

GPU Computing

Laurea Magistrale in Informatica - AA 2021/22

Docente **G. Grossi**

Lezione 6 – Modello di memoria CUDA e 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 prodotto di matrici e di convoluzione

Le memorie della GPU

Definizione e significato

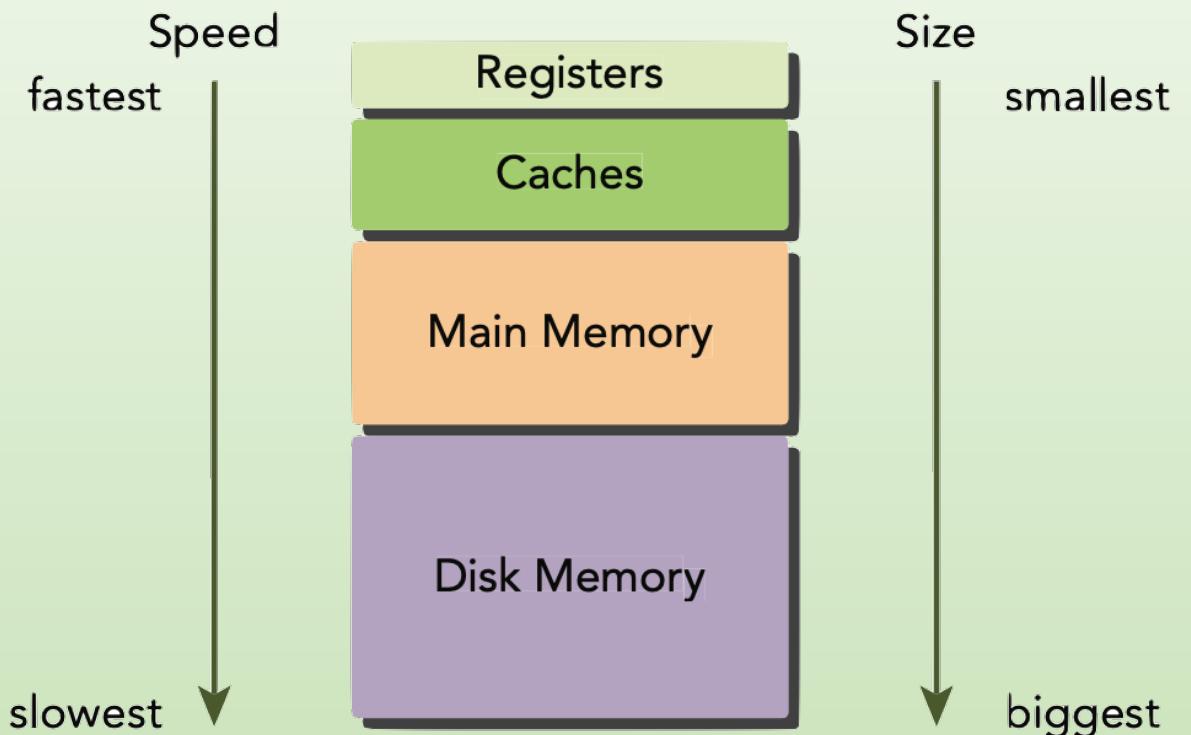
La memoria nei moderni sistemi di calcolo

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

- ✓ Illusion of **large but low-latency** memory
- ✓ **High-latency, high-capacity** (for storage)
- ✓ **Low-latency, low-capacity** (for processor)



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

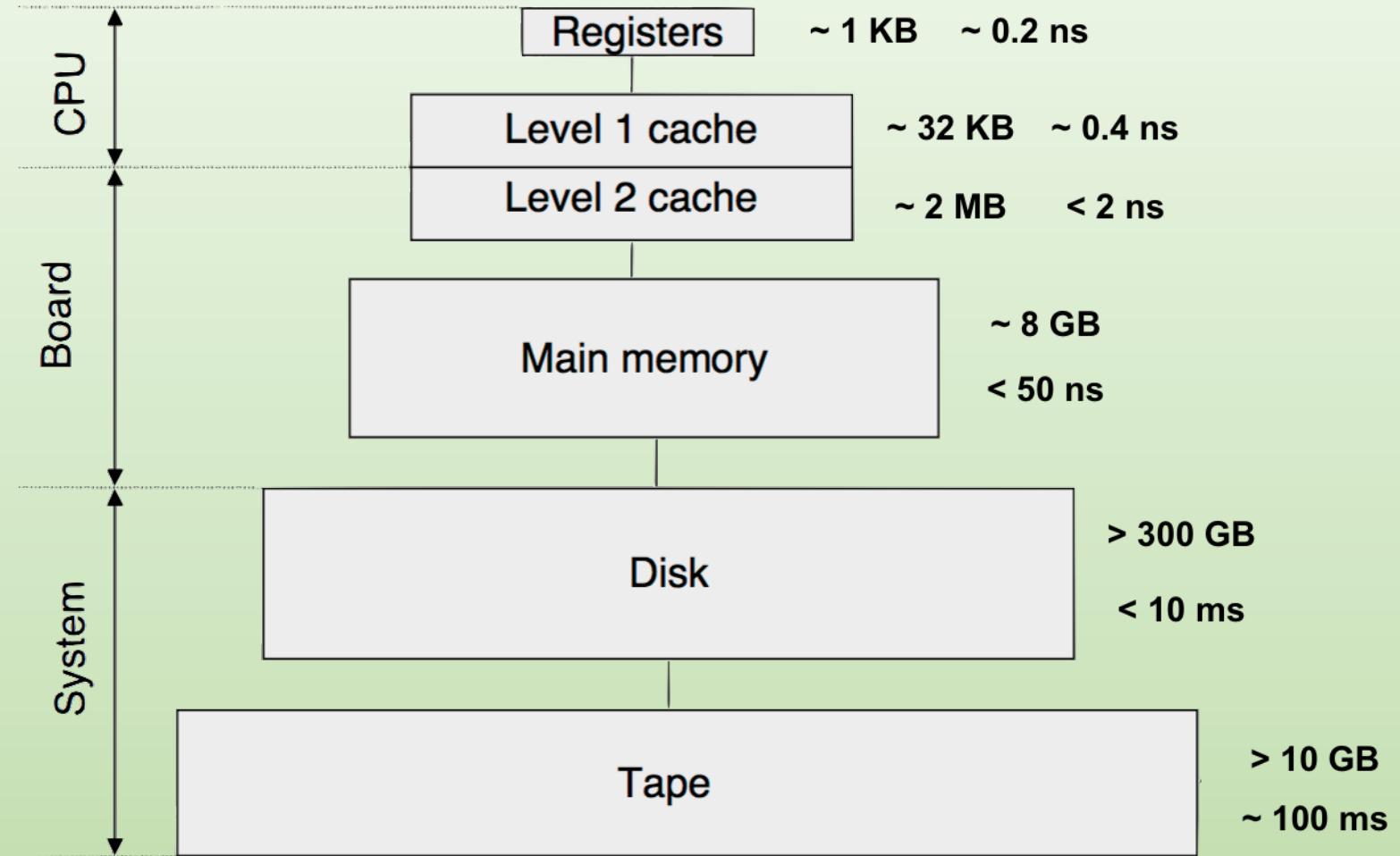
Principio di località e gerarchia di memorie

