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

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

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

- 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 × 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:

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

AI<Soglia: *Memory Bound*
AI>Soglia: *Compute Bound*

- **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

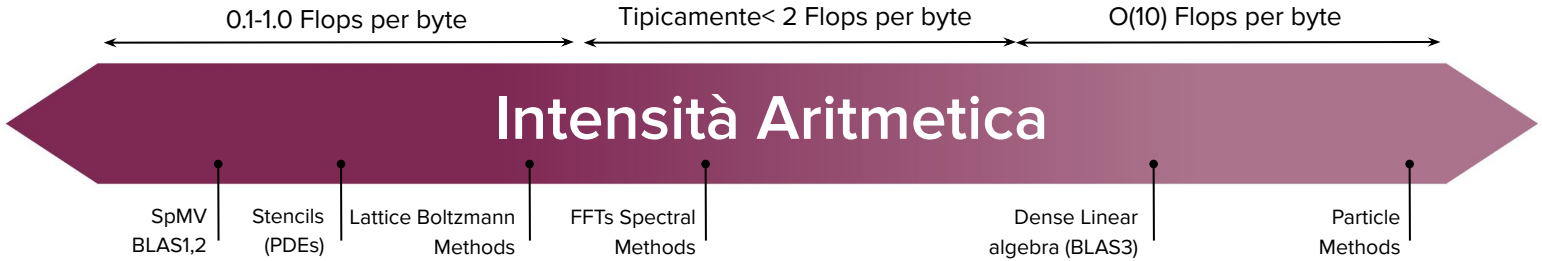
- La soglia dipende dall'hardware specifico (es. GPU), ed è definito dal seguente rapporto:

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

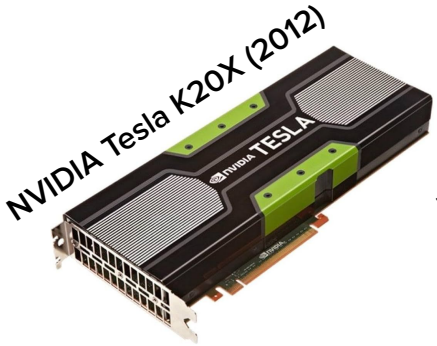
- **Computational Peak Performance**: Massima capacità teorica di calcolo di un dispositivo, misurata in FLOPS.
- **Bandwidth Peak Performance**: Velocità massima con cui i dati possono essere trasferiti tra GPU e memoria principale.

Intensità Aritmetica: Algoritmi e Hardware

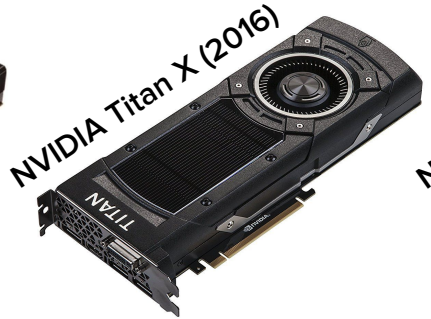
Intensità Aritmetica degli Algoritmi HPC



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):**

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

- **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):

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

- **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.

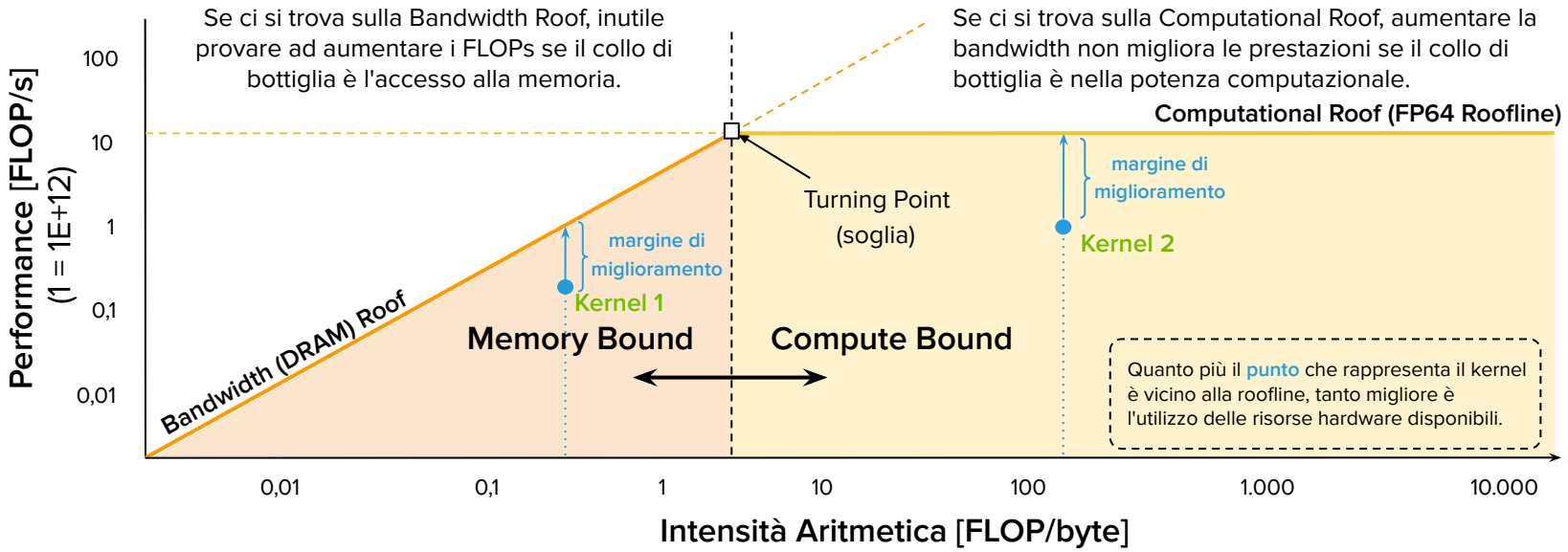
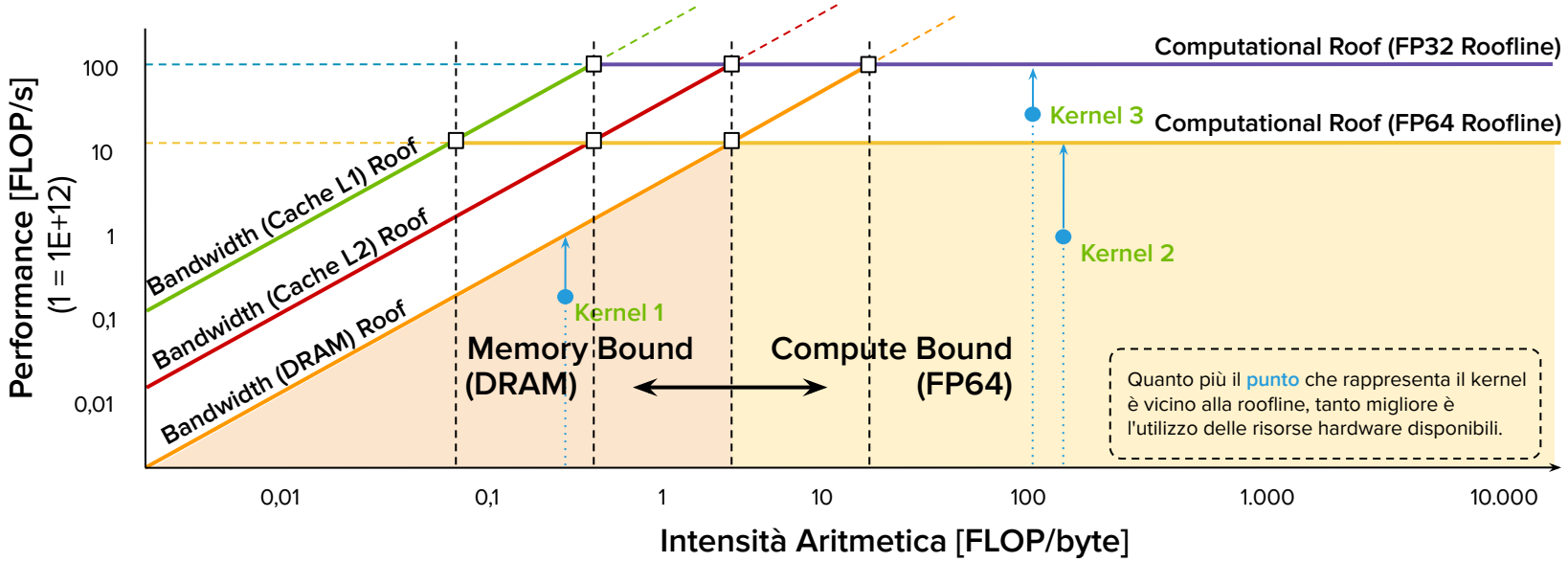


Diagramma Roofline

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).



Panoramica del Modello di Memoria CUDA

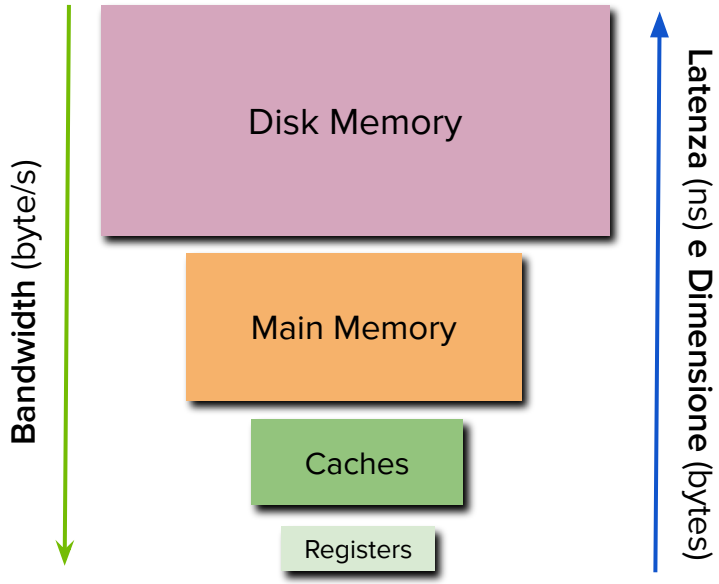
- Modelli di Performance
  - Memory-Bound vs Compute-Bound
  - Intensità Aritmetica e Roofline Model
- **Gerarchia di Memoria CUDA**
  - Organizzazione Gerarchica Completa
  - Scope e Programmabilità
- Gestione della Memoria Host-Device
  - Allocazione e Trasferimenti
  - Pinned Memory
  - Zero-Copy 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

