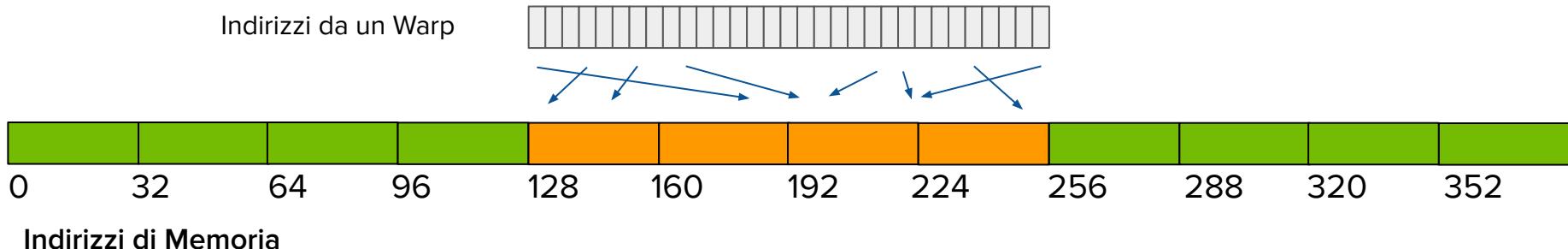
- Granularità dei segmenti di memoria a **32 byte** e non della linea di cache **L1 disabilitata** (128 byte).
- Rappresentazione del caso **ottimale** di accesso alla memoria.
- Il primo indirizzo è un **multiplo** di 32 byte (allineamento rispettato).
- I thread in un warp accedono a **dati contigui** (coalescenza).
- Richiesti **quattro segmenti** da 32 byte → Quattro transazioni coalescenti e allineate da 32 byte per servire l'accesso.
- **Utilizzo del bus al 100%**.
- `int c = a[idx]; // nvcc -Xptxas -dlcm=cg`



**Nota:** Questo comportamento vale anche per CC successive alla 6.0 per il caso cached con granularità 32 byte.

# Uncached Loads: Allineato con Indirizzi Randomizzati

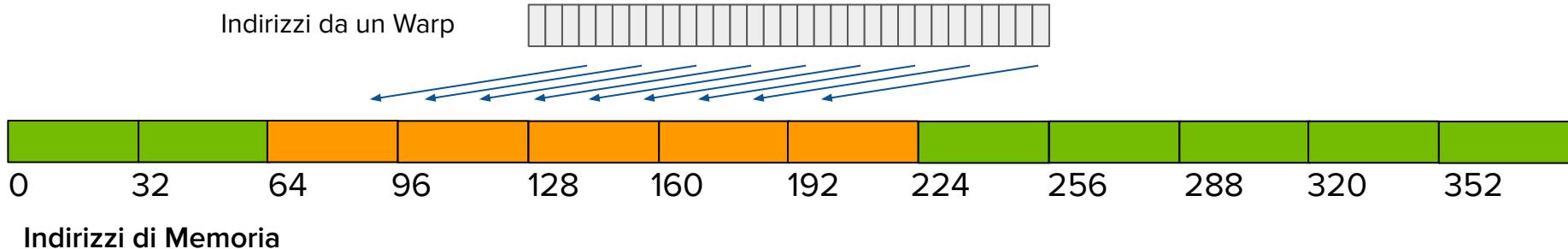
- Il primo indirizzo è un **multiplo** di 32 byte.
- Gli indirizzi richiesti sono **randomizzati** all'interno del range da 128-byte.
- Gli indirizzi ricadono all'interno di **quattro segmenti** da 32 byte.
- Ogni thread richiede un **indirizzo unico** nel range.
- Utilizzo del bus al 100%.**
- La randomizzazione degli accessi **non compromette** le prestazioni del kernel.
- `int c = a[(threadIdx.x * 17) % warpSize]; // nvcc -Xptxas -dlcm=cg`



**Nota:** Questo comportamento vale anche per CC successive alla 6.0 per il caso cached con granularità 32 byte.

# Uncached Loads: Disallineato ma Consecutivo

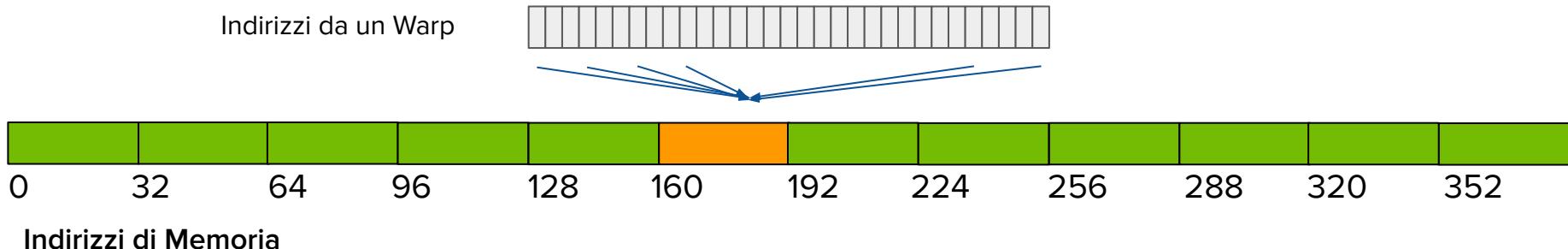
- Warp effettua un caricamento **non allineato** richiedendo **32 elementi consecutivi da 4 byte**.
- Gli indirizzi dei 128 byte richiesti ricadono in massimo **cinque segmenti** da 32 byte.
- **Utilizzo del bus al 80%** (128 byte richiesti, 160 byte caricati).
- I caricamenti “fine-grained” a 32 byte **riducono lo spreco di banda** rispetto ai cached loads da 128 byte su accessi disallineati/non coalescenti (80% vs 50% di utilizzo).
- **Motivo:** Vengono caricati un minor numero di byte non richiesti.
- `int c = a[idx - 2]; // nvcc -Xptxas -dlcm=cg`



**Nota:** Questo comportamento vale anche per CC successive alla 6.0 per il caso cached con granularità 32 byte.

# Uncached Loads: Accesso allo Stesso Indirizzo

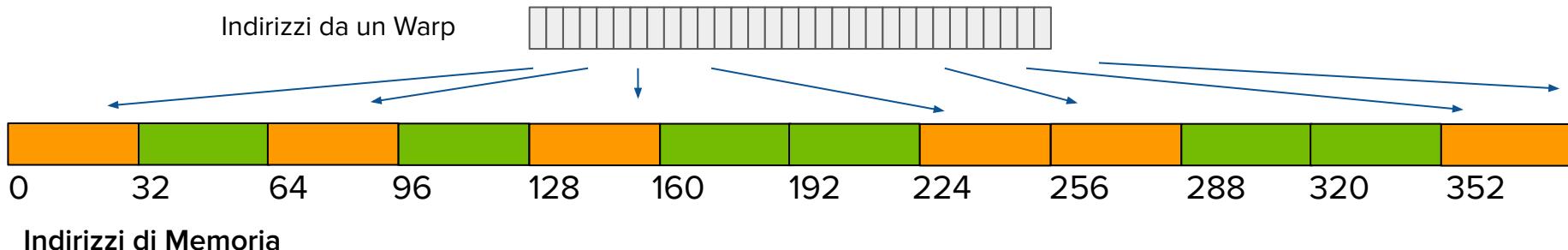
- Tutti i thread in un warp richiedono lo **stesso indirizzo** di memoria.
- L'indirizzo richiesto (anche se disallineato) cade all'interno di un **singolo segmento** da 32 byte.
- **Un solo dato** da 4 byte effettivamente richiesto.
- **Utilizzo del bus** del 12.5% (4 byte richiesti / 32 byte caricati).
- Anche qui, prestazioni **migliori** rispetto ai caricamenti con cache da 128 byte (12.5% vs 3.125%).
- **Motivo:** Minor spreco di larghezza di banda.
- `int c = a[45]; // nvcc -Xptxas -dlcm=cg`



**Nota:** Questo comportamento vale anche per CC successive alla 6.0 per il caso cached con granularità 32 byte.

# Uncached Loads: Accessi Sparsi

- **Worst-case Scenario:** I thread di un warp richiedono **32 indirizzi da 4 byte sparsi** (scattered) nella memoria globale.
- In tale scenario, ogni thread necessita di un solo dato da 4 byte in un segmento che è invece di 32 byte.
- Questo porta a 32 richieste separate da 4 byte ciascuna (totale  $32 \times 32$  byte = 1024 byte caricati; richiesti solo 128).
- Rispetto caso della cached load, c'è un miglioramento grazie alla granularità dei segmenti da 32 byte invece delle linee di cache da 128 byte (**meno sprechi, ma comunque inefficiente**).
- **Utilizzo del bus** dato da:  $128$  byte richiesti /  $(N \times 32$  byte caricati).
- `int c = a[rand()]; // nvcc -Xptxas -dlcm=cg`



**Nota:** Questo comportamento vale anche per CC successive alla 6.0 per il caso cached con granularità 32 byte.

# Scrittura in Memoria Globale

## Caratteristiche generali:

- Prima dell'architettura Volta, la cache **L1** non veniva utilizzata per le operazioni di scrittura in memoria (solo **L2**).
- Da Volta in poi utilizzano la cache L1 in modalità **write-through** (**scrittura simultanea in L1 e L2, poi nella DRAM**).

## Granularità delle operazioni

- Le scritture (store) vengono eseguite a livello di **segmenti con granularità 32 byte**.
- Le transazioni di memoria possono coinvolgere **uno, due o quattro segmenti alla volta**.

## Esempio

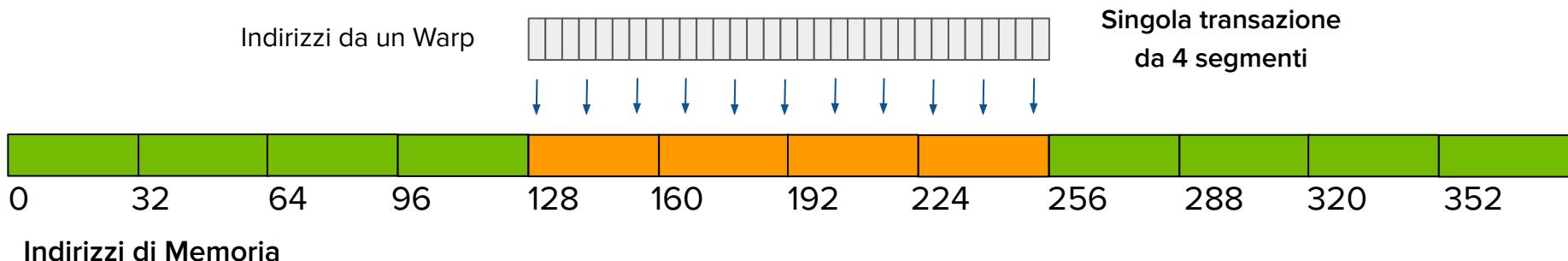
- Se due indirizzi cadono nella stessa regione di 128 byte ma non in una regione allineata di 64 byte:
  - Viene emessa una **singola transazione** di quattro segmenti.
  - **Più efficiente** di due transazioni separate di un segmento ciascuna.

## Ottimizzazione

- Le transazioni più grandi sono **preferite** quando possibile.
- Mira a raggruppare le scritture in **regioni contigue** di memoria.

