GPU Computing

Laurea **Magistrale in Informatica** - AA 2023/24

Docente **G. Grossi**

Lezione 4 – Modello di memoria CUDA: la shared memory

Sommario

- ✓ Il modello di memoria CUDA
- ✓ Tipi di memoria
- ✓ La shared memory: uso e pattern di accesso
- ✓ Allocazione statica e dinamica di shared memory
- ✓ Esercitazione su parallel reduction, prodotto matrici e convoluzione

Le memorie della GPU

Definizione e significato

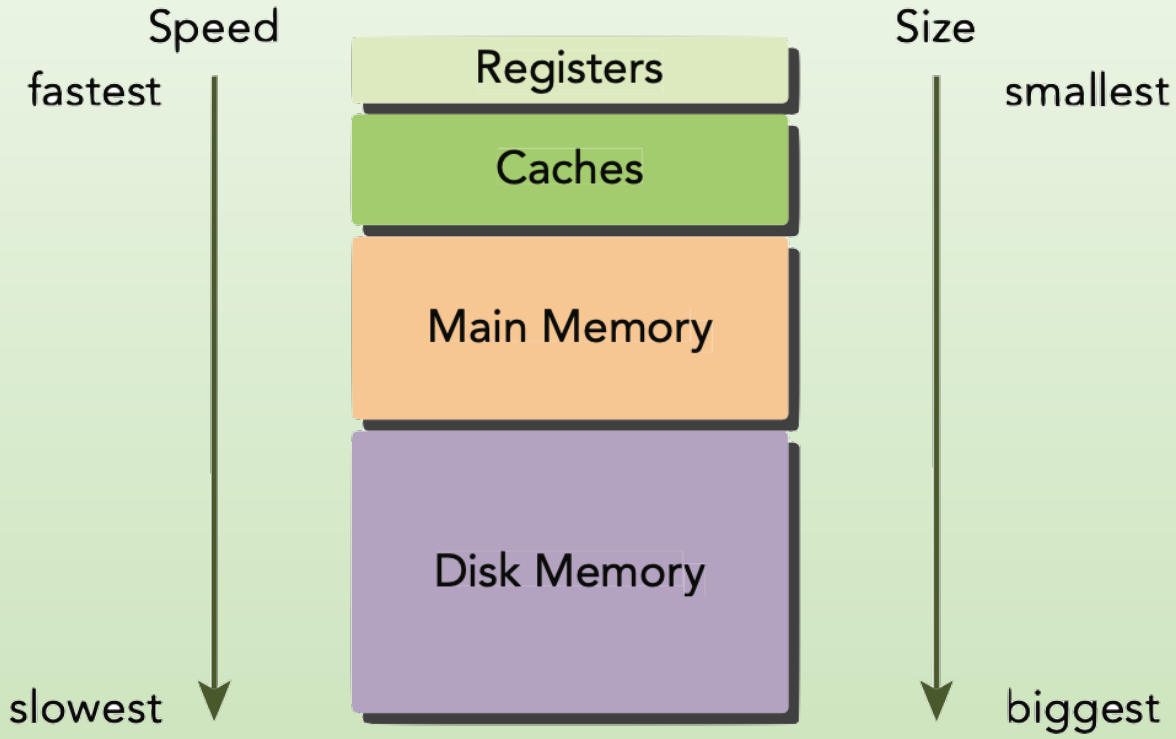
La memoria nei moderni sistemi di calcolo

Elementi chiave nel disegno di **architetture** dei **moderni computer**:

- ✓ Non si possono avere computazioni veloci in assoluto se i dati non possono essere '**spostati**' con sufficiente **velocità**
- ✓ Necessità di avere **grandi quantità** di memoria per applicazioni che coinvolgono big data
- ✓ Le memorie molto **veloci** sono anche molto **costose**
- ✓ Non si può puntare altro che a un **progetto gerarchico**

Carattere della gerarchia

- ✓ Illusione di una **memoria grande** ma a **bassa latenza**
  - **Bassa latenza**, **bassa capacità** (per il processore)
  - **Alta latenza, alta capacità** (per la memoria)



CPU: DRAM (Dynamic Random Access Memory)
GPU: SRAM (Static Random Access Memory)

DRAM vs SRAM

Technology:

- **DRAM** stores data as a charge in a capacitor, requiring periodic refreshing to maintain the data. It's simpler and denser but slower
- **SRAM** uses latches or flip-flops built from transistors to store data, eliminating the need for refreshing. It's larger and more expensive

Performance:

- **DRAM** offers higher density and lower cost per bit, making it ideal for large main memory systems. However, it has slower access times due to refreshing requirements
- **SRAM** provides faster access times and lower latency, making it perfect for applications requiring high speed and performance, like CPU caches, GPU main memory and embedded systems

Power Consumption:

- **DRAM** has lower static power consumption due to simple circuitry. However, refreshing consumes additional power
- **SRAM** has higher static power consumption due to active circuitry maintaining data, which can be a drawback for battery-powered devices

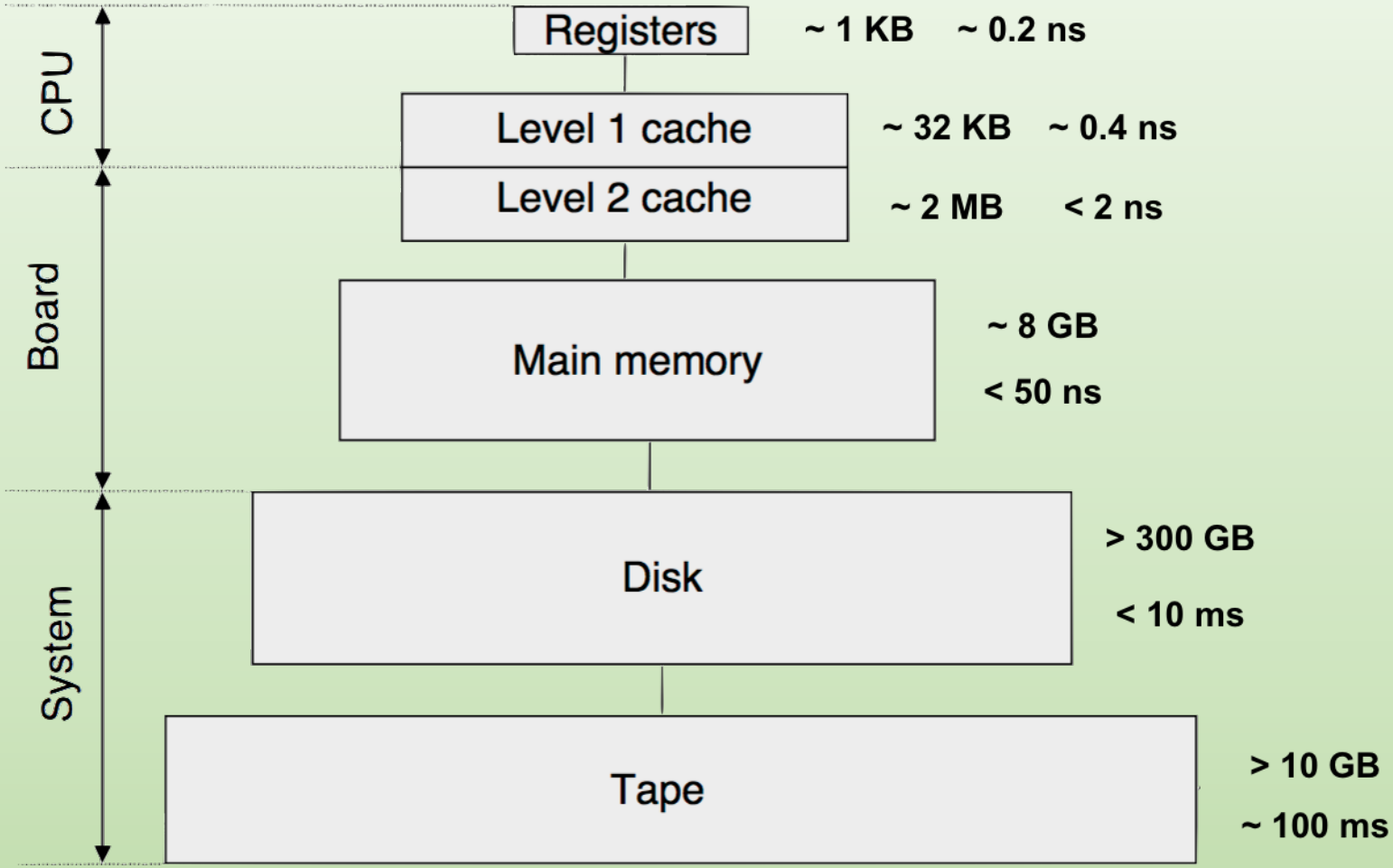
Principio di località e gerarchia di memorie

Località spaziale

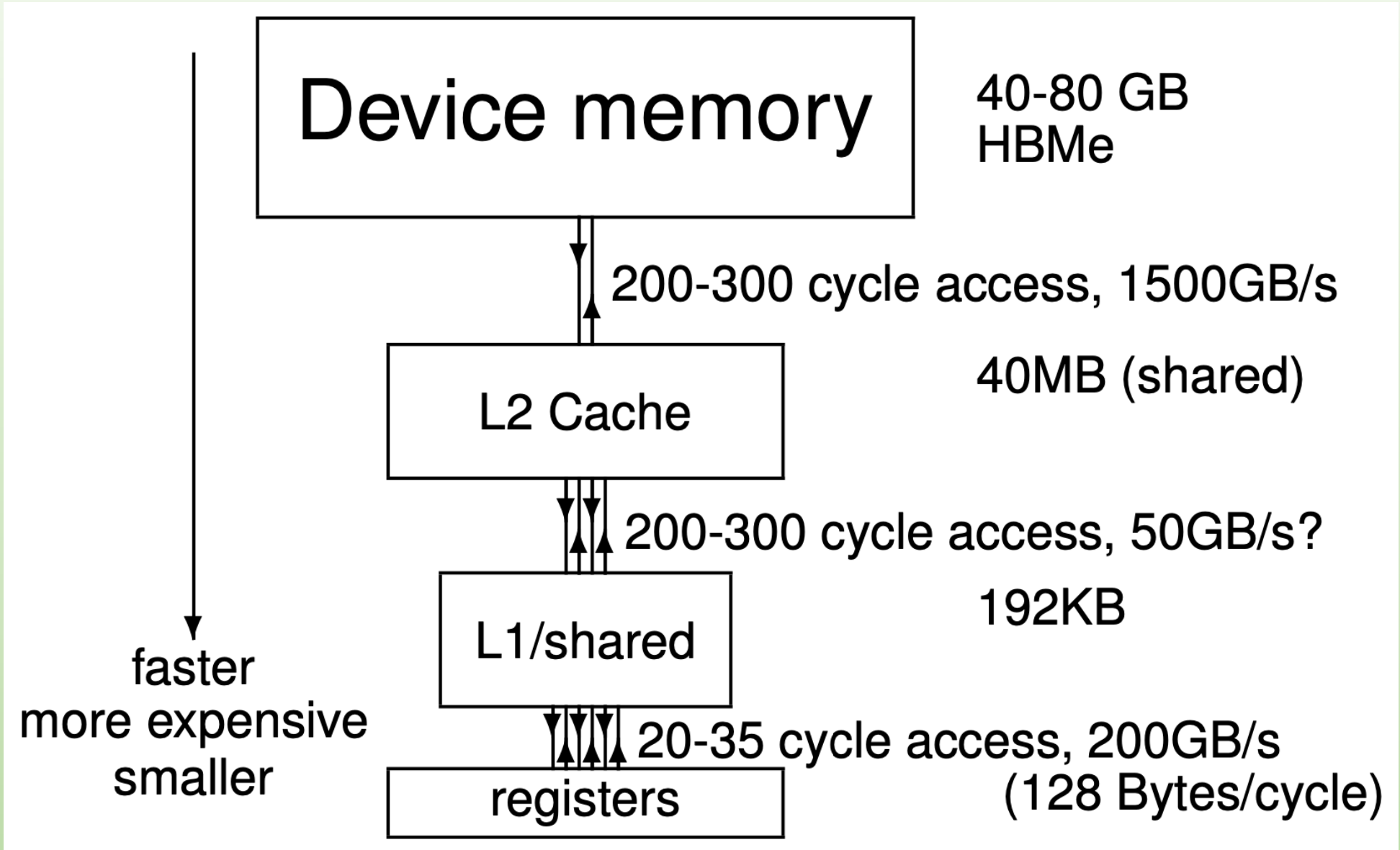
- se l'istruzione con indirizzo $\boldsymbol{i}$ viene eseguita, allora con alta probabilità anche **un'altra l'istruzione** con indirizzo $\boldsymbol{i} + \Delta\boldsymbol{i}$ verrà eseguita (Δi è un intero piccolo)

Località temporale

- se un'istruzione viene eseguita al tempo $\boldsymbol{t}$, allora con alta probabilità la **stessa istruzione** verrà eseguita al tempo $\boldsymbol{t} + \Delta\boldsymbol{t}$ (Δt è un intero piccolo)