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 Memoria DDR e GDDR

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

Confronto tra Memoria GDDR e HBM nelle GPU NVIDIA

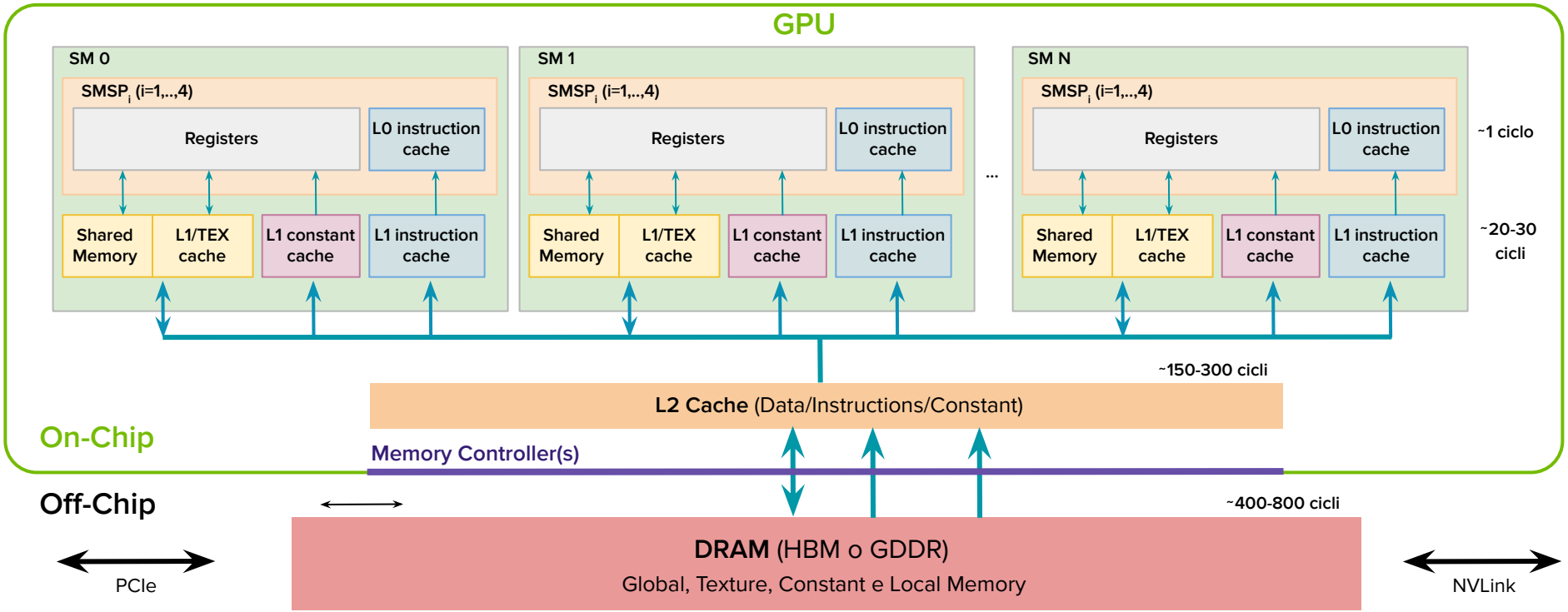
	GDDR (Graphics Double Data Rate)	HBM (High Bandwidth Memory)
Esempio GPU	NVIDIA RTX 4090 Ada (GDDR6X)	NVIDIA H100, GH200 (HBM3, HBM3e)
Utilizzo	Grafica avanzata, AI su piccola scala, rendering	HPC, training AI intensivo, inferenza AI in tempo reale
Architettura	Chip singoli saldati al PCB	Moduli impilati 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

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.

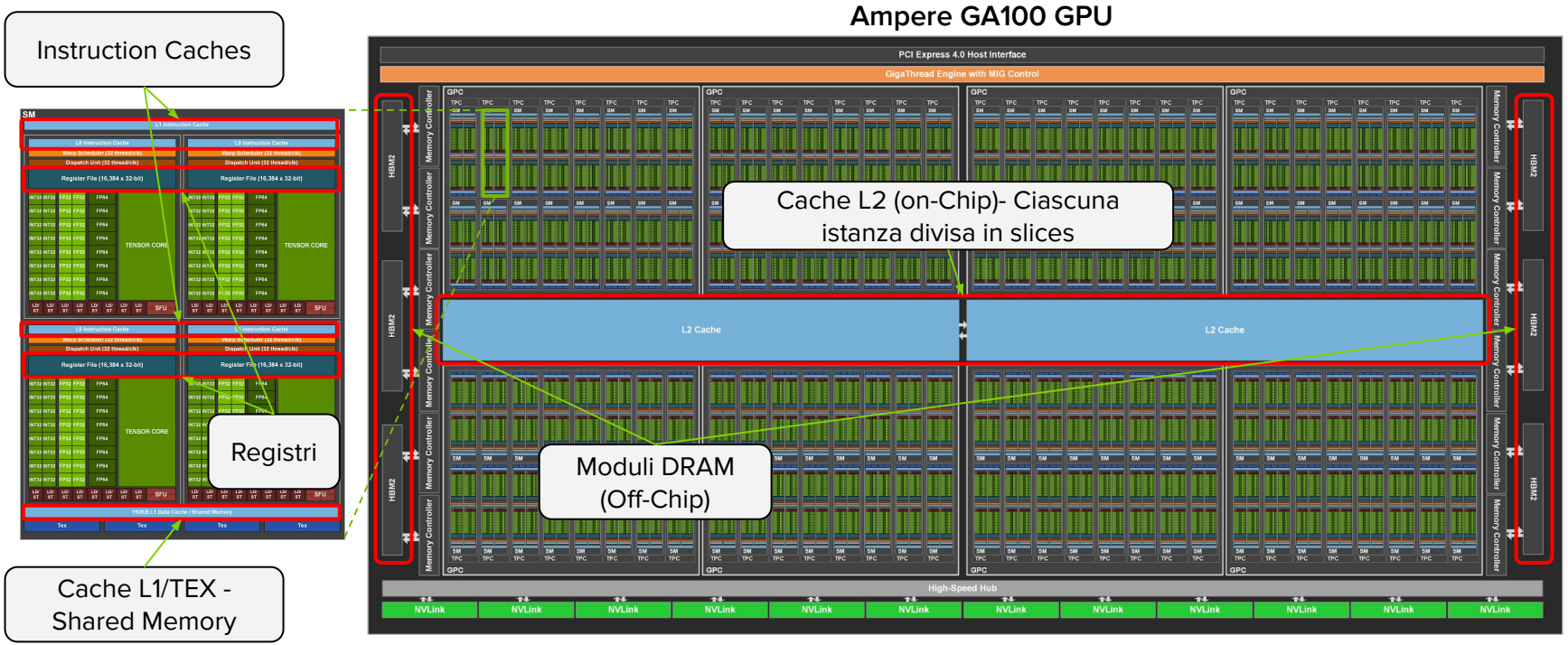
Modelli di Memoria CUDA

- Agli occhi dei programmatori, 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 programmatori, 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 (maggiore 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 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 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.

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
Registro	On	n/a	R/W	Thread	Thread
Condivisa	On	n/a	R/W	Thread nel Blocco	Blocco
Locale	Off	†	R/W	Thread	Thread
Globale	Off	†	R/W	Tutti i Thread + Host	Allocazione Host
Costante	Off	Sì	R	Tutti i Thread + Host	Allocazione Host
Texture	Off	Sì	R	Tutti i Thread + Host	Allocazione Host

† 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
	`float LocalVar`	Registro	Thread	Thread
	`float LocalVar[100]`	Locale	Thread	Thread
`__shared__`	`float SharedVar` †	Condivisa	Blocco	Blocco
`__device__`	`float GlobalVar` †	Globale	Globale	Applicazione
`__constant__`	`float ConstantVar` †	Costante	Globale	Applicazione

† Può essere una variabile scalare o una variabile array

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

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

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

Firma della Funzione (Documentazione Online)

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

Parametri

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

Tipi di Trasferimento (`kind`)

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

Valore di Ritorno

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

Note importanti

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

Deallocazione della Memoria sul Device

Ruolo della Funzione

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

Firma della Funzione (Documentazione Online)

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

Parametri

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

Valore di Ritorno

- **`cudaError_t:`** Codice di errore (**`cudaSuccess`** se la deallocazioneha successo).

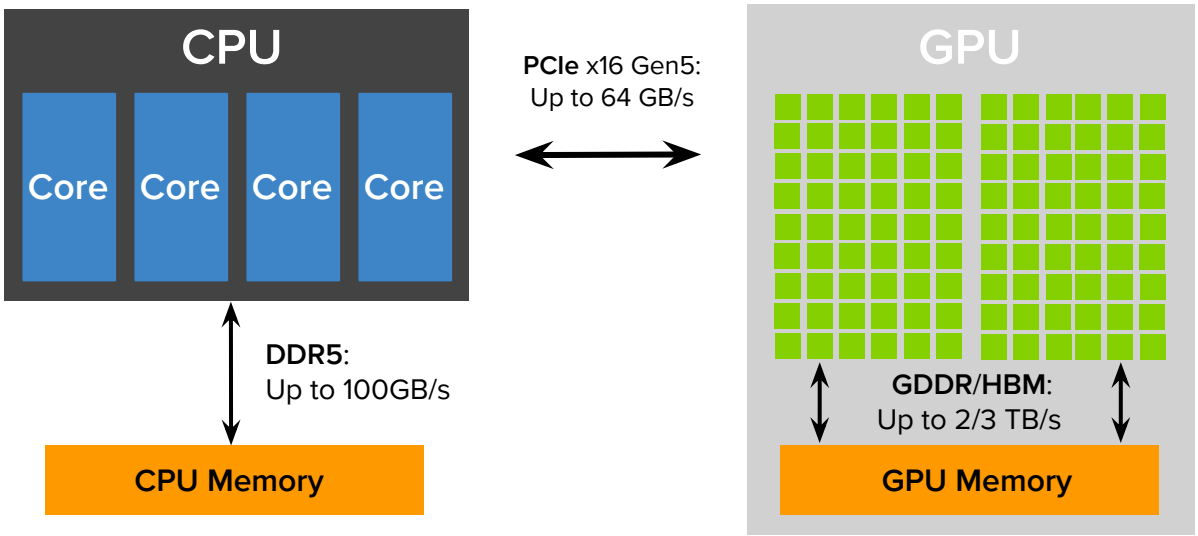
Note Importanti

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

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.



