Introduction to Scientific Programming using GP-GPU (Graphics Processing Unit) and CUDA

Day 2

- Gerarchia di memoria in CUDA
  - *Global Memory*
    - *Sistema di caching*
    - *Accessi alla memoria globale*
  - *Shared Memory*
    - Prodotto Matrice Matrice con *Shared Memory*
  - *Constant Memory*
  - *Texture Memory*
  - *Registers and Local Memory*

La gerarchia di memoria in CUDA

I thread CUDA possono accedere:

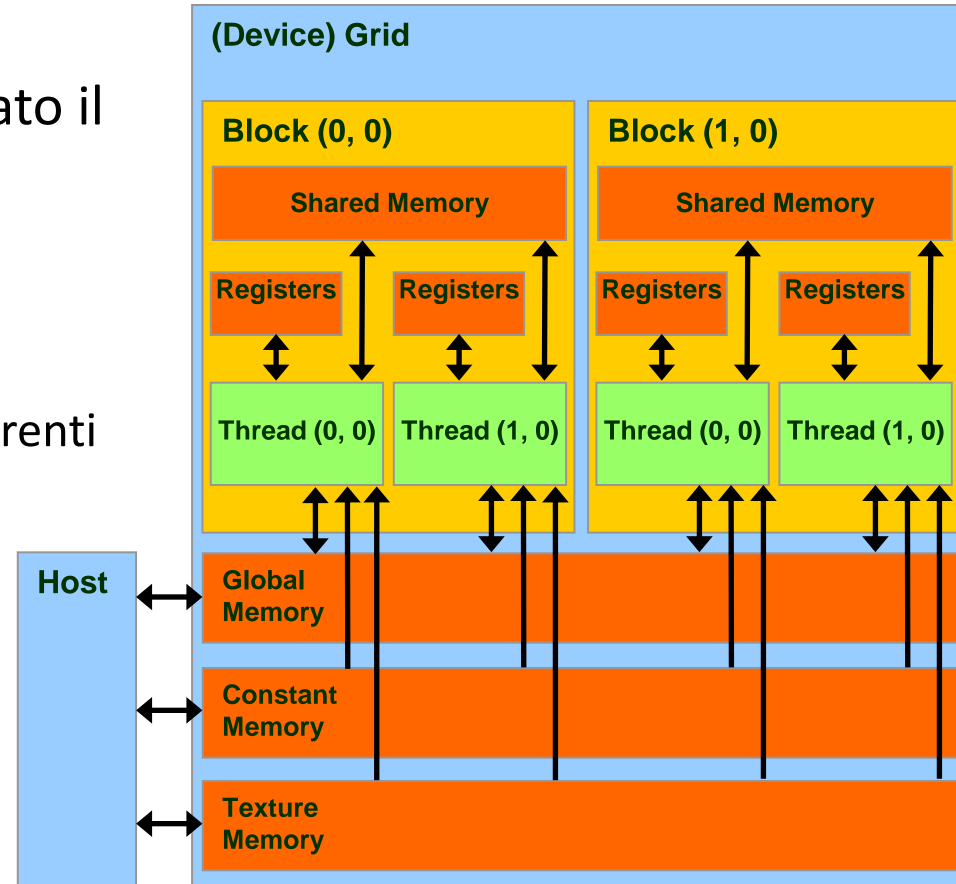
- alle risorse dello Streaming Multiprocessor a cui è stato assegnato il blocco:
  - **Registri**
  - **Memoria Shared**

  NB: Thread appartenenti a blocchi differenti non possono cooperare tra loro

- Tutti i thread dei blocchi possono accedere a:
  - **Memoria Globale**
  - **Memoria Costante** (sola lettura)
  - **Memoria Texture** (sola lettura)

  NB: la CPU può scrivere e inizializzare le memorie costant e la texture

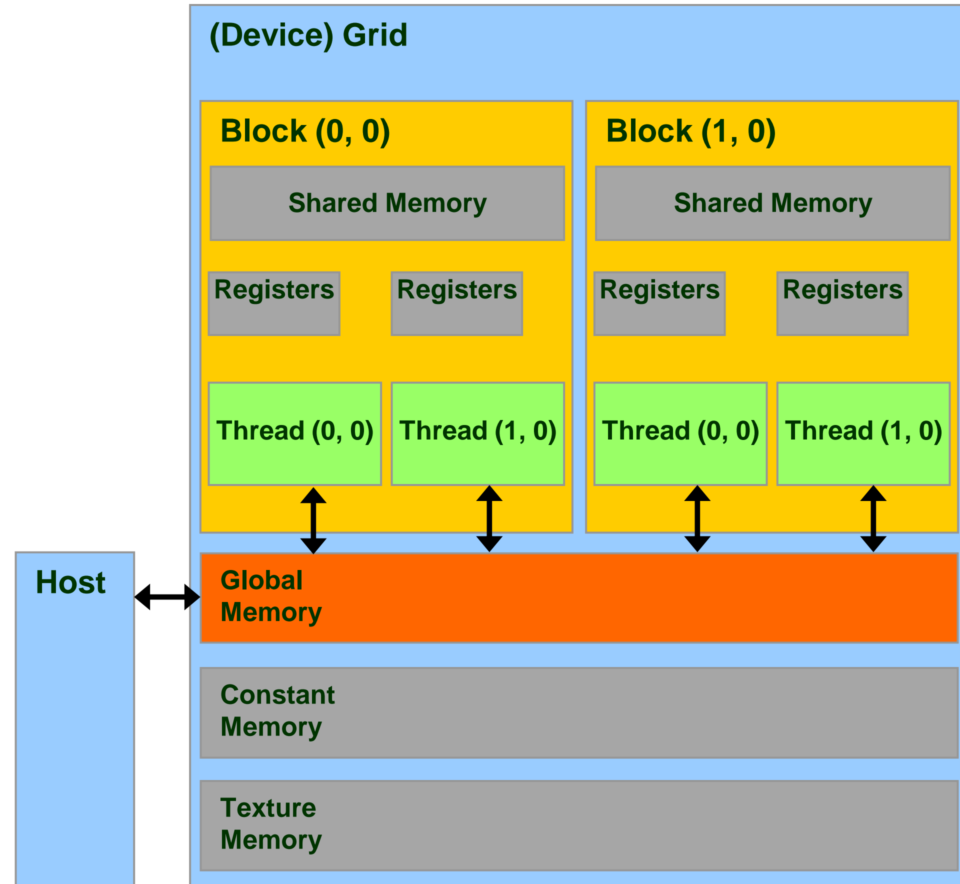
  NB: global, constant e texture sono persistenti



La Memoria Globale in CUDA

- La memoria globale (***Global Memory***) è la memoria più grande disponibile sul *device*
  - Gioca il ruolo che la RAM svolge per la CPU sull'*host*
  - Mantiene il suo stato e i dati tra il lancio di un kernel e l'altro
  - Accessibile in lettura e scrittura da tutti i thread della griglia
  - La sola possibilità di comunicazione in lettura e scrittura tra CPU e GPU
  - **Elevata latenza di accesso** (400-800 cicli di clock)
  - **Elevata bandwidth**

    Throughput: fino a 144-177 GB/s



CINECA SCAI
SuperComputing Applications and Innovation

Allocazione della Memoria Globale

- Per allocare una variabile nella memoria globale:

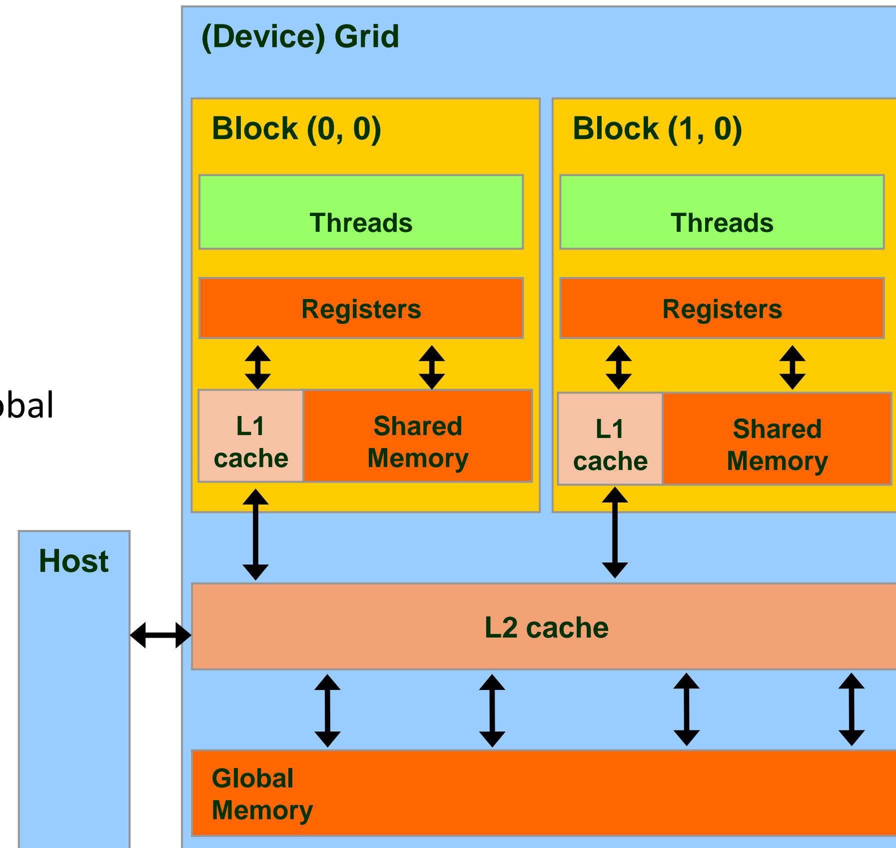
```
__device__  type  variable_name; // statica

oppure

type  *pointer_to_variable;  // dinamica
cudaMalloc((void **) &pointer_to_variable, size);
cudaFree(pointer_to_variable);
```

```
type, device :: variable_name

oppure

type, device, allocatable :: variable_name
allocate(variable_name, size)
deallocate(variable_name)
```

- risiede nello spazio della memoria globale
- ha un tempo di vita pari a quello dell'applicazione
- è accessibile da tutti i thread di una griglia e dall'host

Gerarchia di Cache della Memoria Globale

- Dall'architettura Fermi è stata introdotta una gerarchia di cache tra la memoria globale e lo Streaming Multiprocessor
- 2 Livelli di cache:
  - **L2** : globale a tutti gli SM
    - **Fermi** : [**768 KB**] / **Kepler** : [**1536 KB**]
    - 25 % di latenza 25% in meno rispetto alla Global Memory
    - NB : Tutti gli accessi alla memoria globale passano per la cache L2
    - Incluse le copie da/su Host!
  - **L1** : private ai singoli SM
    - [**16/48 KB**] Configurabile
    - L1 + Shared Memory = 64 KB
    - **Kepler** : configurabile anche a **32 KB**

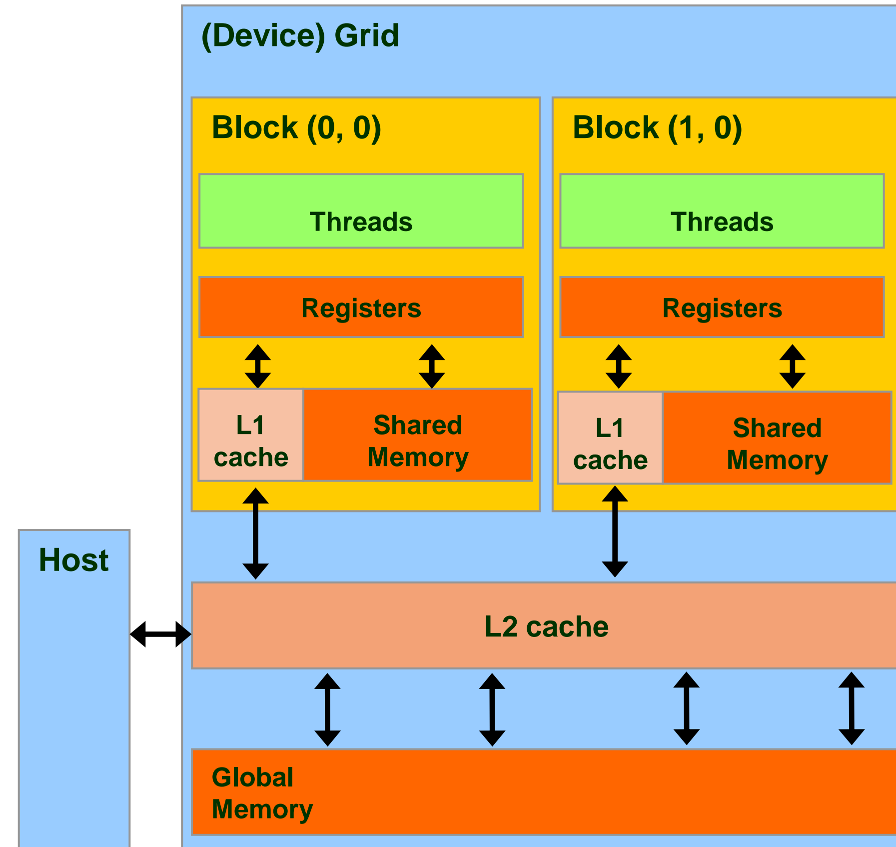



```
cudaFuncSetCacheConfig(kernel1, cudaFuncCachePreferL1);     // 48KB L1 / 16KB ShMem
cudaFuncSetCacheConfig(kernel2, cudaFuncCachePreferShared); // 16KB L1 / 48KB ShMem
```