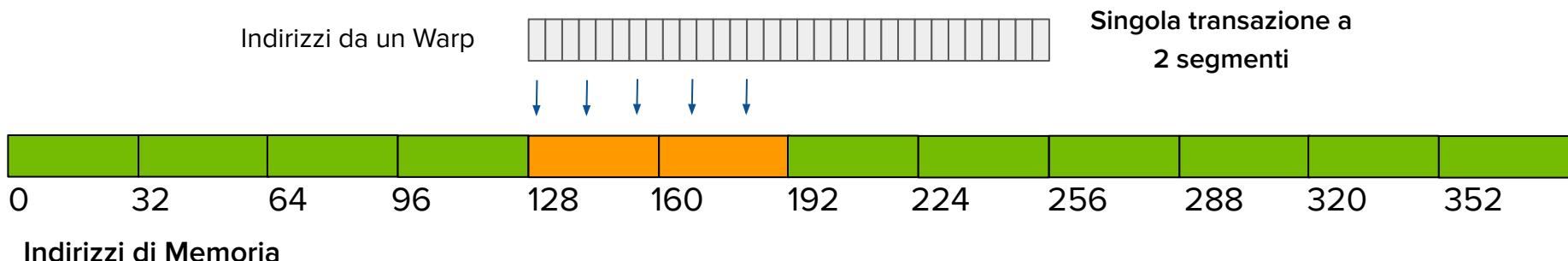
# Scrittura in Memoria: Allineata e Coalescente

- Rappresentazione del **caso ottimale** di scrittura in memoria globale.
- Il primo indirizzo è un **multiplo** della granularità.
- Tutti i thread in un warp accedono a un **intervallo consecutivo** di 128 byte.
- Un warp scrive 128 byte consecutivi (corrisponde esattamente a **quattro segmenti** da 32 byte).
- Servita da una **singola transazione** di quattro segmenti (**utilizzo completo della larghezza di banda**).



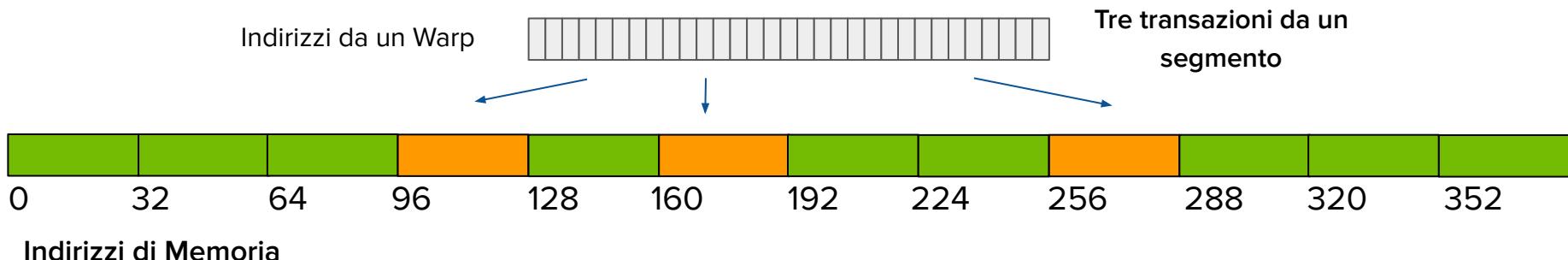
# Scrittura in Memoria: Allineata, Coalescente, Range Limitato

- L'accesso in memoria è **allineato**.
- Gli indirizzi acceduti sono in un **range consecutivo di 64 byte**.
- Corrisponde esattamente a **due segmenti** da 32 byte.
- Servita da una **singola transazione** di due segmenti.



# Scrittura in Memoria: Allineato con Accessi Sparsi

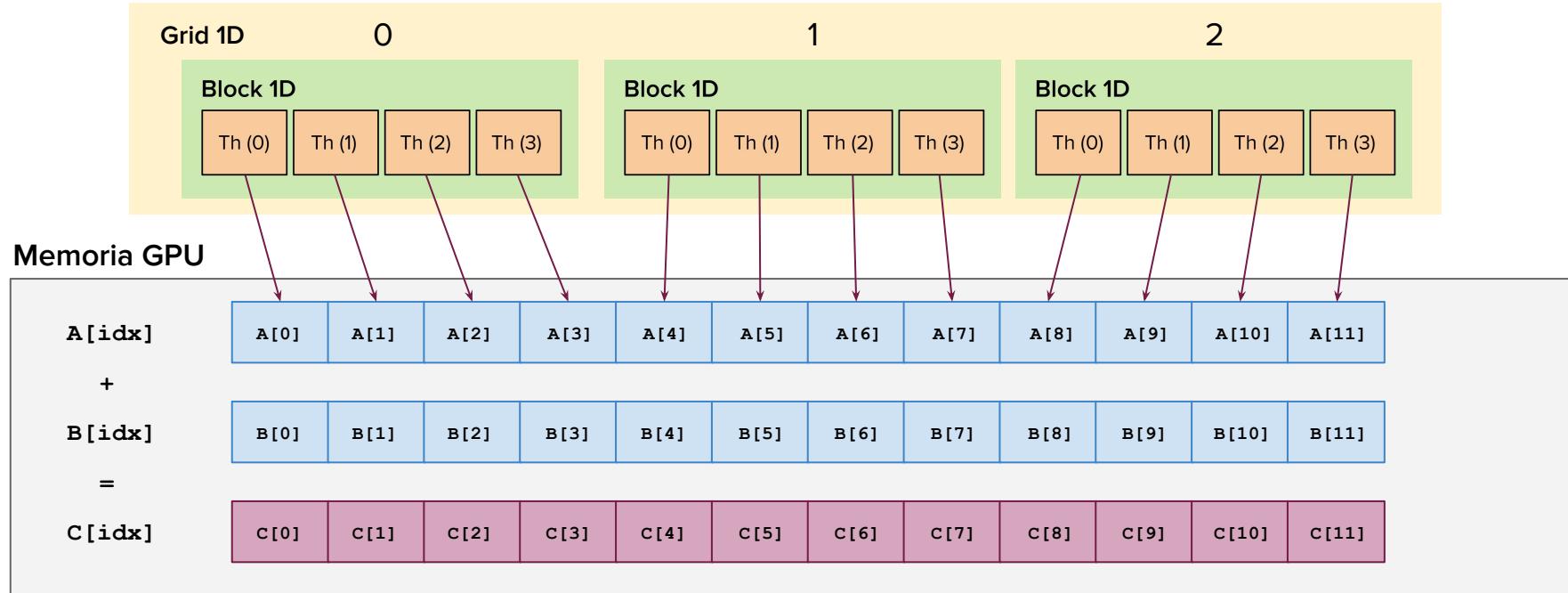
- L'accesso in memoria è **allineato** ma gli indirizzi sono **sparsi** (scattered).
- L'intervallo **totale** coperto è di 192 byte.
- Servita da **tre transazioni** separate di un segmento ciascuna.
- Dimostra l'**impatto negativo** della mancanza di coalescenza.



*Vediamo ora l'**effetto dell'allineamento** e **della coalescenza** su array e matrici:  
due esempi chiave*

# Pattern di Accesso: Somma di Vettori in CUDA C

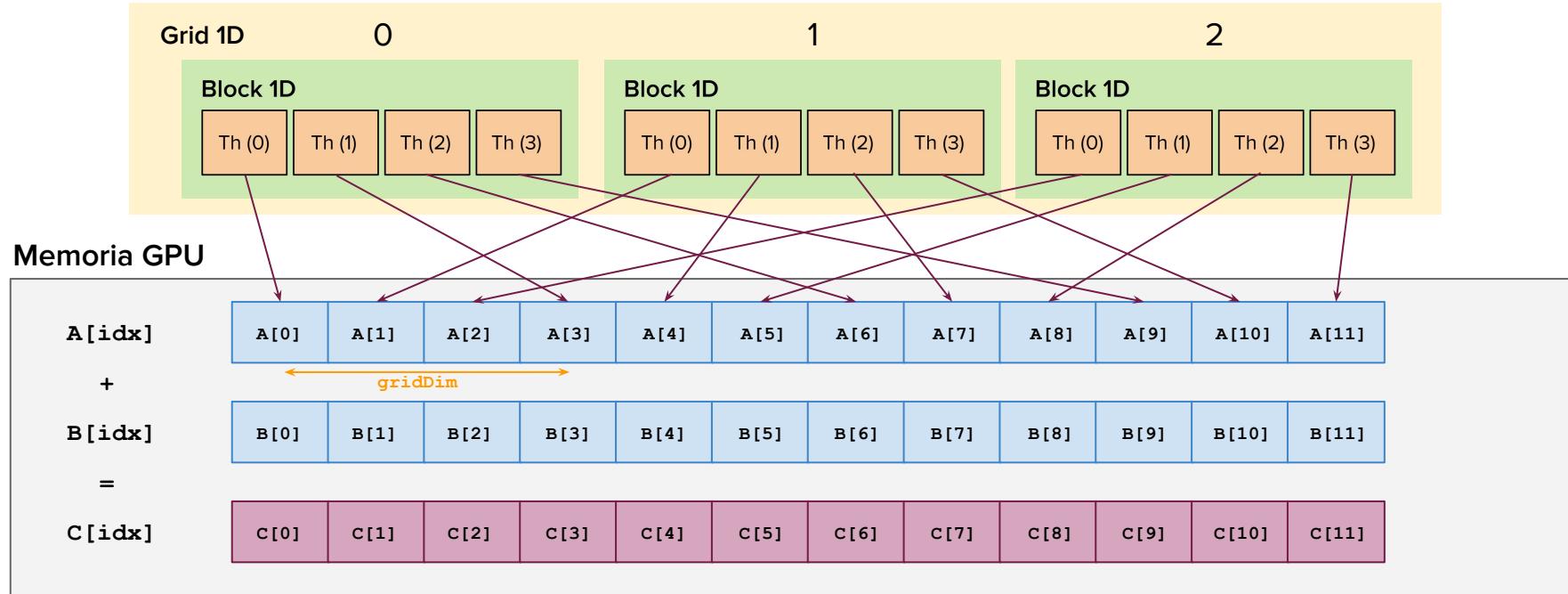
- Allineato e Coalescente: Thread di un blocco accedono in modo **allineato** a **elementi contigui** in memoria.



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

# Pattern di Accesso: Somma di Vettori in CUDA C

- Non Allineato e Non Coalescente: Thread di un blocco accedono a elementi **disallineati** e **non contigui** in memoria.



**idx** = **blockIdx.x** + **gridDim.x** \* **threadIdx.x**

# Pattern di Accesso: Somma di Vettori in CUDA C

## Non Coalescente / Non Allineato

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

// Chiamata del kernel
sum_array_non_coalesced<<<gridDim,blockDim>>>(A,
B, C, N);
```

### Mapping (es. $2^{24}$ elementi, blockSize = 256)

Block 0, Thread 0 → idx = 0  
Block 0, Thread 1 → idx = 65536  
Block 0, Thread 2 → idx = 131072  
...  
Block 1, Thread 0 → idx = 1  
Block 1, Thread 1 → idx = 65537  
Block 1, Thread 2 → idx = 131073  
...

## Coalescente / Allineato

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

// Chiamata del kernel
sum_array_coalesced<<<gridDim,blockDim>>>(A, B,
C, N);
```

### Mapping (es. $2^{24}$ elementi, blockSize = 256)

Block 0, Thread 0 → idx = 0  
Block 0, Thread 1 → idx = 1  
Block 0, Thread 2 → idx = 2  
...  
Block 1, Thread 0 → idx = 256  
Block 1, Thread 1 → idx = 257  
Block 1, Thread 2 → idx = 258  
...

