Modello di Memoria CUDA

back

Indice

Modelli di Performance

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
- Cause comuni:
    - Accessi frequenti alla memoria con latenza elevata.
    - Banda di memoria insufficiente 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 trascorrei più tempo a eseguire calcoli rispetto all'attesa dei dati.
- Cause comuni:
    - Operazione 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	**Bandwidth**
Floating Point Operations per Second	Dati trasferiti al secondo
$\text{FLOPS} = \frac{N_{\text{Floting Point Operations}}}{\text{Tempo Trascorso}}$	$\text{Bandwidth} = \frac{N_{\text{Bytes}}}{\text{Tempo Trascorso}}$
Utilizzata per valutare compute-bound kernel, dove il tempo è dominato dai calcoli.	Utilizzata per valutare memory-bound kernel dove il tempo è dominato dagli accessi in memoria
Unità di misura: MFLOPs, GFLOPs, TFLOPs	Unità di misura: GB/s TB/s
La Peak Performance della GPU rappresenta il limite teorico massimo	La Peak Bandwidth dell'hardware rappresenta il limite teorico massimo raggiungibile

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 in una unità di tempo.
- Kernel Memory Bound: Un kernel è vincolato dalla memoria 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 ha una banda teorica di 177 GB/s, Ampere A100 ha una banda teorica di 1.6 TB/s., Hopper H100 ha una banda teorica di 3.5 TB/s.
- Banda Effettiva:
  - Larghezza di banda realmente raggiunta da un kernel in esecuzione. $$ \text{Banda Effettiva(GB/s)}) = \frac{(\text{byte letti}+\text{byte scritti})x10^{-9}}{\text{tempo trascorso (ns)}} $$
  - **Esempio**: Copia di una matrice 2048 x 2048 contenente interi di 4 byte: $$ \text{Banda Effettiva} = \frac{(2048x2048x4x2)x10^{-9}}{tempo} $$

Il modello di Performance Roofline

Modello Roofline

- Il modello roofline è un metodo grafico utilizzato per rappresentare le prestazioni di un algoritmo in relazione alle capacità di calcolo e memoria di un sistema.
- Utilie 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:

$$ AI = \frac{\text{FLOP}}{\text{Byte Trasferiti}} $$

- FLOPs: Numero di operazioni in virgola mobile o operazioni aritmetiche in generale
- Byte Trasferiti: Numero di byte trasferiti dalla/verso la memoria

Soglia di Intensità Aritmetica

- La soglia diipende dall'hardware specifico ed è definito dal seguente rapporto: $$ \text{Soglia AI} = \frac{\text{Theoretical Computational Peak Performance}}{\text{Bandwidth Peak Performance}}
- Computational Peak Performance: Massima capacità di calcolo della GPU
- Bandwidth Peak Performance: Massima capacità di trasferimento dati della GPU