Località spaziale

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

Località temporale

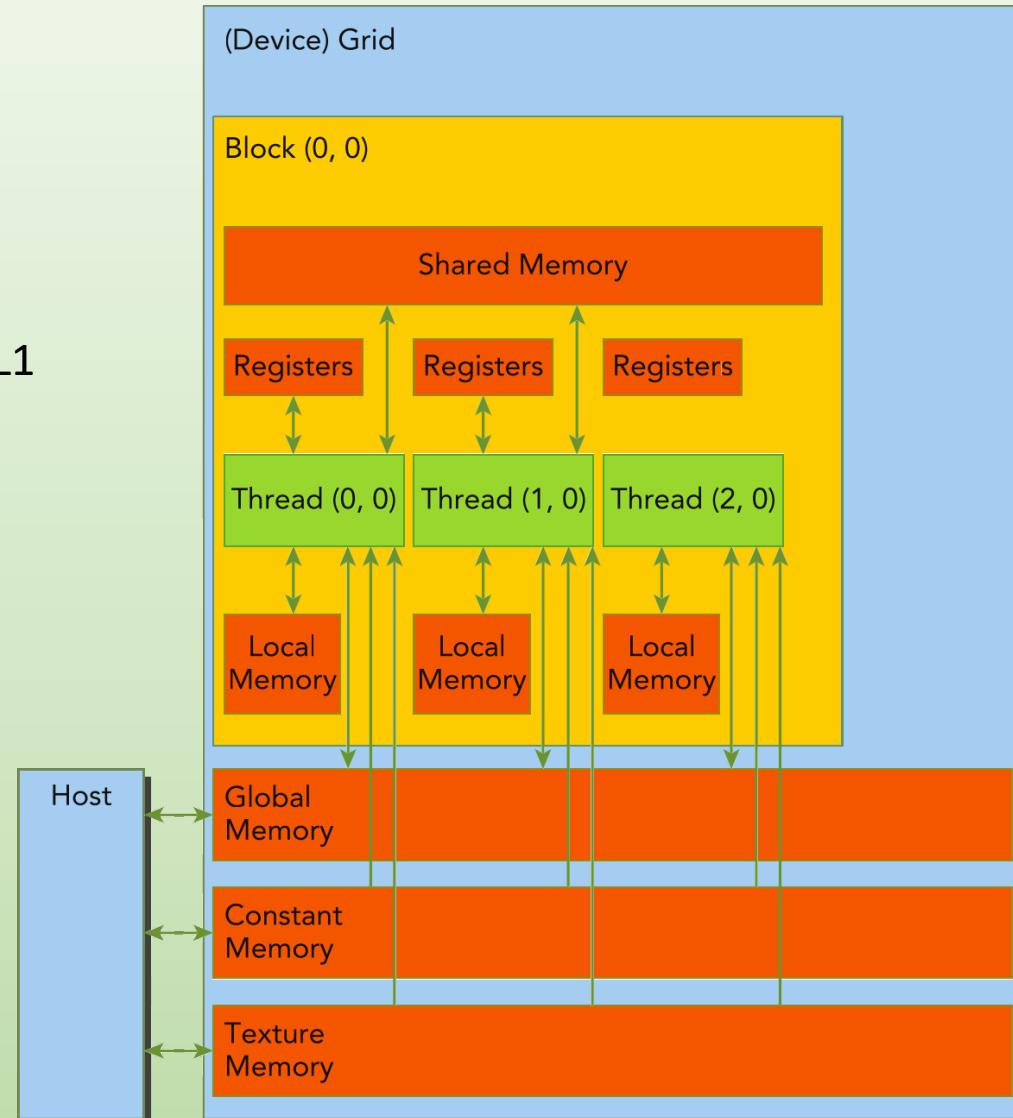
- se un'istruzione viene eseguita al tempo t , allora con alta probabilità la stessa istruzione verrà eseguita al tempo $t + \Delta t$ (Δt è un intero piccolo)



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**



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
 . . .
```

Calcolo max num blocchi

- ✓ Cuda **deviceQuery** sulla GPU (colab) Tesla P100

GTS can assign to
an SM before the
SM continuously
says it can no
longer accept any
more blocks

```
Device 0: "Tesla P100-PCIE-16GB"
    CUDA Driver Version / Runtime Version: 10.2 / 10.2
    CUDA Capability Major/Minor version number: 6.0
        Total number of registers available per block: 65536
        Maximum number of threads per multiprocessor: 2048
        Maximum number of resident blocks per multiprocessor: 32
```

- ✓ Blocchi da 128 threads: $128 \times 11 = 1408$ regs
 - Max num blocks = $65536 / 1408 = 46.54 \rightarrow 46$ blocks
- ✓ Blocchi da 512 threads: $512 \times 11 = 5632$ regs
 - Max num blocks = $65536 / 5632 = 11.63 \rightarrow 11$ blocks

Specifiche tecniche

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

Technical specifications	Compute capability (version)																			
	1.0	1.1	1.2	1.3	2.x	3.0	3.2	3.5	3.7	5.0	5.2	5.3	6.0	6.1	6.2	7.0	7.2	7.5	8.0	8.6
Maximum number of resident grids per device (concurrent kernel execution)	t.b.d.		16	4		32				16	128	32	16	128	16			128		
Maximum dimensionality of grid of thread blocks	2															3				
Maximum x-dimension of a grid of thread blocks	65535															$2^{31} - 1$				
Maximum y-, or z-dimension of a grid of thread blocks															65535					
Maximum dimensionality of thread block															3					
Maximum x- or y-dimension of a block	512														1024					
Maximum z-dimension of a block															64					
Maximum number of threads per block	512														1024					
Warp size															32					
Maximum number of resident blocks per multiprocessor	8				16										32			16	32	16
Maximum number of resident warps per multiprocessor	24	32	48												64			32	64	48
Maximum number of resident threads per multiprocessor	768	1024	1536												2048			1024	2048	1536
Number of 32-bit registers per multiprocessor	8 K	16 K	32 K	64 K	128 K										64 K					
Maximum number of 32-bit registers per thread block	N/A	32 K	64 K	32 K		64 K		32 K	64 K	32 K							64 K			
Maximum number of 32-bit registers per thread	124	63													255					
Maximum amount of shared memory per multiprocessor	16 KB		48 KB		112 KB	64 KB	96 KB	64 KB	96 KB	64 KB	96 KB	(of 128)	64 KB	164 KB	100 KB					
Maximum amount of shared memory per thread block				48 KB										96 KB	48 KB	64 KB	163 KB	99 KB		
Number of shared memory banks	16													32						
Amount of local memory per thread	16 KB													512 KB						
Constant memory size														64 KB						
Cache working set per multiprocessor for constant memory					8 KB					4 KB				8 KB						
Cache working set per multiprocessor for texture memory	6 – 8 KB	12 KB	12 – 48 KB	24 KB	48 KB	N/A	24 KB	48 KB	24 KB	32 – 128 KB	32 – 64 KB	28 – 192 KB	28 – 128 KB							
Maximum width for 1D texture reference bound to a CUDA array	8192		65536											131072						

Register spilling

- ✓ Il **massimo numero** di registri per thread può essere definito manualmente a tempo di compilazione usando l'opzione **-maxrregcount**
- ✓ Indagata a livello di compilazione con l'opzione **--ptxas-options=-v**

```
$ nvcc --ptxas-options=-v -arch=sm_30 mat_prod.cu
ptxas info      : 512 bytes gmem
ptxas info      : Compiling entry function '_Z11matrix_prodPfS_S_' for 'sm_30'
ptxas info      : Function properties for _Z11matrix_prodPfS_S_
                  0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info      : Used 22 registers, 344 bytes cmem[0]
```

```
$ nvcc --ptxas-options=-v -maxrregcount 18 mat_prod.cu
ptxas info      : Used 18 registers, 344 bytes cmem[0]
```

Register bounds

- ✓ Per **minimizzare lo spilling** si può ricorrere al qualificatore **`__launch_bounds__`** nella definizione di kernel