# Pattern di Accesso: Somma di Vettori in CUDA C

Non Coalescente / Non Allineato

```
__global__ void sum_array_non_coalesced(float *A,  
float *B,  
int N)  
{  
    //  
    sum:  
    for (int i = 0; i < N; i++)  
        A[i] = A[i] + B[i];  
}
```

Coalescente / Allineato

```
__global__ void sum_array_coalesced(float *A,  
float *B,  
int N)  
{  
    //  
    sum:  
    for (int i = 0; i < N; i++)  
        A[i] = A[i] + B[i];  
}
```

## Analisi Memoria da Nsight Compute

Dimensione Vettore: 2<sup>24</sup> elementi   Dimensione Blocco: 256   Device: RTX 3090

- Durata [us]:
  - Coalescente / Allineato: 236,80 us
  - Non Coalescente / Non Allineato: 2870,00 us
- Throughput di Memoria [GB/s]:
  - Coalescente / Allineato: 838,14 GB/s
  - Non Coalescente / Non Allineato: 69,66 GB/s
- Peak Memory [%]:
  - Coalescente / Allineato: 92,32%
  - Non Coalescente / Non Allineato: 7,66%
- SOL SM [%]:
  - Coalescente / Allineato: 15,56%
  - Non Coalescente / Non Allineato: 1,49%
- SOL Memory [%]:
  - Coalescente / Allineato: 92,32%
  - Non Coalescente / Non Allineato: 35,77%
- SOL L1/TEX Cache [%]:
  - Coalescente / Allineato: 20,10%
  - Non Coalescente / Non Allineato: 40,99%
- SOL L2 Cache [%]:
  - Coalescente / Allineato: 40,72%
  - Non Coalescente / Non Allineato: 35,77%
- SOL DRAM [%]:
  - Coalescente / Allineato: 92,32%
  - Non Coalescente / Non Allineato: 7,66%

# Problema della Matrice Trasposta

- La trasposizione di una matrice implica lo **scambio di ogni riga con la colonna corrispondente**.

width (W)

height (H)

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47

**A (H x W)**

H

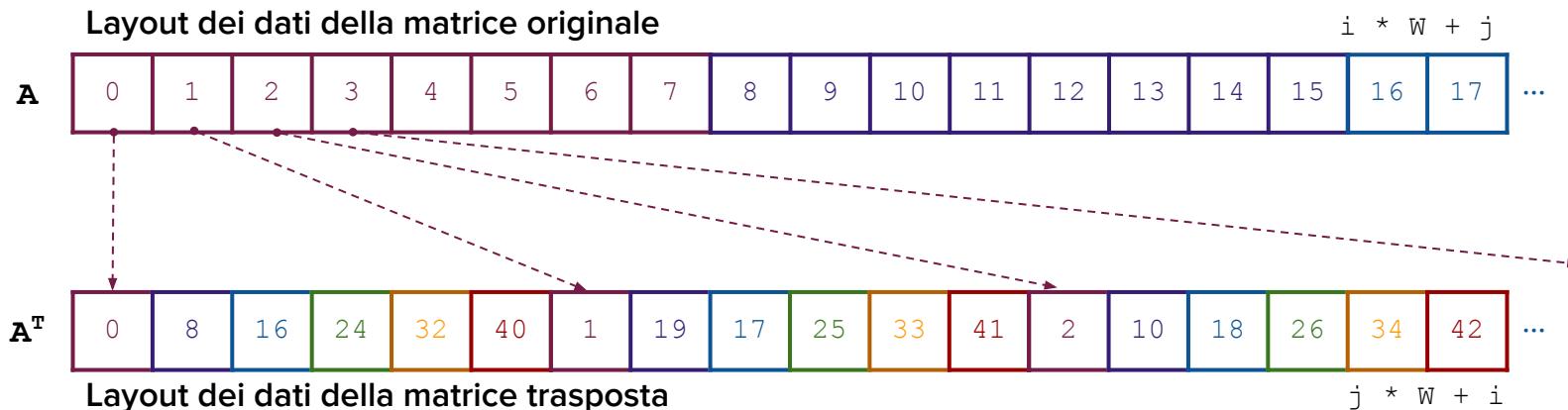
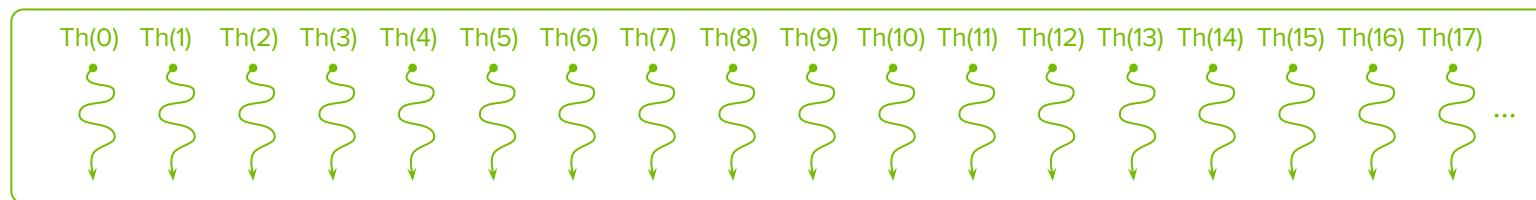
W

$a_{ij} \rightarrow a_{ji}$

0	8	16	24	32	40
1	9	17	25	33	41
2	10	18	26	34	42
3	11	19	27	35	43
4	12	20	28	36	44
5	13	21	29	37	45
6	14	22	30	38	46
7	15	23	31	39	47

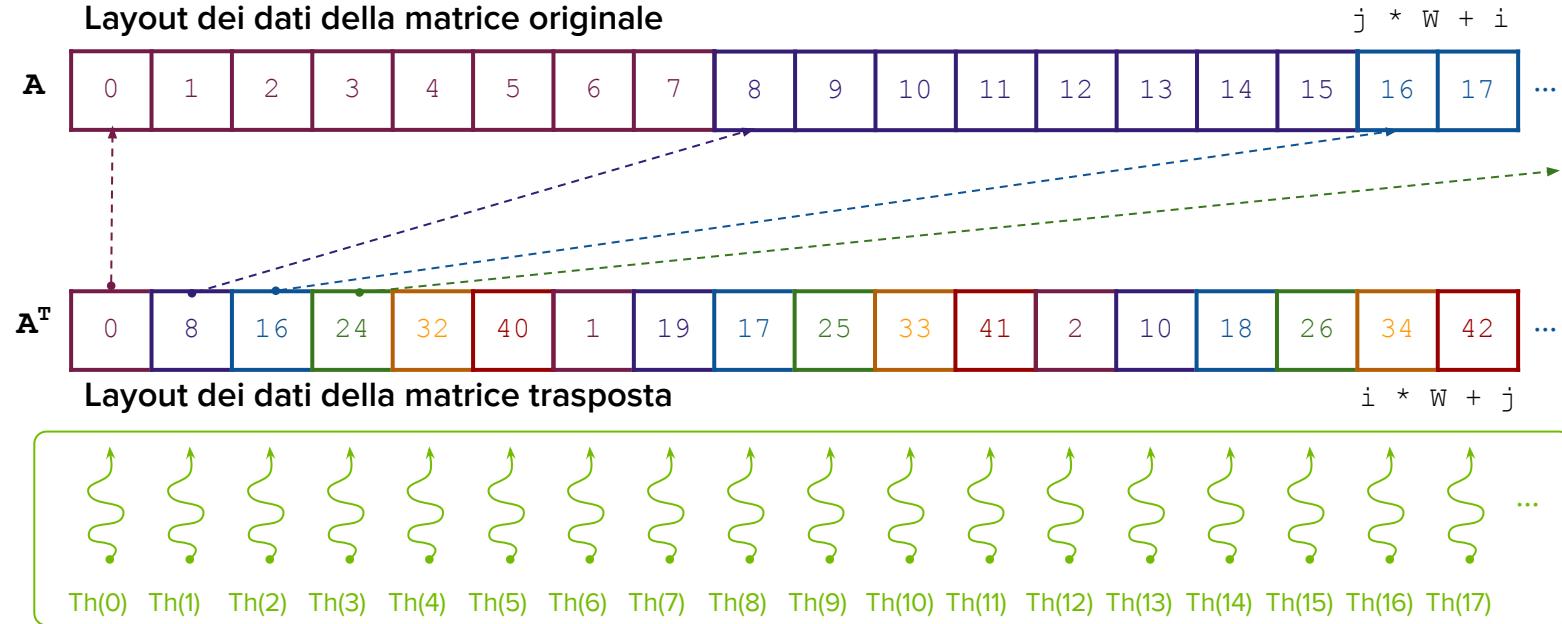
**A<sup>T</sup> (W x H)**

# Pattern di Accesso nel Problema della Matrice Trasposta



- Si ricorda che le matrici vengono memorizzate in memoria come **array 1D** secondo l'approccio *row-major*.
- Osservando i layout delle matrici di input e output, si nota che:
  - **Letture:** accesso per righe nella matrice originale; si ottiene un accesso coalescente.
  - **Scritture:** accesso per colonne nella matrice trasposta; si ottiene un accesso con stride (scattered).

# Pattern di Accesso nel Problema della Matrice Trasposta



- Tuttavia, invertendo il pattern di accesso, si può leggere per colonne e scrivere per righe, spostando l'accesso non coalescente dalla scrittura alla lettura.
- È possibile prevedere le performance relative di queste due implementazioni?

# Pattern di Accesso nel Problema della Matrice Trasposta

Layout dei dati della matrice originale

A	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	...
---	---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	-----

## Osservazione

- Se la cache fosse disabilitata per i caricamenti, le due implementazioni sarebbero teoricamente identiche.
- Con cache abilitata, invece, la lettura per colonna è **più efficiente**:
  - Sebbene le letture per colonna siano non coalescenti, i byte extra caricati in cache potranno servire le letture successive, evitando accessi alla memoria globale.
  - **Esempio:** Eseguendo una lettura non coalescente alla prima colonna {0, 8, 16, 24, 32, 40} tramite multiple transazioni, si caricherebbero in cache tutti i valori delle righe corrispondenti, ovvero i dati {0..7}, {8..15}, {16..23}, {24..31}, {32..39}, {40..47} che possono essere riutilizzati in seguito.
- Tuttavia, **invertendo il pattern di accesso**, si può leggere per colonne e scrivere per righe, spostando l'accesso non coalescente dalla scrittura alla lettura.
- È possibile prevedere le performance relative di queste due implementazioni?

# Pattern di Accesso nel Problema della Matrice Trasposta

## Performance Bounds dei Kernel di Trasposizione

- Si possono creare due kernel che effettuano una semplice **copia della matrice su se stessa** (senza trasporre), usando due diversi pattern di accesso per stabilire dei limiti approssimativi di performance:
  - **Limite superiore:** Copia della matrice caricando e memorizzando **per righe** (Simula le stesse operazioni di memoria della trasposizione ma con soli accessi coalescenti).
  - **Limite inferiore:** Copia della matrice caricando e memorizzando **per colonne** (Simula le stesse operazioni di memoria della trasposizione ma con soli accessi con stride).

## Obiettivo dei Bounds

- I due kernel di copia stabiliscono il range di performance entro cui ci si aspetta che cada il kernel di trasposizione. Qualsiasi implementazione della trasposizione avrà performance comprese tra questi limiti.
  - Il limite superiore rappresenta il caso ideale (tutti accessi coalescenti).
  - Il limite inferiore rappresenta il caso peggiore (tutti accessi non coalescenti).

## Utilizzo nel Confronto

- Se il kernel di trasposizione si avvicina al limite superiore → **Implementazione efficiente**.
- Se si avvicina al limite inferiore → **Spazio per ottimizzazioni**.