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).

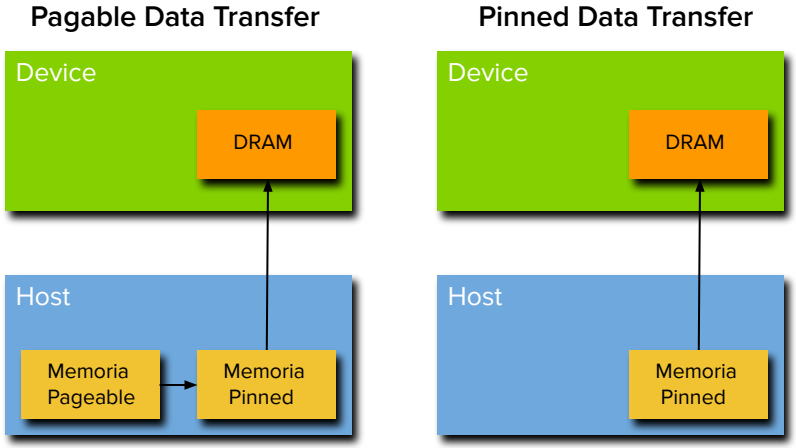
Come avviene allora il trasferimento da Memoria Pageable?

- 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.



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

```
// alloca la memoria sull'host
h_a = (float *)malloc(nbytes);
// alloca 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.

Con Memoria Pinned

```
// alloca memoria pinned sull'host
CHECK(cudaMallocHost((float **)&h_a, nbytes));
// alloca 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

- 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.

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 espliciti tra host e device.

Vantaggi

- **Sfruttamento della memoria host**: Consente di usare la memoria dell'host quando quella del device è insufficiente.
- **Eliminazione trasferimenti espliciti**: 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);
```

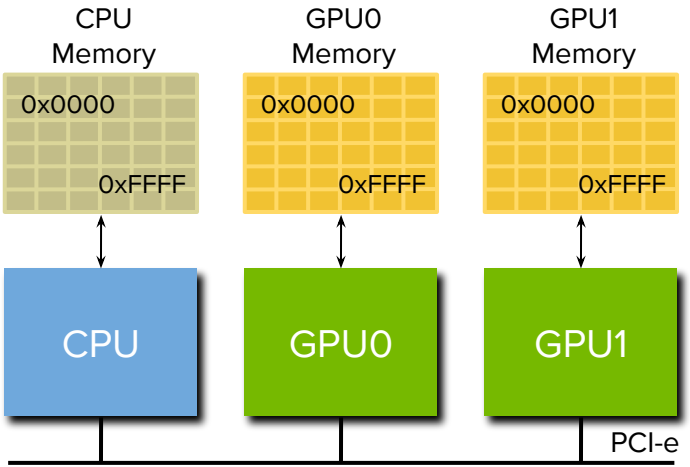
Note

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

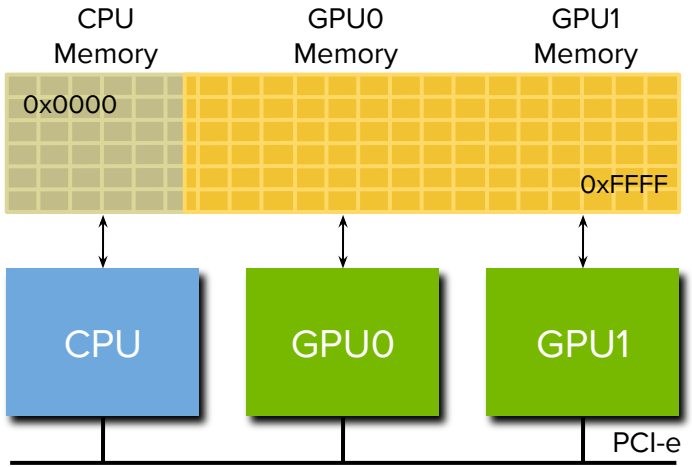
Unified Virtual Addressing (UVA)

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

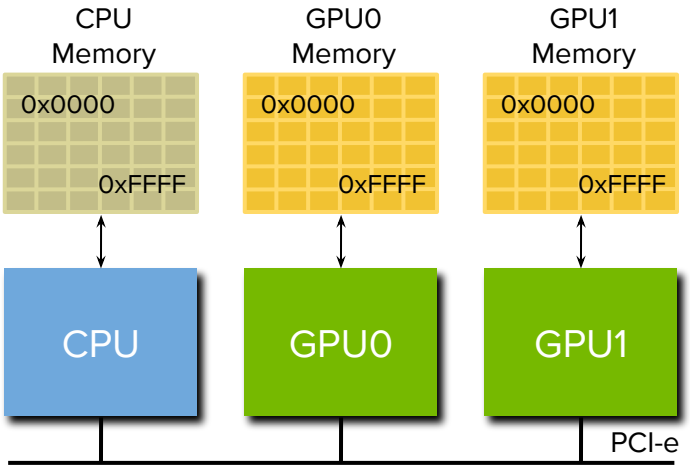


UVA: spazio di memoria singolo

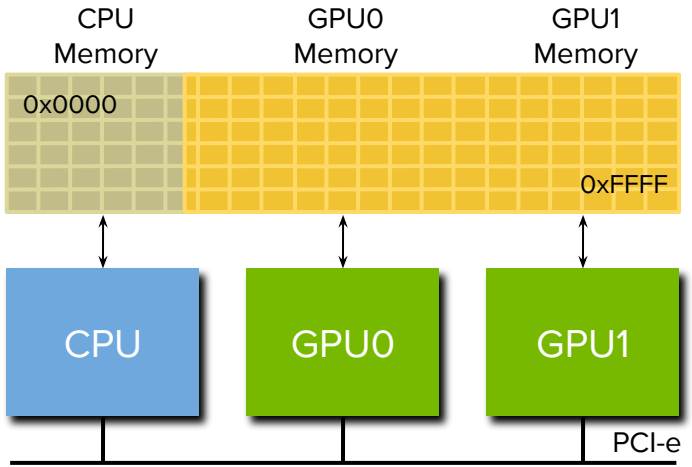
Unified Virtual Addressing (UVA)

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

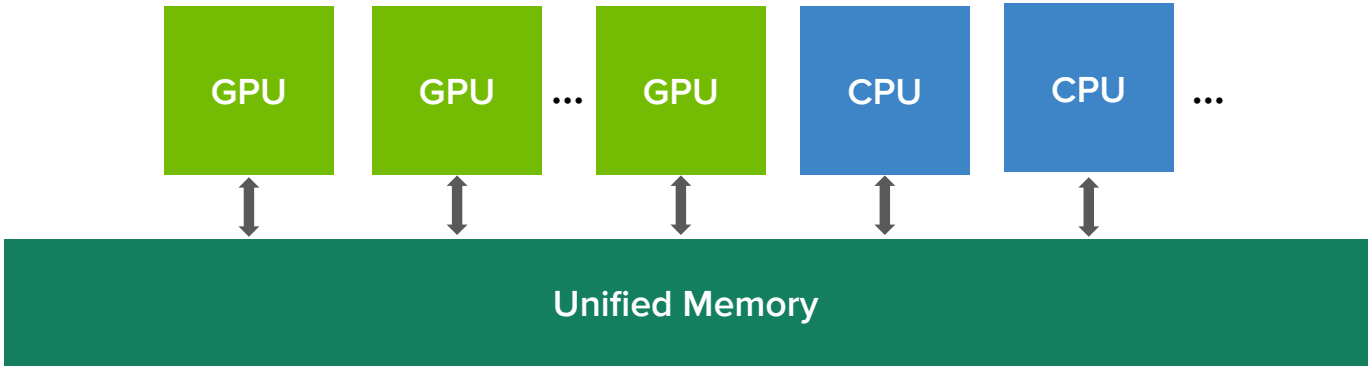


UVA: spazio di memoria singolo

Unified 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.

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.

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;
}
```

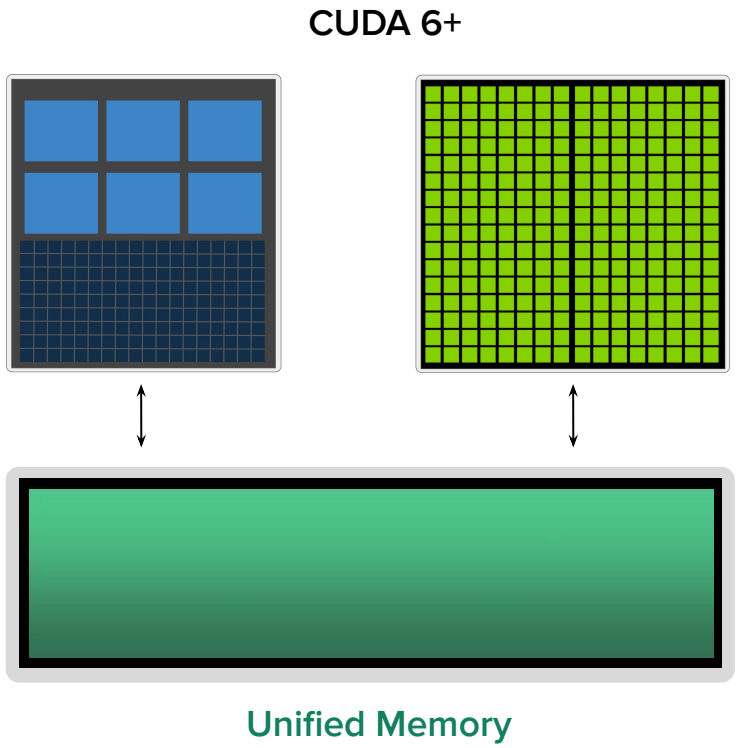
Unified Memory (UM)

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.



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.