Gerarchia di memorie CUDA

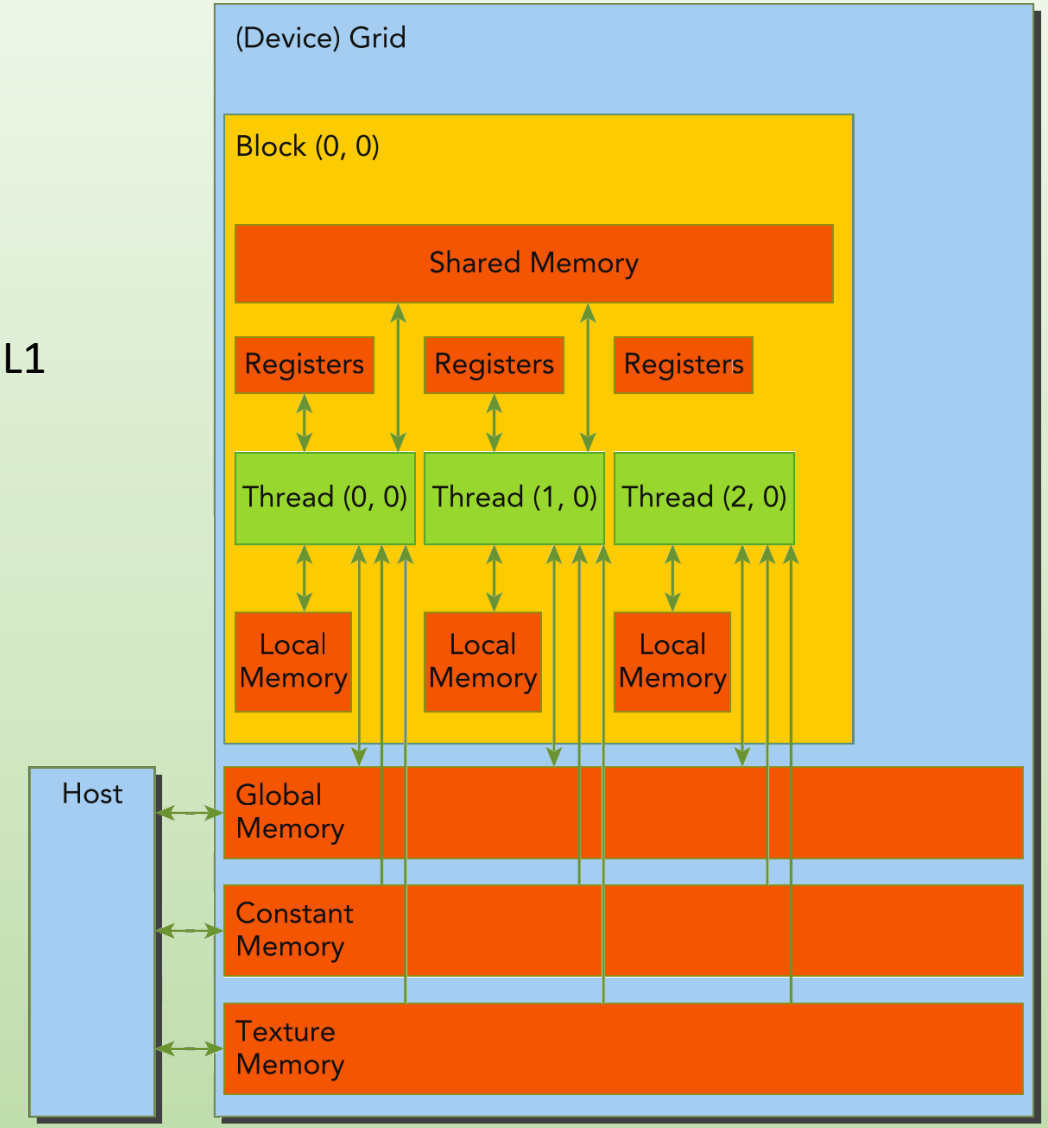


Il modello di memoria CUDA

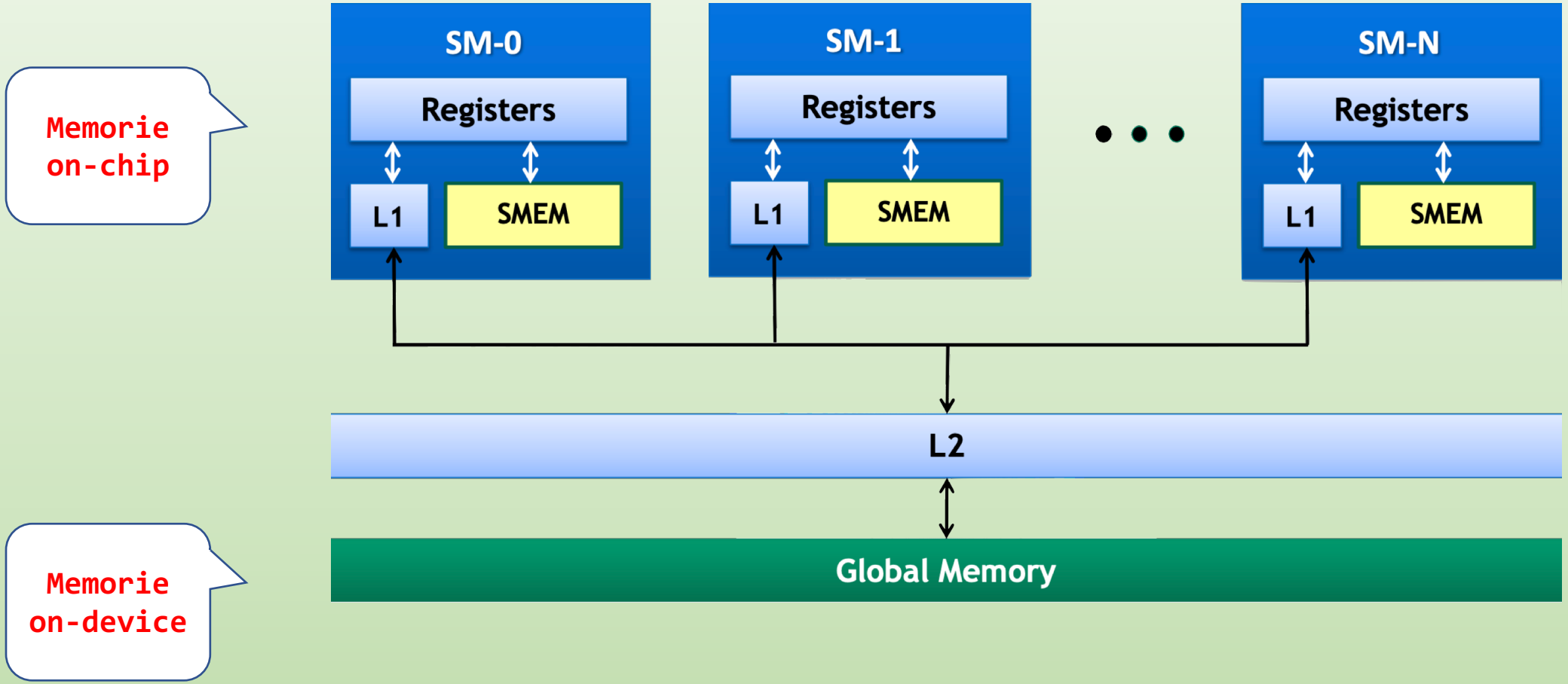
Per il programmatore esistono due tipo di memorie:

- ✓ **Programmabile**: controllo esplicito di lettura e scrittura per dati che transitano in memoria
- ✓ **Non-programmabile**: nessun controllo sull'allocazione dei dati, gestiti con tecniche automatiche (es. le memorie della CPU e le cache L1 e L2 della GPU)

- ✓ **Modello di memoria CUDA**: espone diversi tipi memoria programmabile
  1. **Registri**
  2. **Shared** memory
  3. **Local** memory
  4. **Constant** memory
  5. **Texture** memory
  6. **Global** memory



Suddivisione fisica



Registers

- ✓ Le memorie **più veloci** in assoluto (lifetime: kernel)
- ✓ **Ripartiti** tra i **warp attivi** all'interno del kernel (al più 63 per thread in Fermi e 255 in Kepler+)
- ✓ Una **variabile automatica** dichiarata nel **codice device** e senza che sia specificato nessuno dei qualificatori `__device__` / `__shared__` / `__constant__` generalmente **risiede in un registro**
- ✓ **Meno registri** usa il kernel, **più blocchi** di thread è probabile che risiedano sul multiprocessore (il compilatore usa un'euristica per ottimizzare questo parametro)
- ✓ **Register spilling:** se si usano più registri dei consentiti le variabili si riversano nella **local memory**

Register File: esempio

- ✓ Variabili del kernel **`Vflip`** che il compilatore mappa nei **registri** da **32-bit**
- ✓ 11 vars di tipo **`int` = 11** registri da 32-bit nel RF
- ✓ Blocchi da **128** threads:
  `128 × 11 = 1.3K` `regs`
- ✓ Blocchi da 512 threads:
  `512 × 11 = 5.5K` `regs`

kernel **`Vflip`**

```
. . .
uint i = blockIdx.x;              // block ID
uint j = threadIdx.x;             // thread ID
uint b = blockDim.x;              // block dim
uint x = b * i + j;               // 1D pixel linear index
uint m = (w + b - 1) / b;         // num of blocks in a row
uint r = i / m;                   // row of the source pixel
uint c = x - r * w;               // col of the source pixel
uint s = (w * 3 + 3) & (~3);      // num bytes x row (mult. 4)
uint r1 = h - 1 - r;              // dest. row (mirror)
uint p = s * r + 3*c;             // src byte position of the pixel
uint q = s * r1 + 3*c;            // dst byte position of the pixel
. . .
```

Local Memory

- ✓ Memoria **lenta** come la global memory (collocata off-chip: alta latenza e bassa bandwidth)
- ✓ La local memory è una memoria **locale** ai **thread** (nome non dipende dalla sua locazione)
- ✓ Usata per contenere le variabili automatiche (grandi) non contenute nei registri
  - **Array locali** i cui indici non possono essere determinati a compile-time
  - **Strutture locali grandi** che consumano troppi registri
  - Ogni variabile che non può essere allocata nei registri a causa numero limitato (**register spilling**)
- ✓ La **local memory** risiede nella **device memory**, pertanto gli accessi hanno **stessa latenza** e **ampiezza di banda** della global memory e sono soggetti anche agli stessi vincoli di coalescenza
- ✓ 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**
- ✓ Il compilatore **`nvcc`** si preoccupa della sua allocazione e non è controllata dal programmatore
- ✓ Indagata a livello di compilazione (**`lmem`**) con l'opzione **`--ptxas-options=-v`**

Constant memory

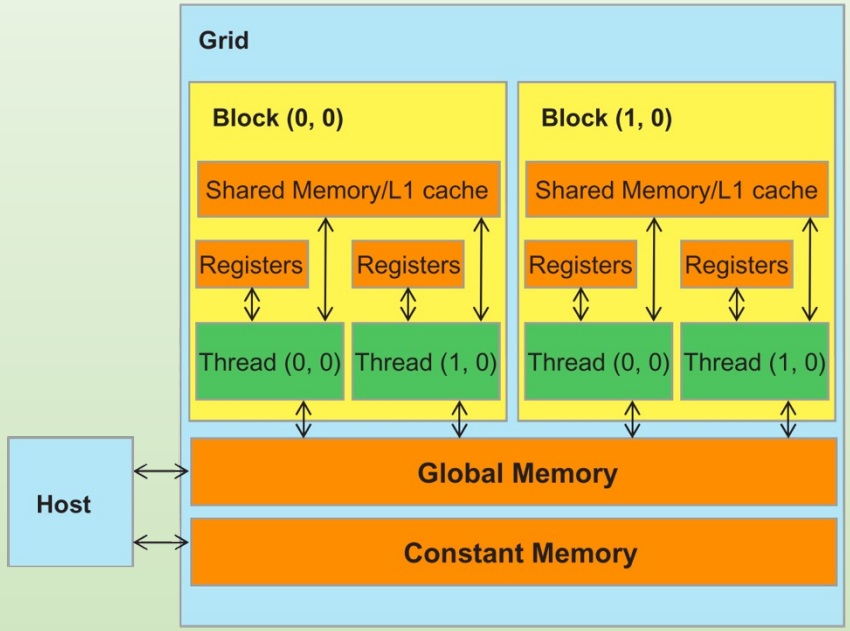
- ✓ Risiede in **device memory** (64K per ogni famiglia) ed ha una cache dedicata in ogni SM
- ✓ Memoria ideale per ospitare dati a cui si accede in **modo uniforme** e in **sola lettura**
- ✓ Una variabile constante è dichiarata con l'attributo: `__constant__`
- ✓ Una variabile costante deve essere dichiara con **scope globale**, al di **fuori** di **qualsiasi kernel**
- ✓ Constant memory è **dichiarata staticamente** ed è **visibile** a tutti i kernel nella stessa unità compil.
- ✓ La constant memory può essere **inizializzata dall'host** usando:

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

- ➢ copia **`count`** byte dalla memoria puntata da **`src`** alla memoria puntata da **`symbol`**, che risiede in memoria globale o costante

Constant memory

- ✓ Il limite di constant memory è pari a **64 KB** per tutte le compute capability (**cache 8 KB**)
- ✓ La constant memory lavora bene quando tutti i thread di un warp leggono dallo **stesso indirizzo** di memoria (caso in cui si raggiunge l'efficienza dei registri)
  - Per **esempio**, un **coefficiente** in una **formula** usato da tutti i thread del warp per calcoli su dati diversi
  - Se ogni thread in un warp legge da **indirizzi differenti**, allora la constant memory non è la miglior scelta perché vengono **serializzati** e ogni singola **read** è inviata a tutti i thread del warp