# Pattern di Accesso nel Problema della Matrice Trasposta

## Kernel Copia per Righe (Accesso Coalescente)

```
__global__ void copyRow(float *out, float *in, const int W, const int H) {
    unsigned int ix = blockDim.x * blockIdx.x + threadIdx.x;
    unsigned int iy = blockDim.y * blockIdx.y + threadIdx.y;

    if (ix < W && iy < H) {
        out[iy * W + ix] = in[iy * W + ix]; // Lettura e scrittura coalescente
    }
}
```

## Kernel Copia per Colonne (Accesso con Stride)

```
__global__ void copyCol(float *out, float *in, const int W, const int H) {
    unsigned int ix = blockDim.x * blockIdx.x + threadIdx.x;
    unsigned int iy = blockDim.y * blockIdx.y + threadIdx.y;

    if (ix < W && iy < H) {
        out[ix * H + iy] = in[ix * H + iy]; // Lettura e scrittura con stride
    }
}
```

# Pattern di Accesso nel Problema della Matrice Trasposta

## Trasposizione Naïve che Legge per Righe e Scrive per Colonne

```
__global__ void transposeNaiveRow(float *out, float *in, const int W, const int H) {
    unsigned int ix = blockDim.x * blockIdx.x + threadIdx.x;
    unsigned int iy = blockDim.y * blockIdx.y + threadIdx.y;

    if (ix < W && iy < H) {
        out[ix * H + iy] = in[iy * W + ix]; // Lettura coalescente, scrittura con stride
    }
}
```

## Trasposizione Naïve che Legge per Colonne e Scrive per Righe

```
__global__ void transposeNaiveCol(float *out, float *in, const int W, const int H) {
    unsigned int ix = blockDim.x * blockIdx.x + threadIdx.x;
    unsigned int iy = blockDim.y * blockIdx.y + threadIdx.y;

    if (ix < W && iy < H) {
        out[iy * W + ix] = in[ix * H + iy]; // Lettura con stride, scrittura coalescente
    }
}
```

# Pattern di Accesso nel Problema della Matrice Trasposta

NVIDIA Nsight Compute

Dim. Matrice (16384,16384) , Dim. Blocco (16,16)

GPU RTX 3090

Kernel	Load HIT Rate Cache L2 (%)	Load HIT Rate Cache L1 (%)	Memory Through. (GB/s)	% Peak Mem	Runtime (ms)
<b>L1 Cache Disabilitata (-Xptxas -dlcm=cg)</b>					
<code>copyRow (Upper Bound)</code>	0	0	830,92	91,37	2,58
<code>copyCol (Lower Bound)</code>	75,00	0	421,04	46,22	5,10
<code>transposeNaiveRow (Strided W/Coalesced R)</code>	0	0	453,93	49,77	4,73
<code>transposeNaiveCol (Strided R/Coalesced W)</code>	75,00	0	529,67	58,16	4,05
<b>L1 Cache Abilitata</b>					
<code>copyRow (Upper Bound)</code>	0	0	830,59	91,37	2,58
<code>copyCol (Lower Bound)</code>	0	75,00	451,50	49,58	4,75
<code>transposeNaiveRow (Strided W/Coalesced R)</code>	0	0	452,11	49,58	4,74
<code>transposeNaiveCol (Strided R/Coalesced W)</code>	0	75,00	560,52	61,60	3,83

# Panoramica del Modello di Memoria CUDA

## ➤ Modelli di Performance

- Memory-Bound vs Compute-Bound
- Intensità Aritmetica e Roofline Model

## ➤ Gerarchia di Memoria CUDA

- Organizzazione Gerarchica Completa
- Scope e Programmabilità

## ➤ Gestione della Memoria Host-Device

- Allocazione e Trasferimenti
- Pinned Memory
- Zero-Copy Memory
- UVA (Unified Virtual Addressing)
- Unified Memory (UM)

## ➤ Global Memory

- Pattern di Accesso
- Lettura Cached vs Uncached
- Scrittura

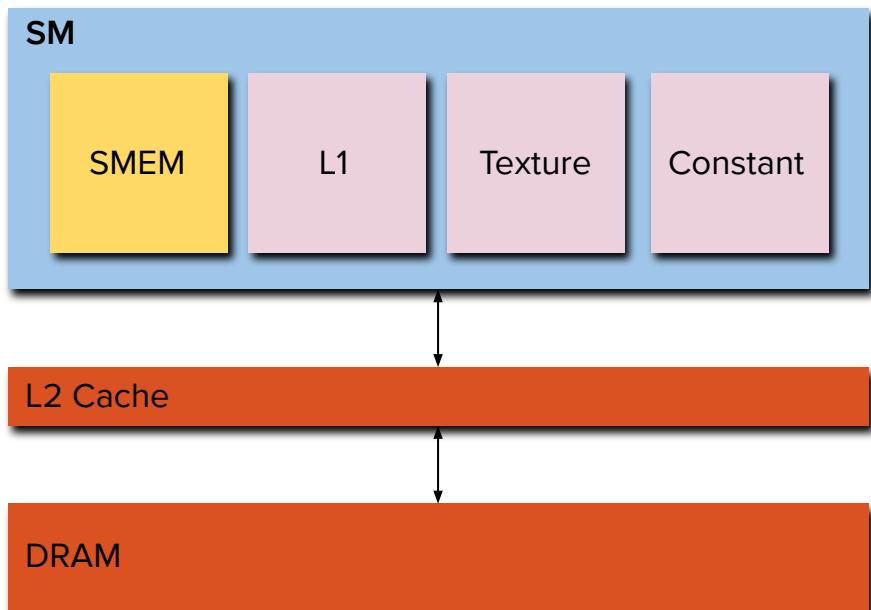
## ➤ Shared Memory

- Memory Banks
- Modalità di Accesso e Bank Conflicts

# Shared Memory (SMEM)

## Perché è Importante?

- Canale di comunicazione per tutti i thread appartenenti ad un blocco.
- Una **cache gestita dal programma** per i dati dalla memoria globale.
- Memoria **scratch pad** (temporanea) per elaborare dati on-chip e migliorare i pattern di accesso alla global memory.
- Aumenta la **banda disponibile** e riduce la **latenza** (20-30 inferiore alla memoria globale), accelerando l'esecuzione del kernel.
- La SMEM si trova **più vicina alle unità di elaborazione di un SM** rispetto alla cache L2 e alla memoria globale, il che contribuisce alla sua bassa latenza.



# Shared Memory (SMEM)

## Allocazione e Accesso

- Una **quantità fissa di SMEM** viene allocata ad ogni blocco di thread all'inizio della sua esecuzione. Questo spazio rimane dedicato al blocco per tutto il suo ciclo di vita nell'SM.
- Tutti i thread del blocco condividono lo **stesso spazio di indirizzamento** della SMEM.
- Gli accessi alla memoria avvengono per warp, idealmente con **una sola transazione** per richiesta. Nel **caso peggiore**, sono necessarie 32 transazioni per warp.
- La SMEM è **gestita esplicitamente** dal programmatore, che decide *quali* dati caricare, *come* organizzarli, e *gestire la sincronizzazione* tra thread del blocco usando **\_\_syncthreads()**.

## Considerazioni Chiave

- La SMEM è una **risorsa limitata**, condivisa tra tutti i blocchi di thread attivi su un SM. La sua dimensione dipende tipicamente dall'architettura GPU e può essere configurabile entro un certo limite (vedere Compute Capability).
- Un uso eccessivo di SMEM può **limitare il parallelismo del dispositivo**, riducendo il numero di blocchi di thread attivi concorrenti in un SM (**minore occupancy**).
- Nelle architetture NVIDIA, lo spazio della SMEM è **tipicamente condiviso fisicamente con la cache L1**, permettendo una configurazione flessibile della suddivisione dello spazio tra i due usi.

# Flusso Tipico di Utilizzo della Shared Memory

## 1. Caricamento in Shared Memory (Global → Shared)

- Ogni thread del blocco carica i dati dalla memoria globale alla shared memory.

## 2. Sincronizzazione Post-Caricamento

- `__syncthreads()` garantisce che tutti i thread del blocco abbiano completato il caricamento dei dati.
- Assicura che i dati necessari siano consistenti per l'elaborazione per tutti i thread del blocco.

## 3. Elaborazione Dati

- Ogni thread del blocco elabora i dati sfruttando la bassa latenza della shared memory.
- Consente il riutilizzo dei dati tra i thread del blocco.

## 4. (Opzionale) Sincronizzazione Post-Elaborazione

- `__syncthreads()` garantisce che i thread abbiano completato le modifiche ai dati, se necessario.
- Usare solo quando i risultati elaborati da un thread sono utilizzati da altri thread dello stesso blocco.

## 5. Scrittura dei Risultati (Shared → Global)

- I thread del blocco collaborano per trasferire i risultati dalla shared memory alla memoria globale.

# Flusso Tipico di Utilizzo della Shared Memory

## Workflow Memoria Condivisa

```
__global__ void kernelExample(float *global_data, float *global_output) {
    // Dichiarazione array in memoria condivisa
    extern __shared__ float shared_data[];

    // Calcolo dell'indice globale
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    // 1. CARICAMENTO (Global → Shared)
    shared_data[threadIdx.x] = global_data[idx];

    // 2. SINCRONIZZAZIONE
    __syncthreads();

    // 3. ELABORAZIONE
    float result = elaborate(shared_data[threadIdx.x]);

    // 4. (Opzionale) SINCRONIZZAZIONE, se altri thread dipendono dai risultati
    __syncthreads();

    // 5. SCRITTURA (Shared → Global)
    global_output[idx] = result;
}
```

# Shared Memory (SMEM)

## Metodi di Allocazione

- **Statica:** La quantità di SMEM da allocare è specificata e nota al momento della compilazione.

```
_shared_ float tile[size_y][size_x];
```

- Se dichiarata all'interno di un kernel, l'ambito di questa variabile è locale al kernel.
- Se dichiarata al di fuori di qualsiasi kernel in un file, l'ambito di questa variabile è globale a tutti i kernel.
- Supporta array 1D, 2D e 3D.

- **Dinamica:** La quantità di SMEM viene specificata nella configurazione di lancio del kernel, prima dell'esecuzione.

```
extern _shared_ int tile[];
```

- Questa dichiarazione può essere fatta all'interno o all'esterno di tutti i kernel.
- La keyword **extern** è utilizzata per dichiarare un array di dimensione non nota al momento della compilazione ma che verrà **determinata a runtime**.
- Poiché la dimensione dell'array è **sconosciuta** a tempo di compilazione, alloca dinamicamente la memoria condivisa specificando la **dimensione in byte** come terzo argomento nella chiamata al kernel:

```
kernel<<<grid, block, isize * sizeof(int)>>>(...)
```

- **Nota:** è possibile dichiarare dinamicamente solo array 1D (è comunque possibile gestire array multidimensionali attraverso calcoli manuali degli indici).

# Shared Memory (SMEM)

## Allocazione Statica

```
// Lato Device

__global__ void kernel(...) {
    __shared__ short array0[128];
    __shared__ float array1[64];
    __shared__ int array2[256];

    // È possibile comunque dichiarare
    // variabili come:
    __shared__ int i;
    __shared__ int array3[10][10];
    __shared__ int array4[10][10][10];
}

// Lato Host:
kernel<<<nBlocks, blockSize>>>(...);
```

## Allocazione Dinamica

```
// Lato Device

__global__ void kernel(...) {
    extern __shared__ float array[];
    // Tutte le variabili dichiarate così
    // iniziano allo stesso indirizzo di memoria

    short* array0 = (short*)array;
    float* array1 = (float*)&array0[128];
    int* array2 = (int*)&array1[64];
    // ...
}

// Lato Host:
size_t smBytes = 128 * sizeof(short) +
                 64 * sizeof(float) +
                 256 * sizeof(int);

kernel<<<nBlocks, blockSize, smBytes>>>(...);
```

*Capire come la shared memory è organizzata internamente è fondamentale per sfruttarne appieno i vantaggi in termini di latenza e bandwidth.*

# Shared Memory Banks

## Cos'è un Memory Bank?

- Per massimizzare la banda larga di memoria, la shared memory è suddivisa in **32 moduli di memoria** di uguale dimensione chiamati **memory bank** (banco di memoria).
- Un "cassetto" che contiene una porzione di dati, con ogni banco capace di servire una **word** (4 o 8 byte, la cui dimensione dipende dalla specifica architettura).

## Perché 32 Banchi?

- Il numero 32 corrisponde al numero di thread presenti in un warp, permettendo l'**accesso simultaneo** alla memoria da parte di tutti i thread.

