

GPU Computing

Laurea Magistrale in Informatica - AA 2018/19

Docente **G. Grossi**

Lezione 6 – la global memory

Sommario

- ✓ La global memory
- ✓ Pinned & unified memory
- ✓ Memory bandwidth
- ✓ Coalescenza e pattern di accesso alla global memory
- ✓ Allocazione statica e dinamica di shared memory
- ✓ Esercitazione su convoluzione e matrice trasposta

Variabili dinamiche e statiche

STATICÀ:

- ✓ Dic. con qualificatore **`__device__`**
- ✓ visibilità: **file scope**

```
__device__ int devCount; // static global var

/* kernel that uses the global var */
__global__ void incGlobalVariable() {
    devCount++; // gloabl var change
}
```

DINAMICA:

- ✓ allocazione da **host...**
- ✓ visibilità: **file scope**

```
// dynamic global var
float *dev = (float *) malloc(nbytes);

// free memory
cudaFree(dev);
```

Uso di memoria globale statica

Copia dati da una variabile device a una host avviene con **cudaMemcpyToSymbol** e **cudaMemcpyFromSymbol** (copia attraverso il simbolo e non l'indirizzo)

Nota: non si può usare dall'host l'op. dereferenz. **&** su una variabile device perché si tratta di un simbolo che fa riferimento a un oggetto nella lookup table della GPU

```
int main(void) {
    // initialize global var
    int count = 0;
    // copy to memory device
    cudaMemcpyToSymbol(devCount, &count, sizeof(int));
    printf("Host: init value of global var: %d\n", count);

    // launch kernel many times
    for (int i = 1; i <= 10; i++) {
        incGlobalVariable<<<1, 1>>>();
        // host gets value
        cudaMemcpyFromSymbol(&count, devCount, sizeof(int));
        printf("Host: number of calls: %d\n", count);
    }
    return EXIT_SUCCESS;
}
```

Accesso a variabile globale statica

Si può tuttavia acquisire l'indirizzo della variabile globale con la CUDA API: **cudaGetSymbolAddress**

Nota: Si può anche avere la dimensione dei dati allocati nella variabile globale con la CUDA API: **cudaGetSymbolSize**

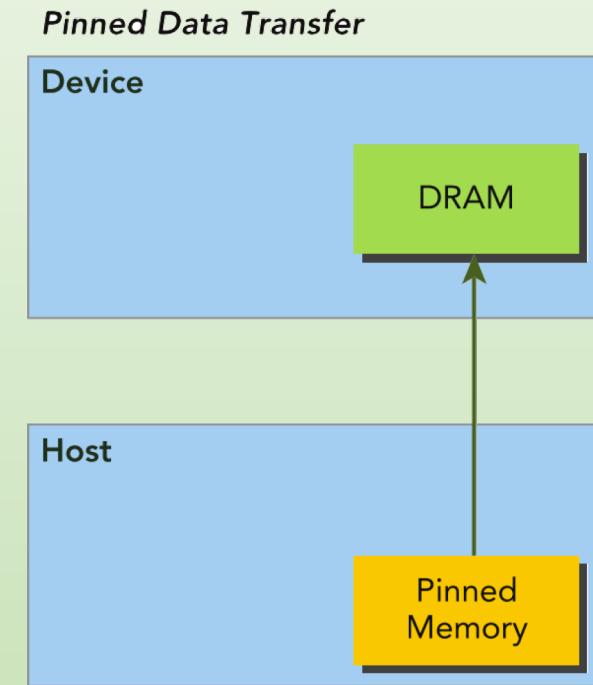
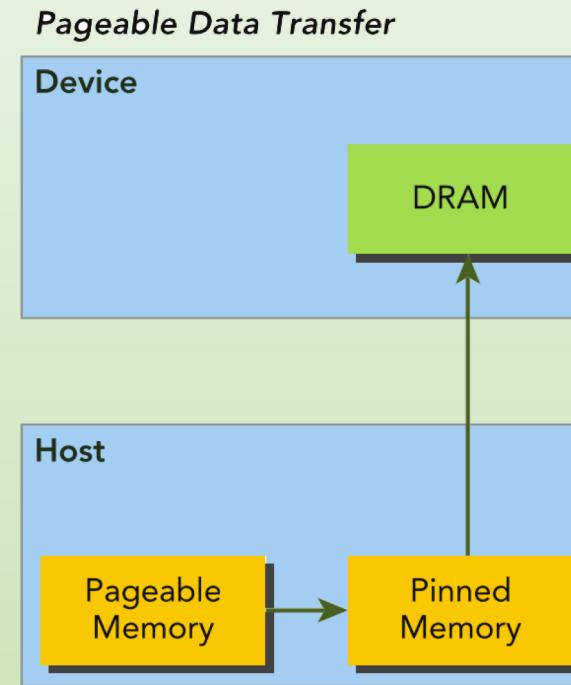
```
 . . .
// using pointer
count = 0;
int *devptr = NULL;
cudaGetSymbolAddress((void**)&devptr, devCount);
cudaMemcpy(devptr, &count, sizeof(int), cudaMemcpyHostToDevice);
// kernel launch
incGlobalVariable<<<1, 1>>>();
cudaMemcpy(&count, devptr, sizeof(int), cudaMemcpyDeviceToHost);
printf("Host: number of calls : %d\n", count);
. . .
```