```
__global__ void
__launch_bounds__(maxThreadsPerBlock, minBlocksPerMultiprocessor) MyKernel(...) {
    ...
}
```

Programming Guide :

If launch bounds are specified, the compiler first derives from them the upper limit **L** on the number of registers the kernel should use to ensure that **minBlocksPerMultiprocessor** blocks (or a single block if **minBlocksPerMultiprocessor** is not specified) of **maxThreadsPerBlock** threads can reside on the multiprocessor. The compiler then optimizes register usage in the following way:

- If the initial register usage is higher than **L**, the compiler reduces it further until it becomes less or equal to **L**, usually at the expense of more local memory usage and/or higher number of instructions

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

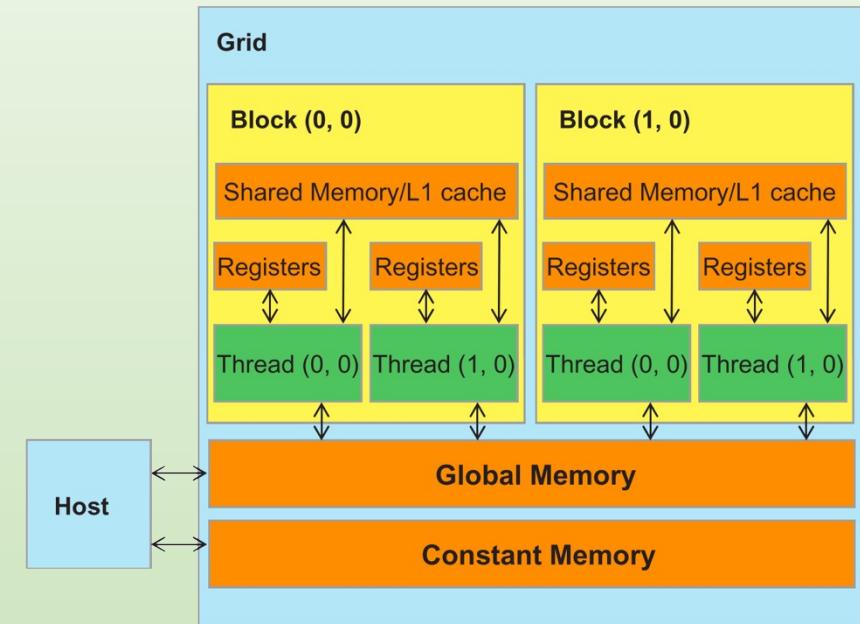
- ✓ Risiede in **device memory** (64K per ogni famiglia) ed ha una cache dedicata in ogni SM
- ✓ Memoria ideale per ospitare dati a cui si accede in **modo uniforme** e in **sola lettura**
- ✓ Una variabile costante è dichiarata con l'attributo: **constant**
- ✓ Una variabile costante deve essere dichiarata 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
- ✓ 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



Texture memory

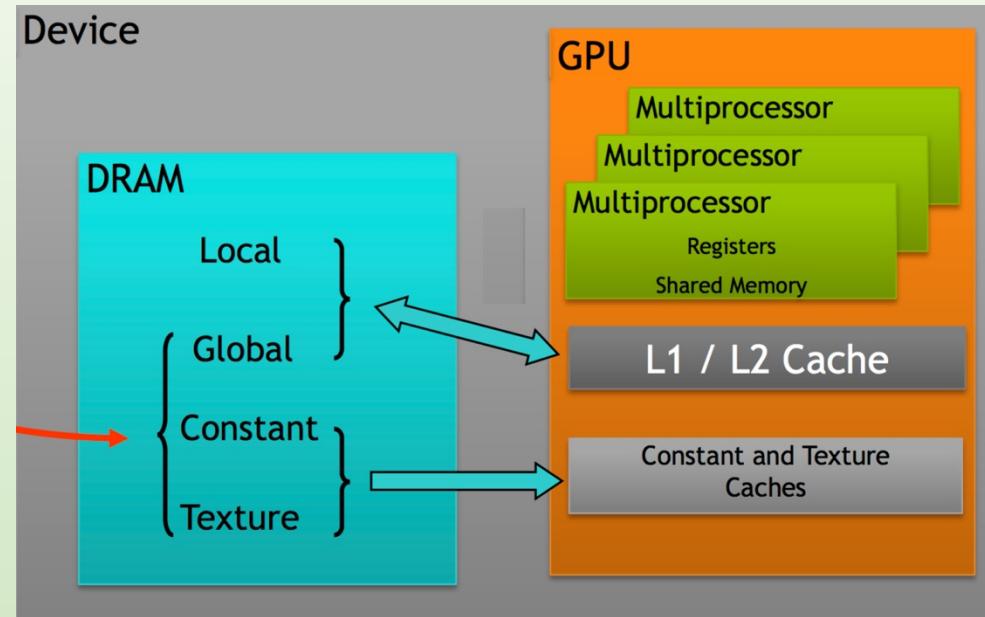
- ✓ La Texture memory risiede nella **device memory** ed ha una cache per-SM, la **read-only** cache ed è acceduta solo attraverso di essa
- ✓ La cache read-only include un supporto hw efficiente per **filtraggio** o **interpolazione** floating-point nel processo di lettura dei dati
- ✓ La Texture memory 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 **migliori prestazioni** rispetto a quelle standard
- ✓ Adatta per applicazioni in cui servono classiche operazioni hw lineari, come **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



Hardware resources are finite

$$\text{Occupancy} = \frac{\text{warp attivi}}{\text{max numero warp}}$$

Limits on the numerator:

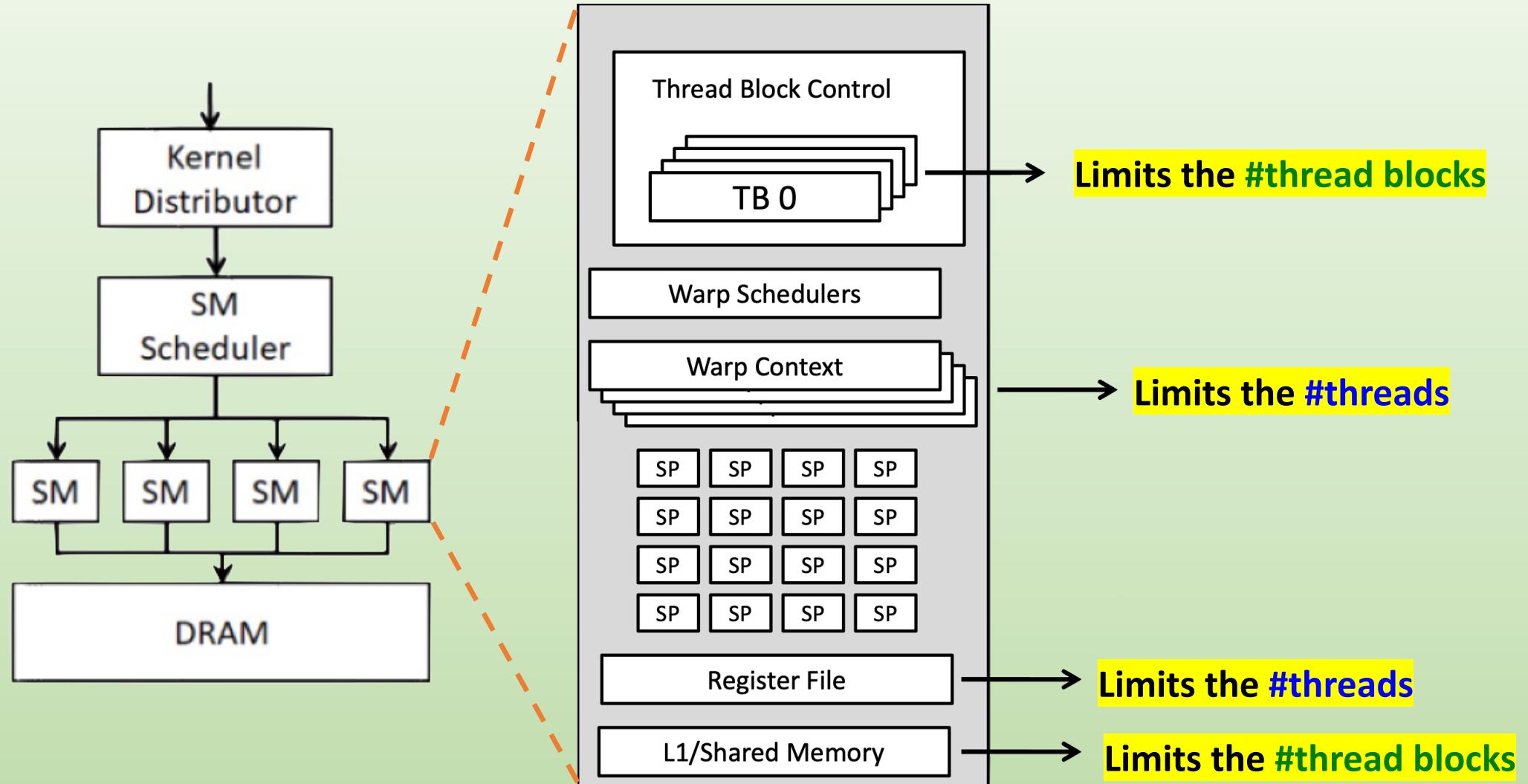
- Shared memory/thread block