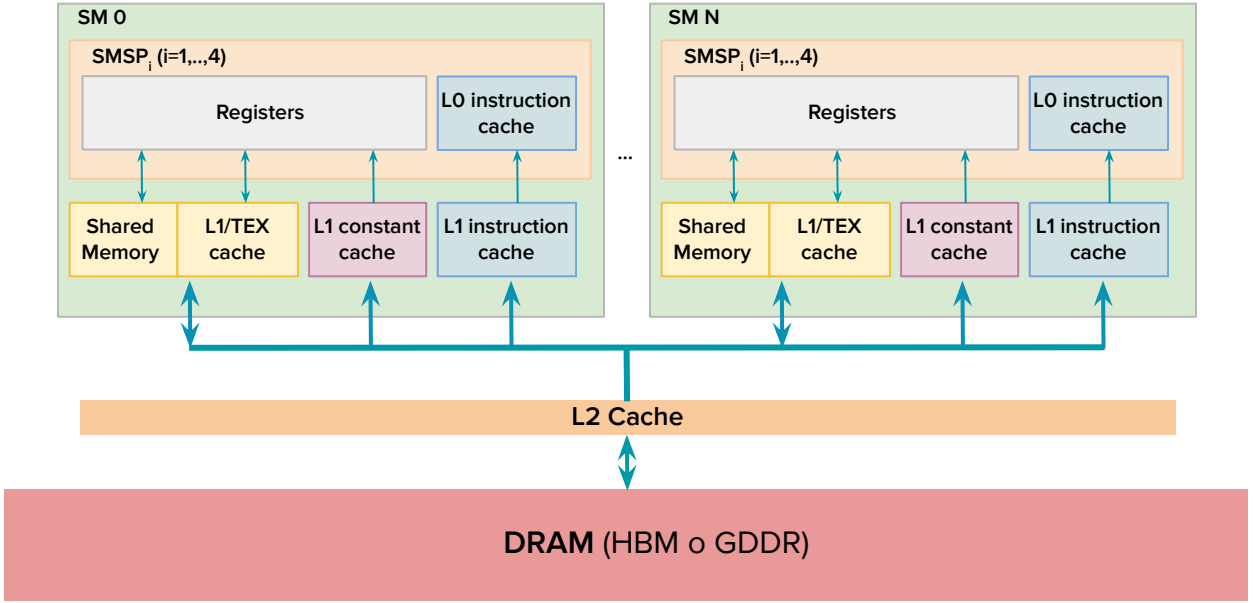
Pattern di Accesso alla Memoria

- 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

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.

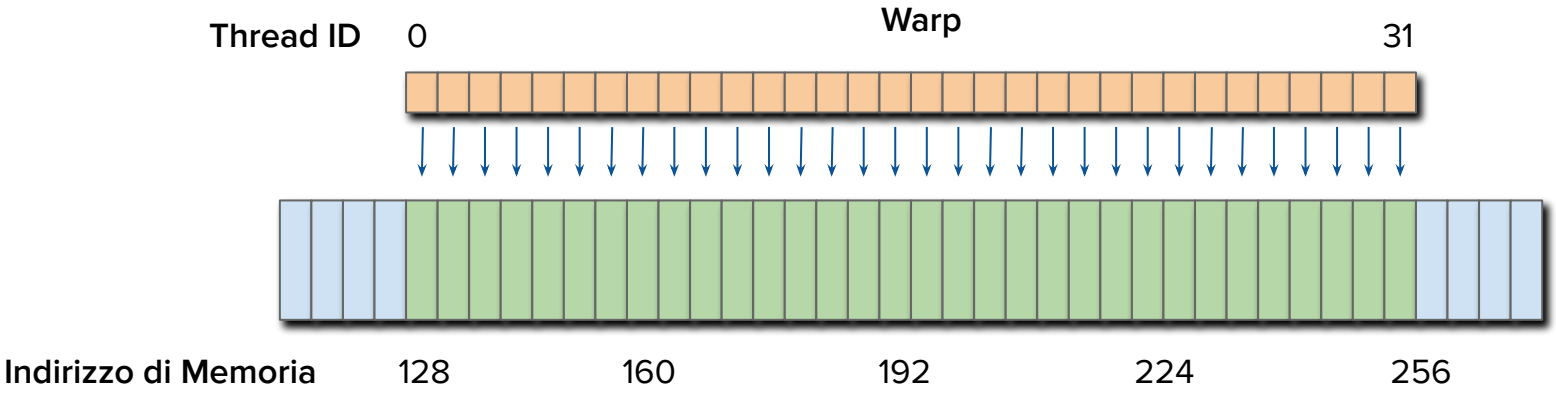
Accessi Allineati e Coalescenti

- 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 ✅

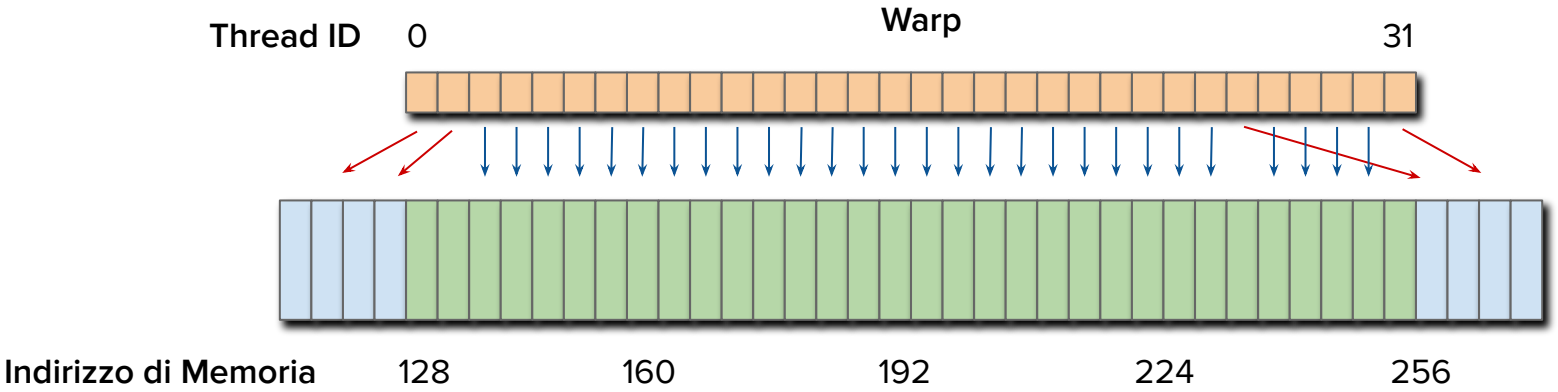
- Una **singola transazione** da 128 byte (4 byte per thread in un warp) recupera tutti i dati necessari.
- Utilizzo **ottimale** della larghezza di banda.
- **Riduzione** del numero totale di transazioni di memoria.
- **Minimizzazione** della latenza di accesso ai dati.



Accessi Disallineati e Non Coalescenti in CUDA

Esempio: Accesso Disallineato e Non Coalescente ❌

- 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).



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 ≥ 6.0**: Cache L1 è abilitata di default.

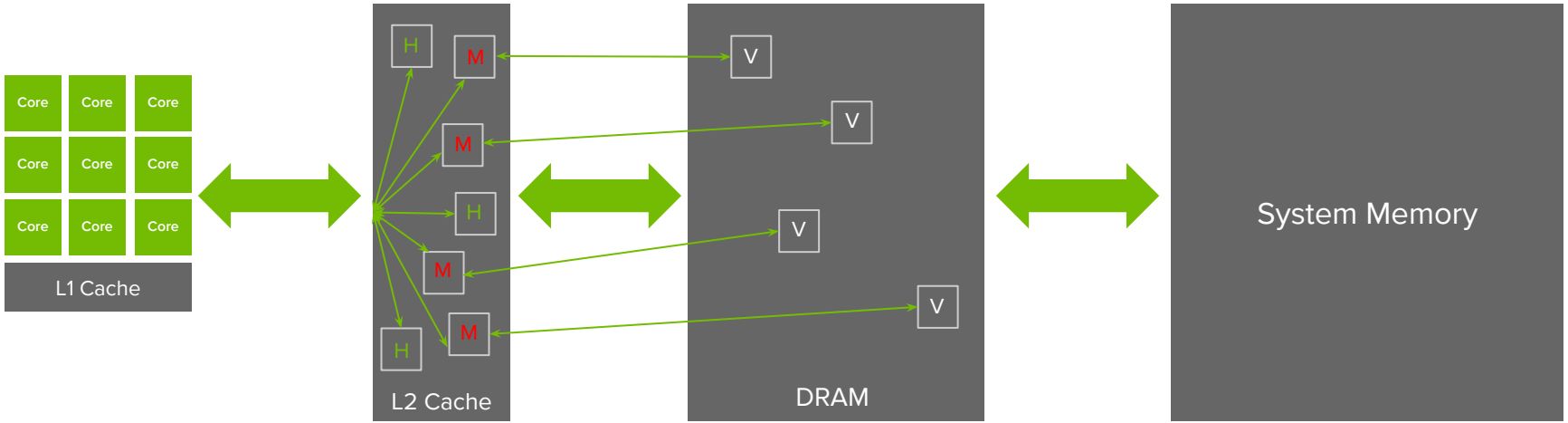
Controllo della cache L1 tramite flag del compilatore nvcc

- **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 ≤ 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.



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**.

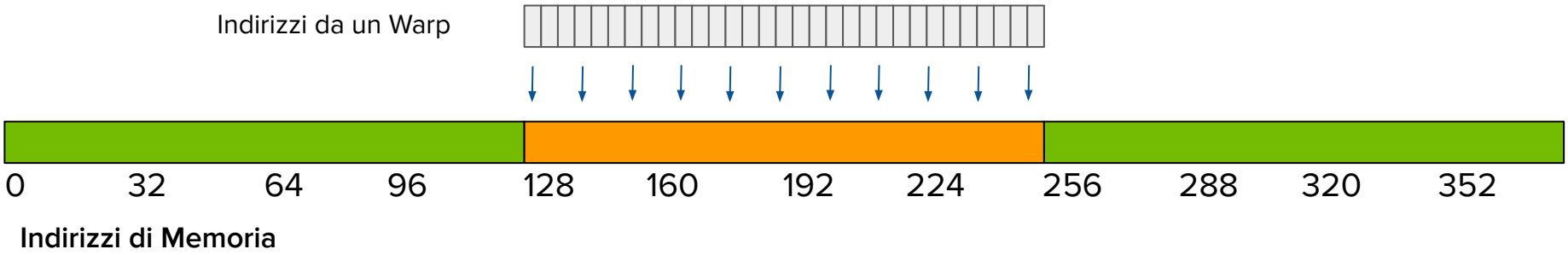
Caratterizzazione dei Pattern di Accesso

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.