## Mappatura degli Indirizzi

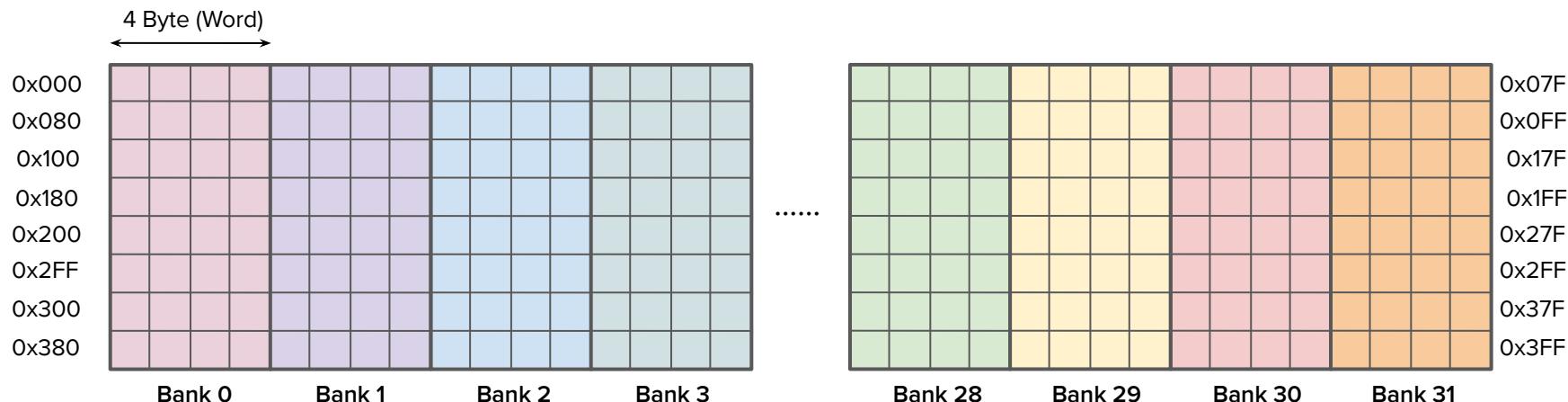
- La memoria condivisa è uno **spazio di indirizzamento lineare (1D)**, ma viene mappata fisicamente sui banchi.
- La mappatura degli indirizzi ai banchi **varia a seconda della compute capability** della GPU.

## Mappatura degli Indirizzi

- **Scenario Ideale:** Se un'operazione di lettura o scrittura (load/store) emessa da un warp accede ad un solo indirizzo per ogni banco, l'operazione è servita da una singola transazione di memoria.
- **Scenario Non Ottimale:** Se un'operazione accede a più indirizzi nello stesso banco, sono necessarie più transazioni di memoria, riducendo l'utilizzo della banda larga.

# Shared Memory Banks

- La shared memory CUDA è **organizzata in 32 banchi paralleli per consentire accessi simultanei**.
- L'indirizzamento è **sequenziale**: word consecutive sono mappate su banchi consecutivi.
- La dimensione totale della shared memory **varia con la Compute Capability** (configurabile per bilanciare l'uso con la L1 cache).
- La figura mostra un **caso semplicistico** di 1KB totale di shared memory con word da 4 byte (1 int, 1 float, 4 char, etc..) distribuite sui 32 banchi, dove l'**organizzazione in 32 banchi paralleli rimane una costante architetturale**.



# Modalità di Accesso alla Shared Memory

## Larghezza del Banco di Memoria

- La **larghezza del banco** di memoria condivisa definisce quali indirizzi di memoria appartengono a quali banchi di memoria.
- La larghezza dei banchi **varia** a seconda della Compute Capability del dispositivo.

## Larghezza di Banda del Banco

- **4 byte (32 bit)** oppure **8 byte (64 bit)**.
- Dispositivi con Compute Capability 3.x (es. Kepler) usano banchi da 8 byte, le architetture più recenti da 4 byte.

## Esempio (Compute Capability 5.x e successive) [[Documentazione Online](#)]

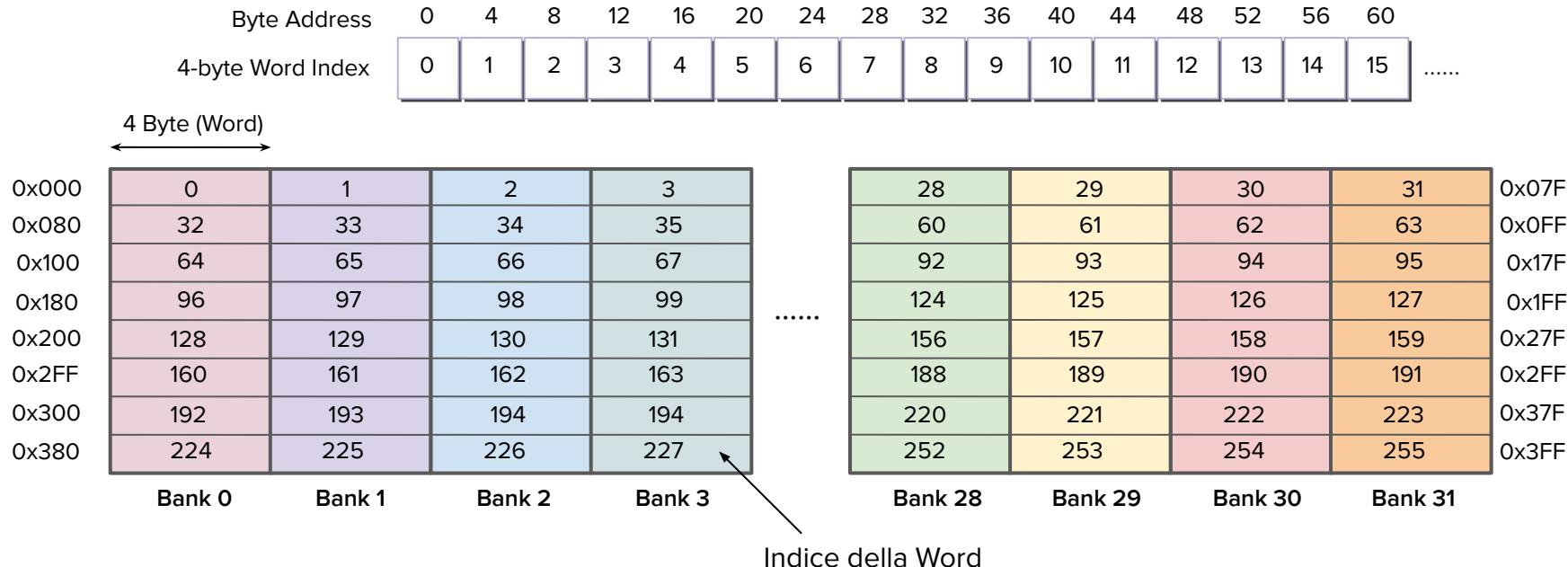
- La larghezza del banco è di **4 byte (32 bit)** e ci sono **32 banchi**.
- Ogni banco supporta trasferimenti paralleli di 32 bit per ciclo di clock.
- Le **parole (word) successive** di 32 bit si mappano alle **banchi successivi**.
- Calcolo dell'Indice del banco:

**indice banco = (indirizzo byte ÷ 4 byte/banco) % 32 banchi**

- L'indirizzo byte è diviso per 4 per ottenere l'indice della parola di 4 byte, e l'operazione modulo 32 converte questo indice in un indice di banco.

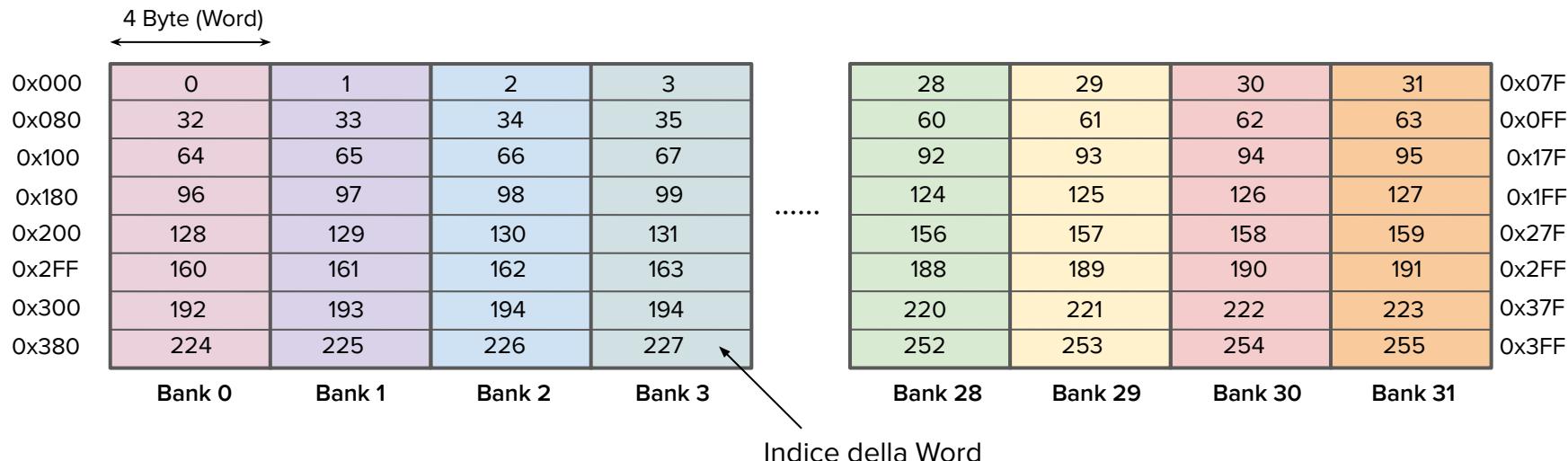
# Mappatura della Memoria Condivisa

- **Mappatura Memoria/Banchi:** Ogni indirizzo viene mappato a un indice di parola e a un banco specifico. Nei dispositivi con larghezza del banco da 4 byte, questo determina la distribuzione delle parole tra i banchi.
  - **Pattern Ciclico:** Ogni 32 parole, la mappatura ai banchi si ripete ciclicamente (es: parola 0 e parola 32 appartengono allo stesso banco).



# Mappatura della Memoria Condivisa

- La Shared Memory può essere indirizzata sia a **byte** che a **intero word**, con i 32 banchi che possono servire simultaneamente tutti i thread di un warp.
- Ad esempio, in un array di float (`_shared_ float arr[256]`), ogni elemento è distribuito ciclicamente sui banchi: **arr[0]** al **Bank 0**, **arr[1]** al **Bank 1**, fino a **arr[31]** al **Bank 31**, poi **arr[32]** torna al **Bank 0**.
- Questa organizzazione garantisce accessi paralleli efficienti quando thread consecutivi accedono a **elementi consecutivi** dell'array.



# Mapping degli Indirizzi e Bank Conflicts nella SMEM

- Il seguente esempio illustra come gli indirizzi consecutivi vengono assegnati ai banchi.