- Registers/thread
- Number of scheduling slots: blocks, warps

Measures how well the concurrency/parallelism is used

Limits on the denominator:

- Memory bandwidth
- Scheduler slots

Hardware resources are finite



Impact of thread block size

Maximum number of resident blocks per multiprocessor	8			16	32			16	32	16
Maximum number of resident warps per multiprocessor	24	32	48		64			32	64	48
Maximum number of resident threads per multiprocessor	768	1024	1536		2048			1024	2048	1536

Example: Volta v100 (CC 7.0)

- max active warps/SM = 64 (limit: warp context)
- max active blocks/SM = 32 (limit: block control)
- ✓ With 512 threads/block how many blocks can execute (per SM) concurrently?
 - Max active warps * threads/warp = $64 * 32 = 2048$ threads → 4 blocks
- ✓ With 128 threads/block?
 - Max active warps * threads/warp = $64 * 32 = 2048$ threads → 16 blocks
- ✓ Consider HW limit of 32 thread blocks/SM @ 32 threads/block:
 - Blocks are maxed out, but max active threads = $32 * 32 = 1024$
 - Occupancy = 0.5 (= $1024 / 2048$)

Impact of #registers per thread

Registers/thread can limit number of active threads!

Maximum number of 32-bit registers per thread block	N/A	32 K	64 K						
Maximum number of 32-bit registers per thread	124	63					255		64 K

Volta v100 (CC 7.0)

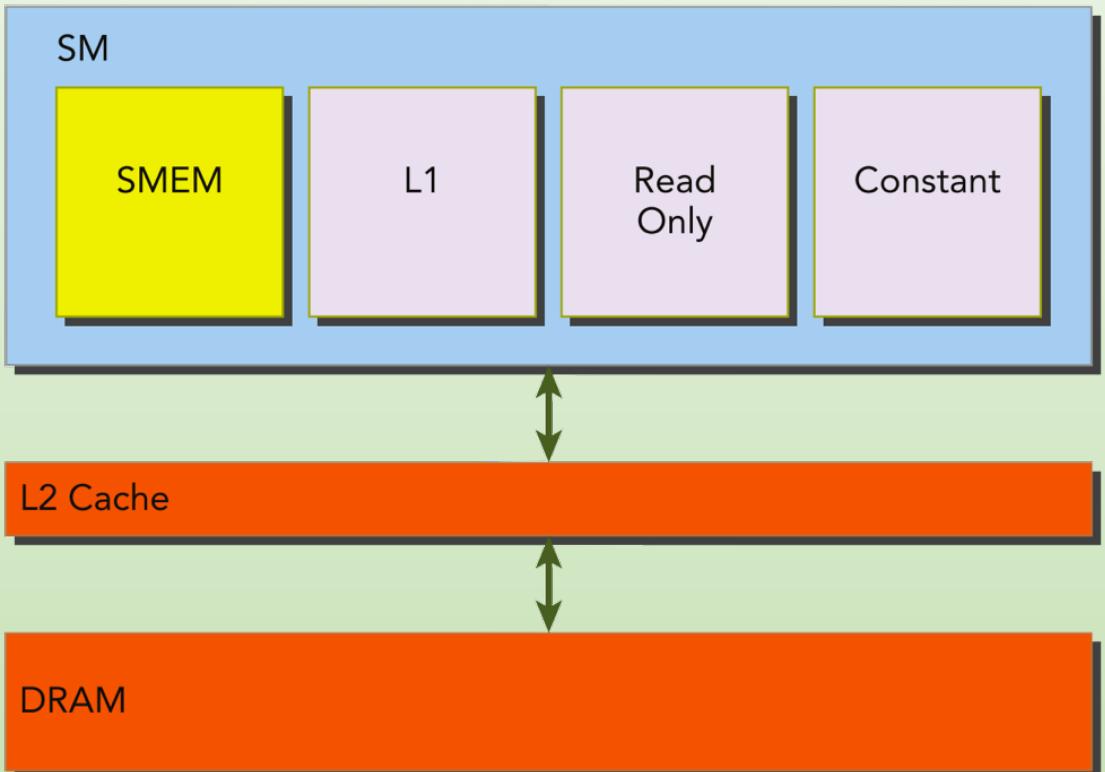
- Registers per thread max: 255
 - Registers per thread block (SM): 64K
- ✓ Assume a kernel uses 32 registers/thread, thread block size of 256
- Thus, a thread block requires 8192 registers for a maximum of 8 thread blocks per SM
 - Uses all 2048 thread slots (8 blocks * 256 threads/block)
 - FULLY Occupied!**
- ✓ What is the impact of increasing number of registers by 2?
- Loss of concurrency of 256 threads!
 - 64 regs/thread * 256 threads/block * 7 blocks/SM = 60k registers,
 - Occupancy drops to .875!

Shared Memory

Definizione e uso

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 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

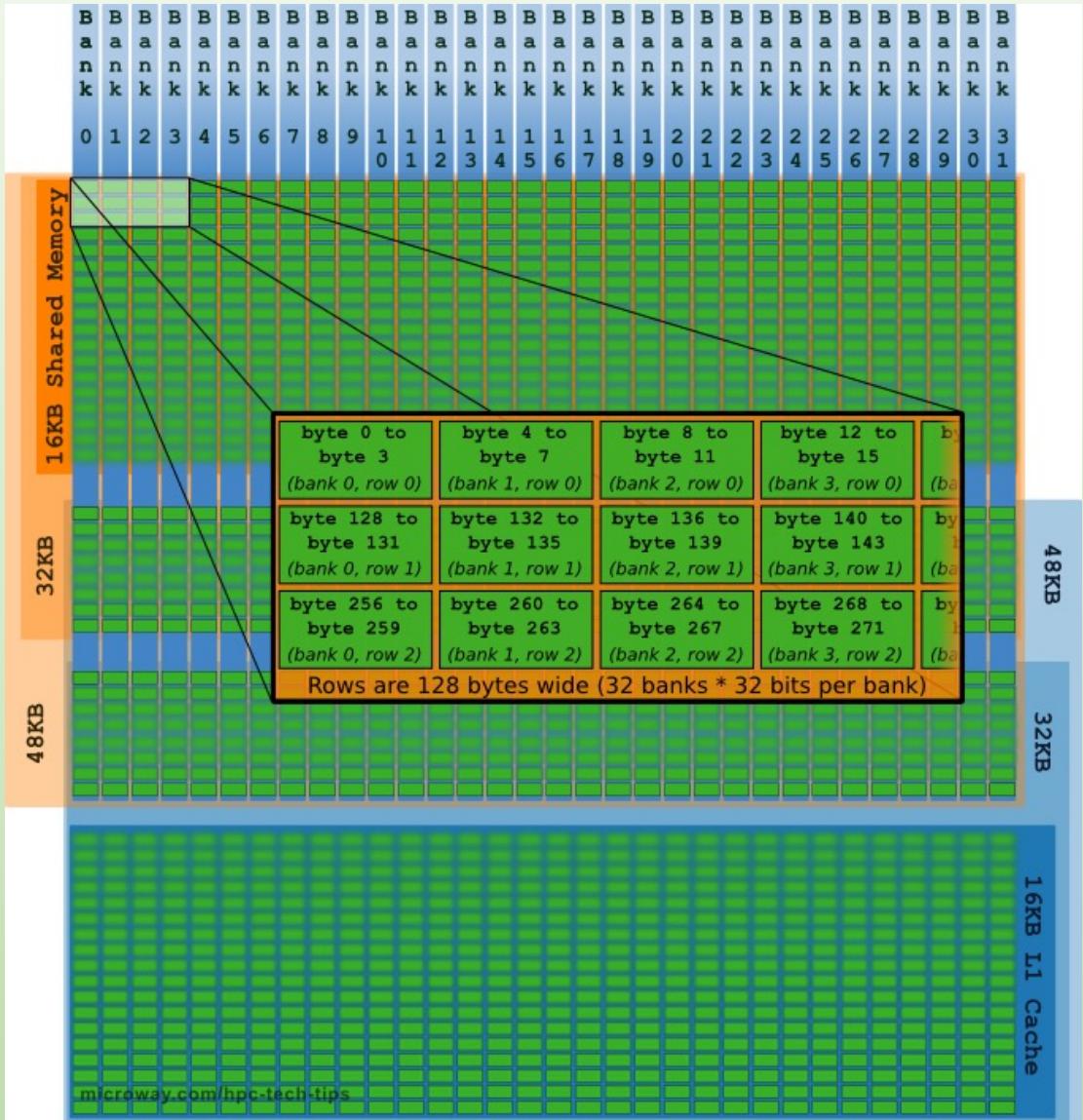