Gerarchia di Cache della Memoria Globale

Due tipologie di ***load***:

- **Caching**
  - Il dato viene prima cercato nella L1, poi nella L2, poi in Global Memory (GMEM)
  - La dimensione delle linee di cache è di **128-byte**
  - Default mode
- **Non-caching**
  - Disabilita la cache L1
  - Il dato viene prima cercato in L2, poi in GMEM
  - La dimensione delle linee di cache è di **32-bytes**
  - Selezionabile a *compile time* con l'opzione di compilazione: `-Xptxas -dlcm=cg`



Una tipologia di ***store:***

- Viene invalidato il dato in L1, e aggiornato direttamente la linea di cache in L2

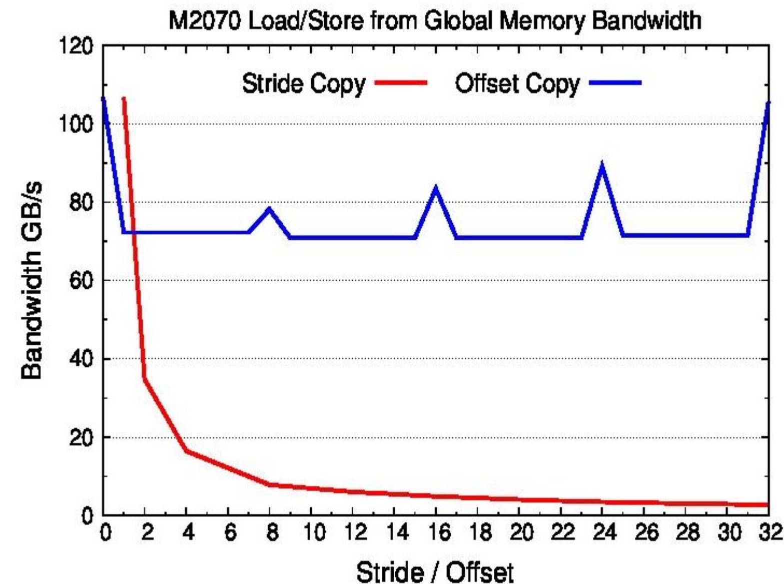
Load/Store in Memoria Globale

```
// strided data copy
__global__ void strideCopy (float *odata, float* idata, int stride) {
    int xid = (blockIdx.x*blockDim.x + threadIdx.x) * stride;
    odata[xid] = idata[xid];
}
```

```
// offset data copy
__global__ void offsetCopy(float *odata, float* idata, int offset) {
    int xid = blockIdx.x * blockDim.x + threadIdx.x + offset;
    odata[xid] = idata[xid];
}
```

Copia con stride	
Stride	**Bandwidth GB/s**
1	106.6
2	34.8
8	7.9
16	4.9
32	2.7

Copia con offset	
Offset	**Bandwidth GB/s**
0	106.6
1	72.2
8	78.2
16	83.4
32	105.7



M2070; Num. Elementi = 16776960; Num. Blocchi = 65535; Dim. Blocco = 256

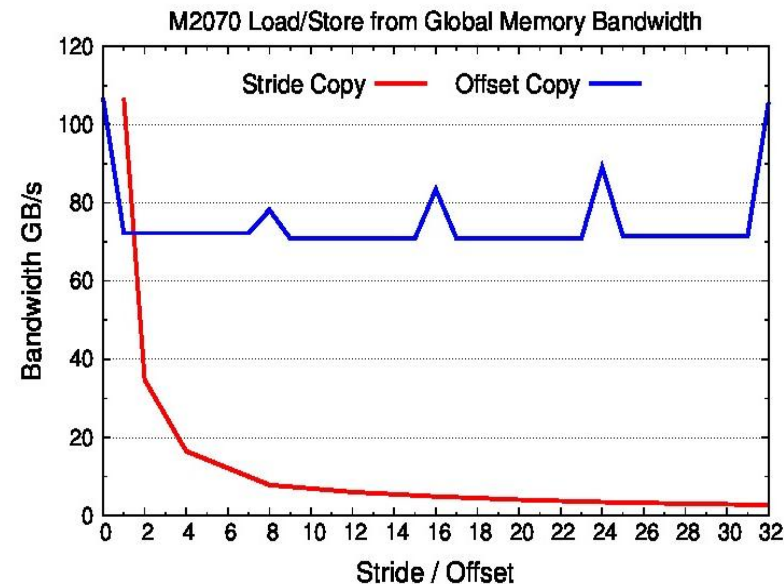
Load/Store in Memoria Globale

```
// strided data copy
__global__ void strideCopy (float *odata, float* idata, int stride) {
    int xid = (blockIdx.x*blockDim.x + threadIdx.x) * stride;
    odata[xid] = idata[xid];
}
```

```
// offset data copy
__global__ void offsetCopy(float *odata, float* idata, int offset) {
    int xid = blockIdx.x * blockDim.x + threadIdx.x + offset;
    odata[xid] = idata[xid];
}
```

Copia con stride	
Stride	**Bandwidth GB/s**
1	106.6
2	34.8
8	7.9
16	4.9
32	2.7

Copia con offset	
Offset	**Bandwidth GB/s**
0	106.6
1	72.2
8	78.2
16	83.4
32	105.7



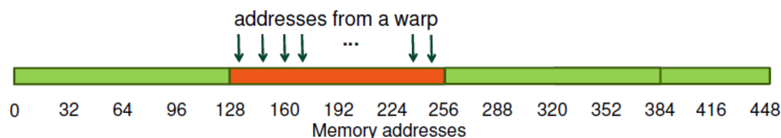
CINECA SCAI SuperComputing Applications and Innovation

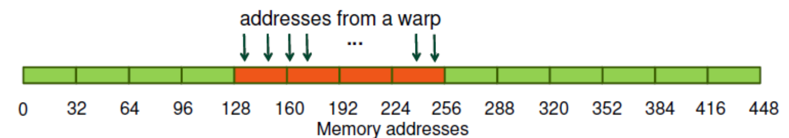
M2070; Num. Elementi = 16776960; Num. Blocchi = 65535; Dim. Blocco = 256

Operazioni di load in Memoria Globale

- Tutte le operazioni di load/store dalla memoria globale vengono istanziate contemporaneamente da tutti i *thread* del *warp*
  1. i *thread* del *warp* calcolano gli indirizzi di memoria a cui accedere
  2. le unità di *load/store* determinano in quali *segmenti* della memoria risiedono i dati richiesti
  3. le unità di *load/store* avviano la richiesta dei segmenti

Warp richede 32 word di 4-byte consecutive e allineate (Richiesta di 128 bytes)	
Caching Load	**Non-caching Load**
Gli indirizzi cadono in una singola linea di cache	Gli indirizzi cadono in 4 segmenti di linee di cache
128 bytes vengono spostati sul bus	128 bytes vengono spostati sul bus
Utilizzo bus: **100%**	Utilizzo bus: **100%**



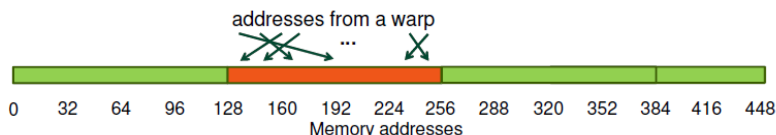


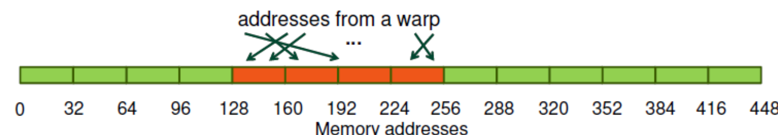
CINECA SCAI
SuperComputing Applications and Innovation

Operazioni di load in Memoria Globale

Warp richiede 32 word di 4-byte permutate e allineate (Richiesta di 128 bytes)

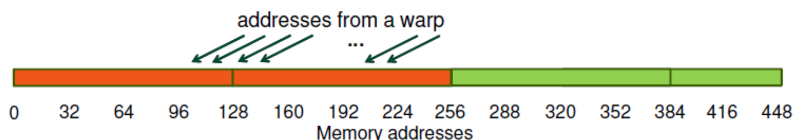
Caching Load	Non-caching Load
Gli indirizzi cadono in una singola linea di cache	Gli indirizzi cadono in 4 segmenti di linee di cache
128 bytes vengono spostati sul bus	128 bytes vengono spostati sul bus
Utilizzo bus: **100%**	Utilizzo bus: **100%**

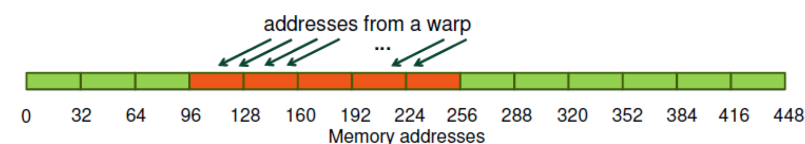




Warp richiede 32 word di 4-bytes consecutive non allineate (Richiesta di 128 bytes)

Caching Load	Non-caching Load
Gli indirizzi cadono in 2 linee di cache	Gli indirizzi cadono al massimo in 5 linee di cache
256 bytes vengono spostati sul bus	160 bytes vengono spostati sul bus
Utilizzo bus: **50%**	Utilizzo bus: **al minimo 80%**

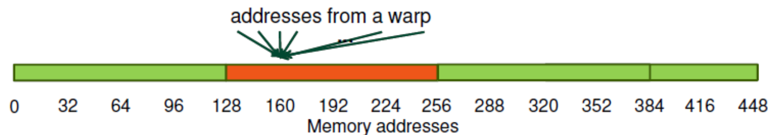


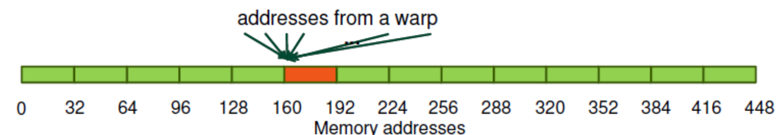


Operazioni di load in Memoria Globale

Tutti i threads di un warp richiedono la stessa word di 4-byte (Richiesta di 4 bytes)

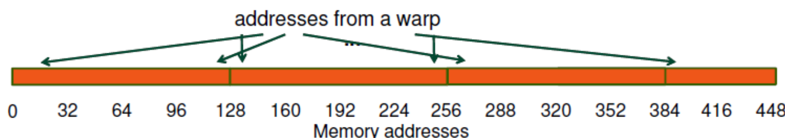
Caching Load	Non-caching Load
Gli indirizzi cadono all’interno di una linea di cache	Gli indirizzi cadono all’interno di una linea di cache
128 bytes vengono spostati sul bus	32 bytes vengono spostati sul bus
Utilizzo bus: **3.125%**	Utilizzo bus: **12.5%**

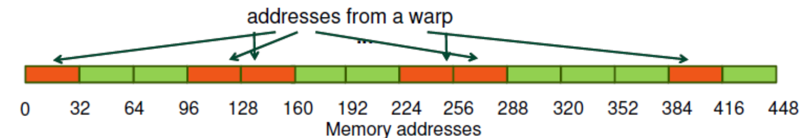




Warp richiede 32 word di 4-bytes sparse (Richiesta di 128 bytes)

Caching Load	Non-caching Load
Gli indirizzi cadono all’interno di N linee di cache	Gli indirizzi ricadono all’interno di N linee di cache
N*128 bytes vengono spostati sul bus	N*32 bytes vengono spostati sul bus
Utilizzo bus: **128 / (N*128)**	Utilizzo bus: **128 / (N*32)**





Garantire l'allineamento dei dati in memoria

- L'allineamento dei dati in memoria è fondamentale per avere accessi *coalesced* e limitare il numero di segmenti coinvolti in una transazione
  - **`cudaMalloc()`** garantisce l'allineamento del primo elemento nella memoria globale, utile quindi per array 1D
  - **`cudaMallocPitch()`** è ideale per allocare array 2D
    - gli elementi sono paddati e allineati in memoria
    - restituisce un intero (`pitch`) per accedere correttamente ai dati