Intensità Aritmetica: Algoritmi e Hardware

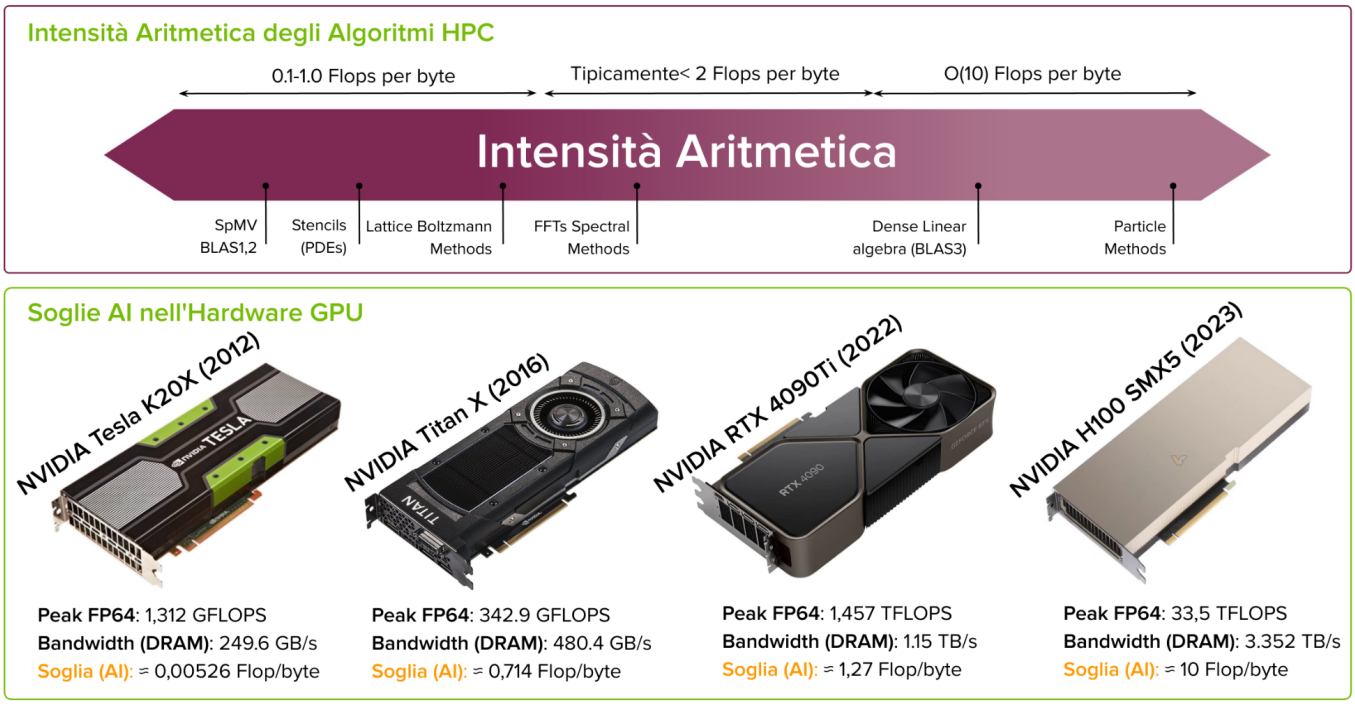
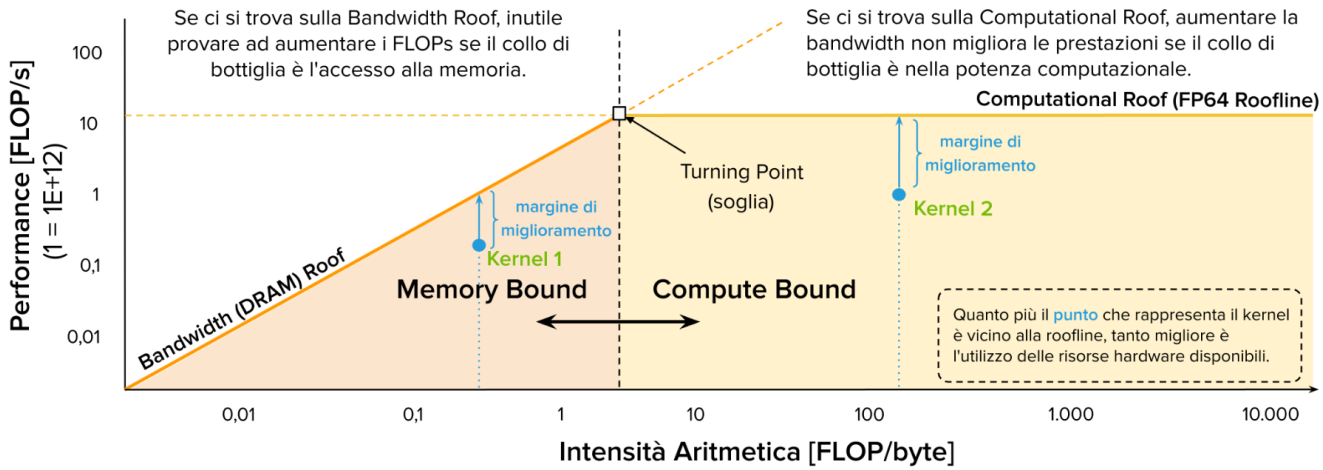