Esempio di allocazione

```
// CUDA global constants
__constant__ int dA;
__constant__ int dB;
__constant__ int dC;
// or
__constant__ float dABC[3]; // 3 elements of type float (12 bytes)

int main(void){
     float hA = 1;
     float hB = 2;
     float hC = 3;
     ...
     cudaMemcpyToSymbol(dA, &hA, sizeof(int));
     cudaMemcpyToSymbol(dB, &hB, sizeof(int));
     cudaMemcpyToSymbol(dC, &hC, sizeof(int));
     ...
     float hABC[] = {1, 2, 3};
     ...
     cudaMemcpyToSymbol(dABC, &hABC, 3 * sizeof(float));
```

cudaGetSymbolAddress() is used to retrieve the address pointing to the memory allocated for a variable declared in global memory space.
The size of the allocated memory is obtained through **cudaGetSymbolSize()**

Texture memory

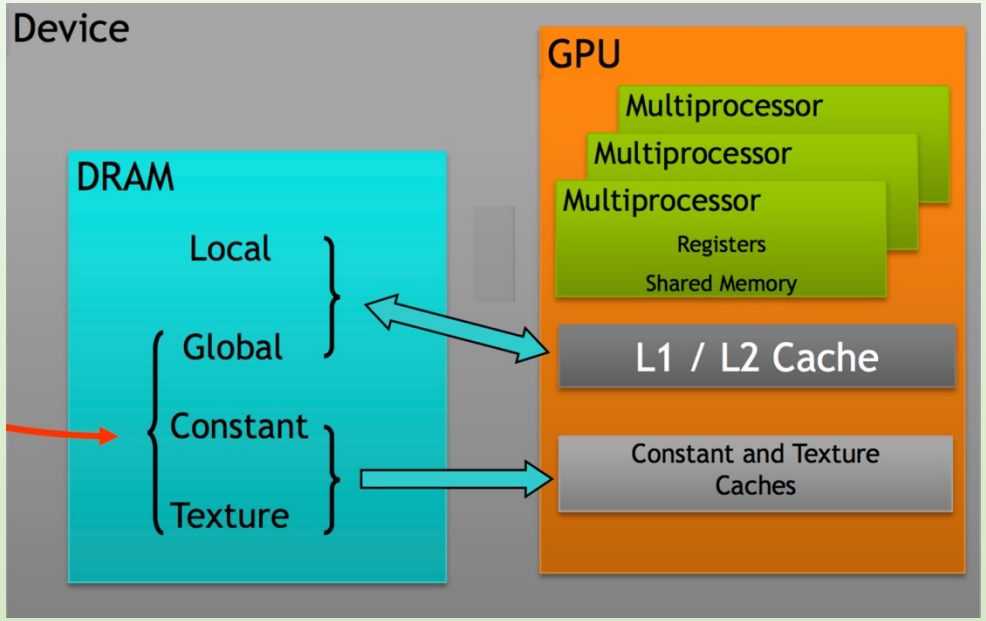
- ✓ La **Texture memory** risiede nella **device memory** e (può avere) una **read-only** cache per-SM ed è acceduta solo attraverso di essa
- ✓ La **cache read-only** include un **supporto hw** efficiente per **filtraggio** o **interpolazione** floating-point nel processo di lettura dei dati
- ✓ La **Texture memory** è ottimizzata per la **località spaziale** 2D, quindi 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
- ✓ Adatta per applicazioni in cui servono classiche **elaborazioni** di **immagini/video**
- ✓ Per altre applicazioni l'uso della texture memory potrebbe essere più **lento** della global memory

Global Memory

- ✓ La global memory è la più **grande**, con più alta **latenza** e più comunemente usata memoria su GPU
- ✓ Nome **global** deriva da **scope** e **lifetime globale** (può essere acceduta da ogni thread in ogni SM)
- ✓ Dichiarazione (codice host):
  - **Statica** `__device__ int a[N];`
  - **dinamica** `cudaMalloc((void **)&d_a, N);` `cudaFree(d_a);`
- ✓ L'**accesso** da parte di **thread** appartenenti a **blocchi distinti** dà potenzialmente modifiche incoerenti
- ✓ La **global memory corrisponde** alla **device memory** ed è accessibile attraverso **transazioni** da **32**, **64**, o **128** byte
- ✓ **L'ottimizzazione** delle transazioni in memoria sono vitali per le prestazioni: quando un warp esegue operazioni di load e store, il numero di transazioni richieste per soddisfare l'operazione dipendono essenzialmente da 2 fattori:
  - **Distribuzione** degli indirizzi attraverso i thread di un warp
  - **Allineamento** degli indirizzi di memoria nelle transazioni

Cache su GPU

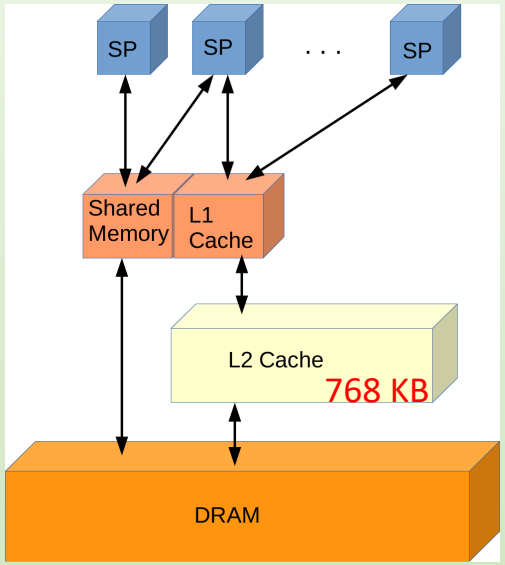
- ✓ Come nel caso di cache su CPU le cache su GPU **non sono programmabili**
- ✓ Ci sono **4 tipi di cache** nelle GPU
  - **L1** (una per ogni SM)
  - **L2** (condivisa tra tutti gli SM)
  - **Read-only constant**
  - **Read-only texture** (**L1** da cc 5.0)



- ✓ **L1 e L2** sono entrambe usate per memorizzare dati in memoria **locale** e **globale**, incluso lo **spilling** dei registri
- ✓ Ogni SM ha anche una **read-only constant** cache e **read-only texture** cache usate per migliorare le prestazioni in lettura dai rispettivi spazi di memoria sul device

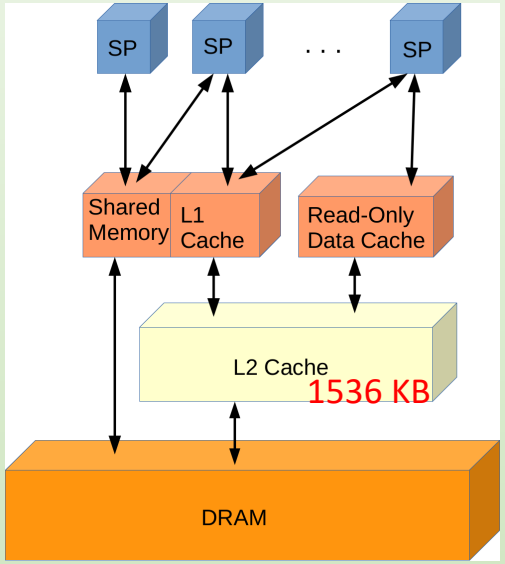
Gestione cache (SMEM, L1, constant)

Fermi



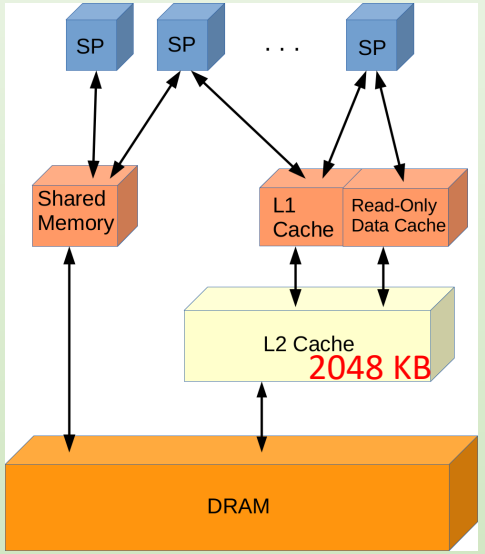
It introduces **64 KB** of configurable **shared memory** and an **L1 cache** per SM, which can be configured as **16 KB** of **L1** cache with **48 KB** of **shared memory**; or **48 KB** of **shared memory** with **16 KB** of **L1 cache**

Kepler



The **64 KB shared memory/L1 cache** is improved by permitting a **32 KB/32 KB split** between the **L1 cache** and **shared memory**

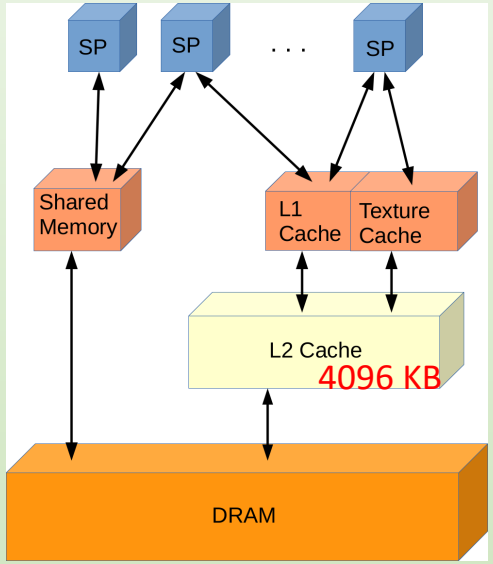
Maxwell



96 KB dedicated shared memory (although each threadblock can only use up to 48 KB), while the **L1 cache** is **shared** with the **texture** caching function

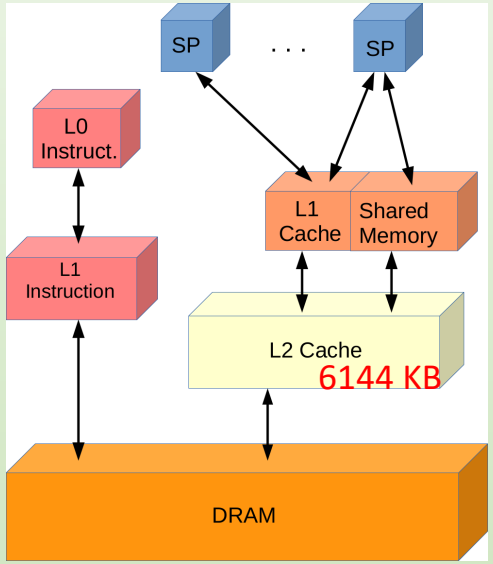
Gestione cache (SMEM, L1, constant)

Pascal



It presents **64 KB** of shared **memory** per SM, and an **L1 cache** that can also serve as **texture cache**

Volta-Turing



new **L0 instruction cache** to provide higher efficiency than previous instructions buffers and a warp scheduler with a dispatch unit

A merged **128 KB L1 Data Cache/shared memory** is introduced, providing **96 KB** of **shared memory**

Salient Features of Device Memory

Cached in L1 and L2 by default except on devices of compute capability 5.x (only L2)

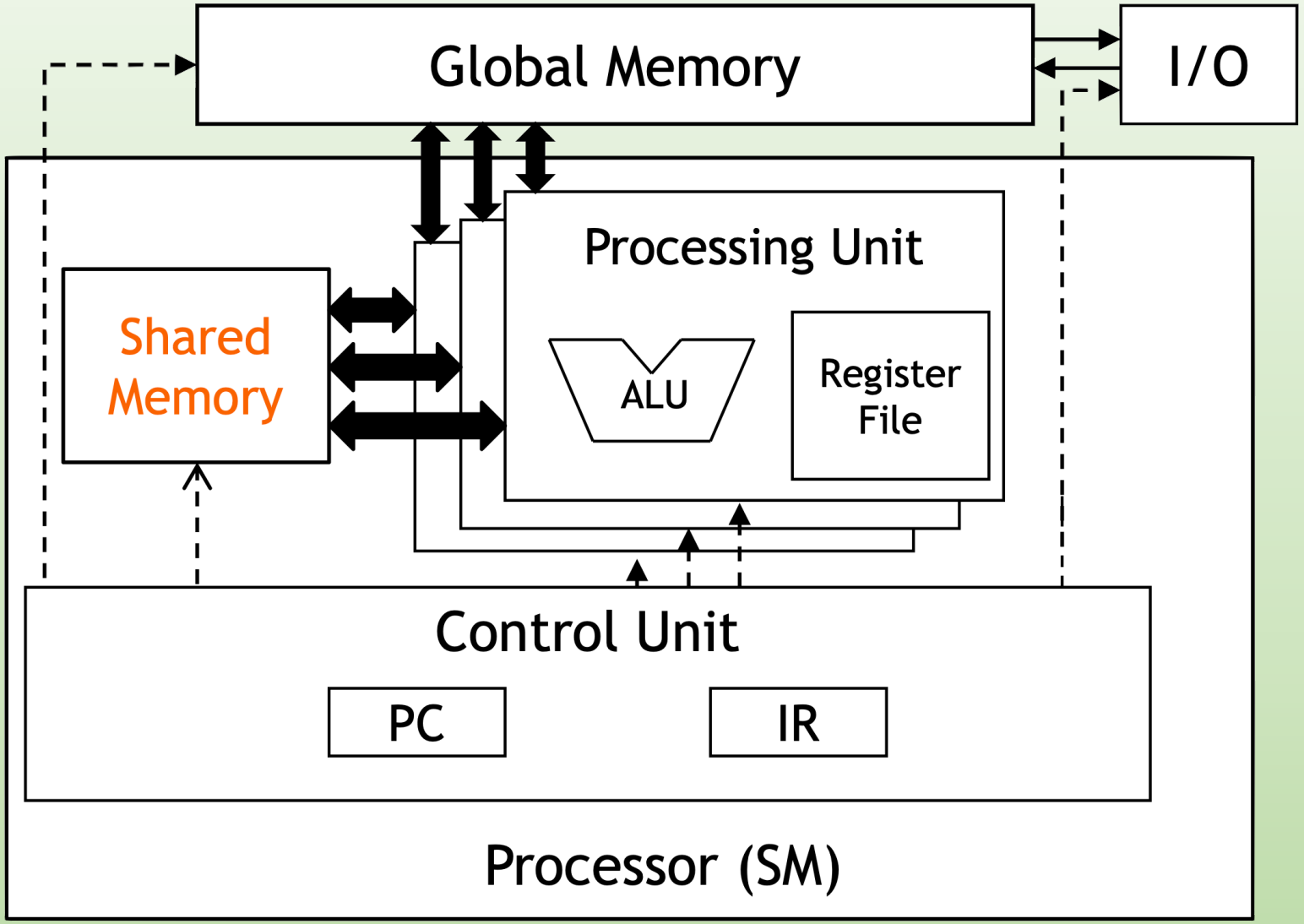
Cached in L1 and L2 by default on devices of compute capability 6.0+