Cached Loads: Allineato e Coalescente

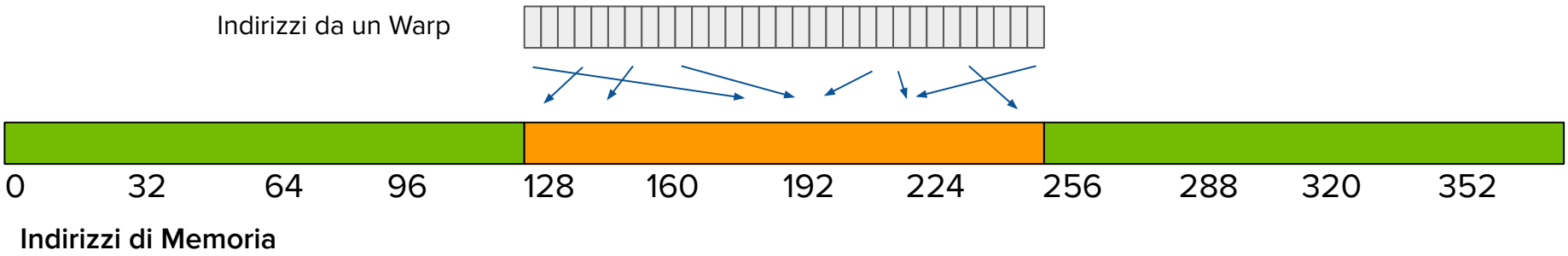
- Caso **ideale** di accesso alla memoria con **transazioni di dimensioni 128 byte**.
- Tutti gli indirizzi richiesti dai thread in un warp cadono all'interno di una **singola linea di cache** da 128 byte.
- Richiede **una sola transazione** di memoria da 128 byte per completare l'operazione di caricamento.
- **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

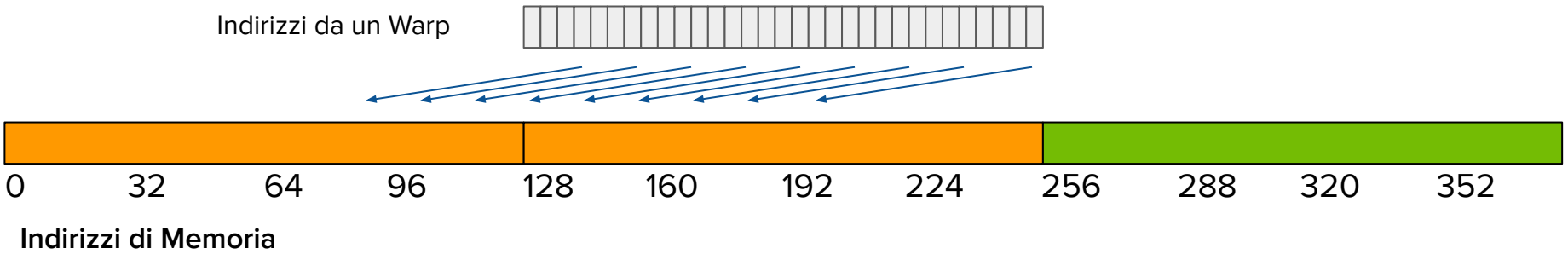
- Gli indirizzi richiesti sono **contigui in memoria**, **non consecutivi rispetto al thread ID** e **distribuiti casualmente**.
- Gli indirizzi sono randomizzati ma **confinati all'interno di una singola cache line** di 128 byte.
- La richiesta genera **una sola transazione di memoria** da 128 byte.
- **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

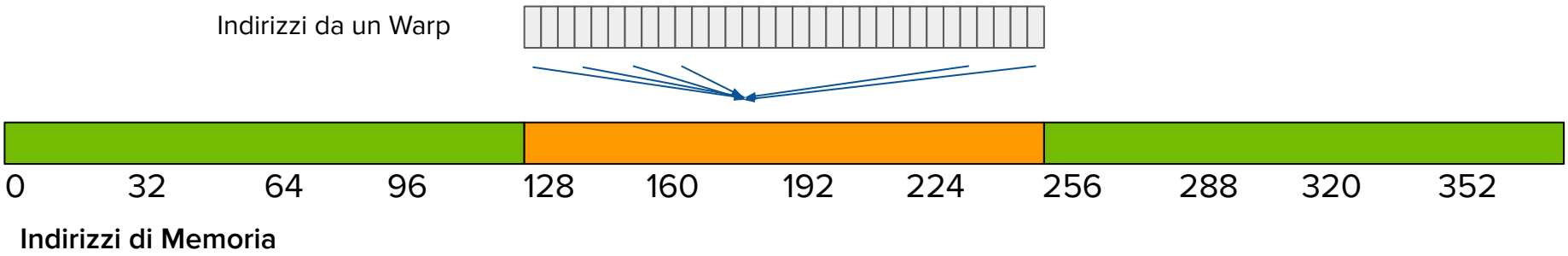
- I thread di un warp richiedono **32 elementi consecutivi di 4 byte** che **non sono allineati**.
- Gli indirizzi richiesti dai thread si estendono su **due segmenti da 128 byte** in memoria globale.
- Il primo indirizzo **non è multiplo** di 128 byte.
- Sono necessarie **due transazioni** da 128 byte 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

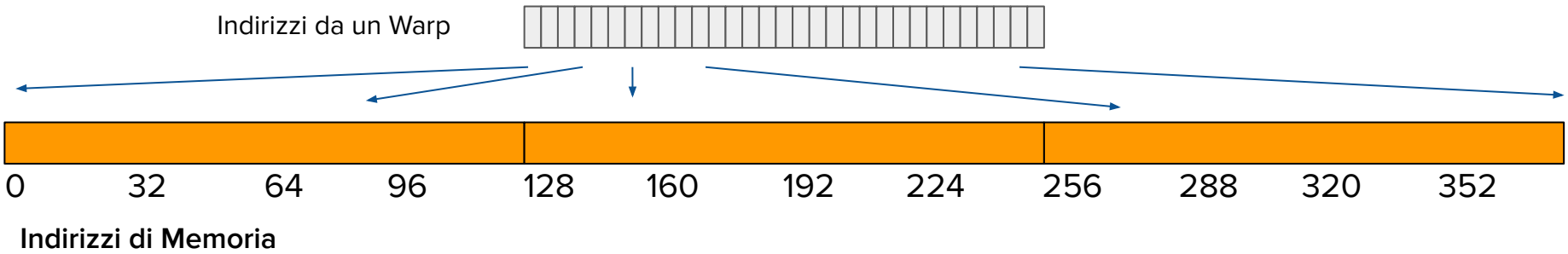
- Tutti i thread in un warp richiedono un dato dallo **stesso indirizzo di memoria**.
- L'indirizzo richiesto cade su una singola linea di cache.
- Richiede **una sola transazione** di memoria da 128 byte.
- **Utilizzo del bus estremamente basso**.
- Per un valore di 4 byte, l'utilizzo effettivo è di **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

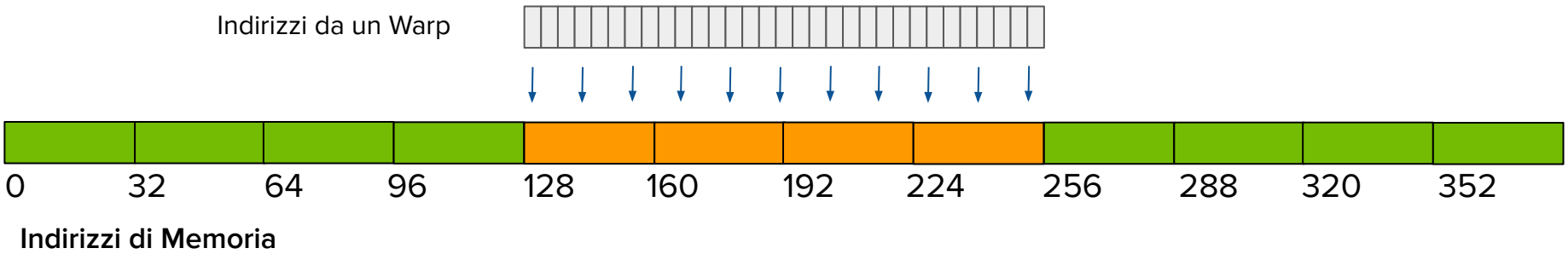
- Gli indirizzi possono estendersi su **N linee di cache**, dove 0 < N ≤ 32.
- Sono necessarie **N transazioni di memoria** per completare una **singola operazione di caricamento**.
- **Worst-case Scenario**: I thread di un warp richiedono **32 indirizzi da 4 byte sparsi** (scattered) nella memoria globale.
- Il totale dei byte richiesti dal warp è solamente di 128 byte (**32 indirizzi × 4 byte per indirizzo**).
- **Utilizzo del bus**: 128 byte richiesti / (N x 128 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

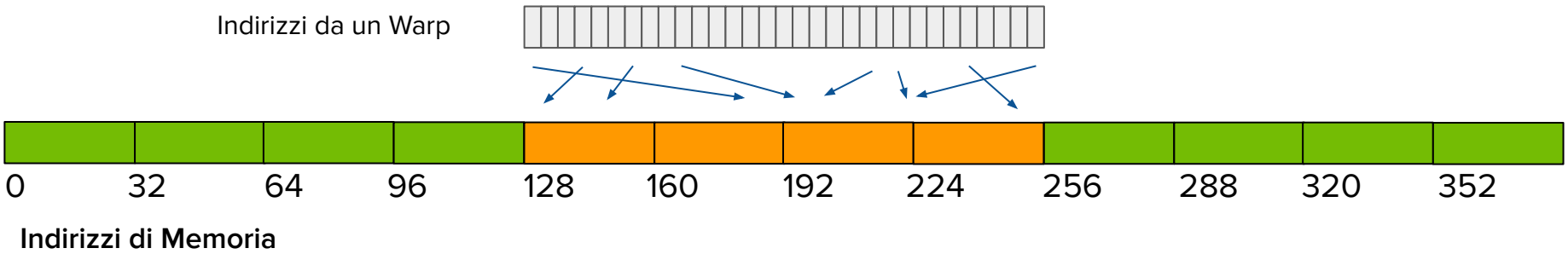
- Granularità dei segmenti di memoria a **32 byte** e non della linea di **cache L1 disabilitata** (128 byte).
- Rappresentazione del caso **ottimale** di accesso alla memoria.
- Il primo indirizzo è un **multiplo** di 32 byte (allineamento rispettato).
- I thread in un warp accedono a **dati contigui** (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 Randomicizzati