Diagramma Roofline

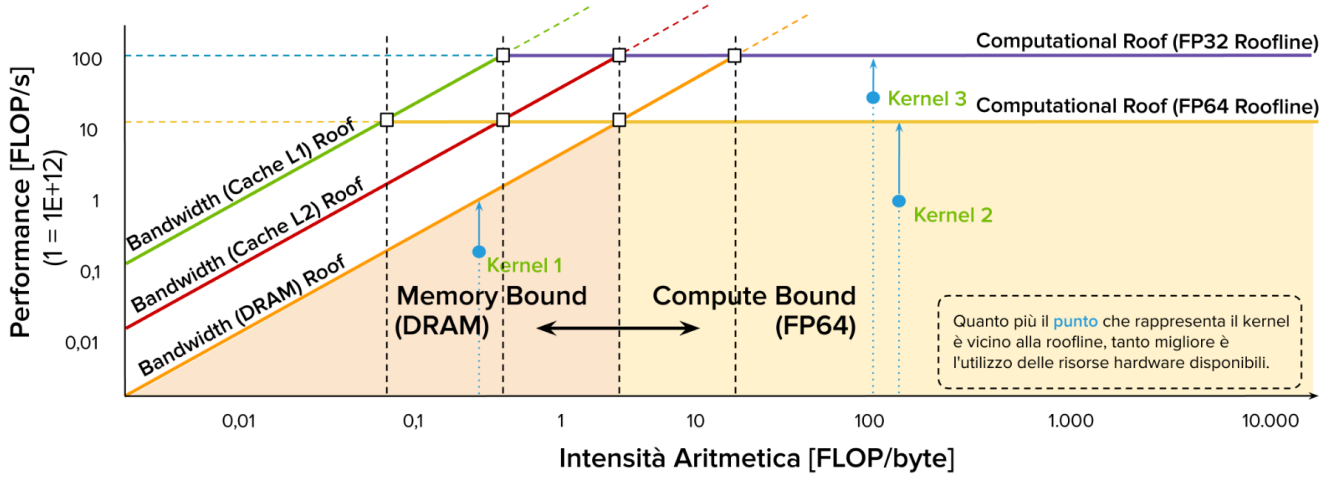
Curve nel Diagramma

- 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. Questa è la massima velocità a cui la GPU può eseguire operazioni in FP64.



Multiple Roofline

- Roofline di Memoria GPU: Diversi limiti 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, FP32, FP64. Migliori prestazioni utilizzando tipi di dati con precisione inferiore.



Gerarchia di Memoria CUDA

Le applicazioni seguono spesso 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, larghezze di banda e capacità:
>   - Livelli Bassi (registri, cache): Bassa latenza, bassa capacità, elevato costo 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 e cache, Dischi e Flash per la memoria lenta e capiente
> - CUDA espone più livelli della gerarchia rispetto ai modelli CPU, offrendo un controllo più esplicito per ottimizzare le prestazioni.

alt text

DDR vs GDDR

	DDR (Double Data Rate)	**GDDR (Graphics DDR)***
Target	CPU	GPU
Utilizzo	sistemi operativi, applicazioni, DB	Gaming, Rendering 3D, AI
Architettura	Ottimizzata per bassa latenza, bus a 64 bit, accessi rapidi nelle operazioni di sistema	Memoria ottimizzata per massimo Throughput con bus ampi (e.g. 384 bit) per garantire banda elevata
Memory clock	Fino a 3600MHz	Fino a 1500 MHz

	DDR (Double Data Rate)	GDDR (Graphics DDR)*
Larghezza di banda	Fino a 100 GB/s	Fino a 1TB/s
Consumo	Basso consumo	Elevato anche in idle
Costo per GB	10€-20€	30€-60€
capacità massima per modulo	Fino a 256GB	fino a 48 GB

GDDR vs HBM

	GDDR (Graphics DDR)	HBM (High Bandwidth Memory)
Esempio GPU	RTX 4090	H100
Utilizzo	Grafica, AI piccola scala, rendering	HPC, Traning AI, Inferenze AI real time
Achitettura	Chip su PCB	Moduli impilati sul die della GPU
Bus Wigth	384 bit	fino a 5120 bit
Banda Passante	Fino a 1TB/s	Fino a 3.5 TB/s
Latenza	più bassa rispetto a HBM	più alta rispetto a GDDR
Capacità massima	fino a 24 GB	Fino a 80-144 GB
Efficienza energetica	consumi maggiori rispetto a HBM	più efficiente, ottimizzato per HPC
Costo	accessibile	elevato