Memory	Location on/off chip	Cached	Access	Scope	Lifetime
Register	On	n/a	R/W	1 thread	Thread
Local	Off	Yes††	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	†	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation

Shared Memory

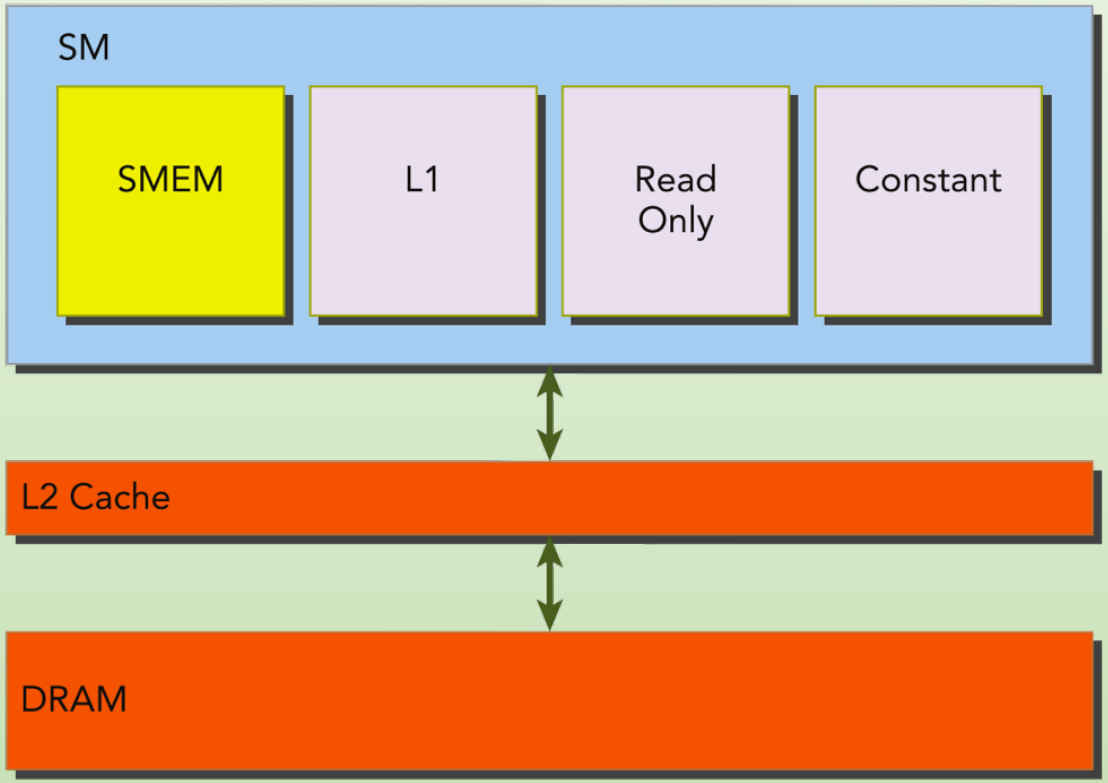
Definizione e uso

Vista hardware delle memorie CUDA



La shared memory (SMEM)

- ✓ Canale di **comunicazione** per thread **intra-block**
- ✓ **Cache** per la **global memory** gestita direttamente dal programmatore
- ✓ Memoria **scratch pad** per elaborare dati on-chip e migliorare i pattern di accesso alla global memory
- ✓ **Latenza** ridotta grosso modo **~30** volte rispetto global memory
- ✓ **Bandwidth** maggiore **~10** volte maggiore rispetto global memory



SMEM

- ✓ Memoria programmabile **on-chip**, quindi con **bandwidth** molto più **alta** e minor latenza della local o global memory
- ✓ E' suddivisa in **moduli** della stessa ampiezza, chiamati **bank** (32)
- ✓ Ogni richiesta di accesso fatta di **N indirizzi** che riguardino **N distinti bank** sono serviti simultaneamente
- ✓ Ogni SM ha una **quantità limitata** di shared memory che viene **ripartita** tra i **blocchi** di thread
- ✓ Prestare attenzione a non **sovra-utilizzare** per non limitare il numero di blocchi/warp attivi
- ✓ Con i **blocchi** di thread condivide anche **lifetime** e **scope**
- ✓ La shared memory serve come base per la **comunicazione inter-thread**: i thread in un block possono cooperare scambiandosi dati memorizzati in shared memory
- ✓ L'accesso deve essere sincronizzato per mezzo della seguente CUDA runtime call:

```
void __syncthreads()
```

Organizzazione

- ✓ La SMEM è suddivisa in blocchi da **4 byte** (chiamate **word**) o da **32 bit**
- ✓ Ogni word può contenere **1 int, 1 float**, **2 shoth**, **4 char**, **2 half float**, **half double** (2 x 32 richiesti), …
- ✓ Se chiedo un **singolo byte** viene comunque letta la **word per intero** cui il byte appartiene
- ✓ La **bandwidth** è di 32 valori ogni 2 cicli di clock
- ✓ Dati 32 bank, ogni **word** è **memorizzata** in **bank distinti** a gruppi di 32 secondo lo schema sotto

L'indirizzo del byte è diviso per 4 e si ha l'indice della word… l'indice del bank è quello della word modulo 32

Byte address	0	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60	
4-byte word index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	.....

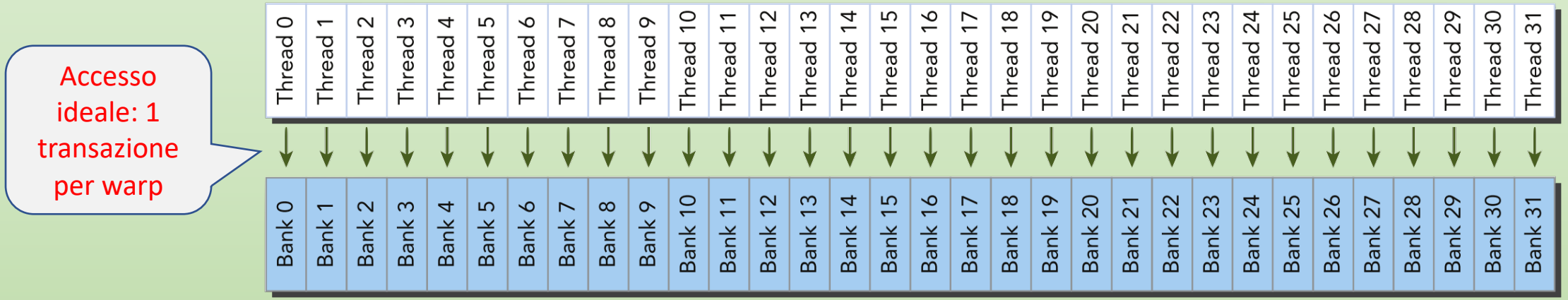
Bank index	Bank 0	Bank 1	Bank 2	Bank 3	Bank 4	Bank 5	Bank 6	Bank 7	Bank 8	Bank 9	Bank 10	Bank 11	.....	Bank 28	Bank 29	Bank 30	Bank 31
4-byte word index	0	1	2	3	4	5	6	7	8	9	10	11	.....	28	29	30	31
	32	33	34	35	36	37	38	39	40	41	42	43	.....	60	61	62	63
	64	65	66	67	68	69	70	71	72	73	74	75	.....	92	93	94	95
	96	97	98	99	100	101	102	103	104	105	106	107	.....	124	125	126	127

SMEM a runtime

- ✓ Viene **ripartita** tra tutti i blocchi residenti in un SM (risorsa critica che limita il parallelismo a livello device)
- ✓ **Maggiore** è la **shared** memory richiesta da un kernel, **minore** è il numero di **blocchi attivi** concorrenti
- ✓ Il contenuto della shared memory ha lo stesso **lifetime** del **blocco** cui è stata assegnata
- ✓ **L'accesso** è per **warp**:
  - caso migliore **1 transazione x 32 thread**
  - caso peggiore **32 transazioni x 32 thread**
- ✓ Nel caso migliore serve **32 thread** del warp in **20 cicli di clock**!

Pattern di accesso

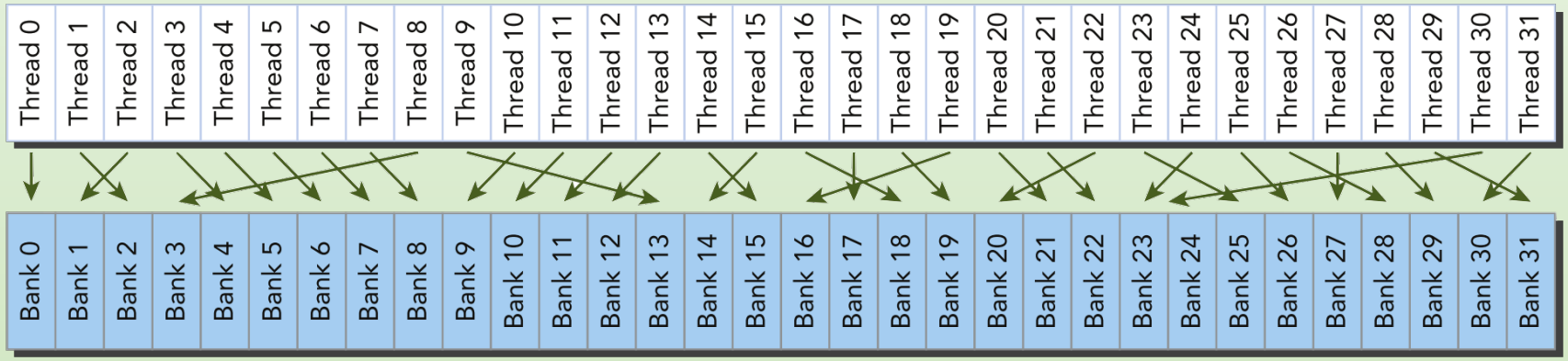
- ✓ Se un'operazione di **load** o **store** eseguita da un warp richiede **al più un accesso per bank**, si può effettuare in una sola transizione il trasferimento dati dalla shared memory al warp
- ✓ In alternativa sono richieste diverse (**>= 2 e <= 32**) transazioni con effetti negativi sulla bandwidth globale



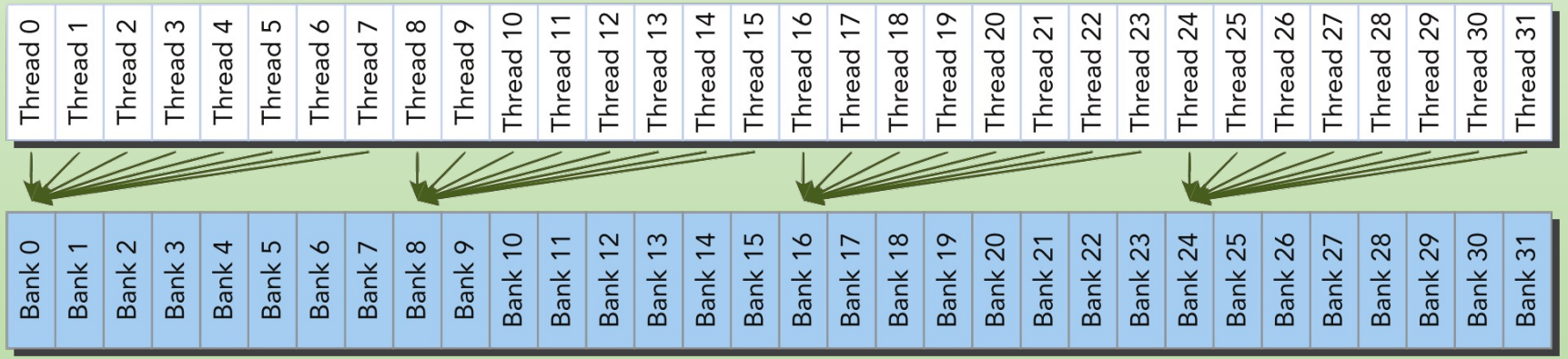
Conflitti

- ✓ **Bank conflict** = diversi indirizzi di shared memory insistono sullo stesso bank
- ✓ L'hardware effettua **tante transazioni** quante ne sono necessarie per **eliminare i conflitti**, diminuendo la bandwidth effettiva di un fattore pari al numero di transazioni separate necessarie