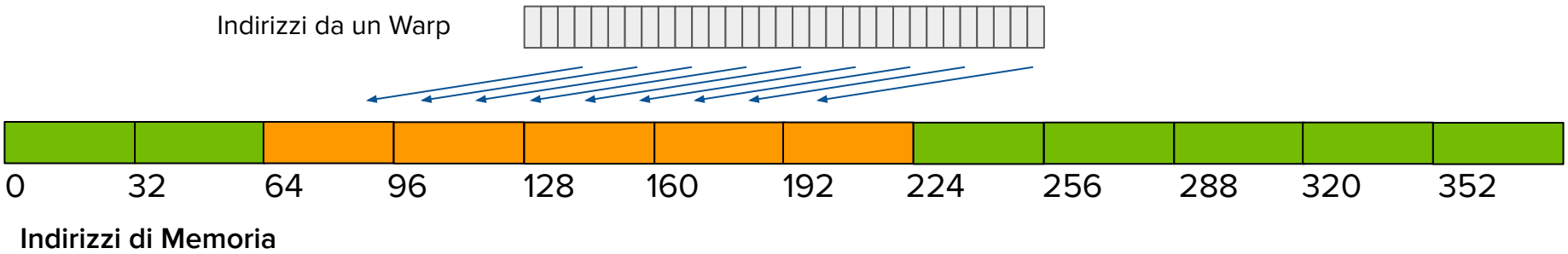
- Il primo indirizzo è un **multiplo** di 32 byte.
- Gli indirizzi richiesti sono **randomizzati** all'interno del range da 128-byte.
- Gli indirizzi ricadono all'interno di **quattro segmenti** da 32 byte.
- Ogni thread richiede un **indirizzo unico nel range**.
- **Utilizzo del bus** al **100%**.
- 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

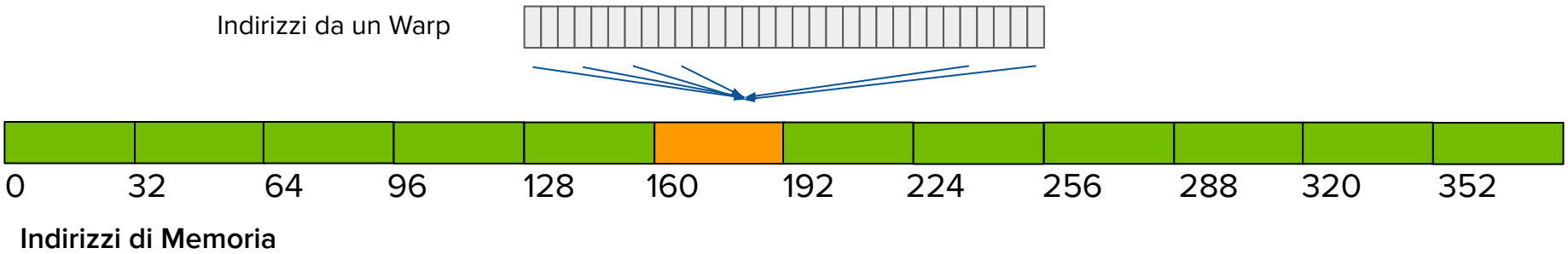
- Warp effettua un caricamento **non allineato** richiedendo **32 elementi consecutivi da 4 byte**.
- Gli indirizzi dei 128 byte richiesti ricadono in massimo **cinque segmenti** da 32 byte.
- **Utilizzo del bus** al **80%** (128 byte richiesti, 160 byte caricati).
- I caricamenti “fine-grained” a 32 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

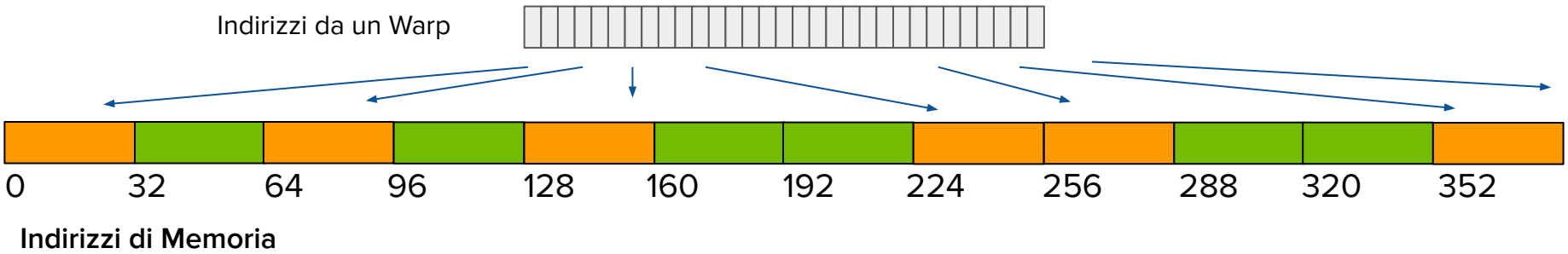
- Tutti i thread in un warp richiedono lo **stesso indirizzo** di memoria.
- L'indirizzo richiesto (anche se disallineato) cade all'interno di un **singolo segmento** da 32 byte.
- **Un solo dato** da 4 byte effettivamente richiesto.
- **Utilizzo del bus** del 12.5% (4 byte richiesti / 32 byte caricati).
- 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 x 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 x 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

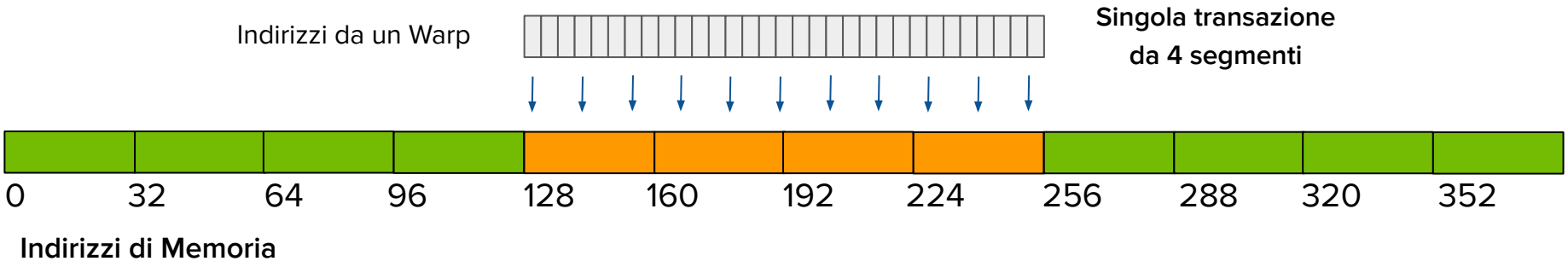
- Se due indirizzi cadono nella stessa regione di 128 byte ma non in una regione allineata di 64 byte:
  - Viene emessa una **singola transazione** di quattro segmenti.
  - **Più efficiente** di due transazioni separate di un segmento ciascuna.

Ottimizzazione

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

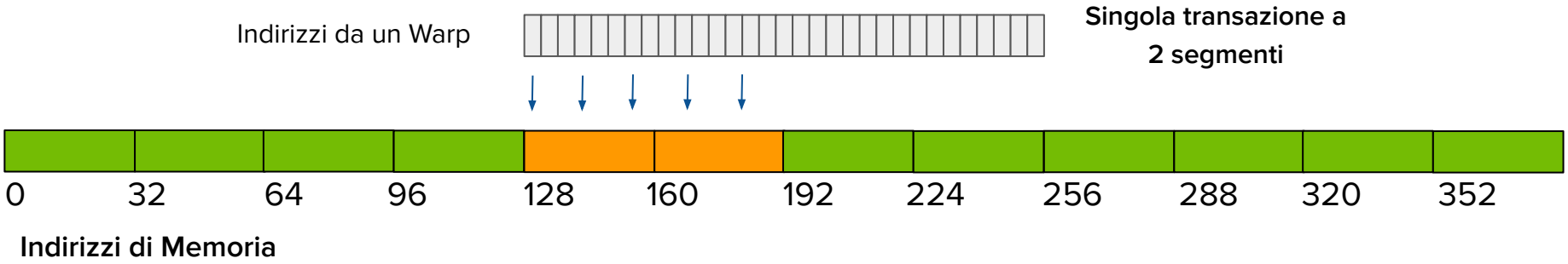
Scrittura in Memoria: Allineata e Coalescente

- 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**).



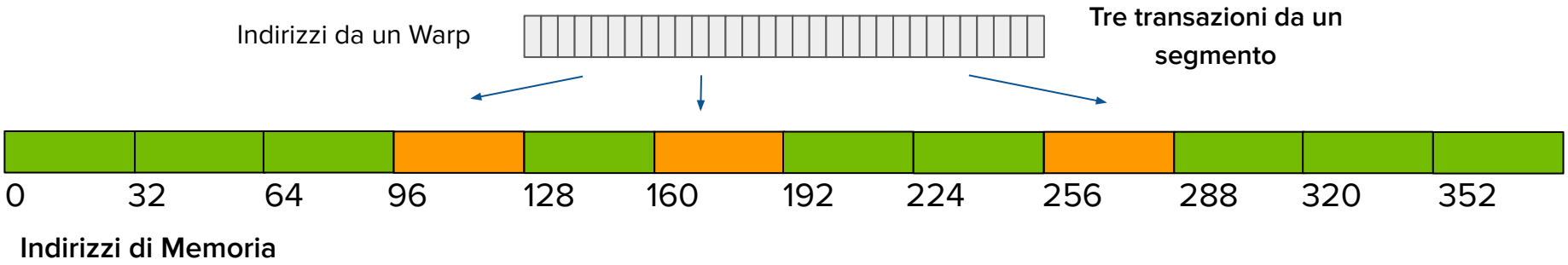
Scrittura in Memoria: Allineata, Coalescente, Range Limitato

- 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: Allineata 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.