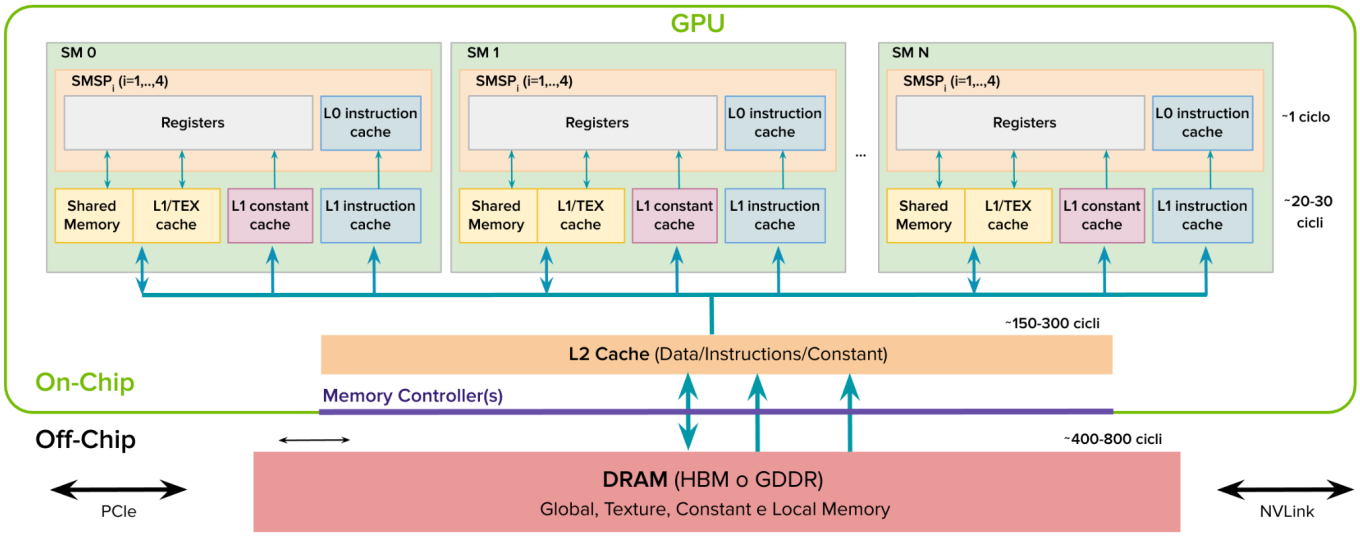
Gerarchia di memoria

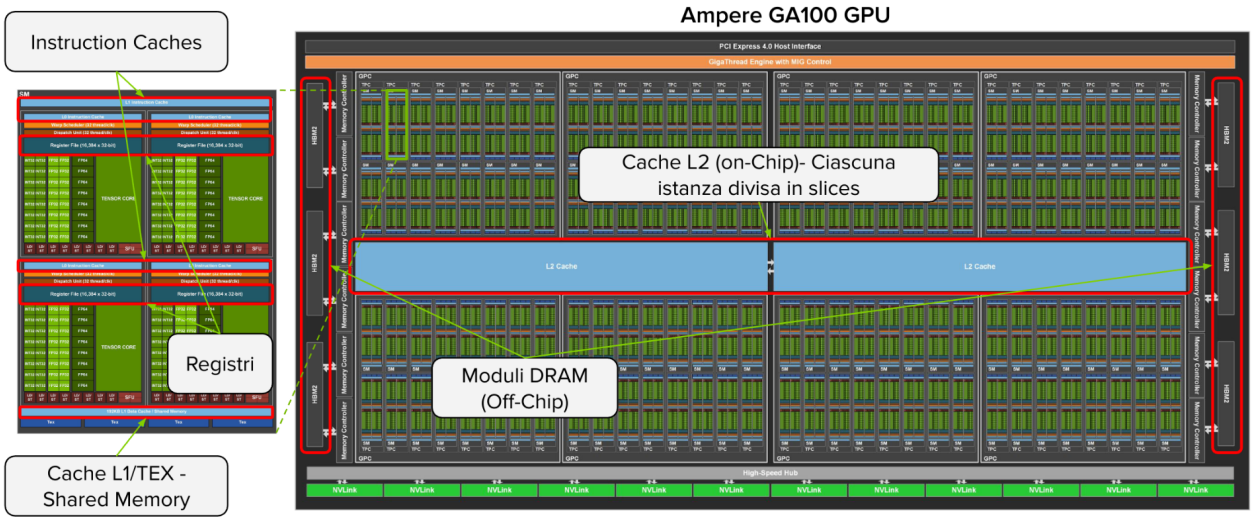
1. registri
    - Mimoria più veloce, privata per ogni thread, variabili temporanee.
2. Shared Memory
    - Memoria condivisa tra i thread di uno stesso blocco, utilizzata per la comunicazione e la cooperazione tra i thread.
3. Caches (L1, L2, Texture, Constant, Instruction)
    - Memoria intermedia automatica che riduce i tempi di accesso ai dati utilizzati frequentemente.
4. Memoria Locale
    - Privata per ogni thread, usata per variabili grandi o register spill (registri non sufficienti).
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 (e.g. immagini)
7. Memoria Globale
    - Memoria più grande e lenta, accessibile da tutti i thread e dalla CPU

Modelli di Memoria CUDA

Due tipi di memoria:

- Programmabile: Controllo esplicito del posizionamento dati, CUDA ne espone diverse tipologie.
- Non Programmabile: Nessun controllo, gestione automatica (e.g. L1, L2)





Registri GPU

- Memoria on chip più veloce sulla GPU (accesso ~1 ciclo)
- 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
- Strettamente privati per thread con durata limitata all'esecuzione del kernel
- Una volta che il kernel ha completato l'esecuzione non è più possible accedere a una variabile di registro

Limiti

- Limite di 63 (Fermi) o 255(kepler e successive) registri per thread
- Allocati dinamicamente tra warp attivi in un SM, influenzando l'occupancy.