Accesso ancora vantaggioso: non è richiesto l'allineamento degli indirizzi (1 transazione)



Accesso seriale: diversi indirizzi nello stesso bank (x 32 indirizzi nello stesso bank si ha broadcast automatico)

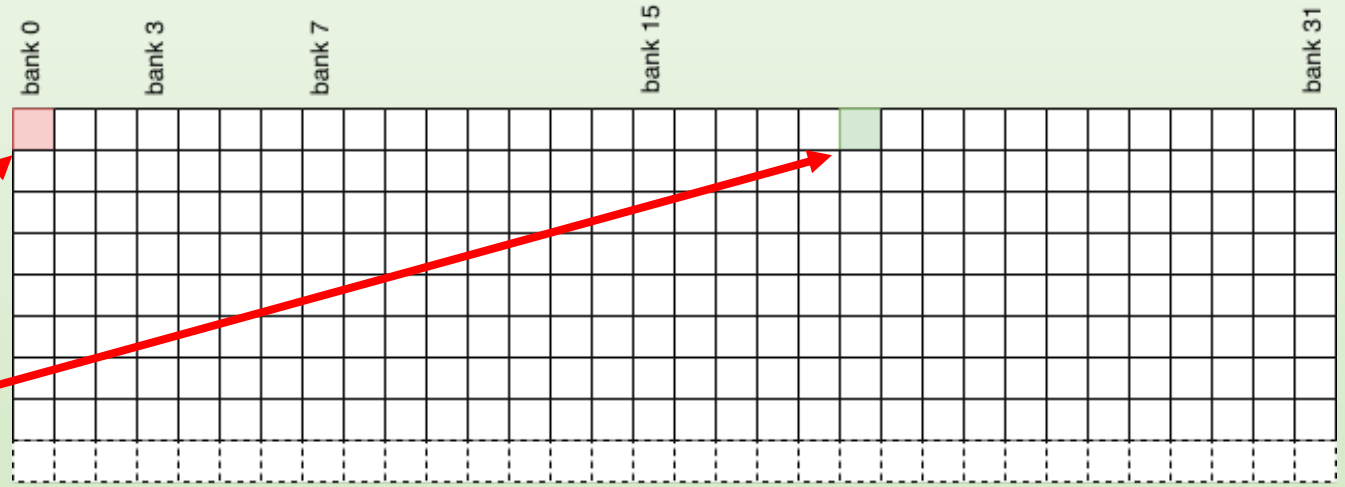


Pattern di accesso ...

- ✓ **Caso broadcast**: un singolo valore letto da tutti thread in un warp da un singolo bank

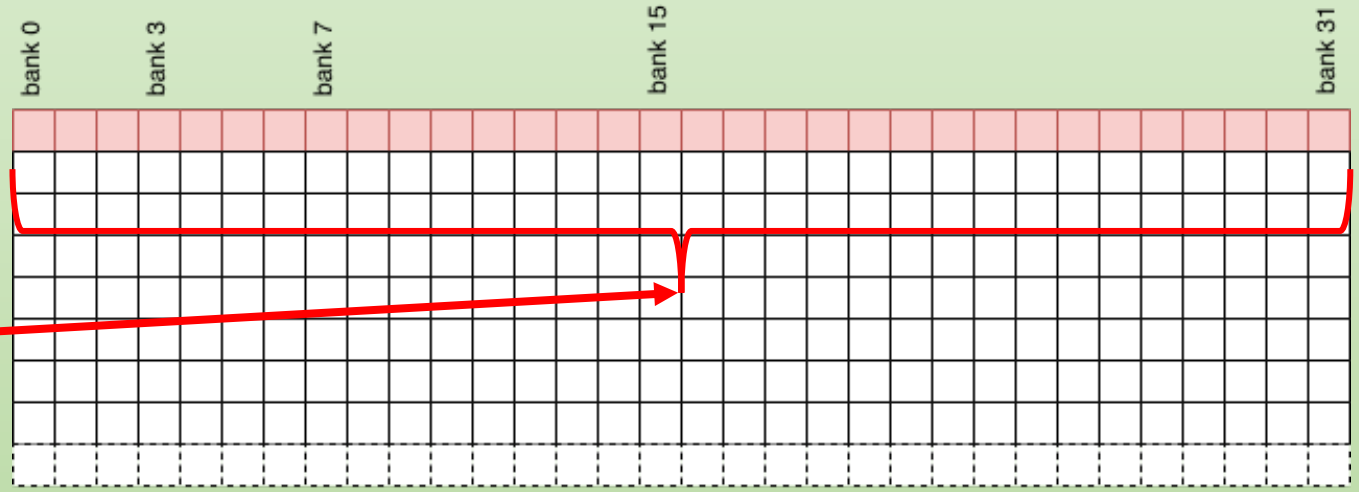
```
float f = smem[threadIdx.x*0];
```

```
float f = smem[20];
```



- ✓ **Caso parallelo**: un singolo valore letto da un singolo bank

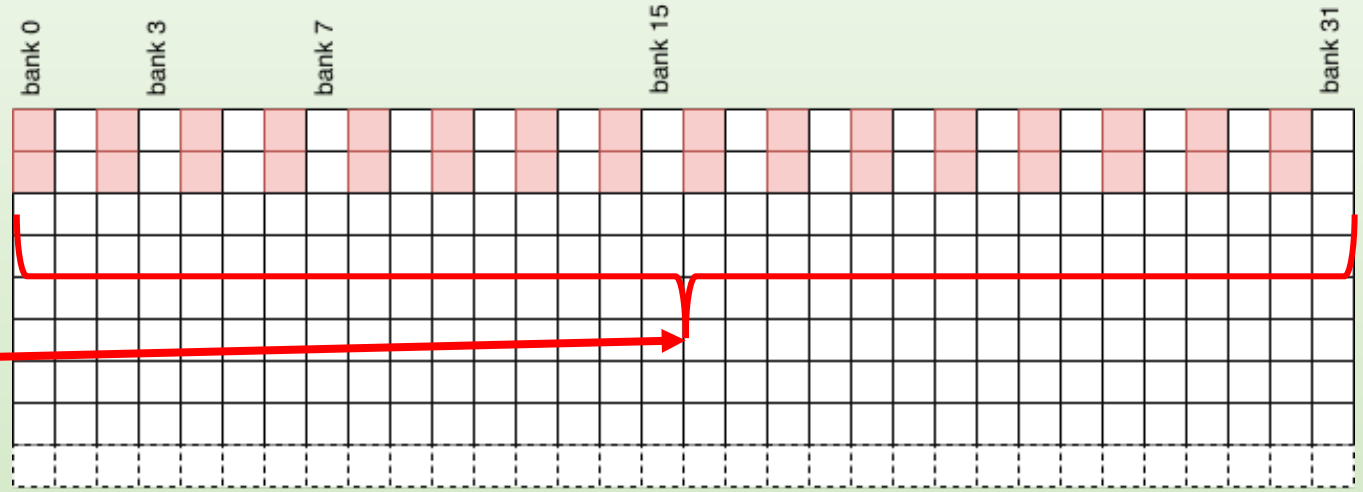
```
float f = smem[threadIdx.x];
```



… pattern di accesso

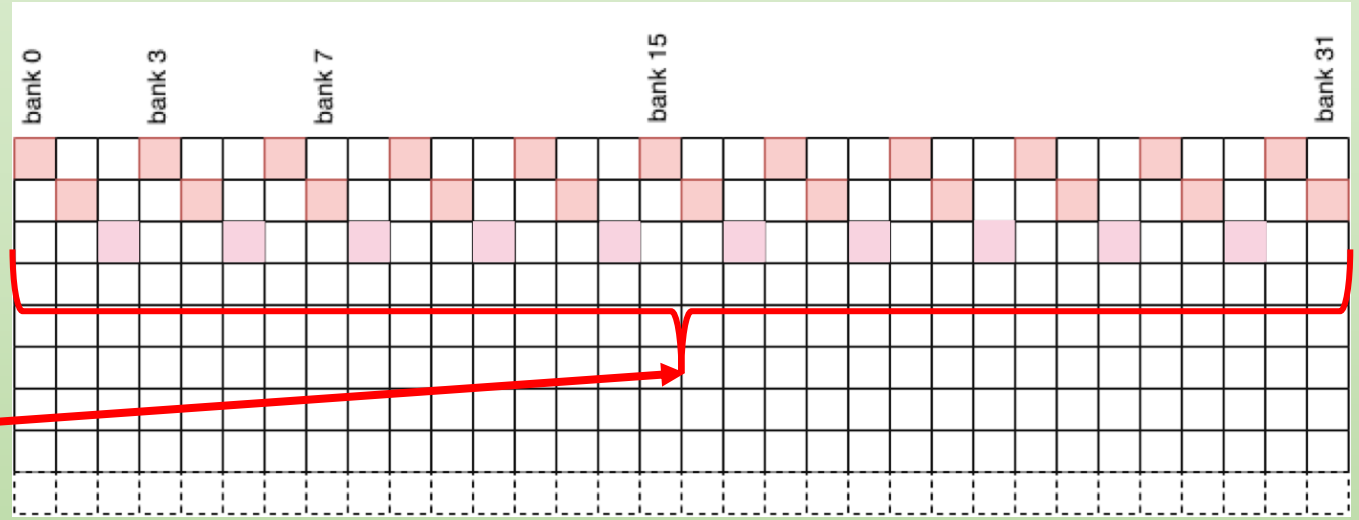
- ✓ **Conflitto doppio**: tutti i thread in un warp richiedono una word di indice doppio il **`threadId.x`** (strutture di 8 byte, per es i double o coppie di float)

```
float f = array_f[threadIdx.x*2];
```



- ✓ **Nessun conflitto**: strutture di 3 word come, ad esempio, le terne di punti nello spazio (X,Y,Z)

```
float f = array_f[threadIdx.x*3];
```



Osservazioni

- ✓ **Latency hiding**: il ritardo che intercorre tra la richiesta avanzata dai thread alla SMEM e l'ottenimento effettivo dei dati non è in generale un problema anche in caso di bank conflict, lo scheduler passa a un altro warp in attesa che quelli sospesi completino il trasferimento dei dati dalla SMEM (in questo modo il ritardo potrebbe essere irrilevante)
- ✓ **Inter-block**: non esiste conflitto tra thread appartenenti a blocchi differenti, il problema sussiste solo a livello di warp dello stesso blocco
- ✓ **Efficienza massima:** Il modo più semplice per avere prestazioni elevate è quello di fare in modo che un warp acceda a word consecutive in memoria shared
- ✓ **Caching**: con lo scheduling efficace le prestazioni anche in presenza di conflitti a livello di SMEM sono di gran lunga migliori se paragonate a quelle in cui cammino che i dati percorrono passa attraverso la cache L2 o peggio, arriva in global memory

Allocazione statica della SMEM

- ✓ Una variabile in **shared memory** può anche essere dichiarata sia **locale** a un kernel sia **globale** in un file sorgente (globale a tutti i kernel dell'applicazione)
- ✓ Una variabile in shared memory è dichiarata con il qualificatore `__shared__`
- ✓ Una variabile in shared memory può essere dichiarata sia **staticamente** sia **dinamicamente**
- ✓ CUDA supporta le dichiarazioni statiche di array **multidimensionali 1D**, **2D** e **3D**
  - dimensione della memoria sia nota a **tempo** di **compilazione**

Esempi allocazione SMEM

Esempi di allocazione statica e uso della SMEM

```
__global__ void mioKernel() {
   __shared__ int i;
   __shared__ float array_f[128];
   __shared__ int array_i[10][10];
    . . .
   array_i[threadIdx.x][threadIdx.y] = 1;
   array_f[threadIdx.x] = 1.0;
   . . .
```

Race condition: due o più thread dello stesso blocco modificano una variabile shared… risultato!?

```
__global__ void mioKernel() {
   __shared__ int i;
   i = threadIdx.x + 1;
   . . .
}
```

Allocazione dinamica della SMEM

- ✓ Se la **dimensione** è **non nota** a tempo di compilazione è possibile dichiarare una variabile **adimensionale** con la keyword `extern`
  - Dinamicamente si possono allocare solo **array 1D**! (Per array multipli vedi prossima slide)
  - può essere sia **interna** al kernel (kernel scope) sia **esterna** (file scope)
  - Es. di array 1D:

```
extern __shared__ int array[];
```

- ✓ Per allocare la **shared memory dinamicamente** (in bytes!) la variabile occorre indicare un terzo argomento all'interno della chiamata del kernel:

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

Allocazione dinamica multipla

Per allocare diversi array nella stessa area di memoria occorre usare esplicitamente gli **offset** all'interno di una singola allocazione globale per tutti i dati

```
short array_s[128];
float array_f[64];
int array_i[256];
```

```
int main(void) {
    int nBytes =
128*sizeof(short)+64*sizeof(float)+256*sizeof(int);
    kernel<<<1, 1, nBytes>>>();  // param. dimensione shared memory
    cudaDeviceSynchronize();
    return 0;
}
```

```
extern __shared__ float array[];  // dynamic shared memory
__global__ void kernel() {
    short* array0 = (short*)array;          // offset inizio short
    float* array1 = (float*)&array0[128];  // offset inizio float
    int* array2 = (int*)&array1[64];        // offset inizio int
    for (int i = 0; i< 128; i++)
        array0[i] = 'a';
    for (int i = 0; i< 64; i++)
        array1[i] = 1.0;
    for (int i = 0; i< 256; i++)
        array2[i] = 1;
}
```

NOTA: stesso modo di organizzare dati di diversi array, come potrebbe essere fatto peraltro in host memory in un buffer sufficientemente capiente

Impiego della shared memory

Uso tipico:

1. Carica i dati dalla device memory alla shared memory
2. Sincronizza i thread del blocco al termine della copia che ognuno effettua sulla shared memory, (così che ogni thread possa elaborare dati certi nel prosieguo)
3. Elabora i dati in shared memory
4. Sincronizza (se necessario) per essere certi che la shared memory contenga i risultati aggiornati
5. Scrivi i risultati dalla device memory alla host memory

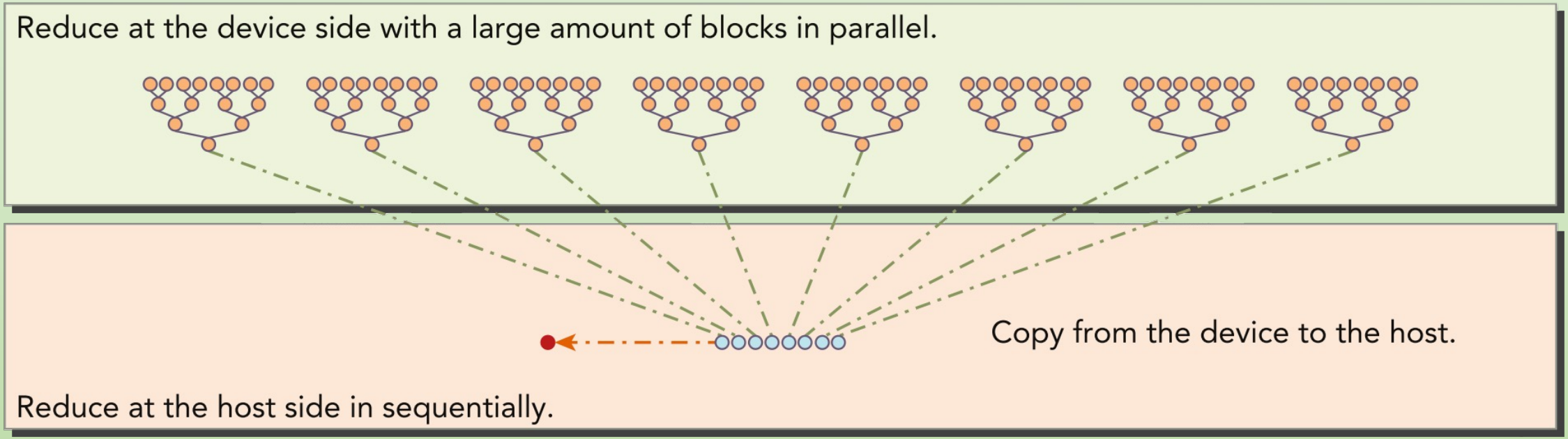
Parallel reduction con SMEM

Accessi coerenti in cache

Somma parziali su blocchi + sincronizzazione

Schema dicotomico:

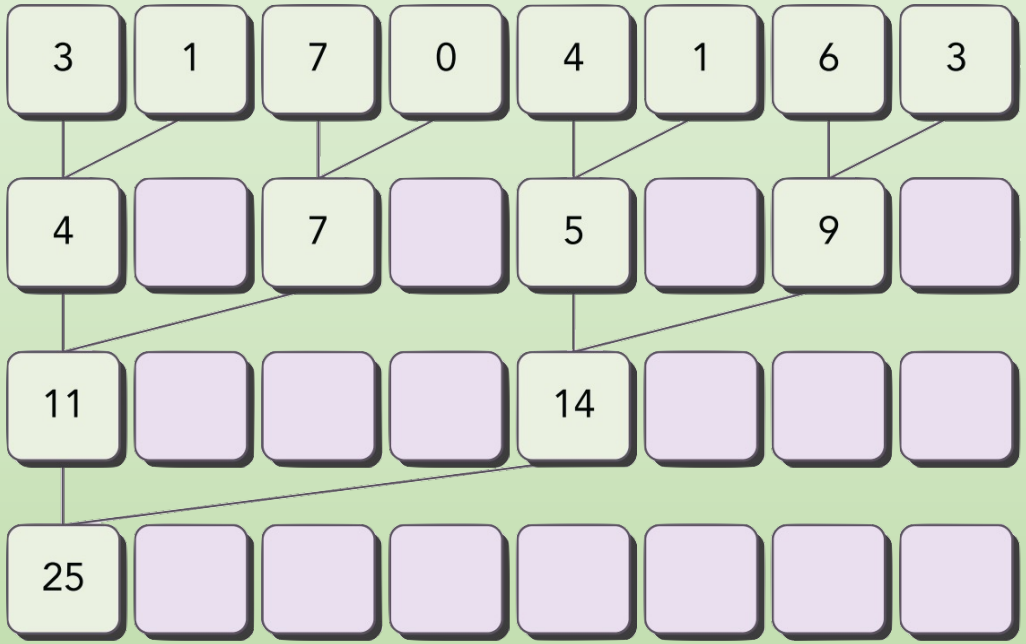
- ✓ **Soluzione locale**: somma parziale sincrona sui blocchi della grid ottenuta per riduzione parallela
- ✓ **Vincolo**: la dimensione di blocco deve essere necessariamente una **potenza di 2**
- ✓ **Svantaggio**: è una soluzione che introduce divergenza crescente a livello di warp (inutilizzo dei thread)



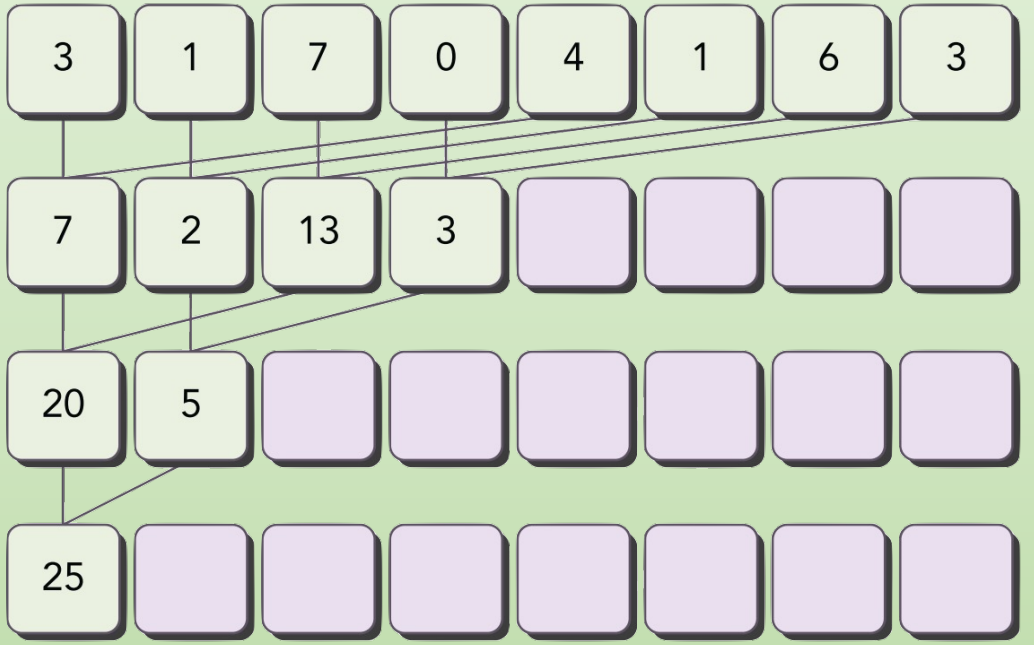
Schemi di parallel reduction

- ✓ Due dei possibili approcci alla risoluzione del problema: quale più efficiente?

Approccio interlacciato

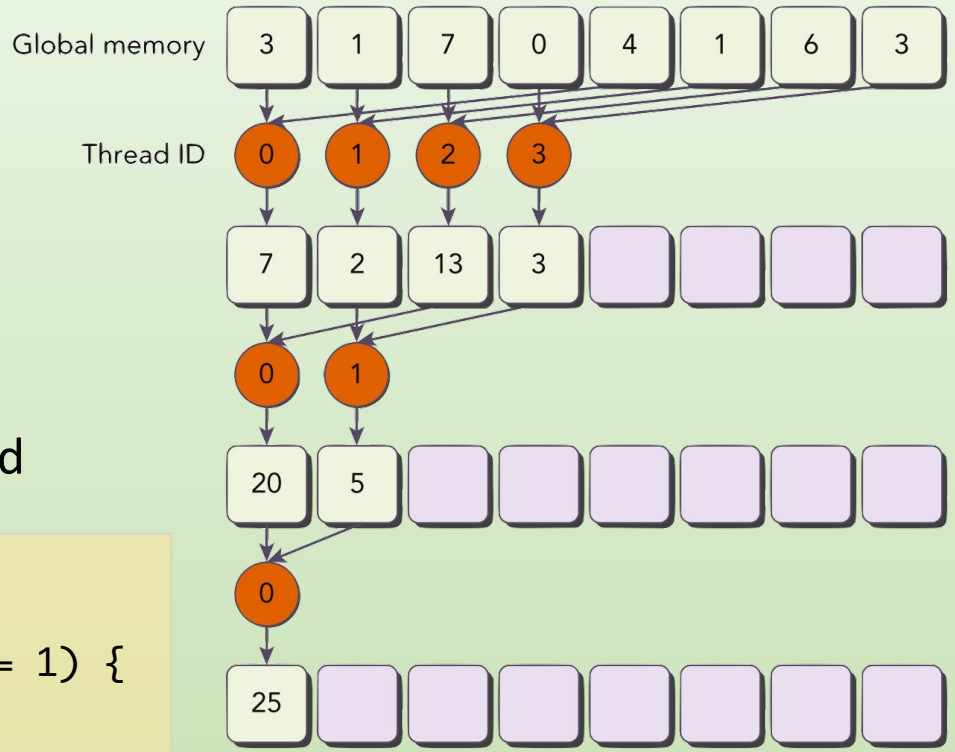


Approccio sequenziale



Schema sequenziale

- ✓ **Soluzione locale**: somma parziale sincrona sui blocchi della grid ottenuta per riduzione parallela
- ✓ **Vincolo**: la dimensione di blocco deve essere necessariamente una **potenza di 2**
- ✓ **Vantaggio**: è una soluzione più efficiente a causa di accesso contiguo a banchi di memoria globale o shared




```
// in-place reduction in global memory
for (int stride = blockDim.x / 2; stride > 0; stride >>= 1) {
   if (tid < stride)
      idata[tid] += idata[tid + stride];
   __syncthreads();
}
```

→ lab4

Par reduction con SMEM

Prodotto convolutivo con SMEM

Possibili approcci 1D e 2D

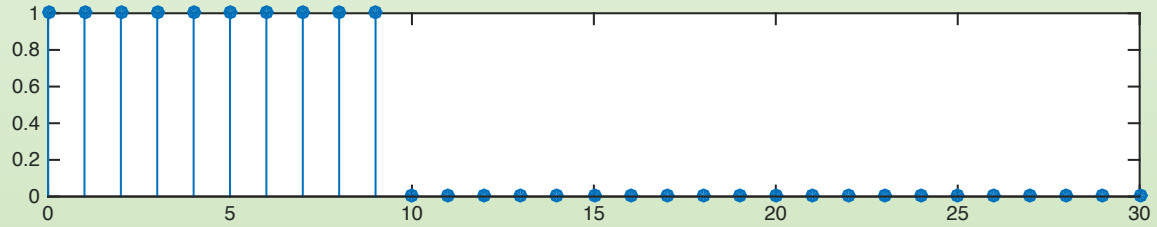
Esempio: somma di convoluzione

✓ **Somma di convoluzione** tra segnali $x(n)$ e $h(n)$ di durata finita N:

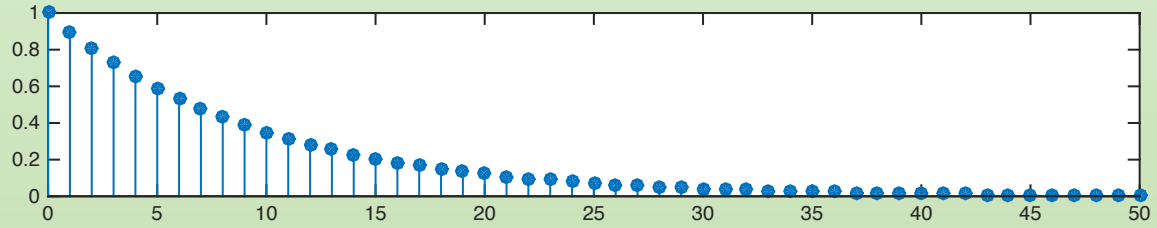
$$y(n) = h(n) * x(n) = \sum_{k=0}^{N-1} h(k)x(n-k)$$