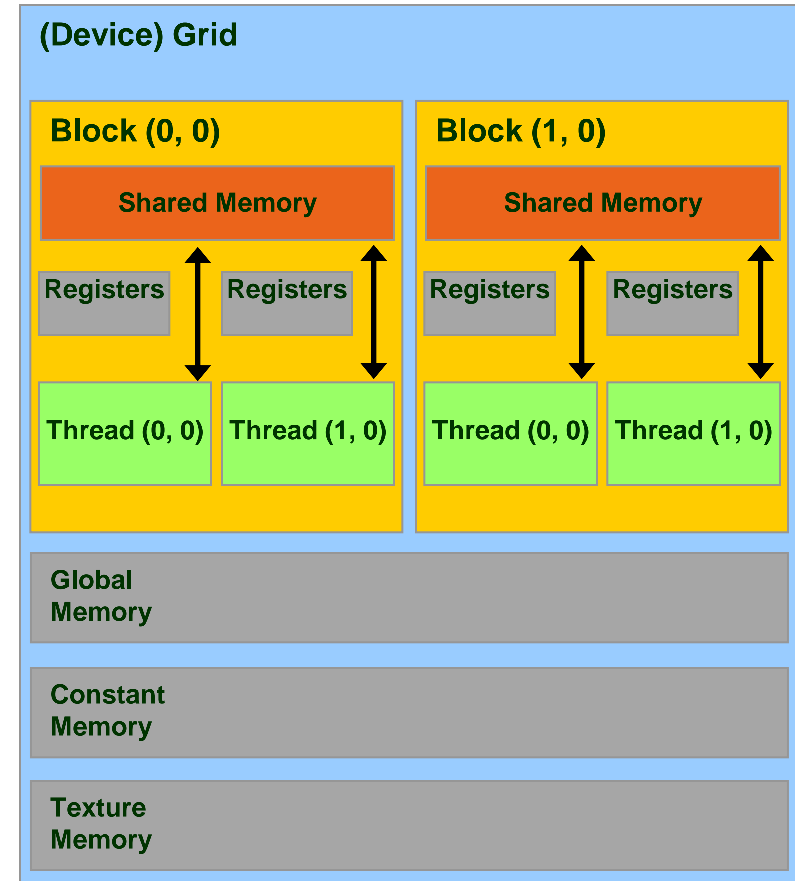
```
// host code
int width = 64, heigth = 64;
float *devPtr;
int pitch;
cudaMallocPitch(&devPtr, &pitch, width * sizeof(float), height);

// device code
__global__  myKernel(float *devPtr, int pitch, int width, int height)
{
  for (int r = 0; r < height; r++) {
    float *row = devPtr + r * pitch;
    for (int c = 0; c < width; c++)
      float element = row[c];
  }
...
}
```

La *Shared Memory* in CUDA

- La memoria condivisa (***Shared Memory***) è una memoria veloce residente su ciascun Streaming Multiprocessor
  - Accessibile in lettura e scrittura dai soli thread del blocco
  - Non mantiene il suo stato tra il lancio di un kernel e l'altro
  - **Bassa Latenza**: 2 cicli di clock
  - Throughput: 4 bytes per banco ogni 2 cicli
  - Default : **48 KB**

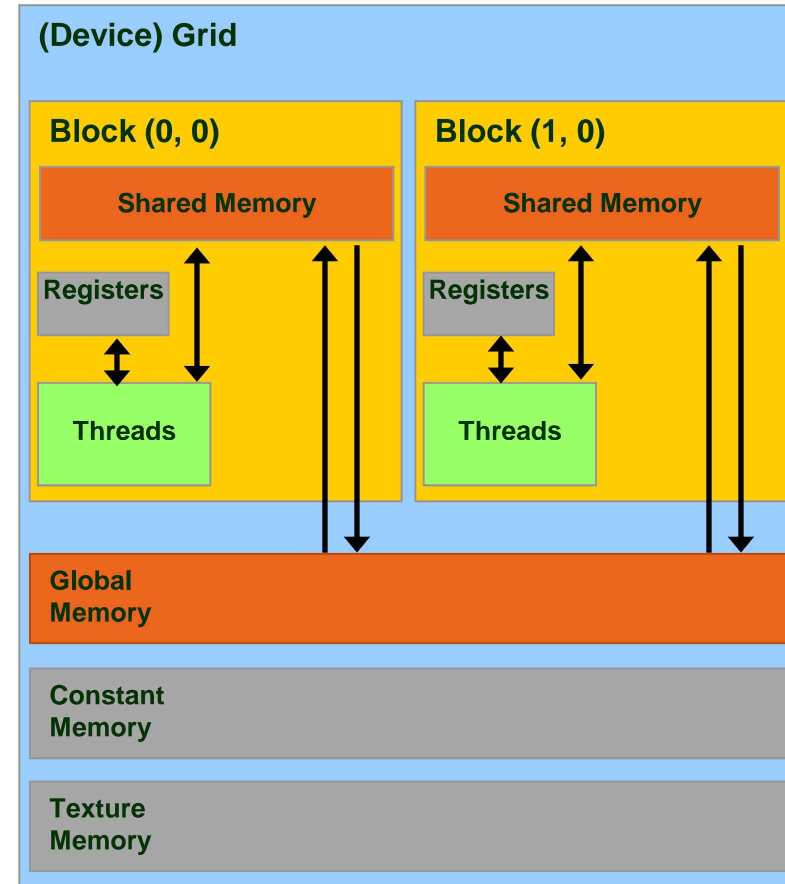
    (Configurabile : 16/48 KB)

    Kepler : anche 32 KB



Tipico utilizzo della memoria shared

- La Memoria Shared è spesso utilizzata :
  - Come una cache gestita dal programmatore per limitare accessi ridondanti alla memoria globale;
  - Per migliorare i pattern di accesso alla memoria globale;
  - Comunicazioni tra thread all'interno di un blocco.
- Modalità di utilizzo:
  - Si caricano i dati nella memoria shared
  - Si sincronizza (se necessario)
  - Si opera sui dati nella memoria shared
  - Si sincronizza (se necessario)
  - Si scrivono i risultati nella memoria globale

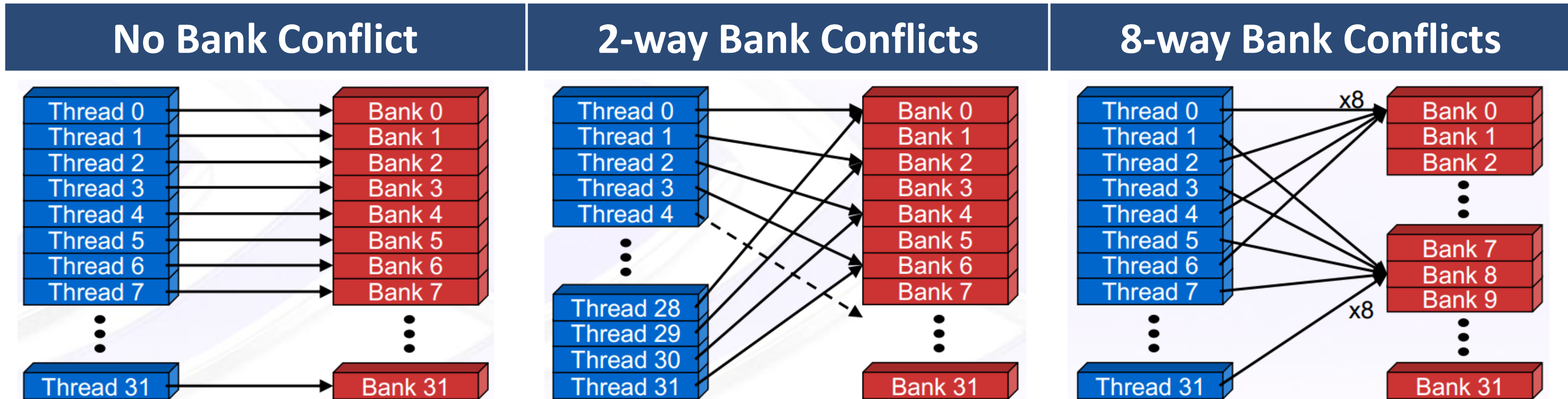


Sincronizzazione dei thread

- `__syncthreads()` | `call syncthreads()` attende che tutti i thread del blocco raggiungano il medesimo punto di chiamata
  - utilizzato per coordinare comunicazioni tra thread
  - è consentito in costrutti condizionali *solo se* valutata allo stesso modo da tutto il blocco
  - "... otherwise the code execution is likely to hang or produce unintended side effects"

La Memoria Shared

- La *shared memory* è organizzata in 32 banchi da 4-byte di ampiezza ciascuno
  - I dati vengono distribuiti ciclicamente su banchi successivi ogni 4-byte
  - Gli accessi alla shared memory avvengono per warp
  - **Multicast** : se n thread del warp accedono allo stesso elemento, l'accesso è eseguito in una singola transazione
  - **Broadcast** : se tutti i thread del warp accedono allo stesso elemento, l'accesso è eseguito in una singola transizione
  - **Bank Conflict** : se due o più thread differenti (dello stesso *warp*) tentano di accedere a dati differenti, residenti sullo stesso banco
    - Ogni conflitto viene servito e risolto serialmente



Allocazione della Memoria Shared

```
// statically inside the kernel
__global__  myKernelOnGPU (...) {
  ...
  __shared__  type  shmem[MEMSZ];
  ...
}

oppure

// dynamically sized
extern  __shared__  type  *dynshmem;

__global__  myKernelOnGPU (...) {
  ...
  dynshmem[i] = ... ;
  ...
}

void myHostFunction() {
  ...
  myKernelOnGPU<<<gs,bs,MEMSZ>>>();
}
```

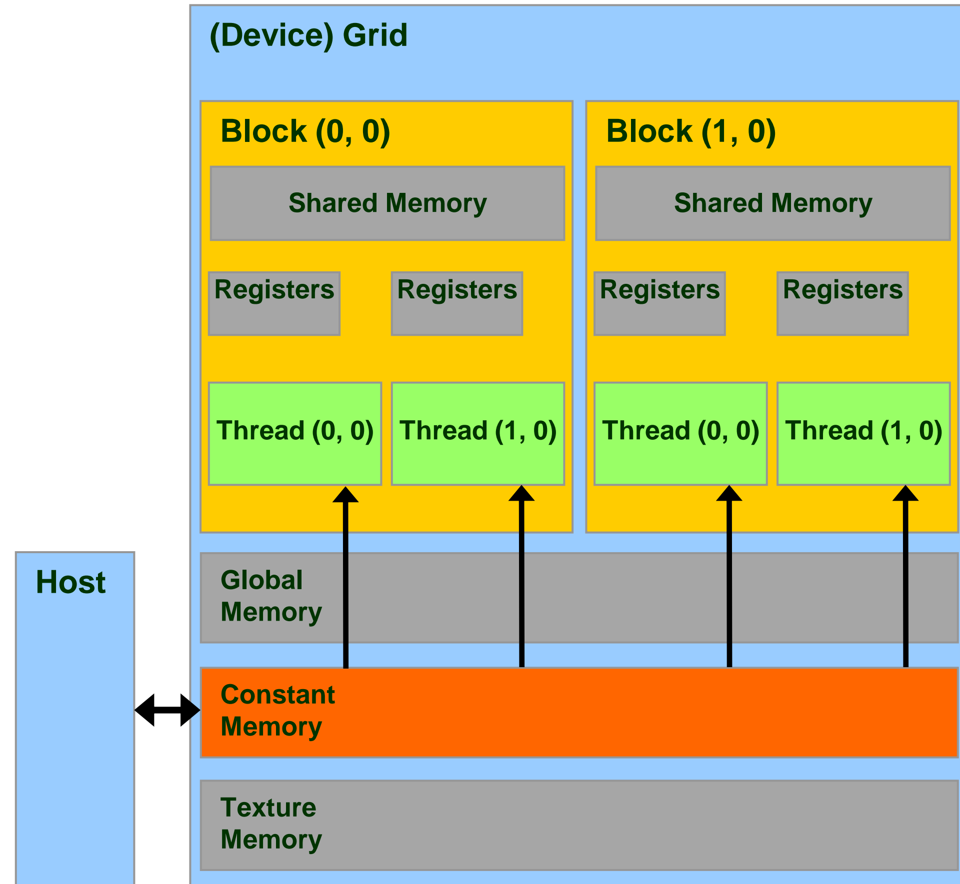
```
! statically inside the kernel
attribute(global)
  subroutine myKernel(...)
  ...
  type, shared:: variable_name
  ...
end subroutine

oppure

! dynamically sized
type, shared:: dynshmem(*)

attribute(global)
  subroutine myKernel(...)
  ...
  dynshmem(i) = ...
  ...
end subroutine
```

- tempo di vita pari a quello del blocco
- accessibile solo dai thread dello stesso blocco

CINECA SCAI SuperComputing Applications and Innovation

La *Constant Memory* in CUDA

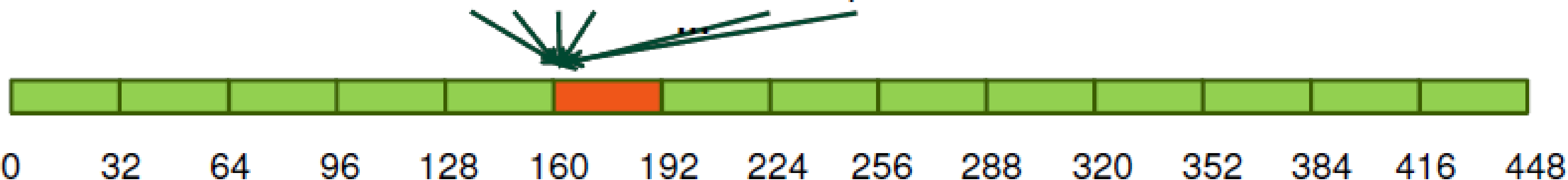
- La memoria costante (***Constant Memory***) è ideale per ospitare coefficienti e altri dati a cui si accede in modo uniforme e in sola lettura
  - i dati risiedono nella memoria globale, ma vi si accede in lettura tramite una *constant-cache* dedicata
  - da utilizzare quando tutti i *thread* di un *warp* accedono allo stesso dato, altrimenti gli accessi sono serializzati
  - In sola lettura (**Read-Only**), viene inizializzata dall'host
  - Dimensione : **64 KB**
  - Throughput: 32 bits per warp ogni 2 clocks