- Minor uso di registri permette di avere più blocchi concorrenti per SM
- Register Spilling: Eccedere il limite hardware sposta automaticamente le variabili dei registri alla memoria locale (~100-300 cicli) riducendo le prestazioni

> **Ottimizzazione**
>
> - Euristiche del compilatore: `nvcc` utilizza euristiche per minimizzare l'uso dei registri evitando 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`: Visualizza l'uso dei registri e la memoria locale
>   - `-maxrregcount=n`: Limita il numero di registri per thread
>   - `-abi=no`: Disabilita l'allocazione di registri per le variabili automatiche

Memoria Locale

- Memoria off-chip (DRAM), 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
- Alta latenza e bassa larghezza di banda, stessa della memoria del device
- Per GPU con compute capability 2.0 o superiore, i dati sono posti in cache 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 copmilazioni
- Grandi strutture o array locali che consumerebbero troppo spazio nei registri
- Variabili che eccedono il limite dei registri del kernel
  - Register spill

> **Prestazioni**
>
> - Preferire uso di registri dove possibile
> - Ristrutturare il codice per ridurre variabili locali di grandi dimensioni
> - Utilizzare shared memory per dati frequentemenenti acceduti

Shared Memory (SMEM) e Cache L1

- Ogni SM ha memoria on-chip limitata, condivisa tra shared memory e cache L1
- Partizionata fra i thread block residenti in un SMEM
- 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
- La shared memory è programmabile con controllo esplicito da parte del programmatore, mentre la cache L1 è automatica per ridurre la latenza alla memoria globale.
- Shared memory è condivisa tra thread di un blocco, cache L1 serve tutti i thread di un S;

- 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 comunicazone tra thread di un blocco. Ciclo di cita legato al blocco di thread, rilasciata al completamento del blocco.
- Cache L1: Gestisce automaticamente la coerenza dei dati, senza bisogno di sincronizzazione esplicita.

Memoria Costante

- Spazio di memoria di sola lettura off-chip (DRAM), accessibile a tutti i thread di un kernel.
- Dimensione totale limitata a 64KB per tutte le copmute capability
- Una porzione della constant memory 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 durante l'esecuzione del kernel

Dichiarazione e Inizializzazione

- Dichiarata con l'attributo `__constant__`
- Inizializzata dall'host usando: `cudaError_t cudaMemcpuToSymbol(const void* symbol, const void* src, size_t count);`
- L'operazione di copia è generalmente sincrona

> **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 di un warp leggono da indirizzi diversi, poichè gli accessi vengono serializzati e ogni lettura viene comunque trasmessa a tutti i thread del warp, potenzialmente sprecando lunghezza di banda.

Memoria Globale

- Memoria più grande con latenza più alta e più comunemente usata sulla GPU
- Memoria principale off-chip (DRAM) della GPU, accessibile tramite transazione da 32, 64 o 128 byte.
- Scope e lifetime globale, Accessibile da ogni thread in ogni SM per tutta la durata della applicazione.

Dichiarazione e Allocazione

- Statica: Usando il qualificatore `__device__` nel codice server
- Dinamica: Allocata dall'host con `cudaMalloc()` e rilasciata con `cudaFree()`
    - Puntatori passati ai kernel come parametri
    - Le allocazioni persistono per l'intera applicazione e sono accessibili ai thread di tutti i kernel

Prestazioni

- Fattori chiave per l'efficienza:
    - Coalescenza: Raggruppare accessi di thread adiacenti a indirizzi contigui

    - Allineamento: Indirizzi di memoria allineati a 32, 64, 128 byte

> **Uso**
>
> - Accessibile da tutti i thread di tutti i kernel, ma richiede attenzione per la sincronizzazione
> - Potenziali problemi di coerenza con accessi concorrenti da blocchi diversi
> - l'efficienza dipende dalla copmute capability del device
> - I dispositivi beneficiano di caching delle transizioni sfruttando la località dei dati.