- ✓ La SMEM dell'arch Fermi sono suddivisi in blocchi da **4 byte** (chiamate **word**) o da **32 bit**
- ✓ Ogni word può contenere **1 int**, **1 float**, **2 shoth**, **4 char**, **2 half float**, ... Se chiedo un singolo byte viene comunque letta la word per intero cui il byte appartiene
- ✓ La **bandwidth** 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 usage

- ✓ A common problem arises when memory is shared: with all memory available to all threads, there will be many threads accessing the data simultaneously
- ✓ To alleviate this potential bottleneck, shared memory is divided into **32 logical banks**
- ✓ Successive sections of memory are assigned to successive banks
- ✓ If there are insufficient registers or shared memory on each SM to process at least one block, the kernel launch will fail

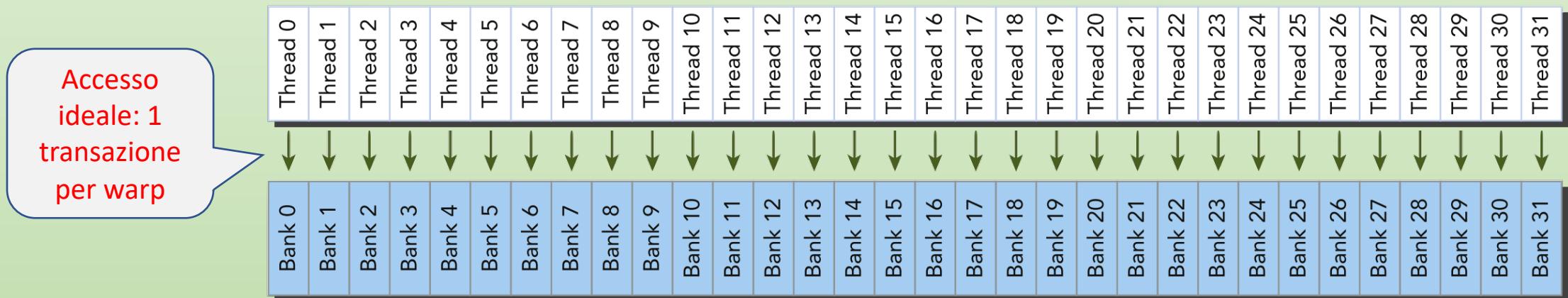


SMEM a runtime

- ✓ 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**
- ✓ Nel caso migliore serve 32 thread del warp in **2 cicli di clock!**

Pattern di accesso

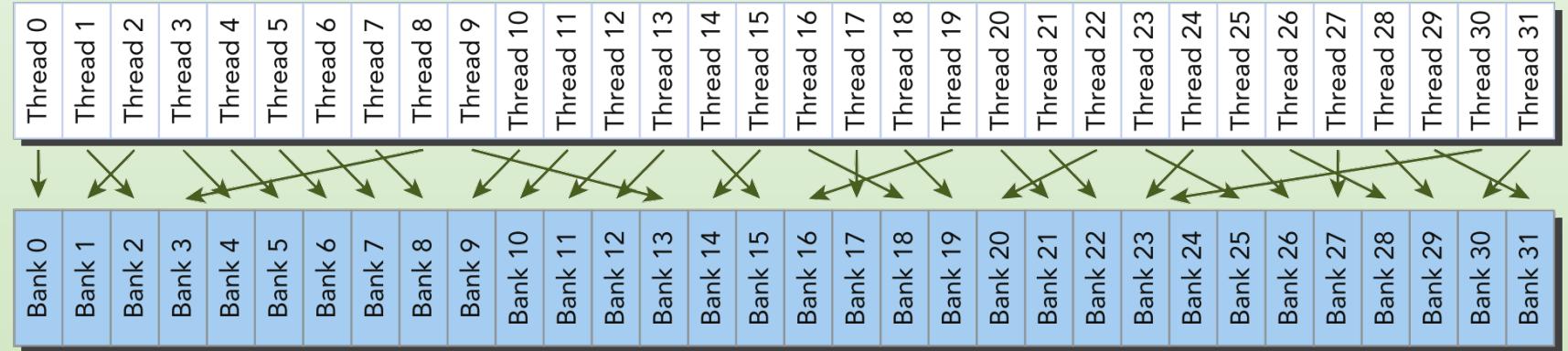
- ✓ Se un'operazione di **load** o **store** eseguita da un warp richiede al più un accesso per bank, si può effettuare in una sola transizione il trasferimento dati dalla shared memory al warp
 - ✓ In alternativa sono richieste diverse (≥ 2 e ≤ 32) transazioni con effetti negativi sulla bandwidth globale



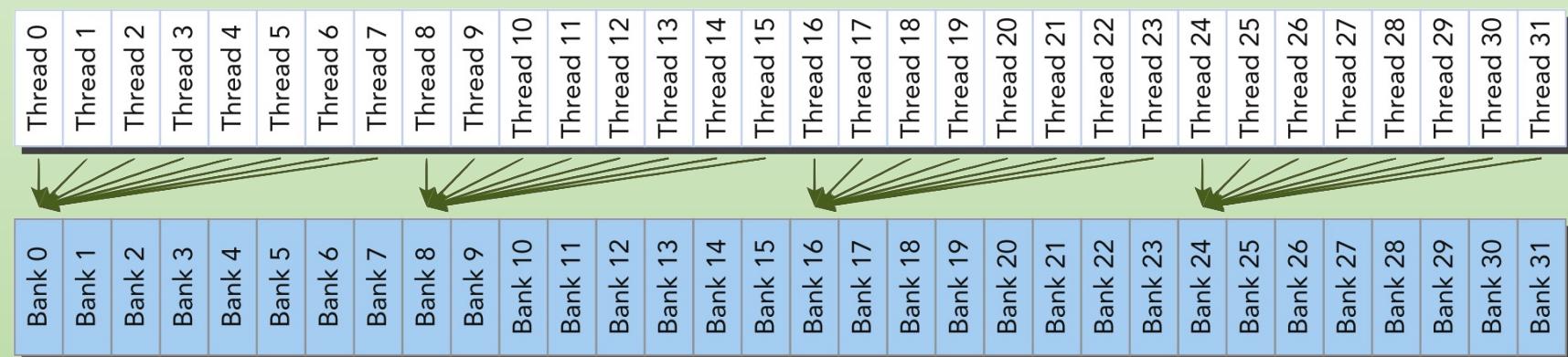
Conflitti

- ✓ **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)

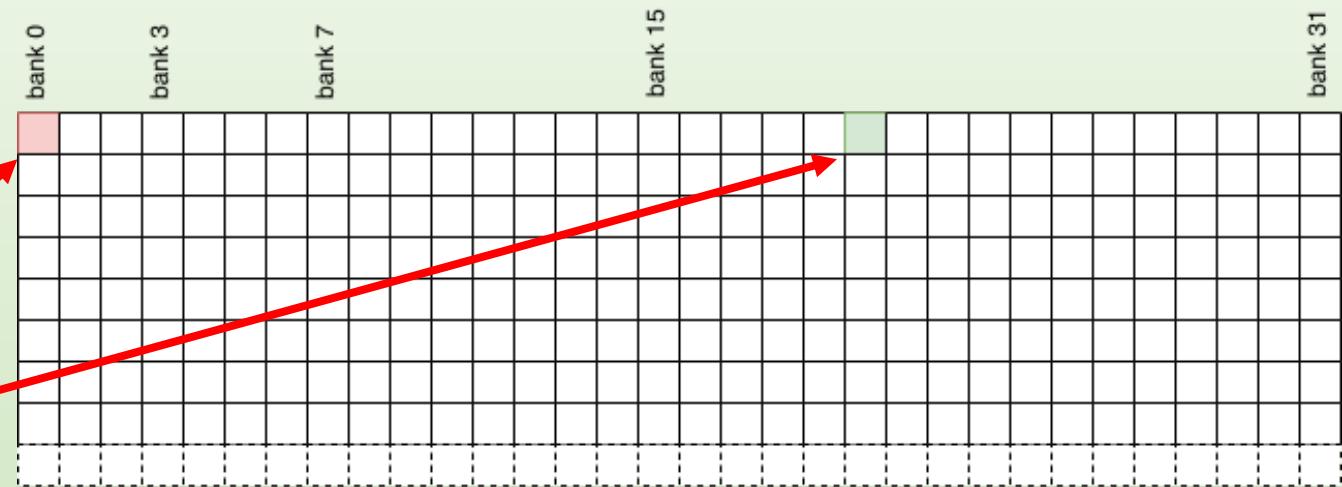


Pattern di accesso ...

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

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

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



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

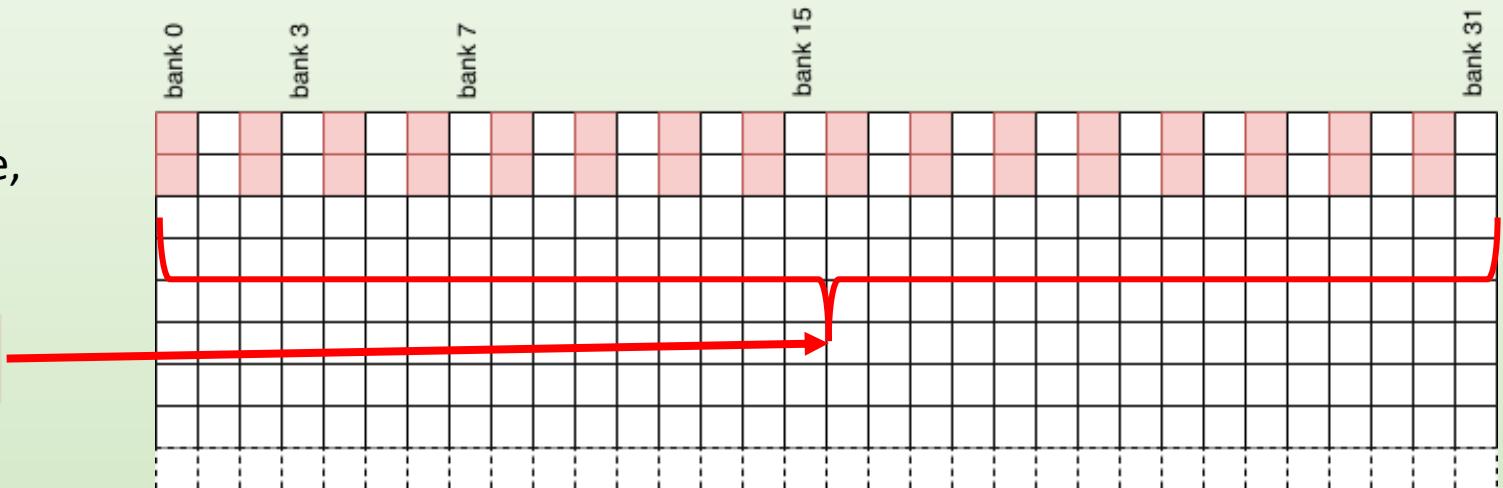
```
float f = array_f[threadIdx.x];
```



... pattern di accesso

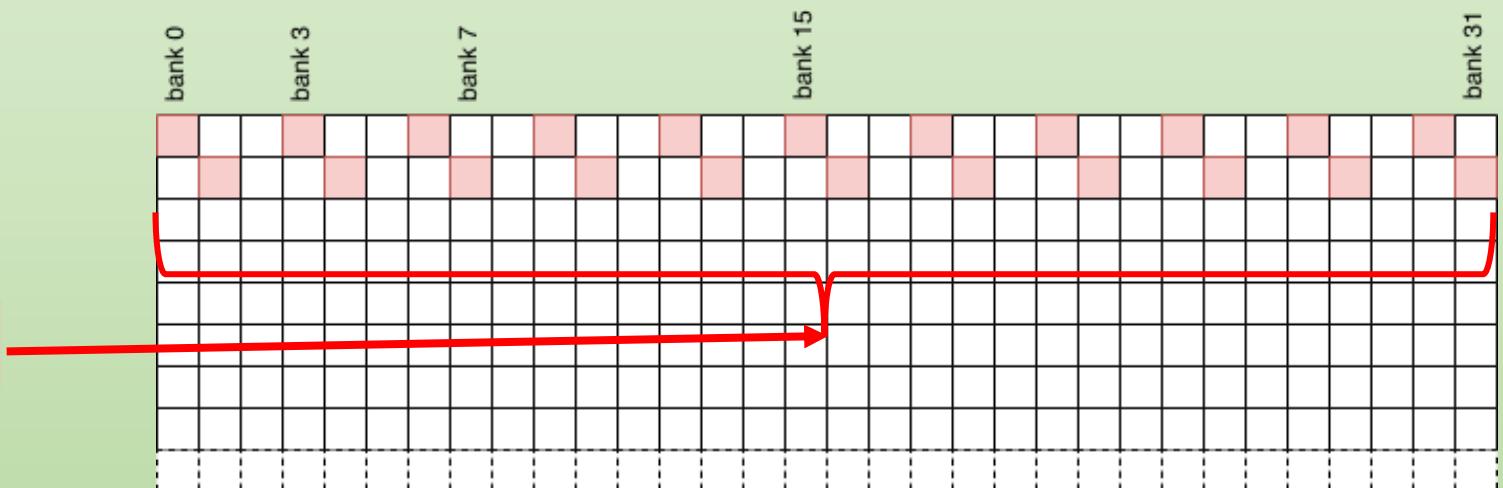
- ✓ **Conflitto doppio:** tutti i thread in un warp richiedono una word di indice doppio il threadIdx.x (strutture di 8 byte, per es i double o coppie di float)