La *Constant Memory* in CUDA

Supponiamo che un kernel venga eseguito con 320 warp per SM e che tutti I thread accedano al medesimo dato

- usando la memoria globale:
  - tutti i *warp* accedono alla memoria globale
  - si genera traffico sul BUS per 320 volte per il numero di accessi
  - (nelle architetture FERMI, la cache L2 allevia questo costo)
- usando la constant memory:
  - il primo *warp* accede alla memoria globale
  - porta il dato in *constant-cache*
  - tutti gli altri *warp* trovano il dato già nella constant-cache (nessun traffico sul BUS)



Allocazione della *Constant Memory*

- Per allocare una variabile nella memoria globale:

```
__constant__  type  variable_name; // statica

cudaMemcpyToSymbol(const_mem, &host_src, sizeof(type), cudaMemcpyHostToDevice);

// attenzione
// non può essere allocata dinamicamente
// attenzione
```

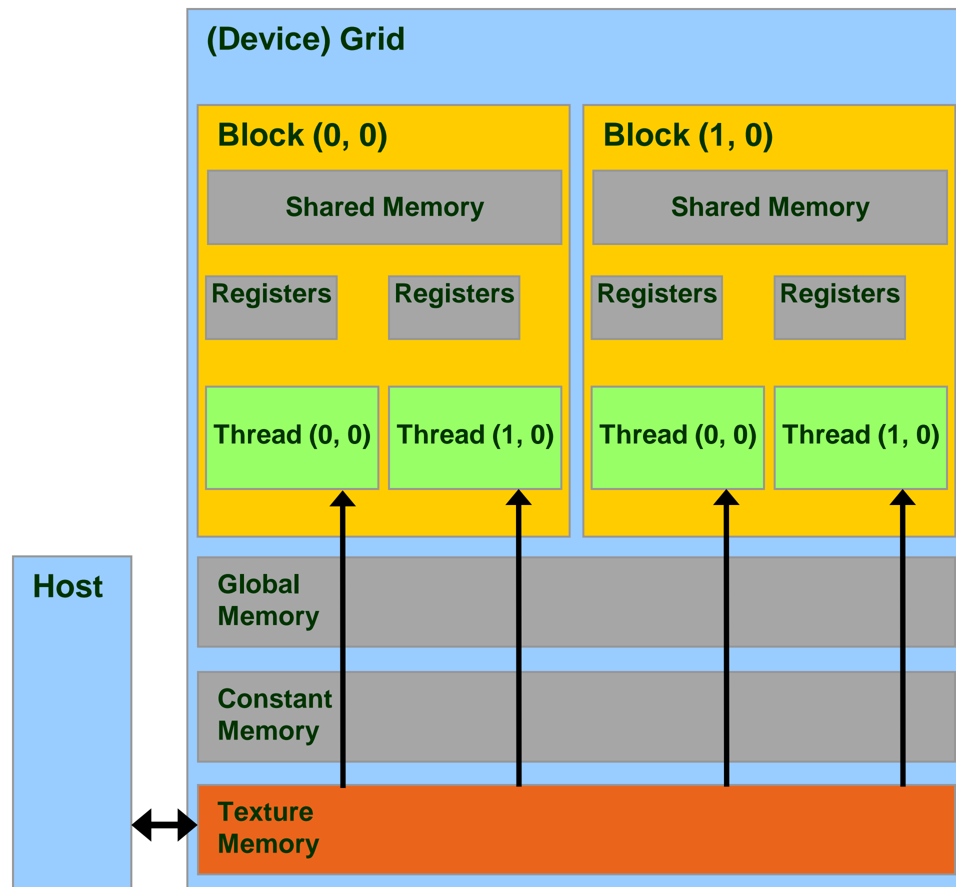
```
type, constant :: variable_name

! attenzione
! non può essere allocatable
! attenzione
```

- risiede nello spazio della memoria costante
- ha un tempo di vita pari a quello dell'applicazione
- è accessibile da tutti i thread di una griglia e dall'host

CINECA SCAI SuperComputing Applications and Innovation

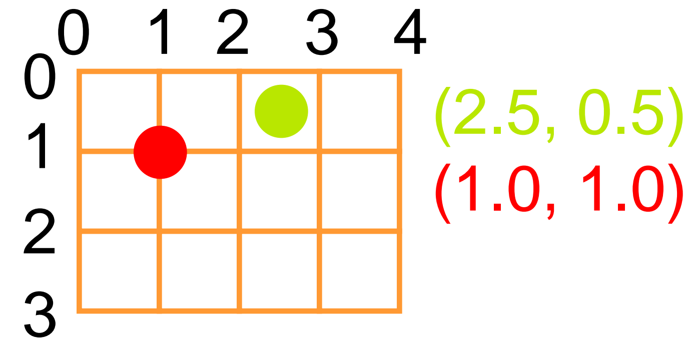
La Memoria Texture in CUDA

- i dati in memoria globale possono essere acceduti **in sola lettura** tramite funzioni **fetch di texture**
- accesso tramite cache (di texture) anche con c.c. inferiori alla 2.x
- calcolo degli indirizzi più efficiente, svolto con hardware apposito
- hardware dedicato a funzionalità speciali:
  - gestione indici fuori dai limiti
  - interpolazione
  - conversione di tipi



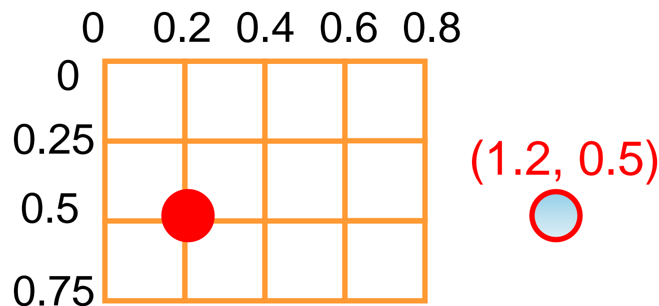
Indirizzamento memoria Texture

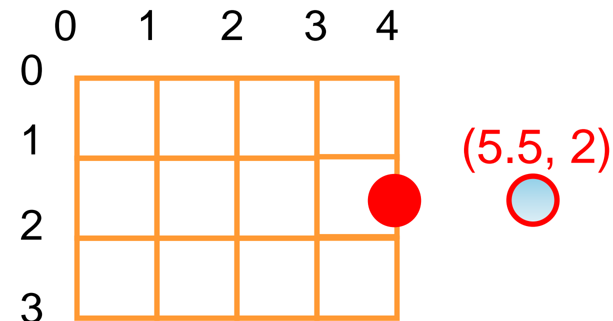
- intero, in 1D: [0,N-1]
- normalizzato, in 1D: [0,1-1/N]
- possibile interpolazione:
  - floor, lineare, bilineare
  - pesi a 9 bit
- addressing mode



Wrap: Le coordinate fuori bordo sono calcolate con l'aritmetica modulare (solo con indirizzamento normalizzato)

Clamp: Le coordinate fuori bordo sono sostituite con il valore più vicino





Passi necessari per usare la Texture

CPU

- Allocare la memoria (globale, pitch linear o cudaArray)

```
cudaMalloc(&d_a, memsize);
```

- Creare a file scope un oggetto “texture reference” (non passabile per argomento):

```
texture<datatype, dim> d_a_texRef;
```

`datatype` non può essere double; `dim` può essere 1, 2 o 3

- Creare un “channel descriptor” per descrivere il formato del valore di ritorno:

```
cudaChannelFormatDesc d_a_desc = cudaCreateChannelDesc<datatype>();
```

- Associare la texture reference alla memoria

```
cudaBindTexture(0, d_a_texRef, d_a, d_a_desc);
```

- Alla fine: disassociare la texture reference (per liberare le risorse):

```
cudaUnbindTexture(d_a_texRef);
```

GPU

- effettuare i caricamenti utilizzando la “texture reference”:
  - `tex1Dfetch(d_a_texRef, indirizzo)`

  per texture in memoria globale lineare, funzionalità limitate (no filtraggio, addressing) , indirizzo intero
  - `tex1d(), tex2D(), tex3D()` - per le pitched linear texture e cudaArray:

Esempio di utilizzo della texture

```
__global__ void shiftCopy(int N, int shift, float *odata, float *idata)
{
  int xid = blockIdx.x * blockDim.x  +  threadIdx.x;
  odata[xid] = idata[xid+shift];
}

texture<float, 1>  texRef;  // CREO OGGETTO TEXTURE

__global__ void textureShiftCopy(int N, int shift, float *odata)
{
  int xid = blockIdx.x * blockDim.x  +  threadIdx.x;
  odata[xid] = tex1Dfetch(texRef, xid+shift);       //  TEXTURE FETCHING
}


...

ShiftCopy<<<nBlocks, NUM_THREADS>>>(N, shift, d_out, d_inp);

cudaChannelFormatDesc d_a_desc = cudaCreateChannelDesc<float>(); // CREO DESC
cudaBindTexture(0, texRef, d_a, d_a_desc);  // BIND TEXTURE MEMORY
textureShiftCopy<<<nBlocks, NUM_THREADS>>>(N, shift, d_out);
```

Memoria Texture in Kepler: aka *Read-only Cache*

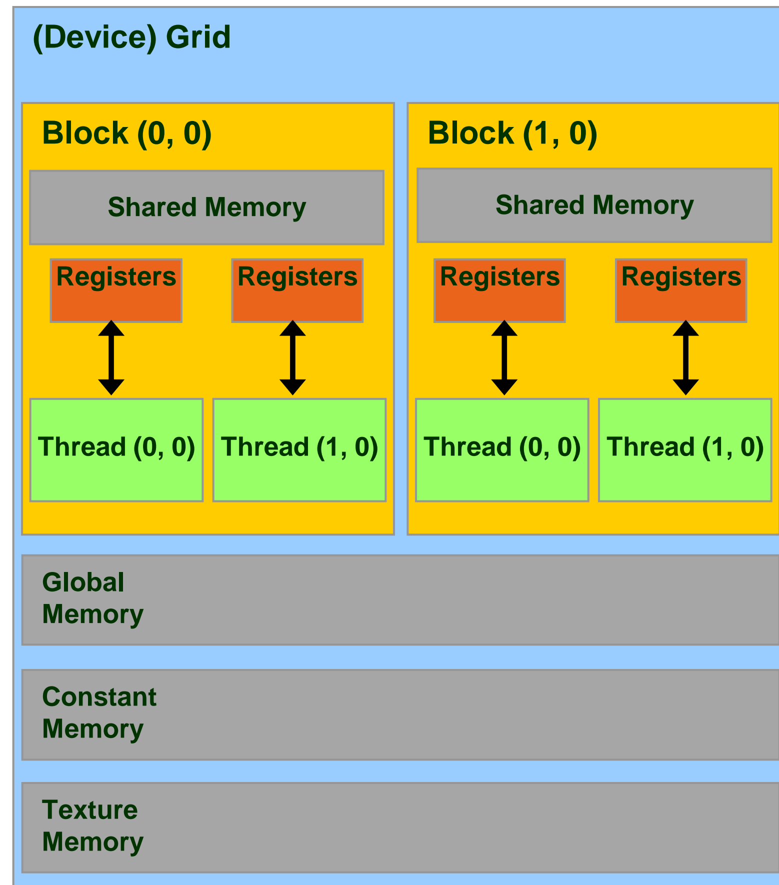
- L'architettura Kepler (cc 3.5) introduce la possibilità di leggere i dati dalla memoria globale sfruttando la *texture cache* :
  - Senza utilizzare il *binding*
  - Senza limitazioni sul numero di texture

```
__global__ void kernel_copy (float *odata, float *idata) {
   int index = blockIdx.x * blockDim.x + threadIdx.x;
   odata[index] = __ldg(idata[index]);
}
```

```
__global__ void kernel_copy (float *odata, const __restrict__
 float *idata) {
   int index = blockIdx.x * blockDim.x + threadIdx.x;
   odata[index] = idata[index];
}
```

I registri in CUDA

- I registri in CUDA sono utilizzati per le variabili locali ad ogni thread
- Tempo di vita di un thread
- Latenza nulla
- **Fermi** : 63 Registri per thread / 32 KB
- **Kepler** : 255 Registri per thread / 64 KB
- Attenzione al *Register pressure*. Uno dei maggiori fattori limitanti
  - Minore è il numero di registri che un kernel usa e maggiore è il numero di thread e di blocchi di thread che possono risiedere per SM.
  - Il numero di registri può essere limitato a *compile time:* `--maxregcount max_registers`
  - Il numero di blocchi attivi può essere forzato tramite il qualificatore `__launch_bounds__`