Cache GPU: Struttura e funzionamento

- 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

- L1
    - più veloce e ogni SM ne ha una propria garantendo accesso rapido ai dati
    - Memorizza dati dia dalla memoria locale che globale, inclusi i dati che non trovano spazio nei registri
- L2
    - Unica e condivisa tra SM, ponte tra le L1 e la memoria globale
    - Memorizza dati provenienti dalla memoria locale, inclusi dati derivanti da register spills
- Constant Cache
    - 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
    - Specializzata per dati di texture, cruciale per rendereing e accessi 2D/3D
    - Supporta funzionalità hardware come interpolazione e filtraggio
    - Nelle ultime architettura NVIDIA, unificata con cache L1

> Su alcune GPU, è possibile configurare se i dati vengono cachati solo sia in L1 che L2 o solo in L2, per ottimizzare le prestazioni in base al pattern di accesso ai dati.

Caratteristiche Gerarchia 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+

Qualificatori di Variabili e Tipi CUDA

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

Gestione delle Memoria Host-Device

Somiglianze con C

- Come in C il prorgammatore deve allorare e deallocare manualmente
- In più è necessario getstire esplicitamente il trasferimento dei dati tra host e device, operazione cruciale per il corretto funzionamento delle applicazioni CUDA **Operazione 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 `cudaMalloc()` e `cudaFree()`
- Trasferimento Dati: Movimentazione esplicita dei dati tramita il bus PCIe, utilizzando `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.
- Sincronizzazioni: Garantire la coerenza tra le memorie può essere non banale

> Evoluzione verso memoria unificata NVIDIA ha gradualmente unificato nel tempo gli spazi di memoria di host e device Tuttavia il trasferimento dei dati manueale rimane un requisito

Allocazione della Memoria sul Device

`cudaMalloc` è una funzione CUDA utilizzata per allocare memoria sulla GPU

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

`cudaMemset` è una funzione CUDA utilizzata per inizializzare la memoria allocata con un valore specifico

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

`cudaFree` è una funzione CUDA utilizzata per deallocare la memoria sulla GPU

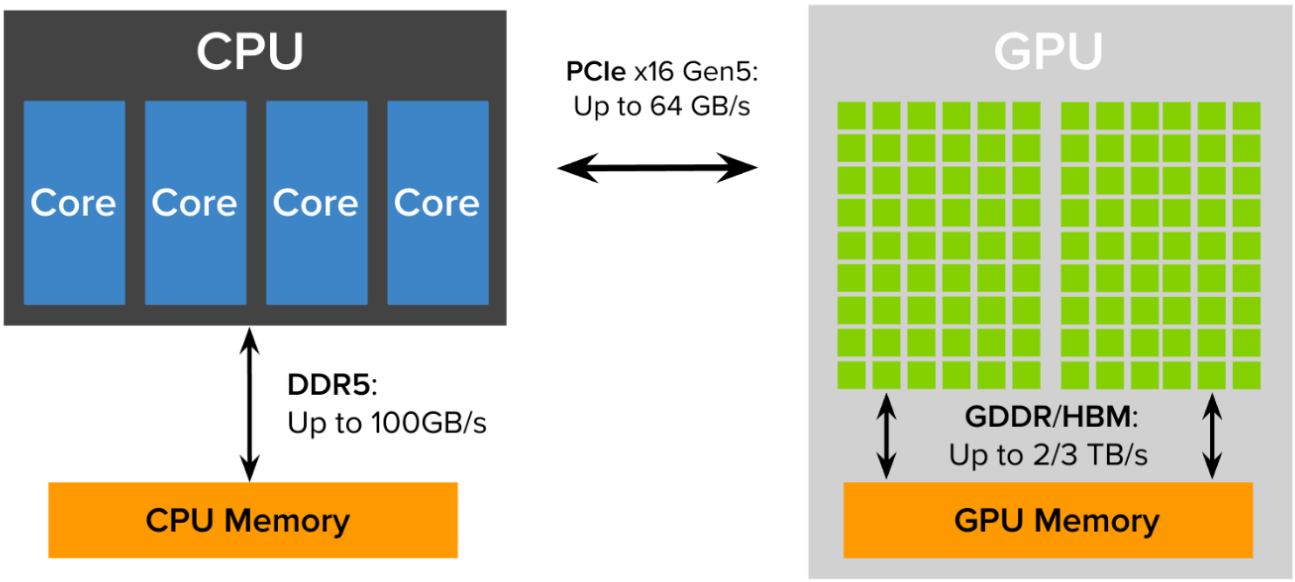
```
cudaError_t cudaFree(void* devPtr);
```

`cudaMemcpy` è una funzione CUDA utilizzata per trasferire dati tra host e device