## Mapping degli Indirizzi ai Banchi

```
__shared__ int arr[256];
int value;

// Ogni indirizzo nella memoria condivisa viene mappato ciclicamente
ai banchi di memoria.
value = arr[0]; // arr[0] è nel banco 0
value = arr[1]; // arr[1] è nel banco 1
value = arr[2]; // arr[2] è nel banco 2
...
value = arr[31]; // arr[31] è nel banco 31
value = arr[32]; // arr[32] è di nuovo nel banco 0
value = arr[33]; // arr[33] è di nuovo nel banco 1
```

# Bank Conflict: Collisioni in Memoria Condivisa

## Cos'è?

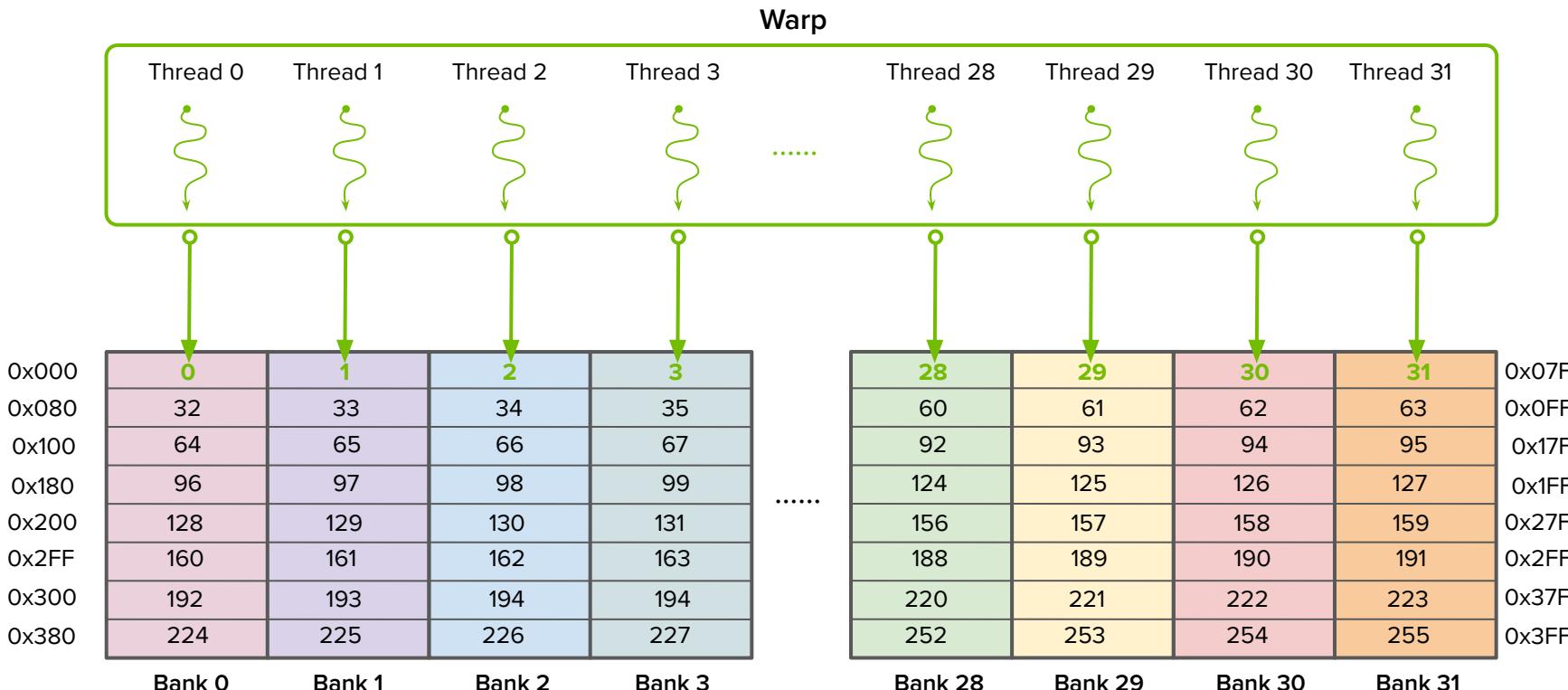
- Un **bank conflict** si verifica quando più thread di un warp accedono a indirizzi diversi nello stesso memory bank.
- L'hardware divide una richiesta con conflitto in **più transazioni separate** (*conflict-free*), riducendo la banda proporzionalmente al numero di transazioni necessarie.
- **Inter-block:** Nessun conflitto tra thread di blocchi diversi, ma **solo a livello di warp** dello stesso blocco.

## Tipi di Accesso

- **Accesso Parallello** (desiderabile)
  - Indirizzi multipli distribuiti su bank diversi.
  - Idealmente, ogni indirizzo in un bank separato.
  - Una singola transazione per servire più o tutti gli accessi.
- **Accesso Seriale**
  - Indirizzi multipli distribuiti nello stesso bank.
  - La richiesta viene serializzata.
  - Nel caso peggiore: 32 transazioni per 32 thread che accedono a locazioni diverse nello stesso bank.
- **Accesso Broadcast**
  - Tutti i thread leggono lo stesso indirizzo in un singolo bank.
  - Una sola transazione, con il dato trasmesso a tutti i thread (broadcast automatico).
  - Efficiente in termini di transazioni, ma potenzialmente basso utilizzo della bandwidth.

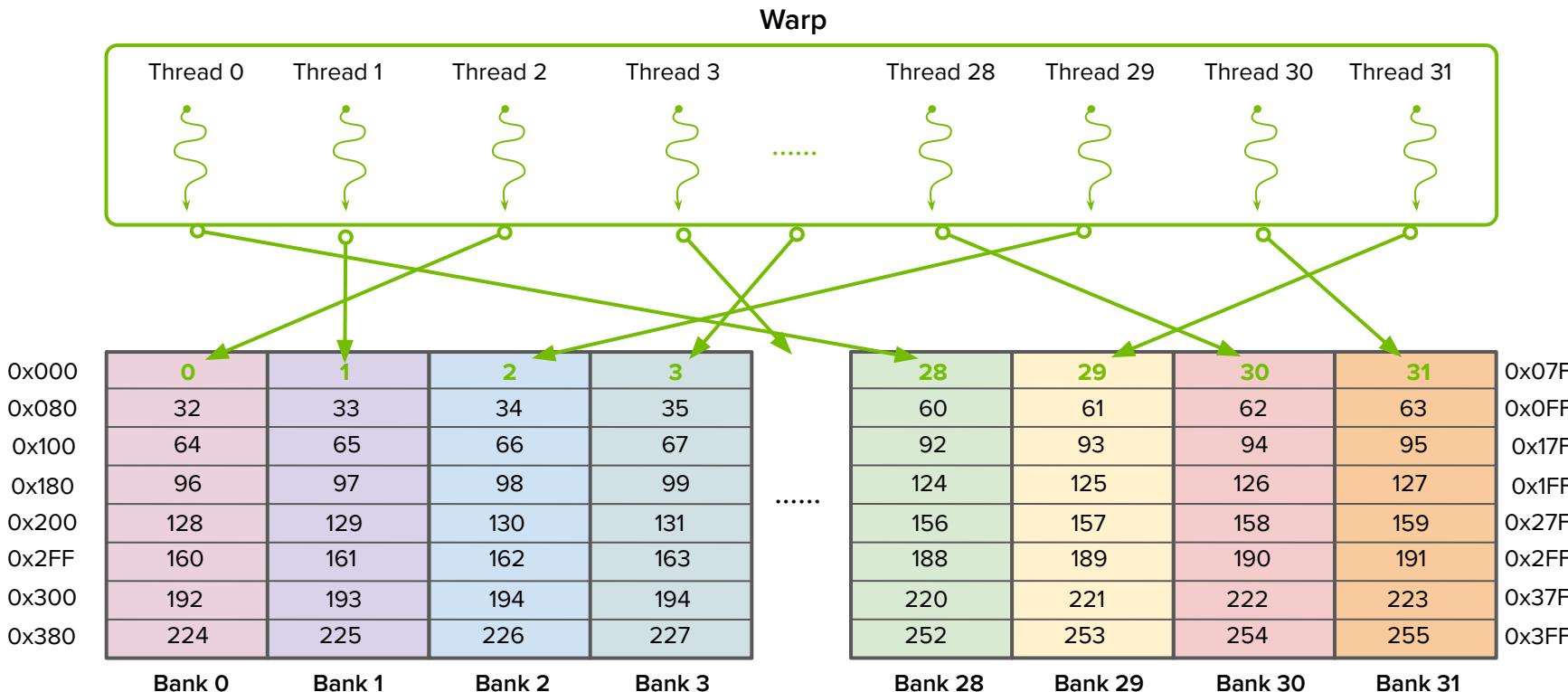
**Nota:** Le prestazioni, anche con conflitti in SMEM, sono comunque nettamente migliori rispetto all'accesso a cache L2 o, peggio, alla global memory.

# Modalità di Accesso - Parallelo ed Ottimale



- Ogni thread accede a una parola di 32 bit in un bank **diverso**.
- **Nessun conflitto**, massima efficienza.

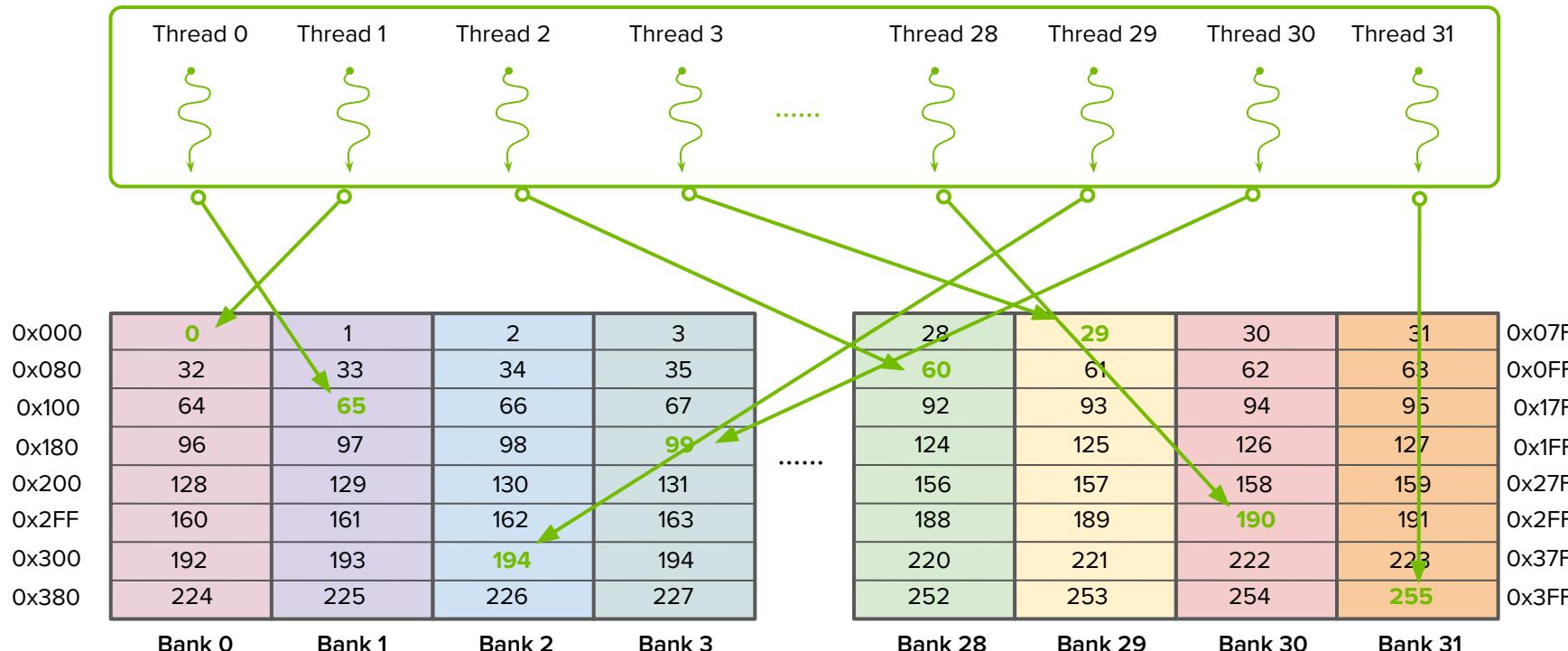
# Modalità di Accesso - Casuale Senza Conflitti



- Pattern di accesso **irregolare** ma **senza conflitti**.
- Ogni thread accede comunque a un **bank diverso**. **Nessun conflitto**, massima efficienza.