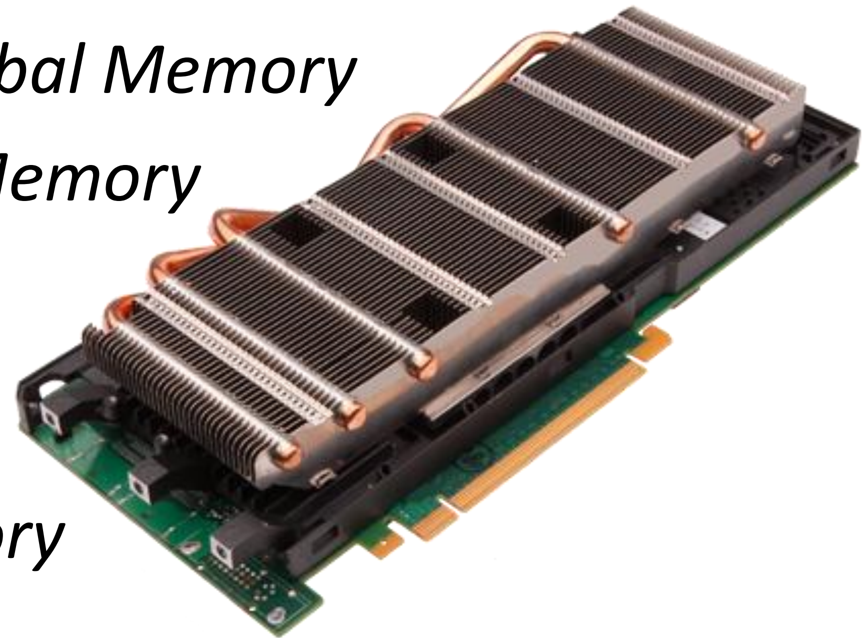

```
__global__ void
__launch_bounds__(maxThreadsPerBlock,
                  minBlocksPerMultiprocessor)
my_kernel( … ) { … }
```

La *Local Memory* in CUDA

- Memoria privata ad ogni thread
- Non esiste fisicamente ma è un area della memoria globale utilizzata per lo *spilling* quando sono terminate le risorse a disposizione dello Streaming Multiprocessor
- L1 *Caching* solo in scrittura
- Per ottenere informazioni sull'utilizzo della memoria locale, costante, shared e sul numero di registri è possibile compilare con l'opzione `--ptxas-options=-v`

```
$ nvcc -arch=sm_20 -ptxas-options=-v my_kernel.cu
...
ptxas info : Used 34 registers, 60+56 bytes lmem, 44+40 bytes
smem, 20 bytes cmem[1], 12 bytes cmem[14]
...
```

- Prodotto Matrice Matrice
  - Limiti all'approccio con *Global Memory*
  - Come sfruttare la *Shared Memory*
  - Cenni di implementazione
- Trasposizione di Matrice
  - Versione con *Shared Memory*
  - *Bank Conflict*

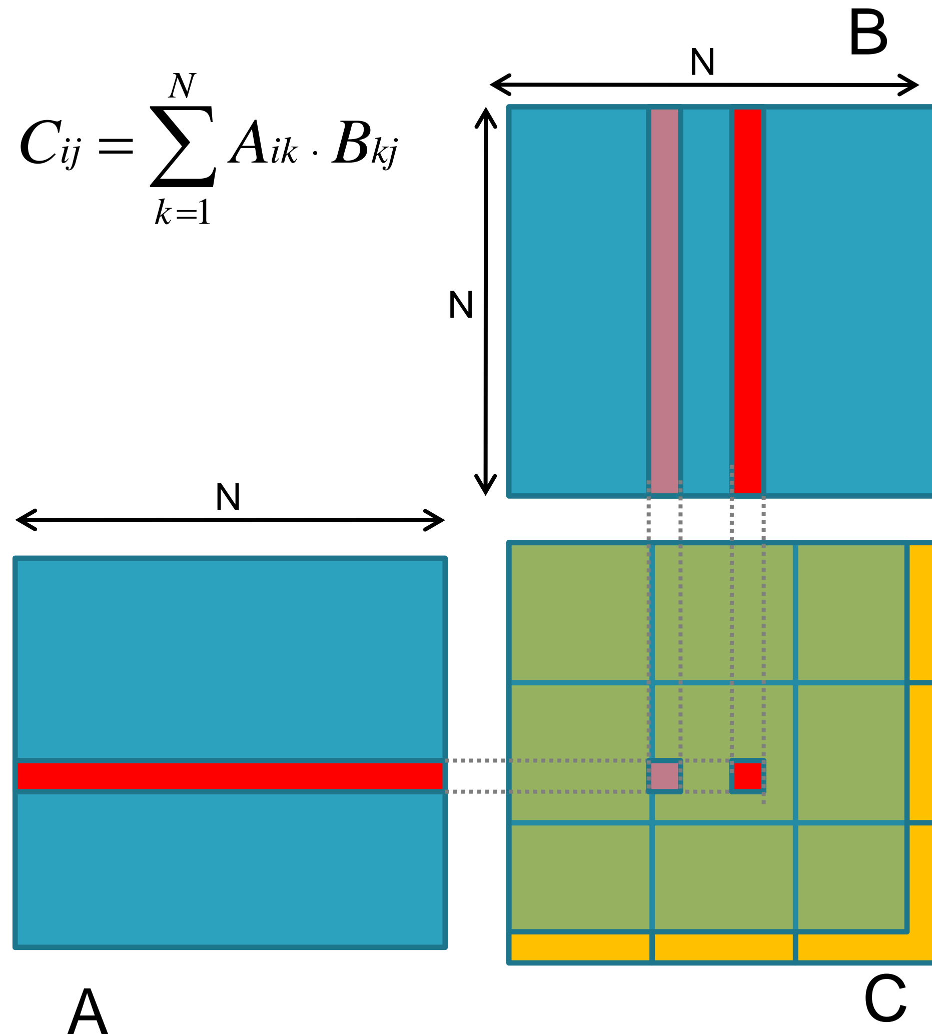


Prodotto matrice-matrice usando memoria globale

- Prodotto Matrice Matrice con *Global Memory*:

  Ogni thread calcola un elemento di C accedendo a 2N elementi (N di A e N di B) e svolgendo 2N operazioni floating-point (N *add* e N *mul*)

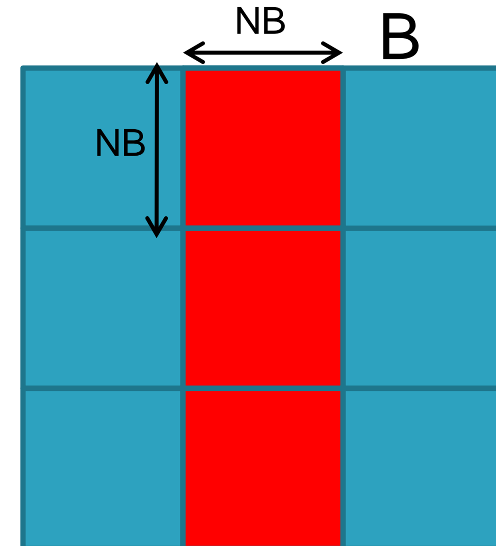
- Poichè gli accessi alla memoria globale sono lenti, consideriamo di usare la memoria condivisa all'interno dei blocchi di thread (*Shared Memory*)
  - L'accesso alla memoria shared è più veloce di quello alla memoria globale
  - I thread caricano in parallelo i dati in memoria shared e poi possono in parallelo effettuare le operazioni da svolgere accedendo ai dati caricati da tutti i thread dello stesso blocco di thread
  - La dimensione della Memoria Shared è di 16/48 Kb a seconda della compute capability e/o della configurazione della scheda

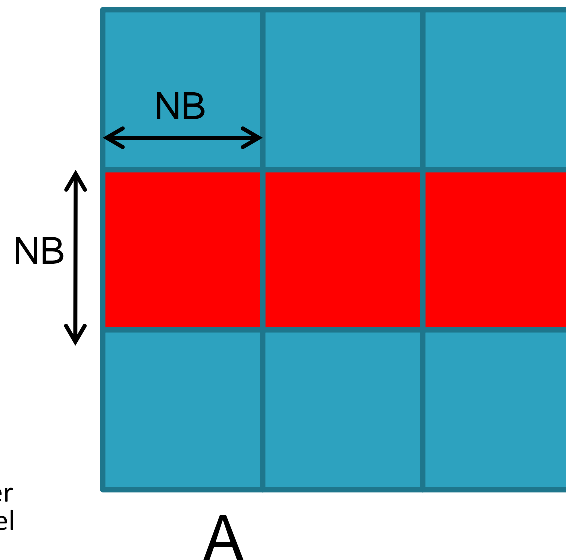


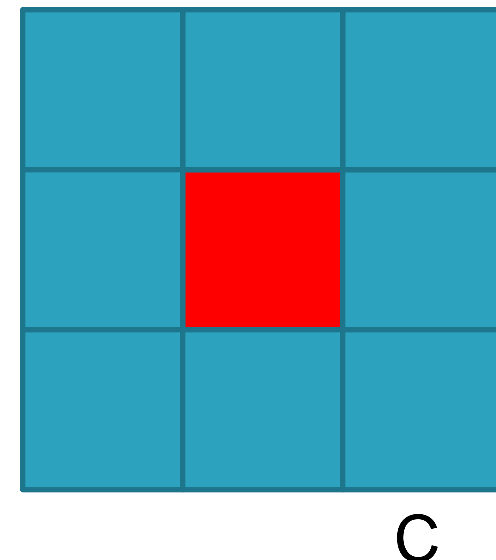
Prodotto matrice-matrice usando memoria shared

- C = A · B, dimensioni (N·N)
- Elaborazione a blocchi
- Per semplicità i blocchi siano di dimensione (NB·NB) con N multiplo di NB
- dimBlock(NB, NB)
- dimGrid(N/NB, N/NB)
- ogni thread block calcola un blocco (NB·NB) della matrice C
- il blocco è il risultato del prodotto delle N/NB sottomatrici di A e di B
- Ogni thread calcola un elemento della matrice
- Il kernel dovrà contenere due macro-fasi:
  1. i thread caricano gli elementi di A e B dalla memoria globale alla memoria shared
  2. i thread calcolano i prodotti necessari per ottenere C estraendo i valori di A e B dalla memoria shared
- Gli elementi di C non vengono accumulati direttamente nella memoria globale ma in variabili di appoggio nei registri e solo alla fine copiati nella memoria globale
- Sono necessarie delle sincronizzazioni:
  - dopo aver caricato la memoria shared per essere garantiti che tutti abbiano caricato
  - dopo aver effettuato il prodotto tra due blocchi per evitare che un thread carichi la memoria shared del passo successivo quando alcuni stanno ancora ultimando il passo precedente

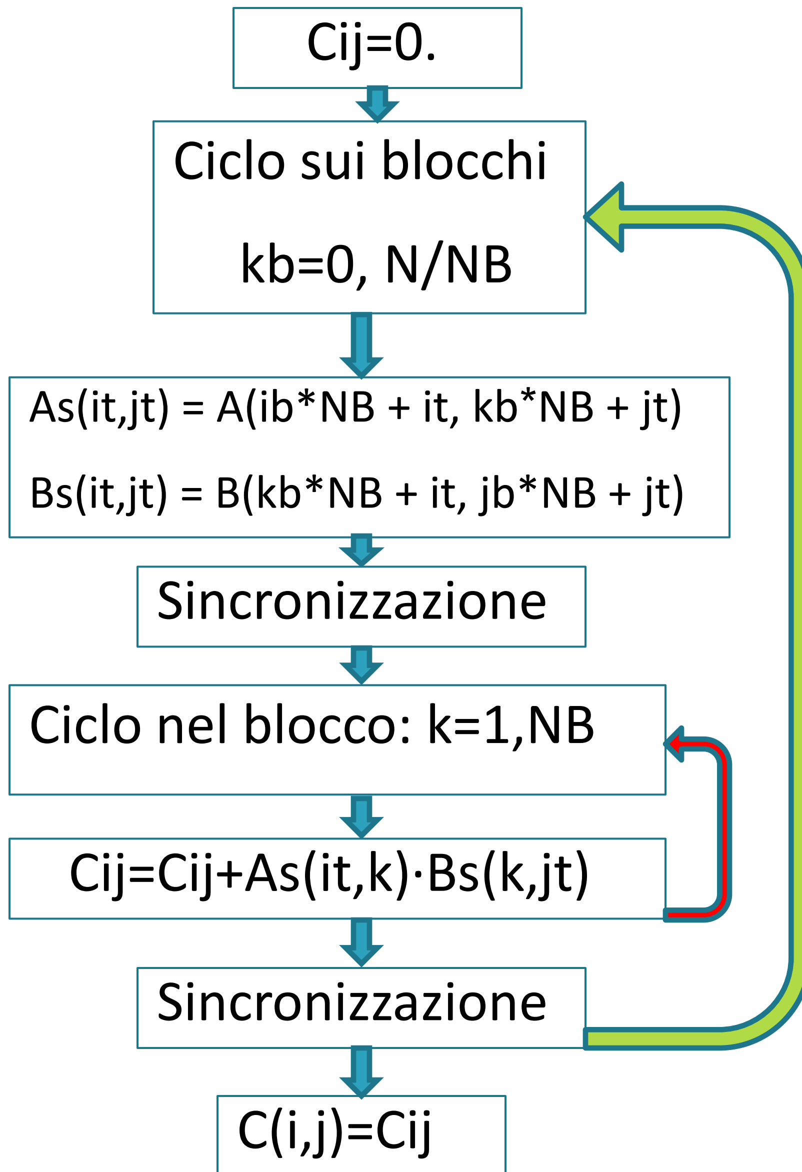
$$C_{ij} = \sum_{S=1}^{N/NB} \sum_{k=1}^{NB} As_{ik} \cdot Bs_{kj}$$