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

Connettività Host-Device e Throughput di Memoria

- La memoria GDDR della GPU offre una larghezza di banda teorica più alta
- Il collegamento PCIe ha una larghezza di banda teorica massima di 64GB/s
- 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

`cudaMallocHost` alloca memoria host pinned che non può essere spostata dal sistema operativo

- Permette 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

Ma

- La eccessiva allocazione riduce la memoria disponibile peggiorando le prestazioni del sistema in caso di alta pressione sulla RAM

Memoria Zero Copy

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

Accesso alla Memoria Zero Copy

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

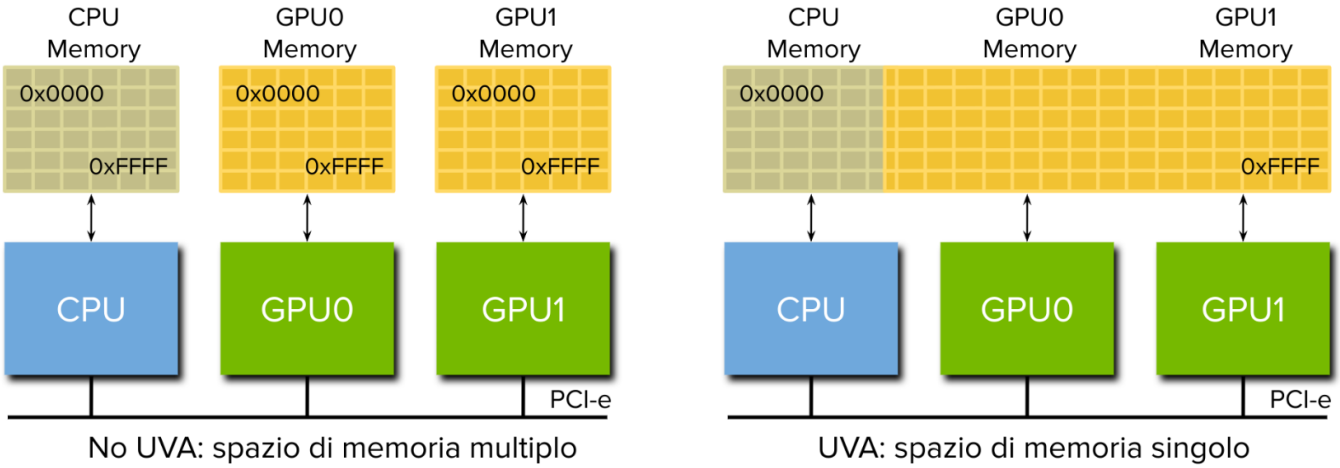
Vantaggi

- Sfruttamento della memoria host: consente di usare la memoria dell'host quando quella del device è insufficientei
- Eliminazione trasferimenti espliciti: Evita la necessità di trasferire esplicitamente i dati tra host e device semplificando il codice e riducendo overhead di gestione della memoria
- Accesso diretto: Utilie 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

> Gli accessi alla memoria devono essere sincronizzato tra host e device per evitare comportamente indefiniti

Unified Virtual Addressing (UVA)

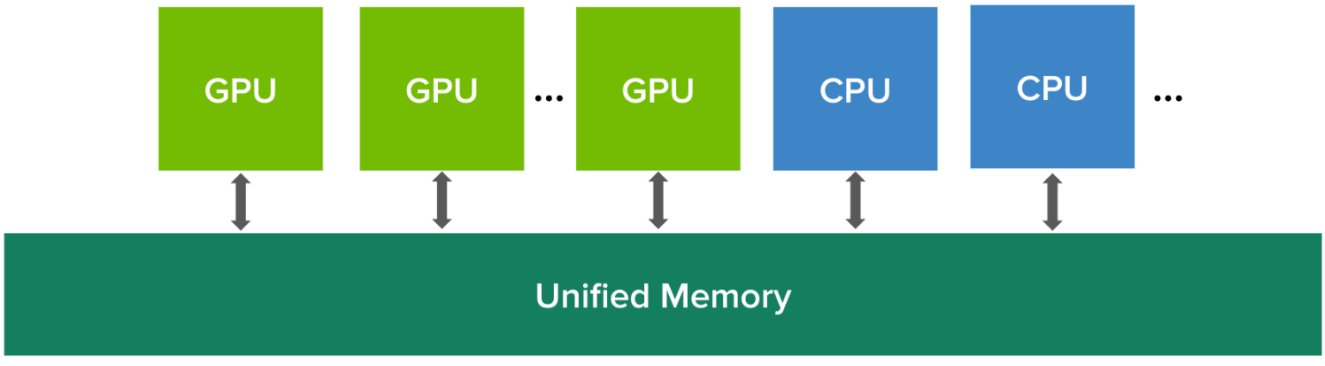
- la Unified Virtual Addressing, è una tecnica che permette alla CPU e alla GPU di condividere lo stesso spazio di indirizzamento virtuale.
- Introdotta in CUDA 4.0 per dispositivi con compute capability 2.0+ e sistemi a 64bit
- Non vi è distinzione tra un puntatore virtuale host e uno device
- Il sistema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