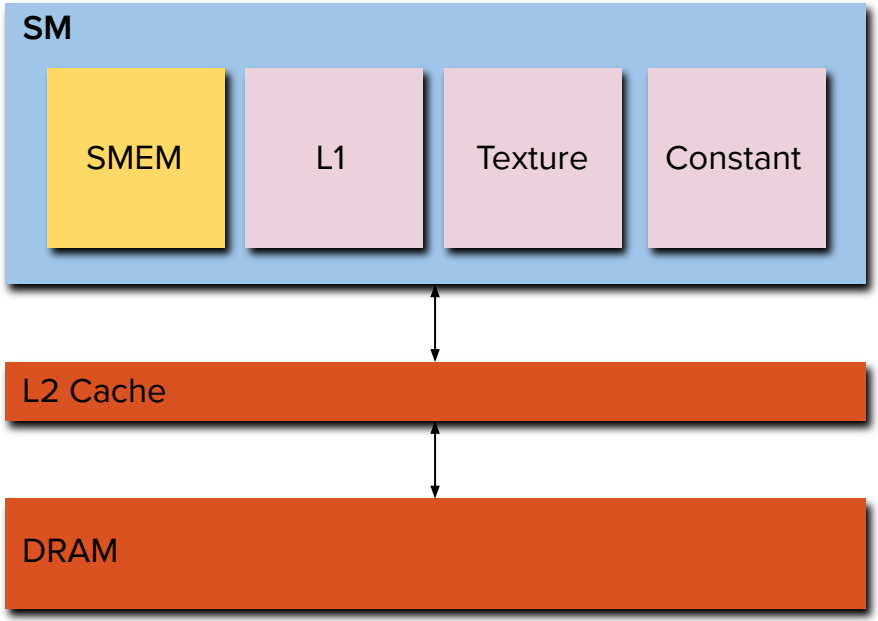
Panoramica del Modello di Memoria CUDA

- ➤ 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.

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 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

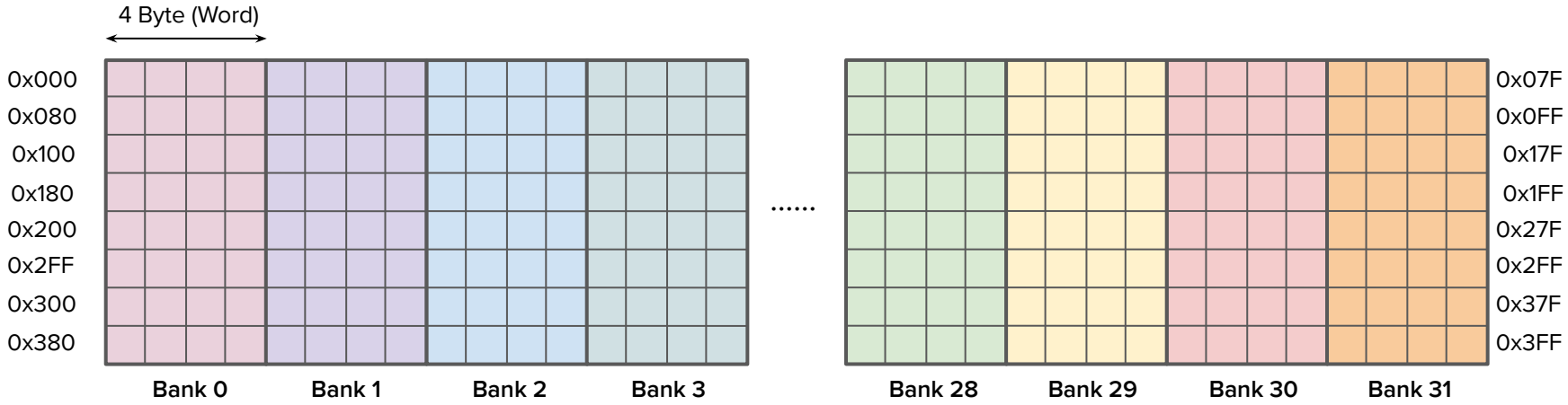
- La memoria condivisa è uno **spazio di indirizzamento lineare** (**1D**), ma viene mappata fisicamente sui banchi.
- La mappatura degli indirizzi ai banchi **varia a seconda della compute capability** della GPU.

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]