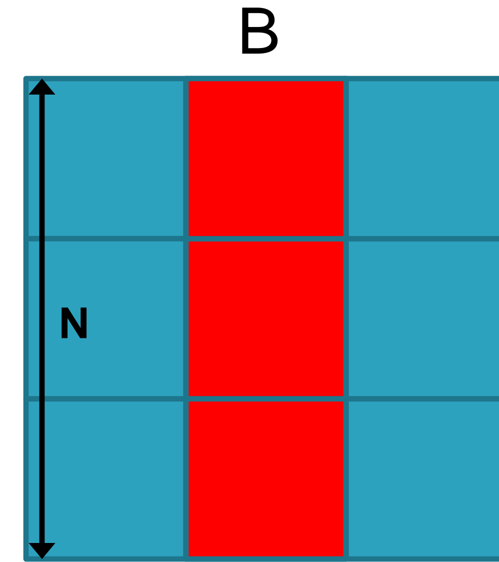


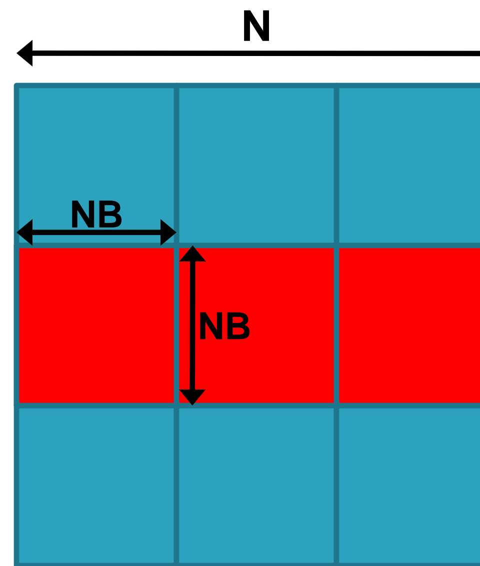
Matrice-matrice con shared: algoritmo del kernel

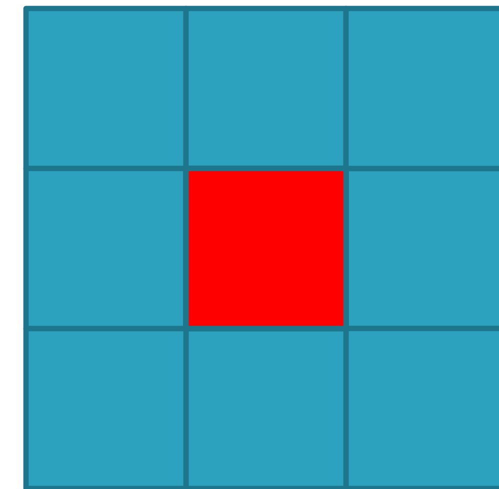



```
it = threadIdx.y
jt = threadIdx.x

ib = blockIdx.y
jb = blockIdx.x
```

```
it = threadIdx%x
jt = threadIdx%y

ib = blockIdx%x - 1
jb = blockIdx%y - 1
```







CINECA SCAI
SuperComputing Applications and Innovation

Prodotto di matrici: lato device

```
// Matrix multiplication kernel called by MatMul_gpu()
__global__ void MatMul_kernel (float *A, float *B, float *C, int N)
{

  // Shared memory used to store Asub and Bsub respectively
  __shared__  float Asub[NB][NB];
  __shared__  float Bsub[NB][NB];

  // Block row and column
  int ib = blockIdx.y;
  int jb = blockIdx.x;

  // Thread row and column within Csub
  int it = threadIdx.y;
  int jt = threadIdx.x;

  int a_offset , b_offset, c_offset;

  // Each thread computes one element of Csub
  // by accumulating results into Cvalue
  float Cvalue = 0;

  // Loop over all the sub-matrices of A and B that are
  // required to compute Csub
  // Multiply each pair of sub-matrices together
  // and accumulate the results

  for (int kb = 0; kb < (A.width / NB); ++kb) {

    // Get the starting address of Asub  and Bsub
    a_offset = get_offset (ib, kb, N);
    b_offset = get_offset (kb, jb, N);

    // Load Asub and Bsub from device memory to shared memory
    // Each thread loads one element of each sub-matrix
    Asub[it][jt] = A[a_offset + it*N + jt];
    Bsub[it][jt] = B[b_offset + it*N + jt];

    // Synchronize to make sure the sub-matrices are loaded
    // before starting the computation
    __syncthreads();

    // Multiply Asub and Bsub together
    for (int k = 0; k < NB; ++k) {
      Cvalue += Asub[it][k] * Bsub[k][jt];
    }
    // Synchronize to make sure that the preceding
    // computation is done
    __syncthreads();
  }

  // Get the starting address (c_offset) of Csub
  c_offset = get_offset (ib, jb, N);
 // Each thread block computes one sub-matrix Csub of C
  C[c_offset + it*N + jt] = Cvalue;

}
```

SCAI
SuperComputing Applications and Innovation

Esempio: Matrix Transpose

```
__global__ void transposeNaive(float *idata, float *odata, int width, int height) {
  int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;

  int index_in  = width * yIndex + xIndex;
  int index_out = height * xIndex + yIndex;

  odata[index_out] = idata[index_in];
}
```

- Tutti gli accessi in lettura sono *coalesced*:
  - ogni *warp* legge una riga di elementi contigui in memoria
    - 32 float risiedono sullo stesso segmento
- Gli accessi in scrittura non sono *coalesced*:
  - il kernel transposeNaive scrive per colonne
    - ogni thread del warp scrive un elemento non contiguo in memoria
    - accede a segmenti differenti dipendente dallo stride
- Il kernel transposeNaive usa 32 scritture differenti per ogni riga letta

Esempio: Matrix Transpose con *Shared Memory*

- Per non incorrere in scritture non-coalesced dovremmo scrivere per righe:
  - riempiamo un tile in shared memory con i dati da scrivere
  - non ci sono penalità di performance di accesso non-contiguo in shared memory
  - eseguiamo la trasposizione in shared memory
  - scriviamo i risultati indietro per righe in modo coalesced

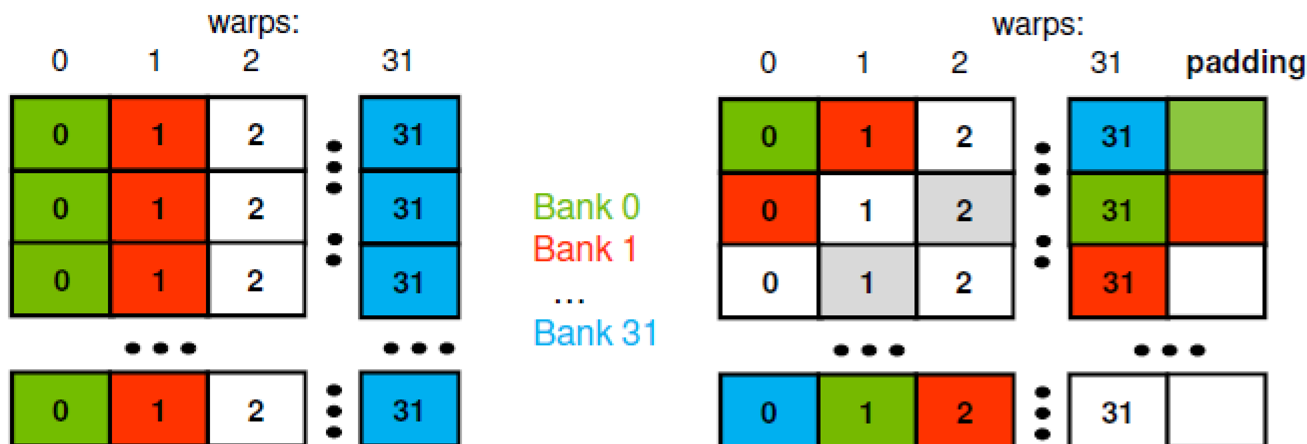
Esempio: Matrix Transpose con *Shared Memory*

```
__global__ void transposeCoalesced(float *idata, float *odata,
                                             int width, int height) {

  __shared__ float tile[TILE_DIM][TILE_DIM];

  int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;

  int index_in  = width * yIndex + xIndex;

  xIndex = blockIdx.y * TILE_DIM + threadIdx.x;
  yIndex = blockIdx.x * TILE_DIM + threadIdx.y;

  int index_out = height * yIndex + xIndex;

  tile[threadIdx.y][threadIdx.x] = idata[index_in];

  __syncthreads();

  odata[index_out] = tile[threadIdx.x][threadIdx.y];
}
```

Evitare il Bank Conflict

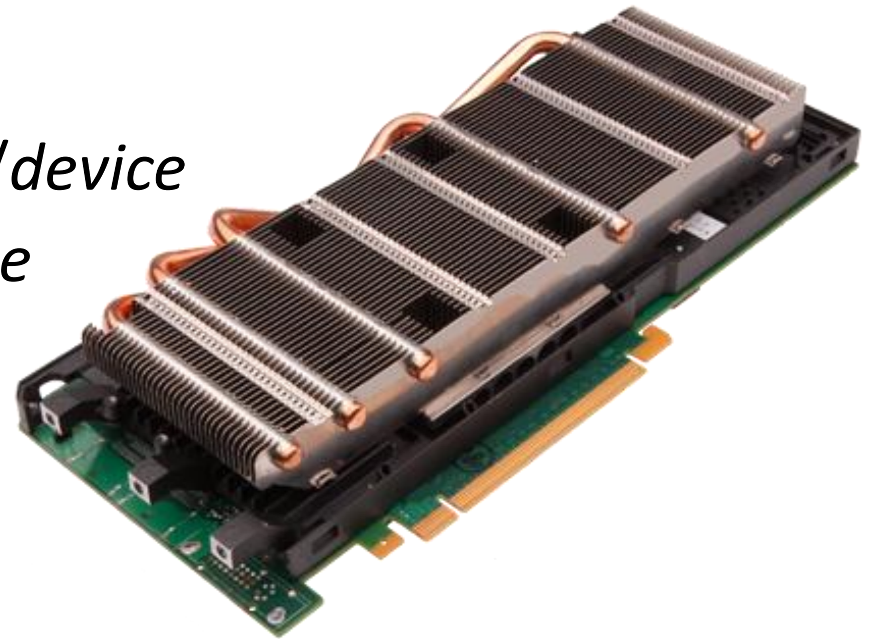
- il kernel coalesced transpose usa un tile in shared memory dimensionato a 32x32 float
  - ogni elemento risiede su un singolo banco (4-byte)
  - dati che distano 32 floats sono mappati sullo stesso banco
  - lettura/scrittura su colonne di questo tile origina bank conflict
- basta usare righe di 33 elementi
  - tutti gli elementi di una colonna vengono distribuiti su banchi differenti

```
__shared__ float tile[TILE_DIM][TILE_DIM+1];
```