- Il runtime gestisce le mappature per `cudaMalloc` e `cudaHostAlloc` in uno spazio unificato.
- Non è ancora possibile deferenziare un puntatore host sul dispositivo o viceversa

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
- 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
- Quando le GPU o CPU accedono a dati non residenti localmente, un page falt avvia il trasferimento automatico dei dati, gestito in modo trasparente dall'applicazione
- Utilizza la Managed Memory, semplificando notevolmente il codice della applicazione e la gestione della memoria



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

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
- Gestione automatica
- Interoperabilità
- Accesso Unificato
- Supporto Completo per CUDA

Allocazione e Migrazione

- 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 nel sistema
- Questo approccio ottimizza l'utilizzo della memoria evitando allocazioni non necessarie a priori
- Le pagine di memoria possono migrare dinamicamente tra CPU e GPU
- Il driver CUDA utilizza euristiche intelligenti per:
    - Mantenere località dei dati
    - Minimizzare i page fault
    - Ottimizzare prestazioni complessive

> Controllo Programmabile
>
> - Gli sviluppatori possono guidare il comportamento del driver usando:
>   - `cudaMemAdvise`
>   - `cudaMemPrefetchAsync`

Global Memory

Pattern di Accesso alla Memoria

- 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 effetto trascurabili

Modello di esecuzione CUDA e Accesso alla Memoria

- Istruzioni ed operazioni di memoria sono emesse ed eseguite per warp
- Ogni thread fornisce un indirizzo di memoria quando deve leggere/scrivere e la dimensione della richiesta del warp dipende dal tipo di dato
- La richiesta è servita da una o più transazioni di memoria
- Una transazione è una operazione atomica di lettura/scrittura tra 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 della Memoria Globale

- Le transazioni di memoria avvengono in blocchi di dimensioni variabili come 128 byte o 32 byte
- Tutti gli accessi alla memoria globale passano attraverso la cache L2
- Molti accessi passano anche attraverso la cache L1, a senconda del tipo di accesso e dalla architettura GPU.

Accessi Allineati e Coalescenti in CUDA

- Accessi allineati alla memoria
    - L'indirizzo iniziale di una transazione di memoria è un multiplo della dimensione della transazione
    - Gli accessi non allineati possono richiedere più transazioni di memoria
- Accessi coalescenti
    - 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
- 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 delle massime prestazioni.

Scrittura in Memoria Globale

- Prima di Volta, la cache L1 non veniva utilizzata per le operazioni di scrittura
- Da Volta in poi utilizzano la cache L1 in modalità write-through
- Le scritture vengono eseguite a livello di segmenti con granularità 32 byte
- Le transazione di memoria possono coinvolgere uno, due o quattro segmente alla volta.
- Le transiazioni più grandi sono preferite quando possibile
- Mira a raggruppare le scritture in regioni contigue di memoria

Shared Memory

- Canale di comunicazione per tutti i thread appartenenti a un blocco
- Una cache gestita dal programma per i dati dalla memoria globale

- Memoria temporanea per elaborare dati on-chip e migliorare i pattern di accesso alla global memory
- Aumenta la banda disponible e riduce la latenza 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

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**
>
> - 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
> - Un uso eccessivo di SMEM può limitare il parallelismo del dispositivo, riducendo il numero di blocchi di thread attivi concorrenti in un SM
> - 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 di utilizzo della SMEM

1. Caricamento in Shared Memory
2. Sincronizzazione Post-Caricamento
3. Elaborazione Dati
4. Sincronizzazione Post-Elaborazione
5. Scrittura in Memoria Globale

Shared Memory Banks

- Per massimizzare la banda larga di memoria, la shared memory è suddivisa in 32 moduli di memoria di uguale dimensione chiamati memory bank
- Contiene una porzione di dati con ogni banco che può servire una word
- 32 perchè è il numero di thread presenti in un warp, permettendo l'accesso simultaneo alla memoria da parte di tutti i thread
- 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.

- Scenario ideale: Se una operazione di lettura o scrittura emessa da un warp accede ad un solo indirizzo per ogni banco l'operazione è servita da una singola transazione di memoria