```
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];
```



Configurazione SMEM / L1 cache

- ✓ Ogni SM ha **64 KB** di memoria **on-chip**, la shared memory e L1 cache condividono questa risorsa hardware
- ✓ CUDA fornisce due metodi per **configurare** la L1 cache e la shared memory:

```
cudaError_t cudaDeviceSetCacheConfig(cudaFuncCache cacheConfig);
```

- Dove l'argomento **cacheConfig** specifica come deve essere ripartita la memoria

<code>cudaFuncCachePreferNone</code> :	no preference(default)
<code>cudaFuncCachePreferShared</code> :	prefer 48KB shared memory and 16 KB L1 cache
<code>cudaFuncCachePreferL1</code> :	prefer 48KB L1 cache and 16 KB shared memory
<code>cudaFuncCachePreferEqual</code> :	prefer 32KB L1 cache and 32 KB shared memory

Criterio:

- ✓ Se si usa molta **shared memory** ovviamente si privilegia la dimensione a 48 KB della stessa a discapito di L1
- ✓ Da Kepler in poi, se si usano **molti registri**, lo spilling dei registri ricade sulla cache L1
- ✓ E' possibile determinare il numero esatto di registri usati dal kernel specificando l'opzione **-Xptxas -v** al momento di lanciare la compilazione con **nvcc** e se ne vengono usati di quelli disponibili si può decidere di aumentare la cache L1
- ✓ E' possibile anche configurare a runtime prima di lanciare il kernel con:

```
cudaError_t cudaFuncSetCacheConfig(const void* func, cudaFuncCache cacheConfig);
```

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, se vi sono molti thread in esecuzione, 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
- ✓ **Clock():** questa funzione può essere usata per misurare il corretto numero di cicli di clock nella GPU per avere tempi di ritardo effettivi della SMEM in presenza di conflitti

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[a][b] = 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[];
```

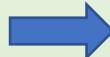
- ✓ Per allocare la **shared memory dinamicamente** la variabile occorre indicare un terzo argomento all'interno della chiamata del kernel:

```
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

CUDA zone...

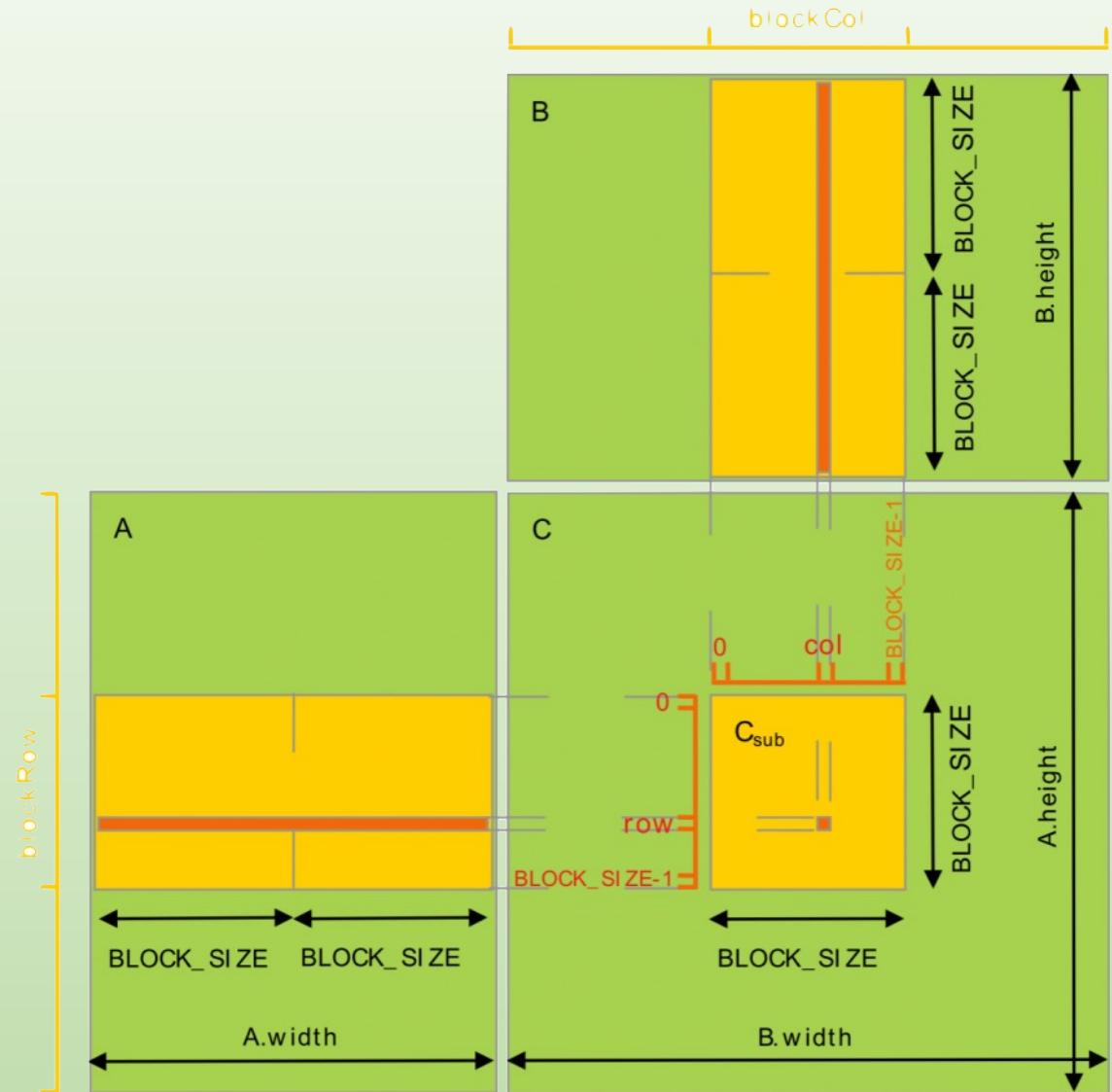
Prodotto e convoluzione matriciale con shared memory

Prodotto matriciale con SMEM

Kernel: ottimizzazione accessi

Prodotto matriciale con SMEM

- ✓ Nella versione senza memoria shared ogni thread è responsabile del calcolo di un elemento della matrice prodotto
- ✓ Come deve essere organizzato il lavoro del thread per sfruttare la memoria shared? Il thread calcola ancora un elemento di output come nel caso precedente?
- ✓ **PASSI:**
 1. Occorre caricare in memoria shared i dati relativi ad ogni blocco prima di svolgere le somme e i prodotti
 2. Ogni thread accumula i risultati di ognuno di questi prodotti (scandendo tutti i blocchi) in un registro
 3. Alla fine scrive su global memory



→ lab5

Prodotto matriciale con SMEM

Prodotto convolutivo con SMEM

Possibili approcci 1D e 2D

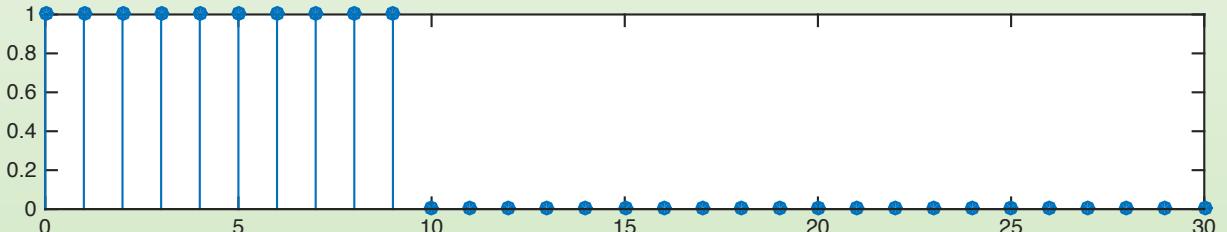
Esempio: somma di convoluzione

Somma di convoluzione tra segnali di durata finita \mathbf{x} e \mathbf{h} :

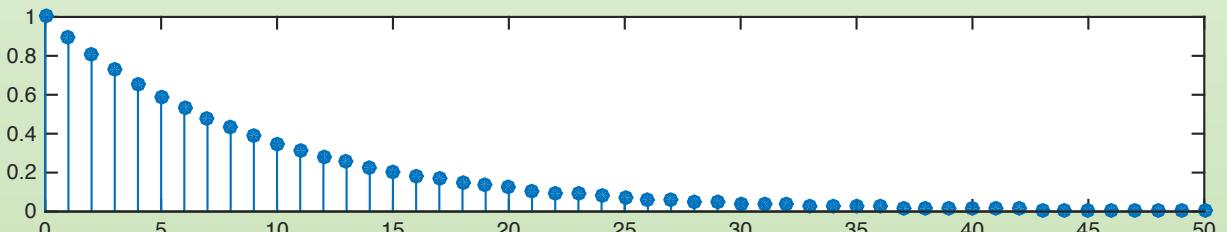
$$y(n) = h(n) * x(n) = \sum_{k=0}^{N-1} h(k)x(n-k)$$

Esempio:

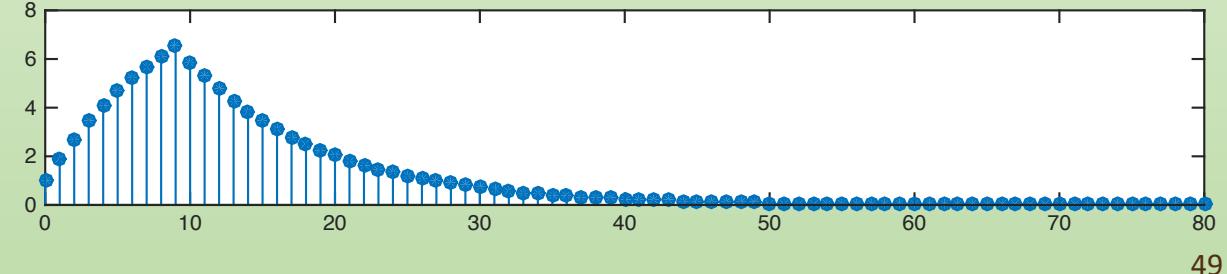
Gradino unitario: $x(n) = u(n) - u(n - 10)$