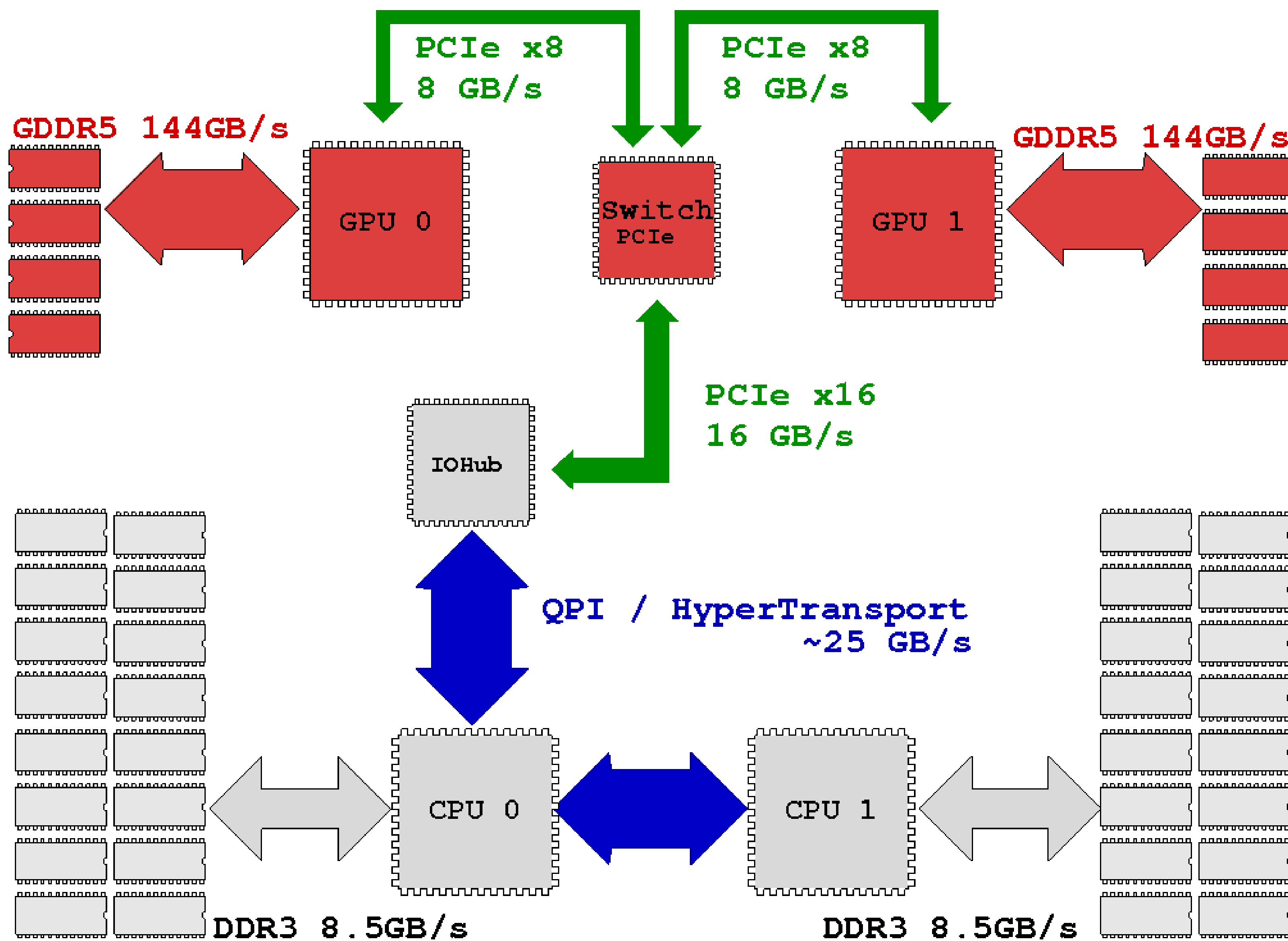


SCAI
SuperComputing Applications and Innovation

- Funzioni sincrone ed asincrone
- Esecuzione concorrente
- Interazione *host/device*
  - Esecuzione concorrente *host/device*
  - Trasferimento dati *host/device*
- Gestione multi-device
- Interazione *device/device*



Schema collegamento fisico *host/device*



Funzioni bloccanti e non bloccanti

- L'*host* si limita a sottomettere comandi ad una coda di esecuzione sul *device*

- Le funzioni del CUDA runtime possono essere suddivise in due categorie:

  - Funzioni **bloccanti** (sincrone): restituiscono il controllo all'*host* thread dopo aver terminato l'esecuzione sul *device*
    - Trasferimenti di memoria da *host* a *device* (>64 KB) e viceversa
    - Allocazione di memoria sul *device*
    - Allocazione di memoria page locked sull'*host*

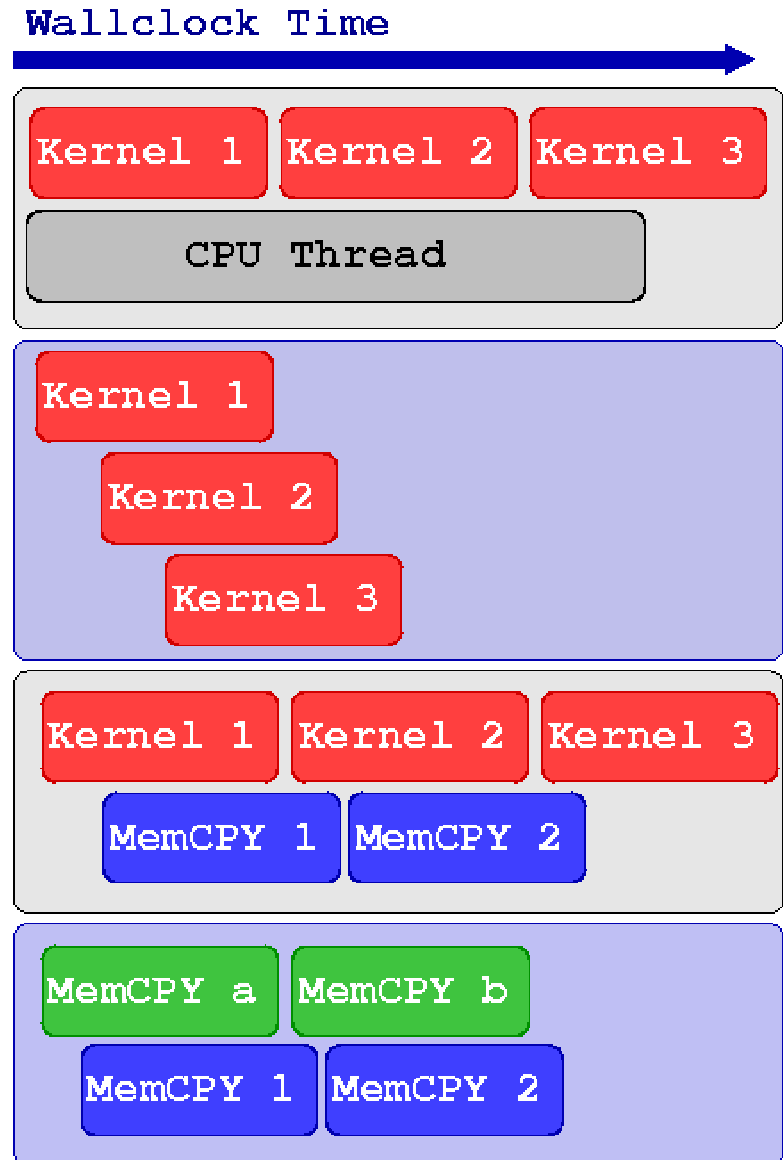
  - Funzioni **non bloccanti** (asincrone): restituiscono immediatamente il controllo all'*host* thread
    - Lancio di kernel
    - Trasferimenti di memoria da *host* a *device* < 64 KB
    - Inizializzazione della memoria sul *device* (cudaMemset)
    - Copie della memoria da *device* a *device*
    - Trasferimenti di memoria asincroni da *host* a *device* e viceversa

- Le funzioni asincrone permettono di realizzare flussi di esecuzione concorrente tra *host* e *device* e/o all'interno del *device*

Esecuzione asincrona e concorrente

L' esecuzione asincrona e concorrente permette di sovrapporre:

1. calcolo sull'*host* e calcolo sul *device*
2. esecuzione di più kernel sulla stessa GPU
3. trasferimento dati dall'*host* al *device* ed esecuzione sul *device*
4. trasferimento dati dall'*host* al *device* e dal *device* all'*host*



Esecuzione concorrente *host/device*

```
cudaSetDevice(0)
kernel <<<threads, Blocks>>> (a, b, c)

// Esegui lavoro utile su CPU mentre è in esecuzione
// il kernel sulla GPU
CPU_Function()

// Blocca la CPU fino a quando la GPU ha finito il calcolo
cudaDeviceSynchronize()

// uso il risultato del calcolo su GPU sull'host
CPU_uses_the_GPU_kernel_results()
```

Poiché il lancio del kernel è non bloccante la CPU può eseguire la funzione `CPU_Function()` mentre la GPU è impegnata nel calcolo del kernel (esecuzione concorrente).

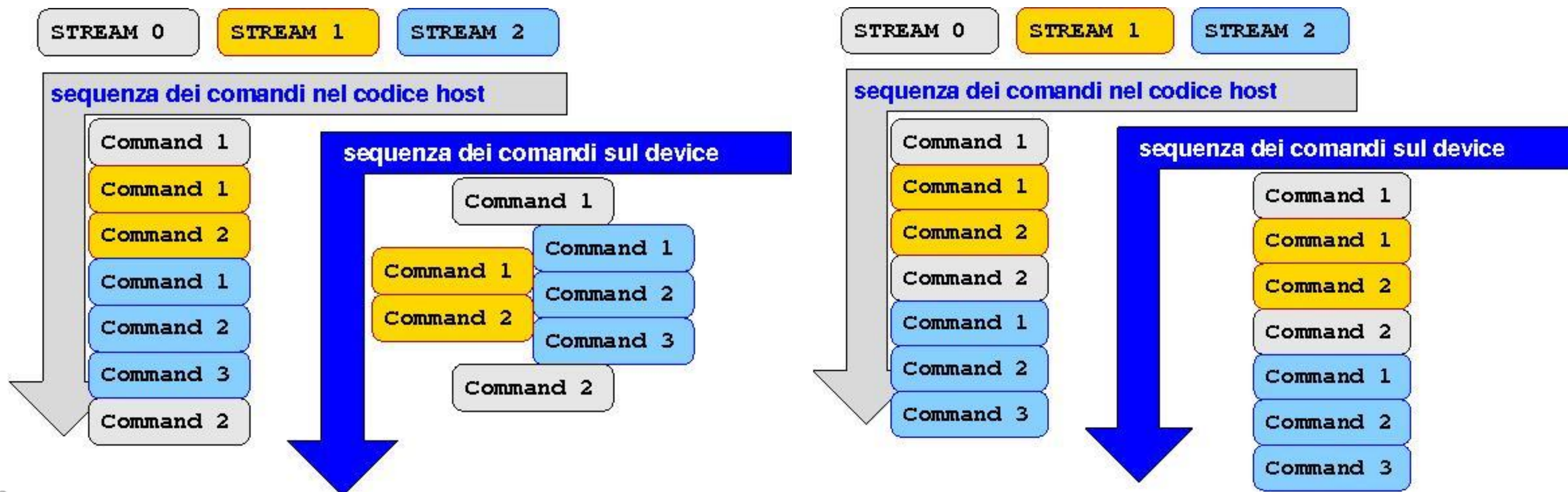
Prima di utilizzare il risultato del kernel è necessario sincronizzare i flussi di esecuzione su *host* e *device*.

CUDA Stream

- In CUDA le operazioni sulla GPU sono implementate utilizzando delle code di esecuzione (**stream**).
- Un comando posto alla fine di una coda di esecuzione verrà eseguito solo dopo che tutti i precedenti comandi assegnati a quello stream sono stati completati (FIFO queue).
- I comandi assegnati a stream differenti potranno essere eseguiti concorrentemente.
- CUDA definisce il **default stream** (stream 0) che verrà utilizzato in assenza di stream definiti dall' utente

CUDA Stream

- Le operazioni assegnate al default stream verranno eseguite solo dopo che tutte le operazioni precedenti assegnate a qualsiasi altro stream saranno state completate.
- Inoltre qualsiasi operazione successiva assegnata a stream definiti dall'utente potrà iniziare solo dopo che tutte le operazioni sul default stream saranno completate.
- I comandi assegnati al default stream agiscono come barriere di sincronizzazione implicite tra le diverse code di esecuzione.