Riassunto dichiarazioni di variabili

QUALIFIER	VARIABLE NAME	MEMORY	SCOPE	LIFESPAN
	float var	Register	Local	Thread
	float var[100]	Local	Local	Thread
<code>__shared__</code>	float var	Shared	Block	Block
<code>__device__</code>	float var	Global	Global	Application
<code>__constant__</code>	float var	Constant	Global	Application

Pinned memory

- ✓ La memoria host allocata è per default **paginabile** (soggetta a page fault per effetto della virtual memory gestita dal sis. op. e non nota al device)
- ✓ La **virtual memory** offre l'illusione di avere molta più memoria di quella fisicamente disponibile come nel caso della **cache L1** (memoria però on-chip)
- ✓ La GPU non può avere accesso sicuro a dati in memoria virtuale, per cui il driver CUDA prima di trasferire allocatione temporaneamente memoria **page-locked** o **pinned** e poi effettua il trasferimento sicuro al device



Gestione della pinned memory

- ✓ La pinned memory può essere **acceduta** direttamente dal device
- ✓ Può essere letta e scritta con più **alta bandwidth** rispetto alla memoria paginabile
- ✓ **Nota:** eccessi di allocazione di pinned memory potrebbero far degradare le prestazioni dell'host (ridurre la memoria paginabile inficia l'uso della virtual memory)
- ✓ Per allocare esplicitamente la memoria pinned:

prototipo-> `cudaError_t cudaMallocHost(void **devPtr, size_t count);`

- ✓ La memoria pinned può essere deallocata con:

prototipo-> `cudaError_t cudaFreeHost(void *devPtr);`

Trasferimento pinned più efficiente

L'uso della pinned memory rende più efficiente il trasferimento. Esempio su **Tesla K40**

```
$ nvcc -O3 -arch=sm_30 peag_mem_tranf.cu
```

```
$ nvprof peag_mem_tranf 1GB
Profiling result:
Time(%)      Time      Avg      Min      Max   Name
 54.41%    515.63ms  515.63ms  515.63ms  515.63ms  [CUDA memcpy HtoD]
 45.59%    432.06ms  432.06ms  432.06ms  432.06ms  [CUDA memcpy DtoH]
```

```
// allocate the host memory
float *h_a = (float*)malloc(nbytes);
// allocate the device memory
float *d_a;
cudaMalloc((void**)&d_a, nbytes);
// transfer data from the host to the device
cudaMemcpy(d_a, h_a, nbytes, cudaMemcpyHostToDevice);
// transfer data from the device to the host
cudaMemcpy(h_a, d_a, nbytes, cudaMemcpyDeviceToHost);
// free memory
cudaFree(d_a);
free(h_a);
```

```
$ nvcc -O3 -arch=sm_30 pinned_mem_tranf.cu
```

```
$ nvprof peag_mem_tranf 1GB
Profiling result:
Time(%)      Time      Avg      Min      Max   Name
 47.46%    172.86ms  172.86ms  172.86ms  172.86ms  [CUDA memcpy HtoD]
 52.54%    191.39ms  191.39ms  191.39ms  191.39ms  [CUDA memcpy DtoH]
```

```
// allocate pinned host memory
float *h_a;
cudaMallocHost((void**)&h_a, nbytes);
// allocate device memory
float *d_a;
cudaMalloc((float**)&d_a, nbytes);
// transfer data from the host to the device
cudaMemcpy(d_a, h_a, nbytes, cudaMemcpyHostToDevice);
// transfer data from the device to the host
cudaMemcpy(h_a, d_a, nbytes, cudaMemcpyDeviceToHost);
// free memory
cudaFree(d_a);
cudaFreeHost(h_a);
```