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



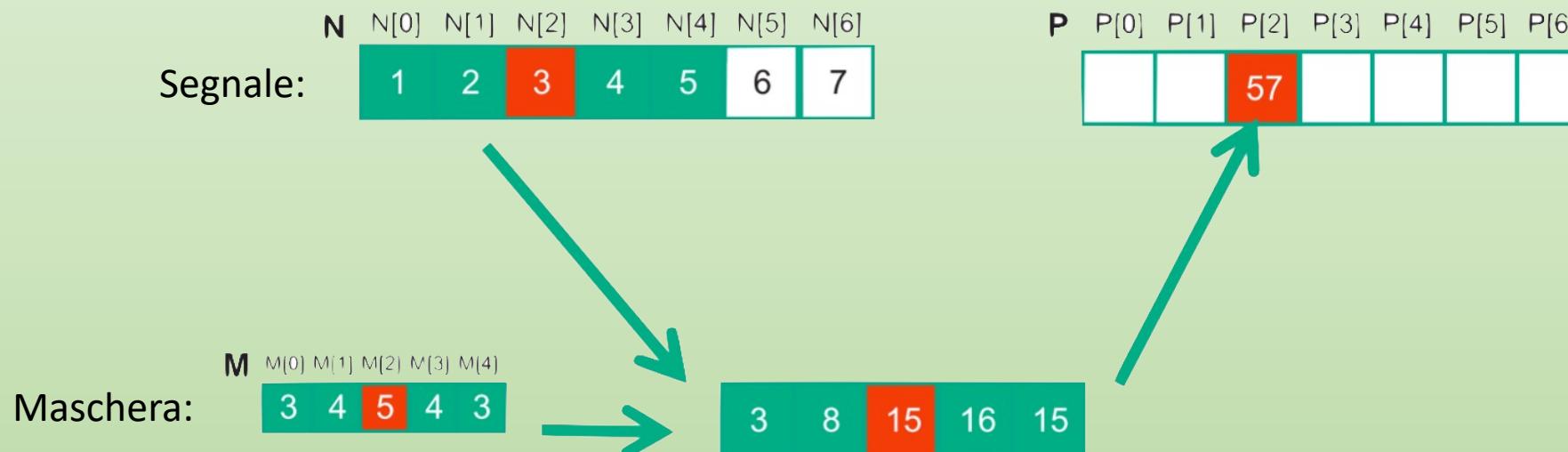
Risultato: $y(n) = h(n) * x(n)$



Implementazione

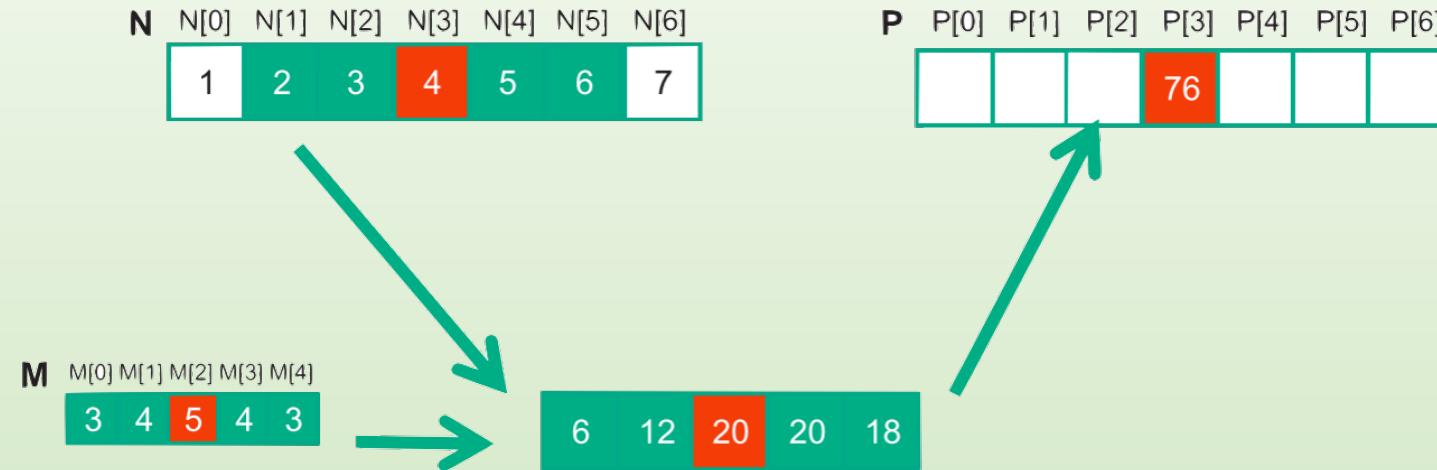
Sistema FIR: $y(n) = \sum_{k=0}^{N-1} h(k)x(n-k) = \sum_{k=0}^{N-1} b_k x(n-k)$

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

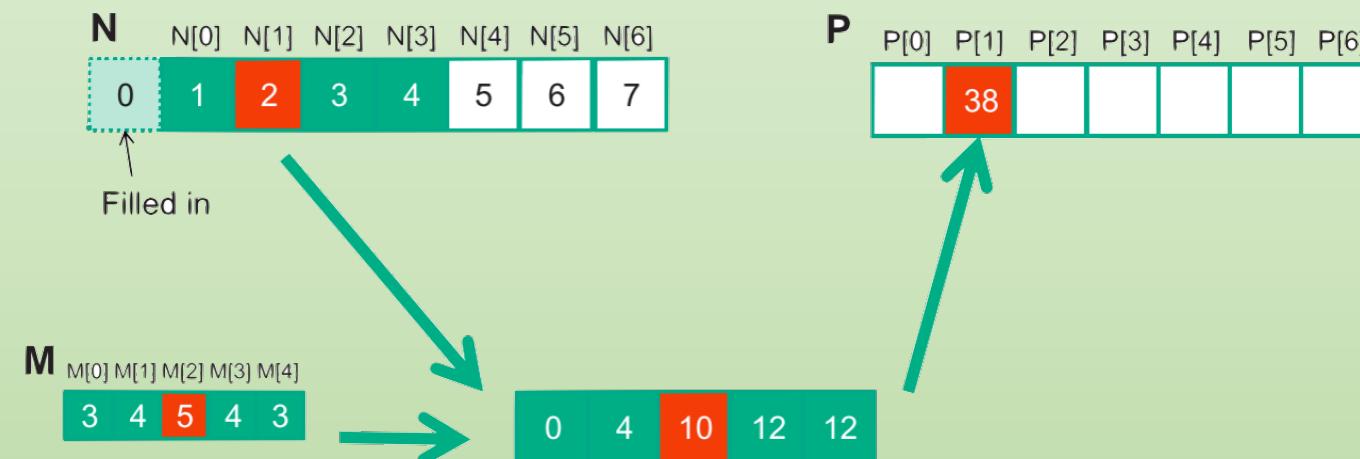


Il problema del bordo

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
	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

				3	2	1		

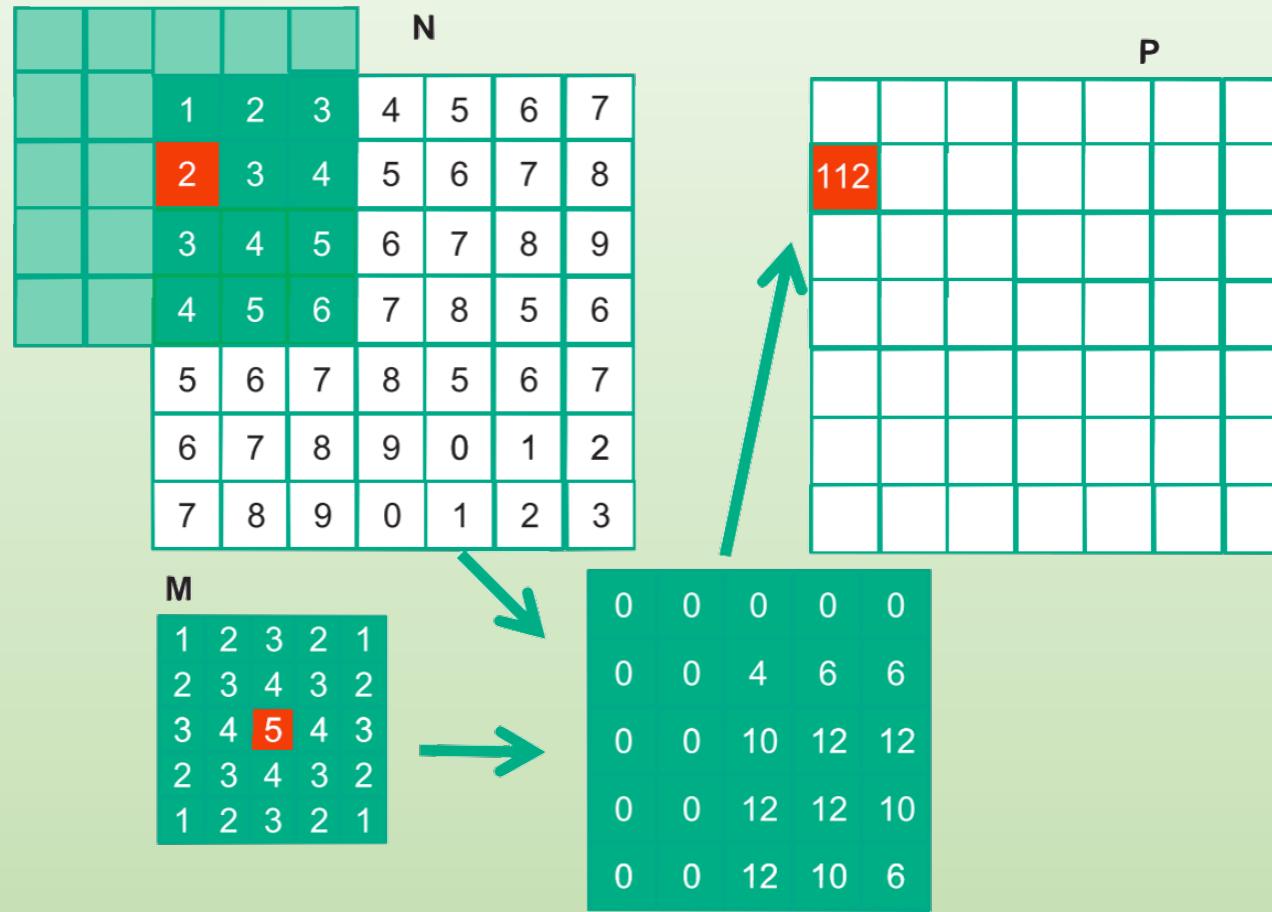
Maschera array 2D

M	1	2	3	2	1
	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



Convoluzione parallela

Problema **ideale** per parallel computing!

- ✓ Primo passo è determinare la **mappa** tra thread ed elementi di output
- ✓ Semplice approccio nei casi 1D e 2D è organizzare i thread in grid corrispondenti in modo tale che ogni thread calcoli un elemento della matrice di convoluzione (**esempio sotto**)
- ✓ **Problema:** non soddisfa per inefficienza negli accessi alla global memory!

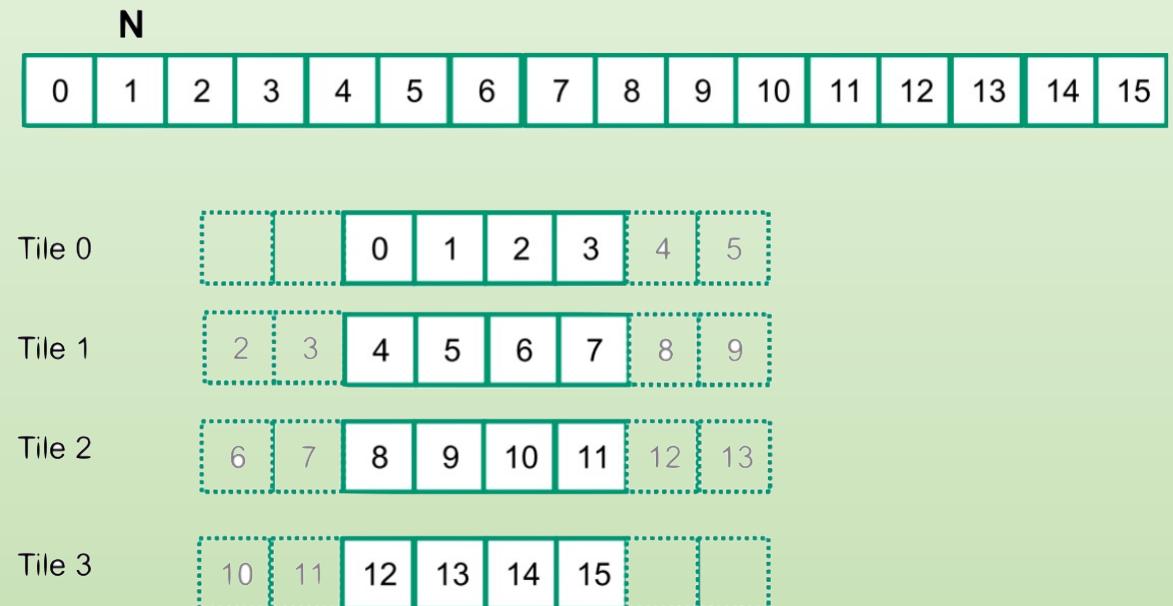
```
__global__ void convolution_1D_basic_kernel(float *N, float *M, float *P, int Mask_Width, int Width) {  
    int i = blockIdx.x*blockDim.x + threadIdx.x;  
    float Pvalue = 0;  
    int N_start_point = i - (Mask_Width/2);  
  
    for (int j = 0; j < Mask_Width; j++) {  
        if (N_start_point + j >= 0 && N_start_point + j < Width) {  
            Pvalue += N[N_start_point + j]*M[j];  
        }  
    }  
    P[i] = Pvalue;  
}
```

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**
- ✓ 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 + MAX_MASK_WIDTH - 1



```
__shared__ float share_mem[TILE_SIZE1 + MAX_MASK_WIDTH - 1];
```

→ lab5

Convoluzione con tiling

Riferimenti bibliografici

Testi generali:

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>
 4. Specifics techniques (page wiki): <https://en.wikipedia.org/wiki/CUDA> Version, features, and specifications