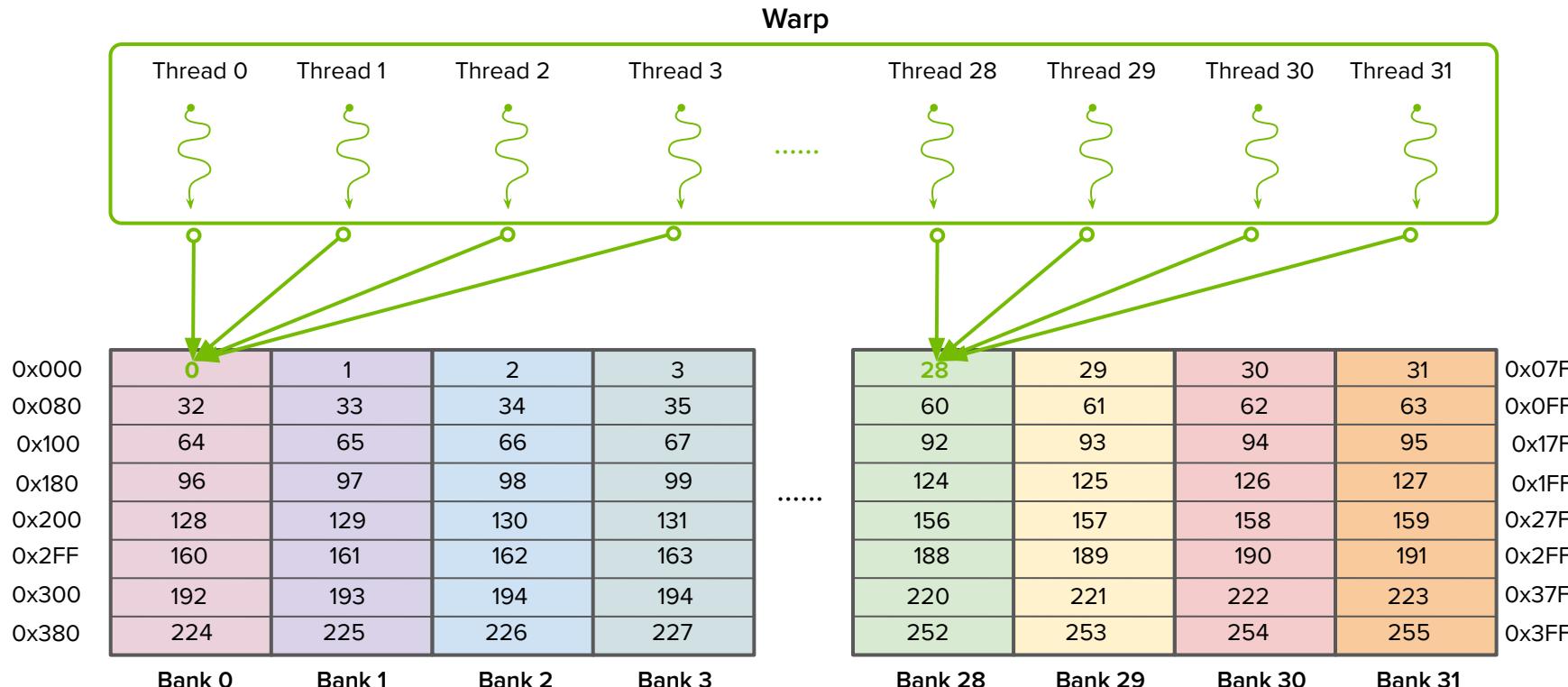
# Modalità di Accesso - Casuale Senza Conflitti

Warp



- Ogni thread accede a una parola di 32 bit in un bank **diverso**.
- Ogni thread accede comunque a un bank **diverso**. **Nessun conflitto**, massima efficienza.

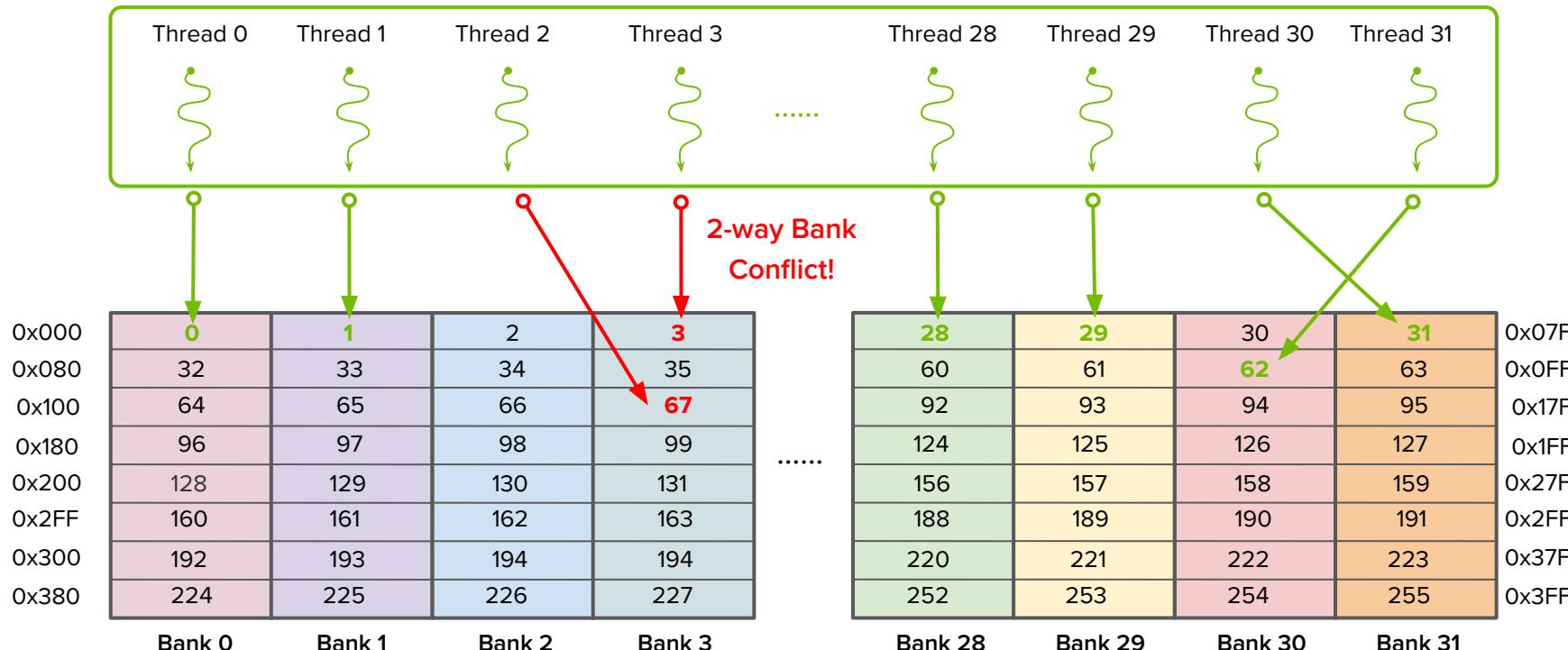
# Modalità di Accesso - Irregolare con Potenziali Conflitti (1/2)



- Più thread accedono allo stesso bank.
- **(Primo Scenario) Broadcast senza conflitti** (stesso indirizzo nel bank) → Massima efficienza ma spreco di bandwidth.

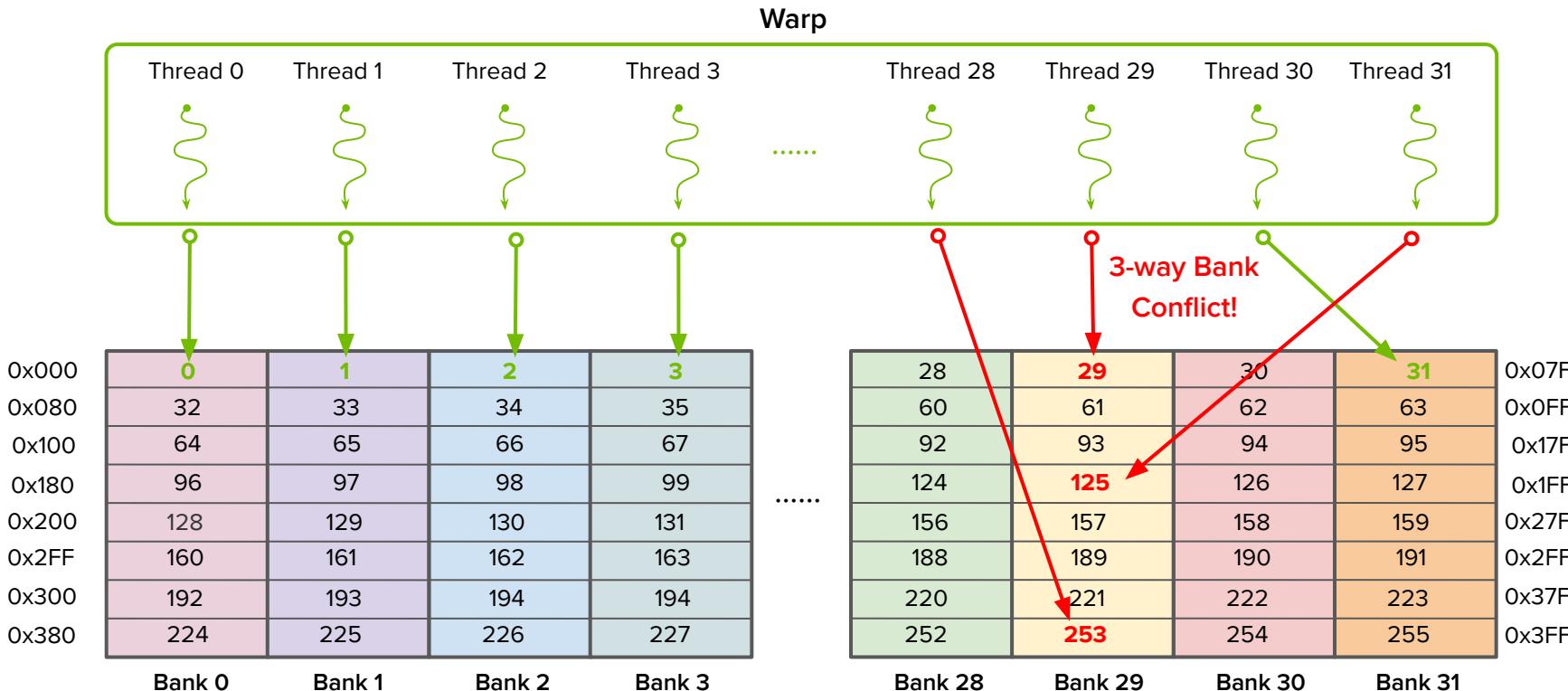
# Modalità di Accesso - Irregolare con Potenziali Conflitti (2/2)