✓ **Esempio**:

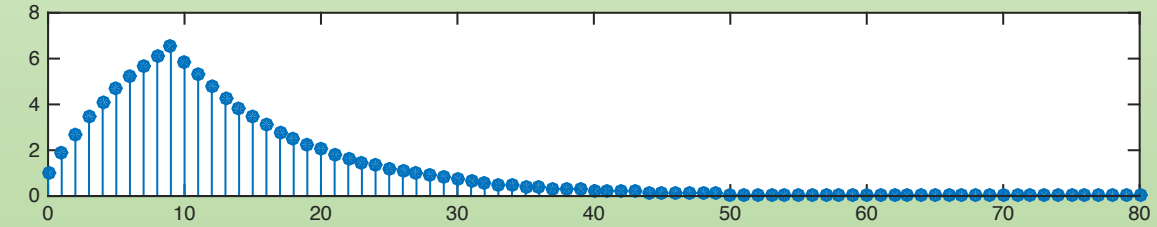
- Gradino unitario: $x(n) = u(n) - u(n-10)$



- Esponenziale decrescente: $h(n) = 0.9^n u(n)$



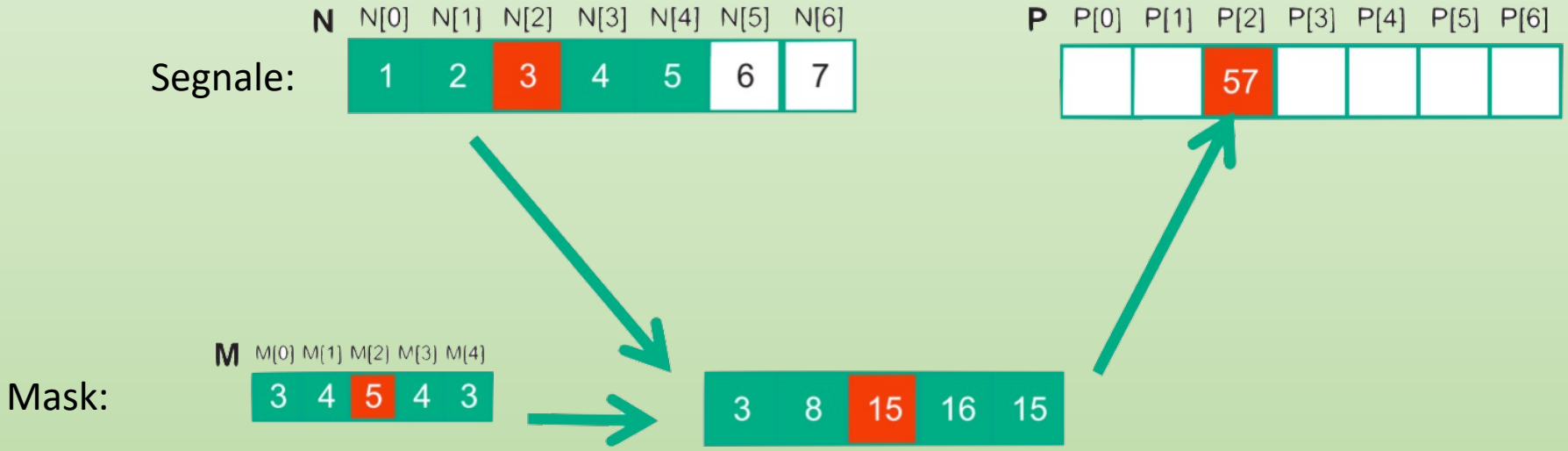
- Risultato: $y(n) = h(n) * x(n)$



Implementazione

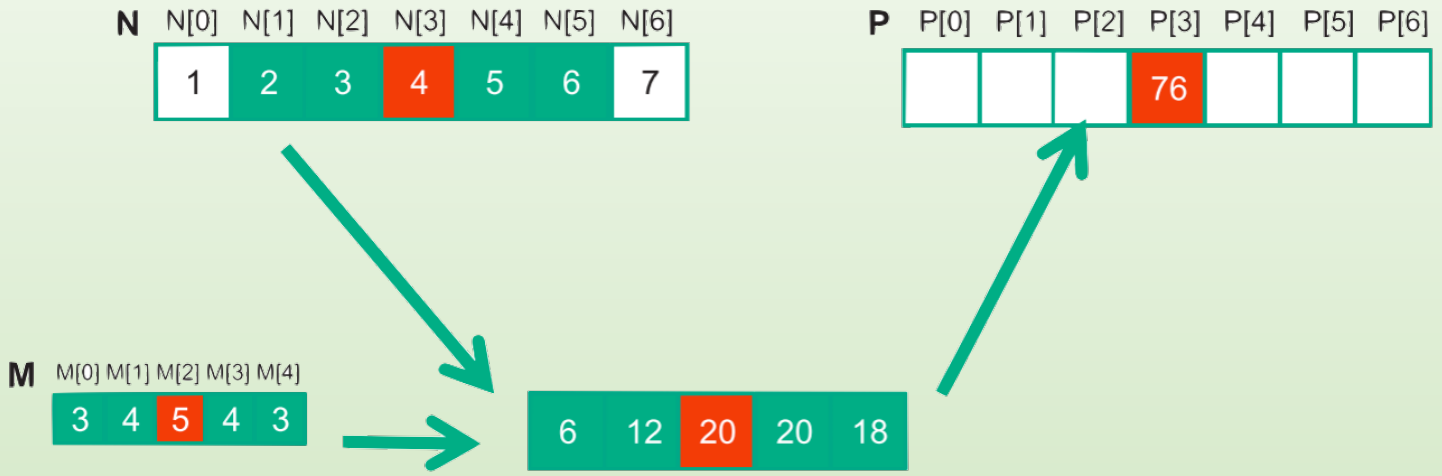
Sistema FIR: $$y(n) = \sum_{k=0}^{N-1} h(k)x(n-k) = \sum_{k=0}^{N-1} b_k x(n-k)$$

Es: media mobile: $$y(n) = \frac{1}{N} \sum_{k=0}^{N-1} x(n-k)$$

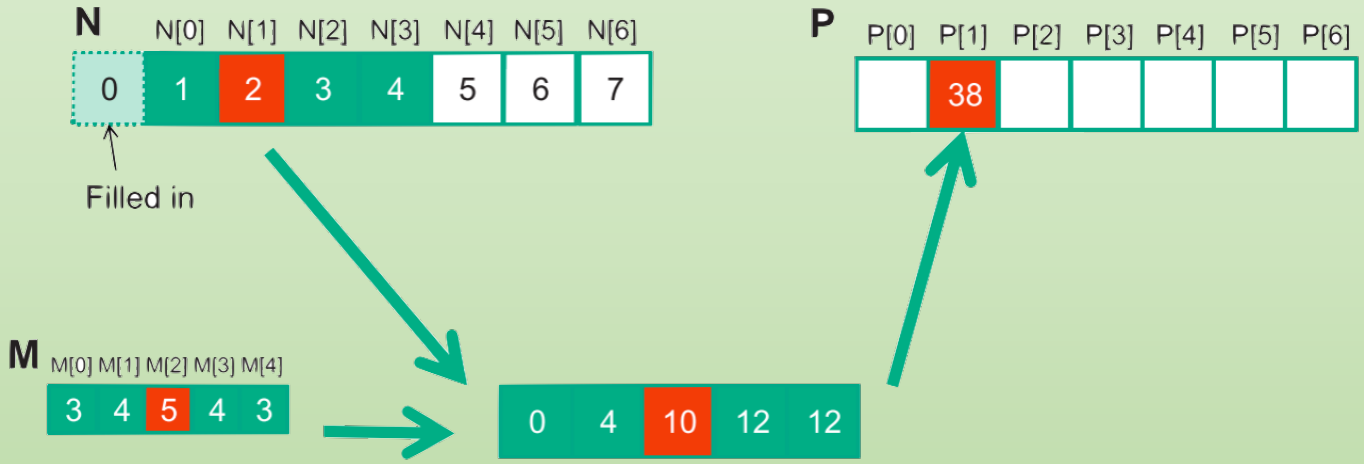


Il problema del bordo

Calcolo elemento P[3] come somma pesata di elementi di N e M



Calcolo elemento P[1] come somma pesata di elementi di M e parte di N (problema del bordo)



Convoluzione 2D - immagini

Matrice array 2D

N

1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	5	6
5	6	7	8	5	6	7
6	7	8	9	0	1	2
7	8	9	0	1	2	3

P

		321				

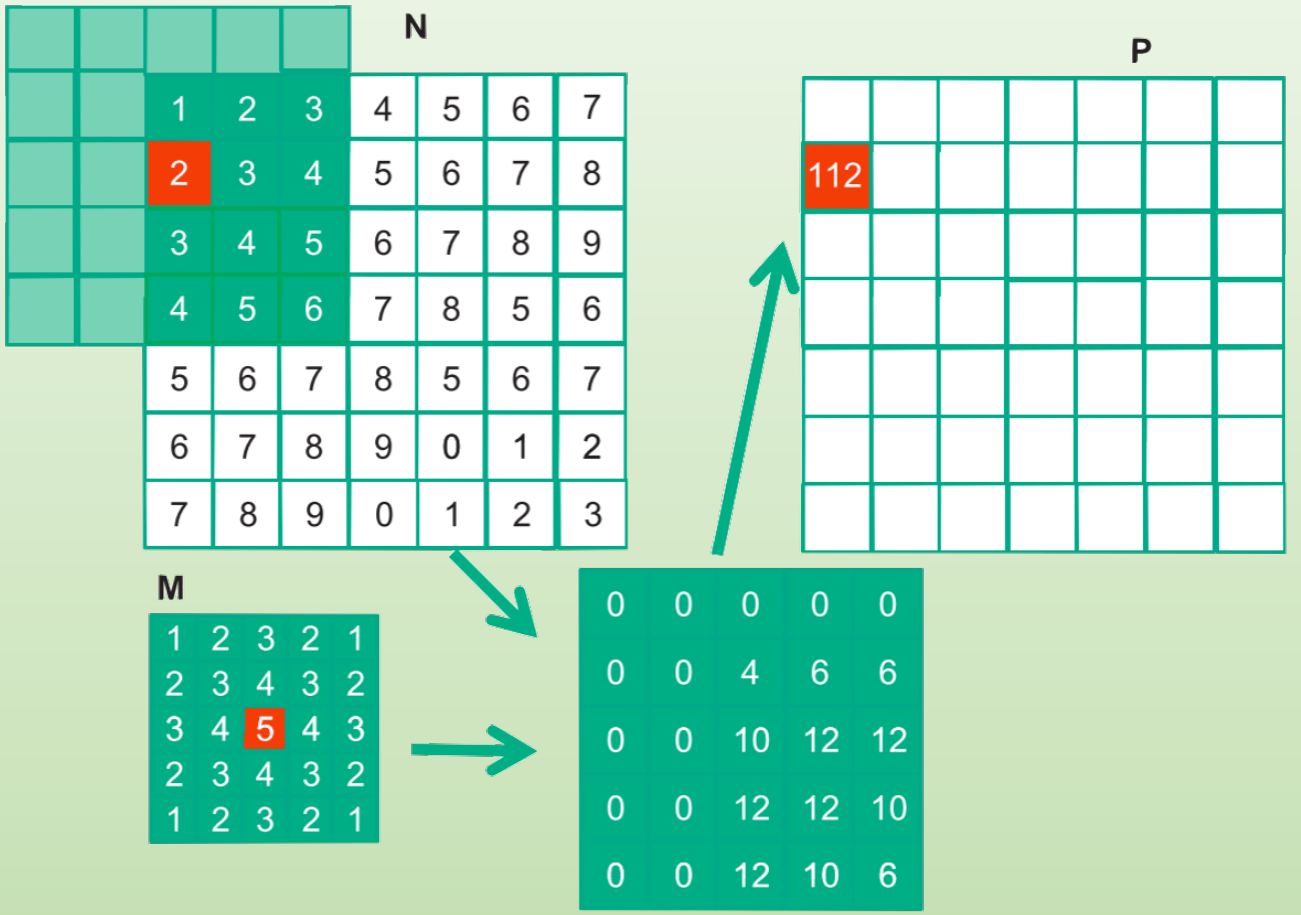
Maschera array 2D

M

1	2	3	2	1
2	3	4	3	2
3	4	5	4	3
2	3	4	3	2
1	2	3	2	1

1	4	9	8	5
4	9	16	15	12
9	16	25	24	21
8	15	24	21	16
5	12	21	16	5

Problemi di bordo 2D

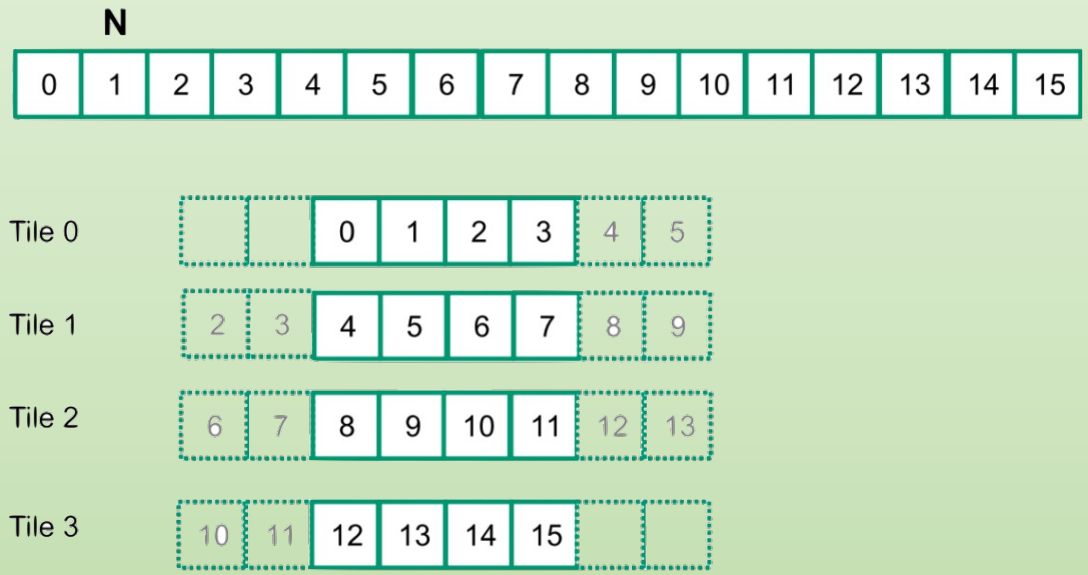


Strategia basata su tiling

- ✓ Array di sedici elementi **1D**
- ✓ Grid = **4 blocchi di 4 thread** ciascuno (dimensione tile)
- ✓ Ci sono **4 tile in output** (la prima copre da P[0] a P[3], la seconda da P[4] a P[7], etc.)
- ✓ Quale strategia per ridurre accesso a global memory?