Sincronizzazioni in CUDA

- Sincronizzazioni **Esplicite** :
  - `cudaDeviceSynchronize()`
    - Blocca l'host finchè ogni chiamata a funzioni CUDA non è terminata sul device
  - `cudaStreamSynchronize(stream)`
    - Blocca l'host finchè ogni chiamata sullo stream non è terminata
  - `cudaStreamWaitEvent(stream, event)`
    - Ogni comando assegnato allo stream non è eseguito finchè l'evento non è completato
- Sincronizzazioni **Implicite** :
  - Ogni comando assegnato allo stream di default
  - Allocazioni di memoria page-locked
  - Allocazioni di memoria sul device
  - Settings del device
  - …

CUDA Stream

- Stream management:
  - Creazione: `cudaStreamCreate()`
  - Sincronizzazione: `cudaStreamSynchronize()`
  - Distruzione: `cudaStreamDestroy()`
- A seconda della compute capability, gli stream permettono di implementare diversi modelli di esecuzione concorrente:
  - esecuzione di più kernel sulla stessa GPU
  - trasferimento asincrono di dati *host*-to-*device* e/o *device*-to-*host* ed esecuzione sulla GPU e sull'*host*
  - trasferimento dati *host*-to-*device* e *device*-to-*host*

Esecuzione concorrente dei kernel

```
cudaSetDevice(0)

cudaStreamCreate(stream1)
cudaStreamCreate(stream2)

// lancio concorrente dello stesso kernel
Kernel_1<<<blocks, threads, SharedMem, stream1>>>(inp_1, out_1)
Kernel_1<<<blocks, threads, SharedMem, stream2>>>(inp_2, out_2)

// lancio concorrente di kernel diversi
Kernel_1<<<blocks, threads, SharedMem, stream1>>>(inp, out_1)
Kernel_2<<<blocks, threads, SharedMem, stream2>>>(inp, out_2)

cudaStreamDestroy(stream1)
cudaStreamDestroy(stream2)
```

Trasferimenti asincroni di dati

- Per trasferire memoria in modo asincrono è necessario che la memoria sull'*host* sia di tipo page-locked (anche detta pinned)
- Per la gestione della memoria page-locked il runtime CUDA mette a disposizione le funzioni:
  - `cudaMallocHost()` permette di allocare memoria page-locked sull'*host*
  - `cudaFreeHost()` deve essere chiamata per deallocare memoria page-locked
  - `cudaHostRegister()` permette di convertire un area di memoria in memoria page-locked
  - `cudaHostUnregister()` riconverte la memoria page-locked in memoria paginabile
- La funzione `cudaMemcpyAsync()` permette di eseguire i trasferimenti asincroni da *host* a *device* e viceversa
- Perché un trasferimento sia asincrono è necessario che avvenga su uno stream diverso dal Default Stream
- L'uso della memoria page-locked consente il trasferimento di dati da *host* a *device* e viceversa con una bandwidth maggiore

Trasferimenti asincroni di dati

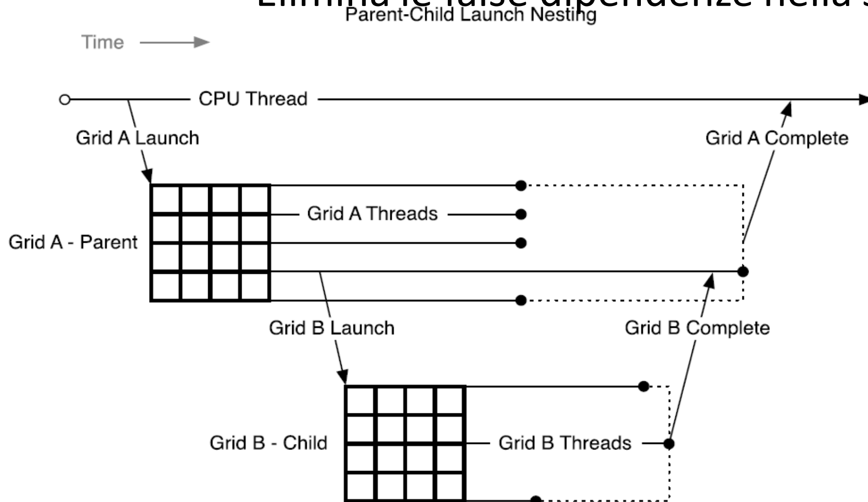
```
cudaStreamCreate(stream_a)
cudaStreamCreate(stream_b)

cudaMallocHost(h_buffer_a, buffer_a_size)
cudaMallocHost(h_buffer_b, buffer_b_size)

cudaMalloc(d_buffer_a, buffer_a_size)
cudaMalloc(d_buffer_b, buffer_b_size)

// trasferimento asincrono e concorrente H2D e D2H
cudaMemcpyAsync(d_buffer_a, h_buffer_a, buffer_a_size,
 cudaMemcpyHostToDevice, stream_a)
cudaMemcpyAsync(h_buffer_b, d_buffer_b, buffer_b_size,
 cudaMemcpyDeviceToHost, stream_b)

cudaStreamDestroy(stream_a)
cudaStreamDestroy(stream_b)

cudaFreeHost(h_buffer_a)
cudaFreeHost(h_buffer_b)
```

Aumentare la *concurrency* con i CUDA streams

```
cudaStream_t stream[4];
for (int i=0; i<4; ++i) cudaStreamCreate(&stream[i]);

float* hPtr; cudaMallocHost((void**)&hPtr, 4 * size);

for (int i=0; i<4; ++i) {
  cudaMemcpyAsync(d_inp + i*size, hPtr + i*size,
                  size, cudaMemcpyHostToDevice, stream[i]);

  MyKernel<<<100, 512, 0, stream[i]>>>(d_out+i*size, d_inp+i*size, size);

  cudaMemcpyAsync(hPtr + i*size, d_out + i*size,
                  size, cudaMemcpyDeviceToHost, stream[i]);
}
cudaDeviceSynchronize();

for (int i=0; i<4; ++i) cudaStreamDestroy(&stream[i]);
```

Kepler : *dynamic parallelism*

- Con l'architettura Kepler (cc 3.5) e CUDA 5.x è stato introdotto il *dynamic parallelism*:
  - Un kernel CUDA può essere invocato all'interno di una altro kernel CUDA
  - Si possono invocare kernel con griglie/blocchi di thread dimensionate dinamicamente
  - Sfrutta la tecnologia *Hyper-Q* :
    - HW dedicato per lo *scheduling* del lavoro sottomesso al device
    - Gestisce fino a 32 code di esecuzione sul device (con Fermi unica pipeline di esecuzione)
    - Processi/thread differenti possono controllare il medesimo device
    - Elimina le false dipendenze nella sottomissione di comandi a differenti stream



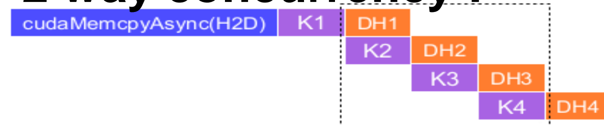

```
// codice Host:
ParentKernel<<<256, 64>>(data);

// codice Device:
__global__ ParentKernel(void *data){
  ChildKernel<<<16, 1>>>(data);
}

__global__ ChildKernel(void* data){
...
}
```

Concurrency

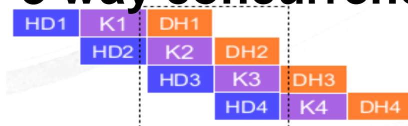
- Concurrency: abilità di elaborare più comandi CUDA simultaneamente
  - **Fermi** : fino a 16 kernel CUDA / **Kepler** : fino a 32 kernel CUDA
  - 2 copie Host/device in direzioni opposte
  - Elaborazione sull'host
- Requisiti:
  - I comandi CUDA devono essere assegnati a stream differenti dal default
  - Le copie Host/Device devono essere asincrone e la memoria host deve essere page-locked
  - Ci devono essere sufficienti risorse HW a disposizione:
    - Shared Mem., Registri, Blocchi, Bus PCI, …

Serial :

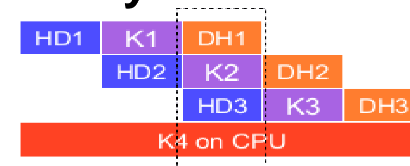
2 way concurrency :



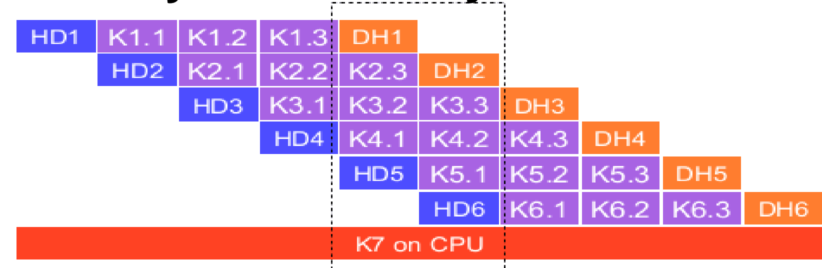
3 way concurrency :



4 way concurrency :



4/+ way concurrency :



Device management

Il runtime CUDA consente di gestire una o più GPU (programmazione Multi-GPU):

- CUDA 3.2 e precedenti
  - È necessaria una programmazione multi thread e/o multi processo per poter gestire contemporaneamente più GPU
- CUDA 4.0 e successivi
  - Nuove funzionalità del runtime consentono ad un singolo *host* thread di gestire tutte le GPU presenti sul sistema
  - È possibile continuare ad usare un approccio multi thread e/o multi processo per la gestione delle GPU

Device management

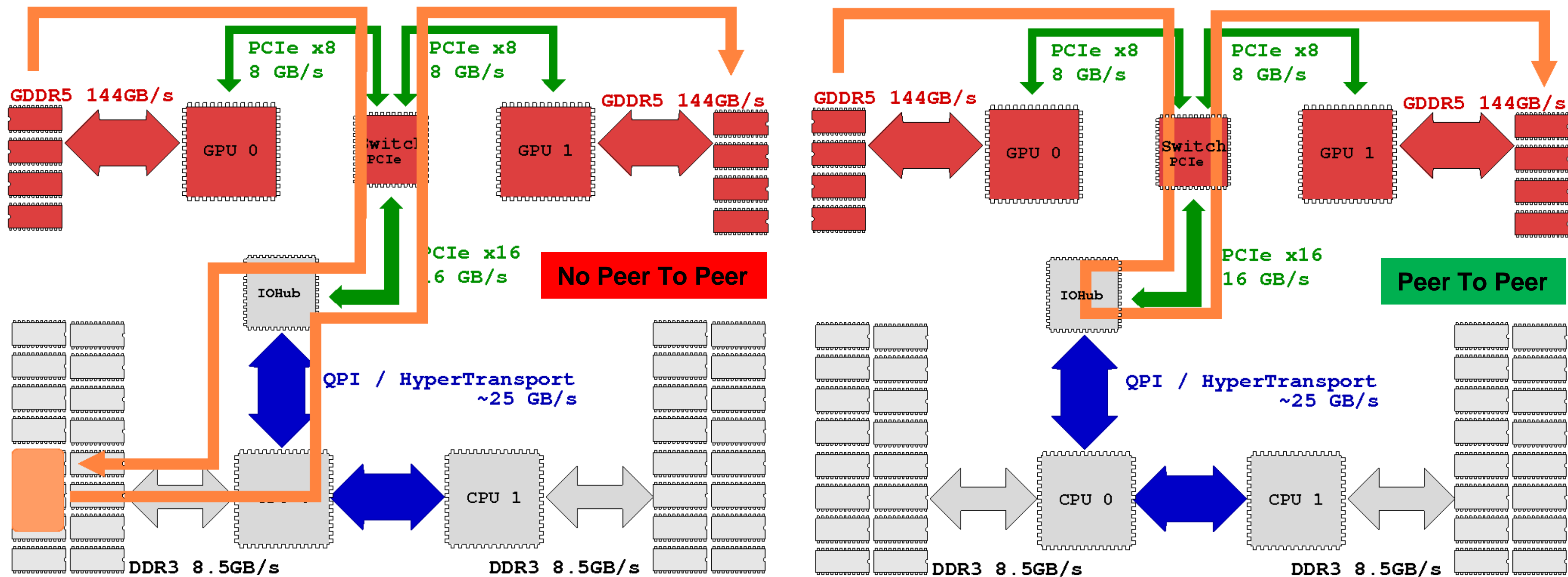
```
cudaDeviceCount(number_gpu)
cudaGetDeviceProperties(gpu_property, gpu_ID)

cudaSetDevice(0)
kernel_0 <<<threads, Blocks>>> (a, b, c)

cudaSetDevice(1)
kernel_1 <<<threads, Blocks>>> (d, e, f)

For each device:
   cudaSetDevice(device)
   cudaDeviceSynchronize()
```

Il runtime permette di:

- Conoscere il numero di GPU installate
- Conoscere le caratteristiche di una GPU
- Selezionare una GPU
- Inviare comandi alla GPU selezionata
- Sincronizzare le GPU utilizzate nel calcolo

Trasferimento e accesso dati peer to peer

- Un *device* può trasferire dati e accedere alla memoria di un altro *device* in maniera diretta
- Il trasferimento diretto non richiede un buffer di appoggio sull'*host*
- Il trasferimento diretto è più efficiente di uno che utilizzi un buffer di appoggio sull'*host*
- L'accesso diretto evita la copia di dati da *device* a *device*



Trasferimento dati peer to peer

```
gpuA=0, gpuB=1
cudaSetDevice(gpuA)
cudaMalloc(buffer_A, buffer_size)

cudaSetDevice(gpuB)
cudaMalloc(buffer_B, buffer_size)

cudaSetDevice(gpuA)
cudaDeviceCanAccessPeer(answer, gpuA, gpuB)

If answer is true:
 cudaDeviceEnablePeerAccess(gpuB, 0)
 // la gpuA esegue la copia da gpuA a gpuB
 cudaMemcpyPeer(buffer_B, gpuB, buffer_A, gpuA, buffer_size)
 // la gpuA esegue la copia da gpuB a gpuA
 cudaMemcpyPeer(buffer_A, gpuA, buffer_B, gpuB, buffer_size)
```

Accesso dati peer to peer

```
gpuA=0, gpuB=1
cudaSetDevice(gpuA)
cudaMalloc(buffer_A, buffer_size)

cudaSetDevice(gpuB)
cudaMalloc(buffer_B, buffer_size)

cudaSetDevice(gpuA)
cudaDeviceCanAccessPeer(answer, gpuA, gpuB)

If answer is true:
 cudaDeviceEnablePeerAccess(gpuB, 0)
 // la gpuA esegue il kernel che accede sia alla sua memoria
 // che direttamente alla memoria di gpuB
 kernel<<<threads, blocks>>>(buffer_A, buffer_B)
```