Warp



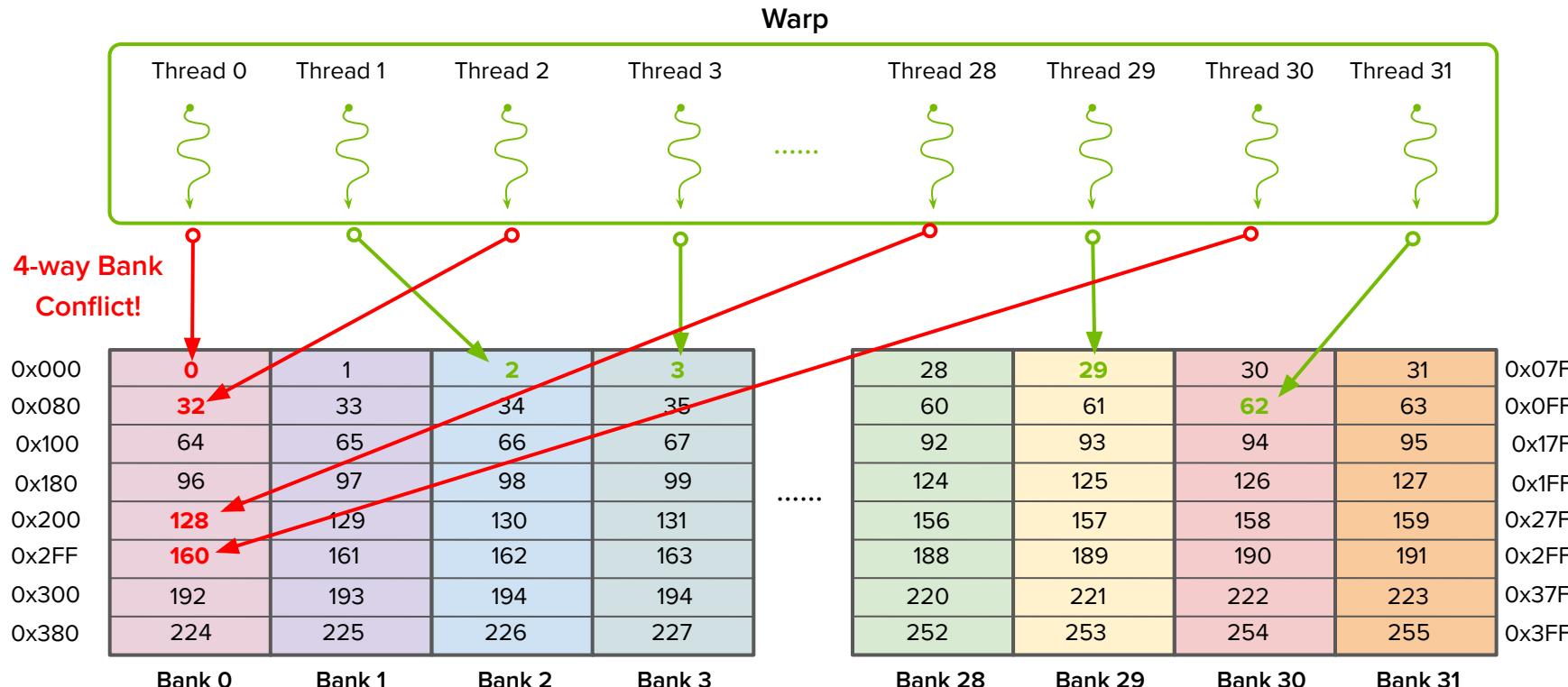
- Più thread accedono allo stesso bank.
- (Secondo Scenario) Conflitto di bank (indirizzi diversi nello stesso bank) → Inefficiente.

# Modalità di Accesso - Irregolare con Potenziali Conflitti (2/2)



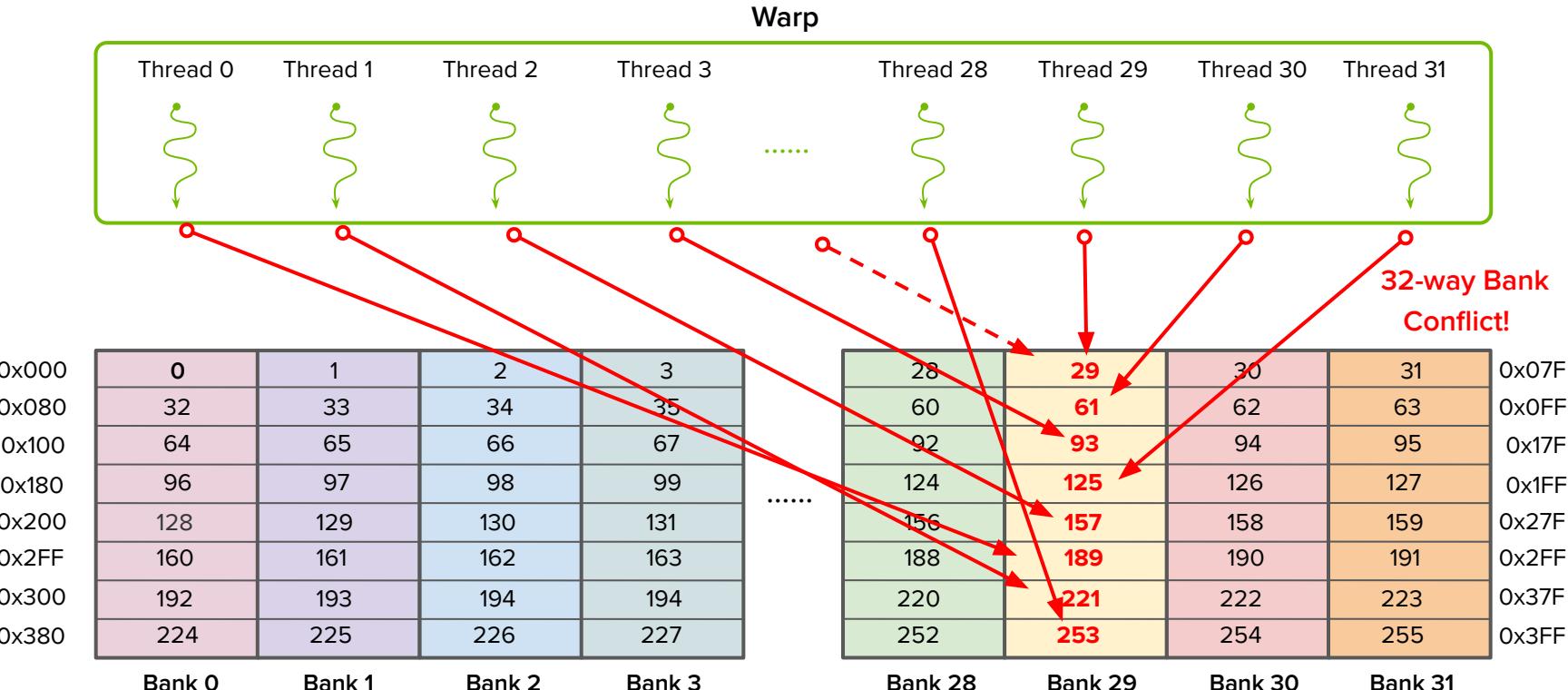
- Più thread accedono allo stesso bank.
- (Secondo Scenario) Conflitto di bank (indirizzi diversi nello stesso bank) → Inefficiente.

# Modalità di Accesso - Irregolare con Potenziali Conflitti (2/2)



- Più thread accedono allo stesso bank.
- **(Secondo Scenario) Conflitto di bank (indirizzi diversi nello stesso bank)** → Inefficiente.

# Modalità di Accesso - Irregolare con Potenziali Conflitti (2/2)



- Worst-case Scenario: Tutti thread accedono allo stesso bank.
- (Secondo Scenario) Conflitto di bank (indirizzi diversi nello stesso bank) → Estremamente inefficiente.

# Mapping degli Indirizzi e Bank Conflicts nella SMEM

- Il pattern di accesso dei thread di un warp alla memoria condivisa determina i conflitti di banco.

## Conflitti di Banco: Effetti dei Pattern di Accesso

```
__shared__ int arr[2048];
int value;

// Accesso senza conflitti di banco: ogni thread accede ad un banco diverso.
// Tutte le transazioni sono gestite in parallelo.
value = arr[threadIdx.x];

// Accesso con conflitti di banco a 2 vie: due thread accedono allo stesso banco.
// Questo causa una serializzazione a 2 livelli.
value = arr[threadIdx.x * 2];

// Accesso con conflitti di banco a 32 vie: tutti i thread del warp accedono allo stesso banco.
// Questo causa una serializzazione a 32 livelli.
value = arr[threadIdx.x * 32];
```

- Nota:** La presenza di conflitti di banco può essere analizzata e verificata tramite **Nsight Compute**.

# Configurare la Shared Memory

## Configurare la Memoria Condivisa [[Documentazione Online](#)]

- Le GPU moderne utilizzano **banchi di dimensione fissa a 32 bit (4 byte)** e una **cache unificata** (shared memory + cache L1) configurabile dinamicamente.
- La capacità di SMEM **varia** in base alla [Compute Capability](#) (es: Hopper/CC 9.0 → la cache unificata ha dimensione massima 256 KB e la capacità della SMEM può essere impostata a 0, 8, 16, 32, 64, 100, 132, 164, 196 o 228 KB).
- La configurazione della shared memory viene **gestita automaticamente dal driver CUDA** per ottimizzare le prestazioni e l'esecuzione concorrente, che però non ha sempre una visione completa del workload.
- È possibile fornire **suggerimenti** per kernel specifici tramite:

```
cudaFuncSetAttribute(kernel_name, cudaFuncAttributePreferredSharedMemoryCarveout, carveout);
```

- Dove **carveout**:
  - Può essere specificato come **percentuale intera della capacità massima supportata**.
  - Valori predefiniti: {`cudaSharedmemCarveoutDefault(-1)`, `cudaSharedmemCarveoutMaxL1(0)`, `cudaSharedmemCarveoutMaxShared(100)`}
  - Se la percentuale richiesta non corrisponde a una capacità supportata, **viene arrotondata alla capacità superiore disponibile**.
  - Il carveout è un **hint** per il driver, che può scegliere una configurazione diversa da quella suggerita.

# Configurare la Shared Memory

## Imposta la Preferenza per la Shared Memory

```
// Codice Device
__global__ void MyKernel(...)

{
    __shared__ float buffer[BLOCK_DIM];
    ...
}

// Codice Host
int carveout = 50; // preferire la capacità di memoria condivisa al 50% rispetto al massimo
// Valori di Carveout:
// carveout = cudaSharedmemCarveoutDefault;    // (-1)
// carveout = cudaSharedmemCarveoutMaxL1;        // (0)
// carveout = cudaSharedmemCarveoutMaxShared; // (100)
cudaFuncSetAttribute(MyKernel, cudaFuncAttributePreferredSharedMemoryCarveout, carveout);
MyKernel <<<gridDim, BLOCK_DIM>>>(...);
```

# Configurare la Shared Memory

## Allocazione di Shared Memory Estesa [Documentazione Online]

- Le allocazioni di shared memory oltre **48 KB** per blocco sono specifiche dell'architettura.
- Richiedono sempre l'utilizzo di **memoria condivisa dinamica**.
- Necessitano di una **configurazione esplicita** tramite **cudaFuncSetAttribute ()**.

### Imposta la Preferenza per la Shared Memory

```
// Codice Device
__global__ void MyKernel(...)

{
    extern __shared__ float buffer[];
    ...
}

// Codice Host
int maxbytes = 98304; // 96 KB
cudaFuncSetAttribute(MyKernel, cudaFuncAttributeMaxDynamicSharedMemorySize, maxbytes);
MyKernel <<<gridDim, blockDim, maxbytes>>>(...);
```

# Confronto tra Memoria Condivisa e Cache L1

## Trade-off tra Memoria Condivisa e Cache L1

- La configurazione ottimale dipende dall'intensità di utilizzo della shared memory e della cache L1 nel kernel:
  - **Più Memoria Condivisa:**
    - Ideale quando i kernel fanno un uso intensivo della memoria condivisa per ridurre la latenza negli accessi alla memoria globale.
    - L'uso intensivo della shared memory può limitare l'occupancy (risorsa on-chip condivisa fra blocchi).
  - **Più Cache L1:**
    - Preferibile quando i kernel accedono frequentemente a dati globali con una buona località spaziale.
    - Utile per ridurre lo spilling dei registri nella memoria globale, migliorando le prestazioni complessive.

## Differenze tra Memoria Condivisa e Cache L1

- Sebbene condividano lo stesso hardware on-chip, ci sono differenze fondamentali:
  - **Accesso:**
    - La memoria condivisa usa 32 banchi per l'accesso parallelo.
    - La cache L1 si basa su linee di cache (es. 128 byte) per il caricamento dei dati.
  - **Controllo:**
    - La memoria condivisa offre pieno controllo programmabile su cosa viene memorizzato e dove.
    - La cache L1 è gestita automaticamente dall'hardware, senza intervento del programmatore.

# Riferimenti Bibliografici

## Testi Generali

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