STRATEGIA:

- ✓ **Block_size = tile**
- ✓ **MASK_WIDTH =** dim della mask (dispari)
- ✓ Caricare in memoria shared tutti gli elementi necessari a calcolare tutti gli output associati a un blocco
- ✓ La dimensione della shared memory è pari alla taglia del tile pari a:
  TILE_SIZE + MASK_WIDTH - 1




```
__shared__ float share_mem[TILE_SIZE + MASK_WIDTH - 1];
```

Convoluzione parallela

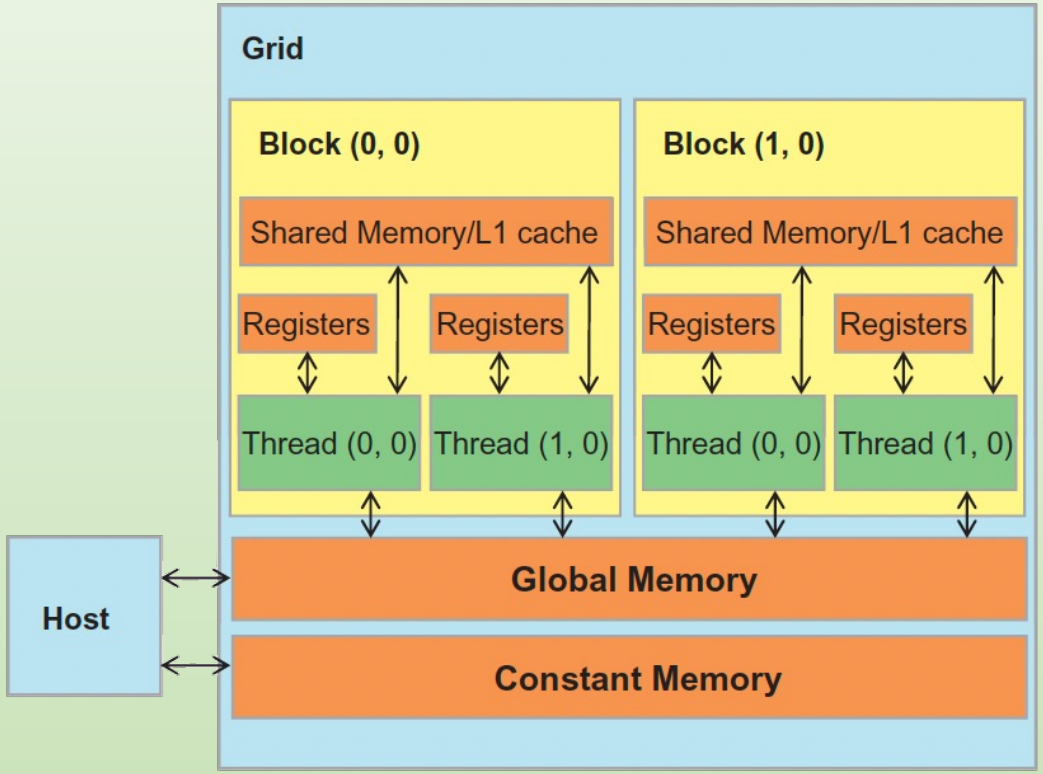
Problema **ideale** per parallel computing!

- ✓ Primo passo è determinare la **mappa** tra thread ed elementi di output
- ✓ Semplice approccio: ogni thread calcola un elemento della matrice di convoluzione
- ✓ **Problema**: non soddisfa per inefficienza negli accessi alla global memory!

```
__global__ void convolution_1D_basic_kernel(float *N, float *M, float *P, int Mask_Width, int Width) {
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  float Pvalue = 0;
  int start_point = i - (Mask_Width/2);

  for (int j = 0; j < Mask_Width; j++) {
    if (start_point + j >= 0 && start_point + j < Width)
      Pvalue += N[start_point + j] * M[j];
  }
  P[i] = Pvalue;
}
```

Constant memory and caching

- ✓ The size of the **Mask** array is typically **small** (compared to data)
- ✓ The **contents** of **Mask** are **not changed** throughout the execution of the kernel
- ✓ **All threads** need to access the **mask elements**
- ✓ Even better, all threads **access** the **Mask** elements in the **same order** starting from `M[0]` …



Codice CUDA (1)

MASK size

TILE size

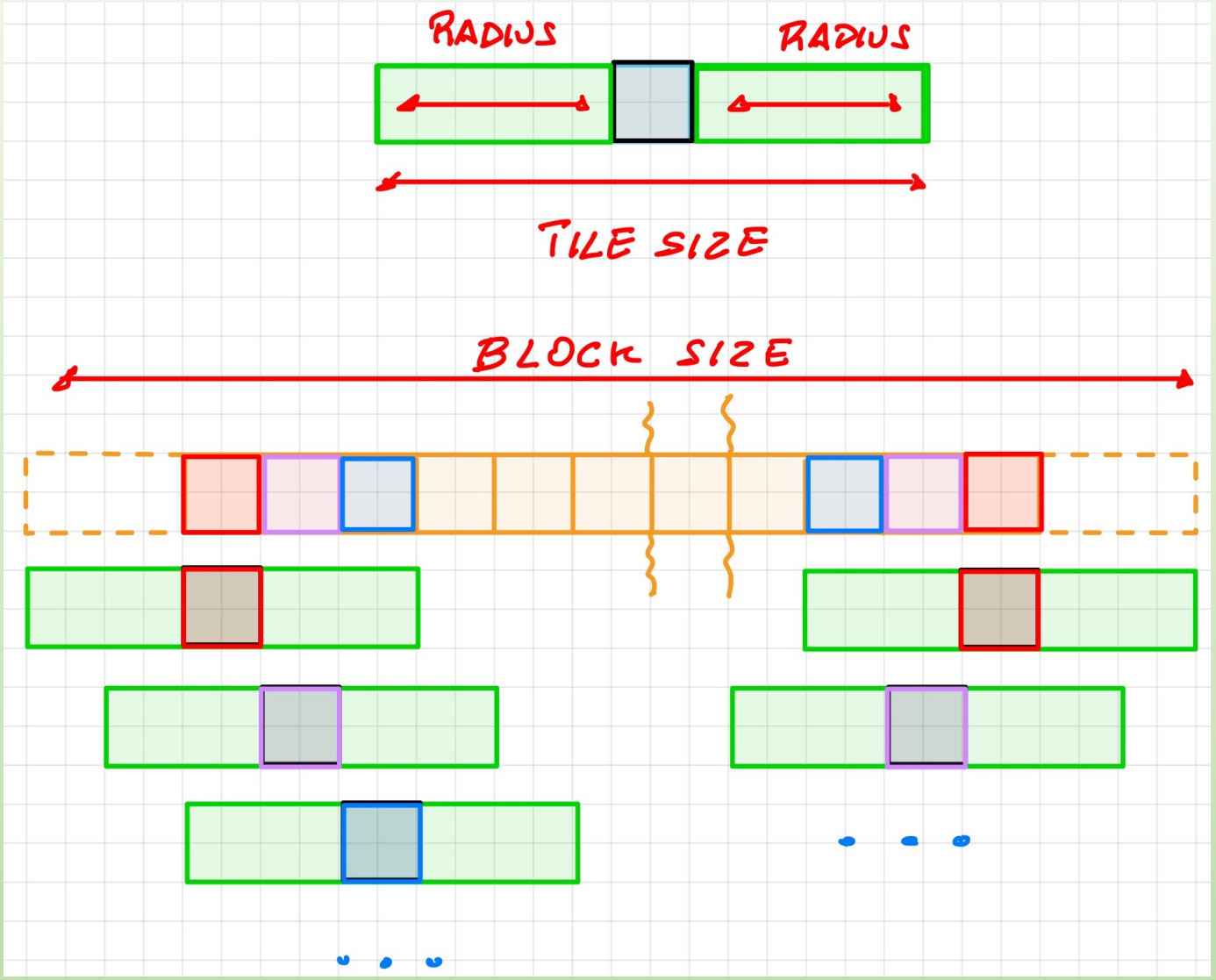
Constant memory for mask

Host allocation for constant memory

Device copy of constant memory

```
#define MASK_RADIUS   5
#define MASK_SIZE     2 * MASK_RADIUS + 1
#define BLOCK_SIZE    128
#define TILE_WIDTH    BLOCK_SIZE + MASK_SIZE - 1

__device__ __constant__ float d_mask[MASK_SIZE];

// MAIN: convolution 1D
int main(int argc, char **argv) {

  // allocate host memory
  float *h_mask = (float *) malloc(nBytes_mask);

  // copy data from host to device
  cudaMemcpy(d_data, h_data, nBytes, cudaMemcpyHostToDevice);
  cudaMemcpyToSymbol(d_mask, h_mask, nBytes_mask);
. . .
```

Picture



Codice CUDA (2)

conv1D kernel
Hold if
MASK_RADIUS < BLOCK_SIZE

shared memory
for a TILE

Left halo check

center halo check

```
// kernel for 1D convolution: MASK_RADIUS < BLOCK_SIZE
__global__ void conv1D(float *result, float *data, int n) {
  unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

  // shared memory size = BLOCK_SIZE + MASK
  __shared__ float tile[TILE_WIDTH];

  // boundary
  int left = blockIdx.x * blockDim.x - MASK_RADIUS;
  int right = (blockIdx.x + 1) * blockDim.x;

  // left halo
  if (threadIdx.x < MASK_RADIUS)
    tile[threadIdx.x] = left < 0 ? 0 : data[left + threadIdx.x];

  // center
  tile[threadIdx.x + MASK_RADIUS] = data[i];
```

Codice CUDA (3)

right halo check

Compute convolution

Store final result

```
  // right halo
  if (threadIdx.x >= blockDim.x - MASK_RADIUS)
    tile[threadIdx.x + MASK_SIZE - 1] = right >= n ? 0 :
                        data[right + threadIdx.x - blockDim.x + MASK_RADIUS];

  __syncthreads();

  // convolution: tile * mask
  float sum = 0;
  for (int i = -MASK_RADIUS; i <= MASK_RADIUS; i++)
    sum += tile[threadIdx.x + MASK_RADIUS + i] * d_mask[i + MASK_RADIUS];

  // final result
  result[i] = sum;
}
```

Codice CUDA (4)

MASK size

TILE size

Host allocation for constant memory

Device copy for constant memory

```
#define MASK_RADIUS   5
#define MASK_SIZE     2 * MASK_RADIUS + 1
#define BLOCK_SIZE    128
#define TILE_WIDTH    BLOCK_SIZE + MASK_SIZE - 1

// MAIN: convolution 1D
int main(int argc, char **argv) {

. . .

  // grid configuration
  dim3 block(BLOCK_SIZE);
  dim3 grid((n + BLOCK_SIZE - 1) / BLOCK_SIZE);

  // kernel launch
  convolution1D<<<grid, block>>>(d_result, d_data, n);
```

→ lab4

Convoluzione con tiling

Riferimenti bibliografici

Testi generali:

1. J. Cheng, M. Grossman, T. Mckercher, Professional Cuda C Programming, Wrox Pr Inc. (1^ ed), 2014 (cap 3)
2. ×□□W !"#$%&'()*+,-./0123456789:;□ÿ□□<=□>□□?@ABCDEFGHIJ□KLMN□O□Q□□□□□#&):□□'0<68□9,24.5□-$%(□/13□=□>□?@□A 3-4)

NVIDIA docs:

1. CUDA Programming: *http://docs.nvidia.com/cuda/cuda-c-programming-guide/*
2. CUDA C Best Practices Guide: *http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/*
3. CUDA compilation process: *https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#using-separate-compilation-in-cuda*