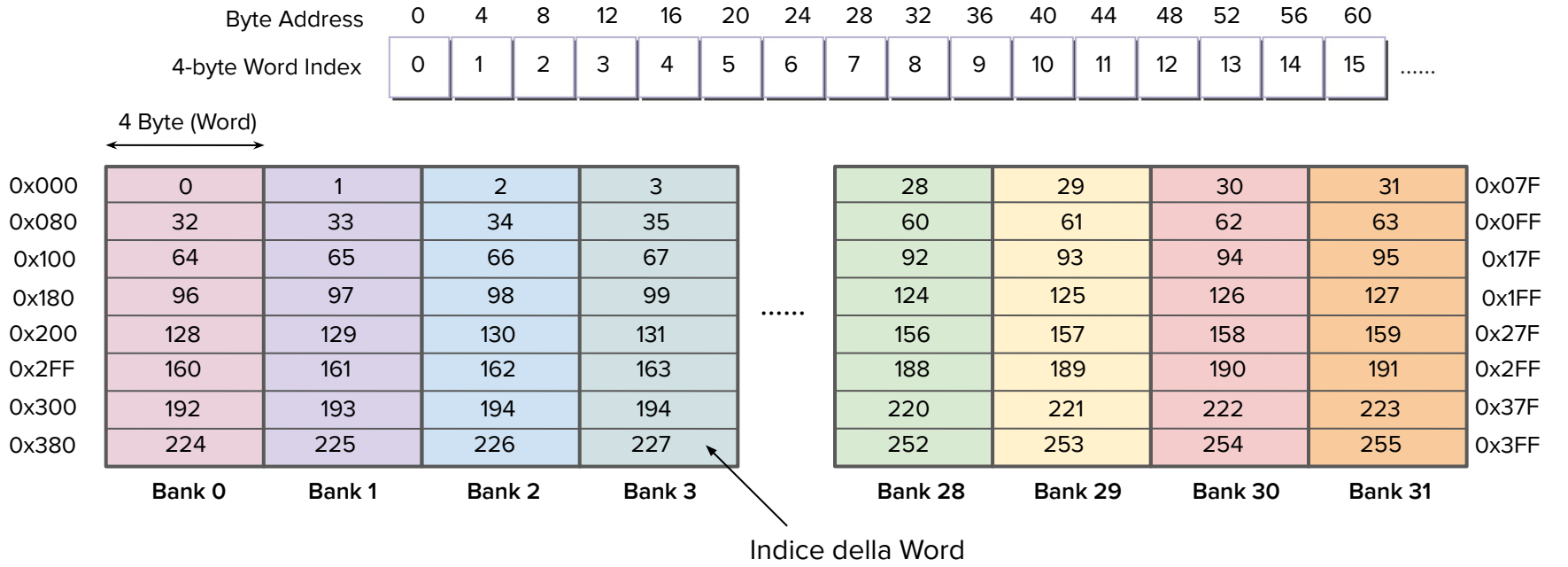
- La larghezza del banco è di **4 byte (32 bit)** e ci sono **32 banchi**.
- Ogni banco supporta trasferimenti paralleli di 32 bit per ciclo di clock.
- Le **parole (word) successive** di 32 bit si mappano alle **banchi successivi**.
- Calcolo dell'Indice del banco:

```
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.

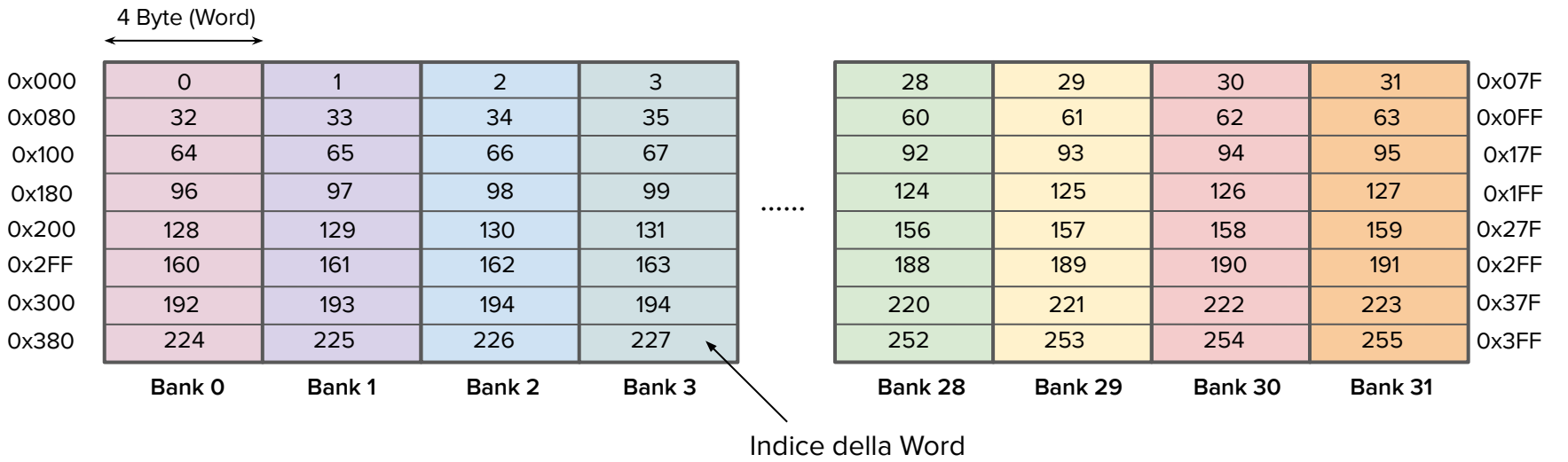
Mappatura della Memoria Condivisa

- **Doppia mappatura**: La figura mostra la relazione tra indirizzo byte, indice di parola (word index) e indice di banco (bank index) nei dispositivi con architetture con larghezza del banco pari a 4 byte.
- **Wrap around**: 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 **intere 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.



Bank Conflict: Collisioni in Memoria Condivisa

Cos'è?

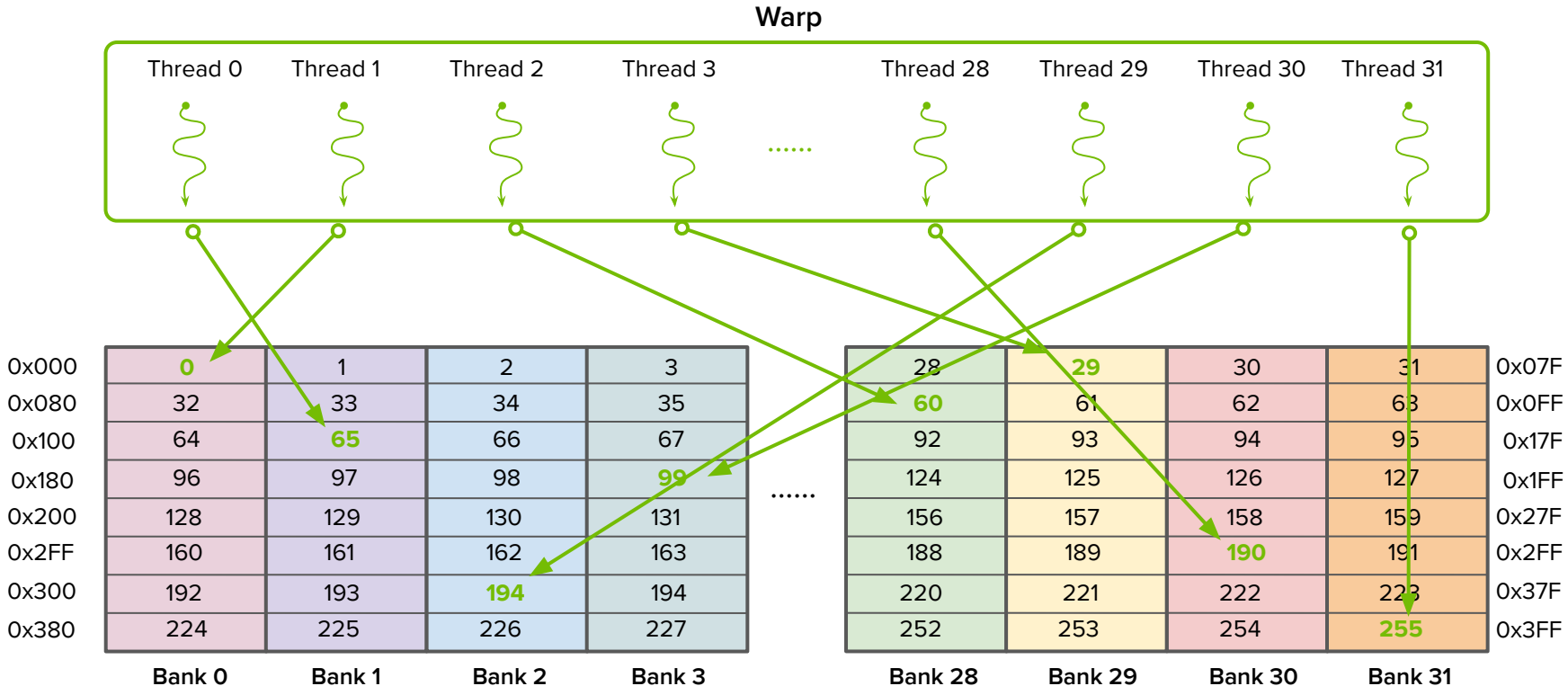
- Un **bank conflict** si verifica quando più thread di un warp accedono a indirizzi diversi nello stesso memory bank.
- L'hardware divide una richiesta con conflitto **in più transazioni separate** (*conflict-free*), riducendo la banda proporzionalmente al numero di transazioni necessarie.
- **Inter-block**: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

Tipi di Accesso

- **Accesso Parallelo** (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.

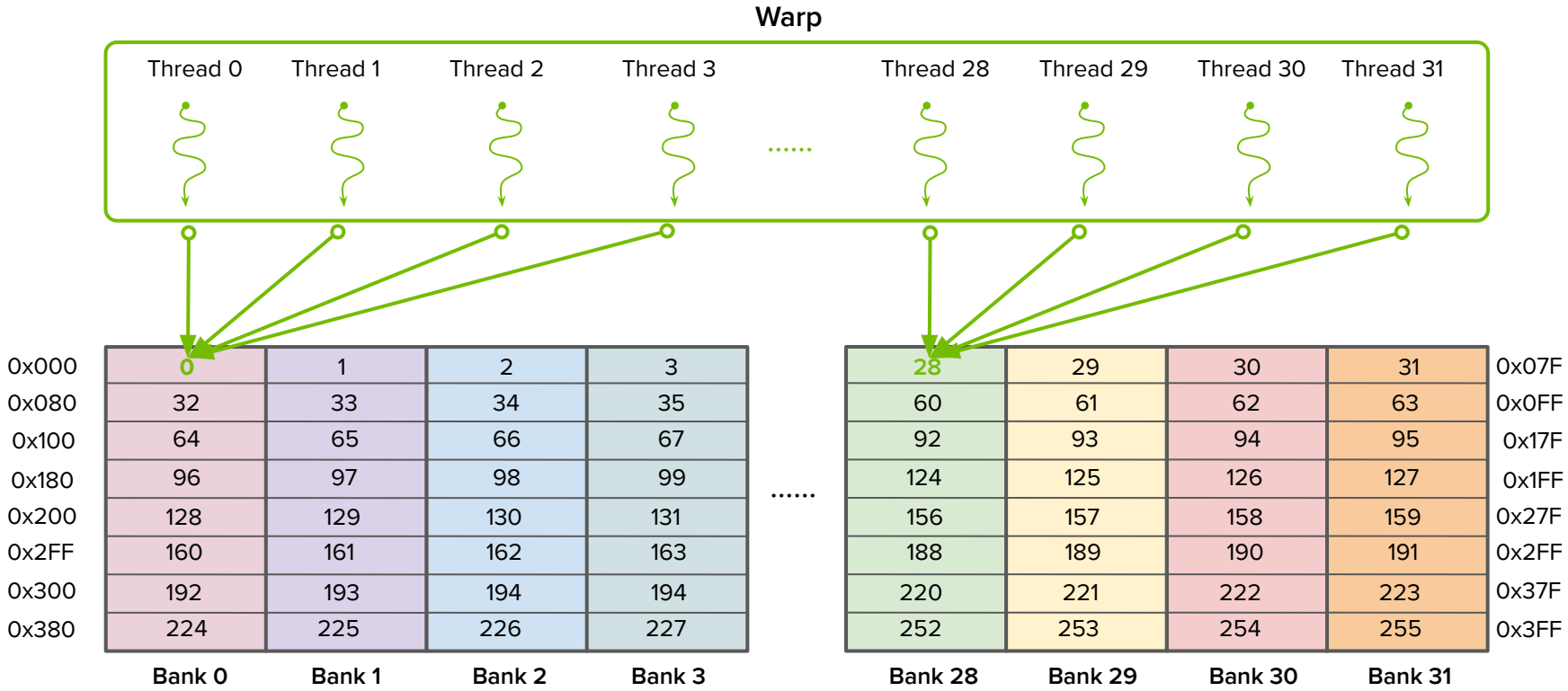
Nota: Le prestazioni, anche con conflitti in SMEM, sono comunque nettamente migliori rispetto all'accesso a cache L2 o, peggio, alla global memory.

Modalità di Accesso - Casuale Senza Conflitti



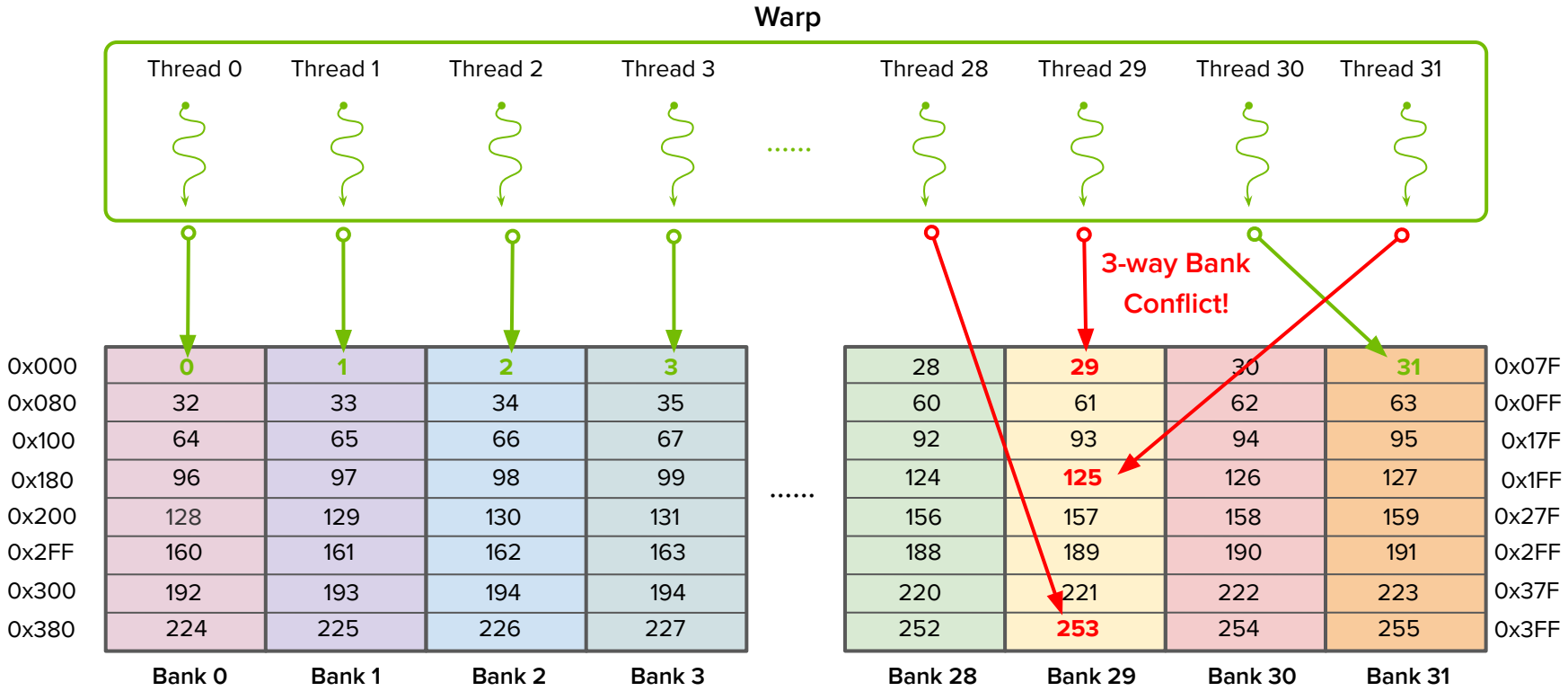
- Ogni thread accede a una parola di 32 bit in un bank **diverso**.
- Ogni thread accede comunque a un **bank diverso**. **Nessun conflitto**, massima efficienza.

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)



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

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.

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^ edizione)
- Kirk, D. B., Hwu, W. W. (2013). **Programming Massively Parallel Processors**. Morgan Kaufmann (3^ edizione)

NVIDIA Docs

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

Risorse Online

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