Zero-copy memory

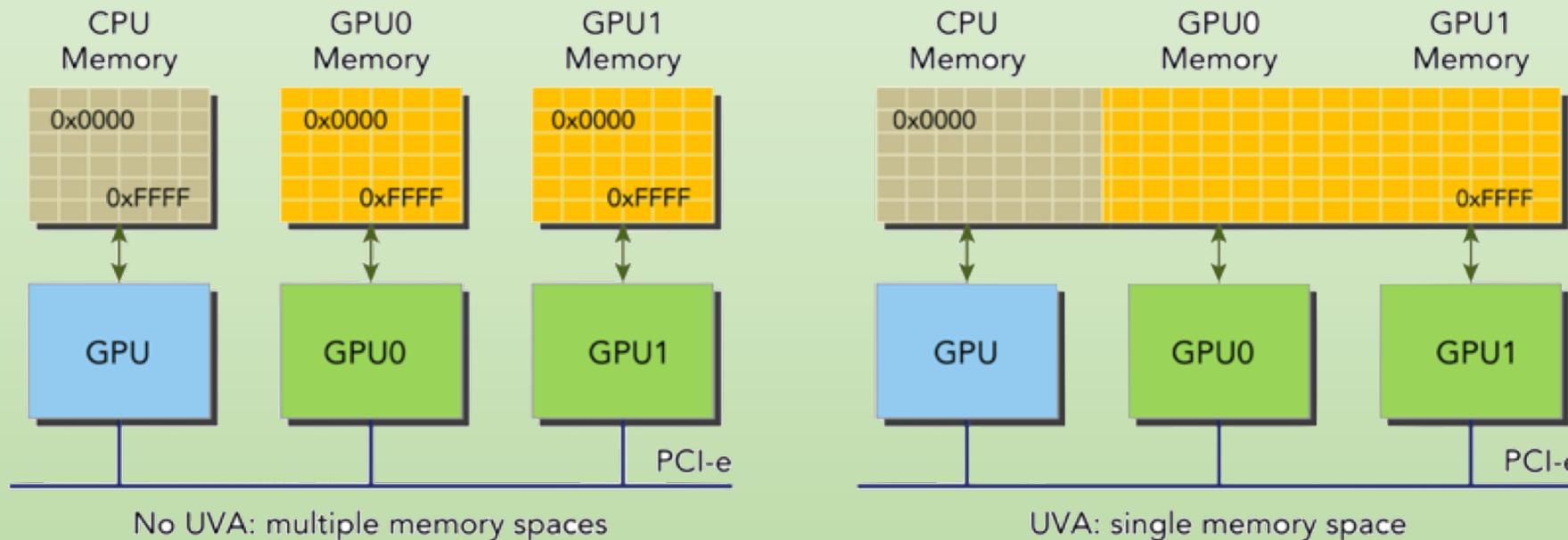
- ✓ Un blocco di memoria host (page-locked) può essere **mappata nello spazio indirizzamento del device:**
 - consente al kernel di **accedere alla host memory** direttamente dalla **GPU**
 - usa la flag **cudaHostAllocMapped** in **cudaHostAlloc()**
- ✓ Un blocco ha **due indirizzi** (e due puntatori)
 - uno in host memory, reso da **cudaHostAlloc()** o **malloc()**
 - uno in device memory, reso da **cudaHostGetDevicePointer()** usato dal kernel
 - questo (page-locked) mapping di memoria deve essere abilitato dalla chiamata **cudaSetDeviceFlags()** con il **cudaDeviceMapHost** flag prima di ogni altra CUDA call
 - Altrimenti **cudaHostGetDevicePointer()** restituisce null

(S)Vantaggi della zero-copy

- ✓ Sfruttare la memoria host quando è insufficiente quella del device
- ✓ evitare il trasferimento esplicito tra host e device
- ✓ migliorare il tasso di trasferimento attraverso bus PCI
- ✓ poiché la memoria è condivisa tra host e device, le applicazioni devono sincronizzare gli accessi usando gli stream o eventi per evitare read-after-write, write-after-read, or write-after-write hazards
- ✓ Si possono avere problemi con le operazioni atomiche perché non sono atomiche per l'host

Unified Virtual Addressing (UVA)

- ✓ Compute capability 2.0 and later support a special addressing mode called **Unified Virtual Addressing (UVA)**.
- ✓ UVA, introduced in **CUDA 4.0**, is supported on **64-bit** Linux systems
- ✓ With UVA, **host memory** and **device memory** share a **single virtual address space**



Unified Memory

- ✓ Simpler programming and memory model:
 - **Single pointer** to data, accessible anywhere
 - Eliminate need for **cudaMemcpy()**
 - Greatly simplifies code porting
 - First introduced in CUDA 6.0
- ✓ Performance through data locality:
 - Migrate data to accessing processor
 - Guarantee global coherency
 - Still allows **cudaMemcpyAsync()** hand tuning

Gestione

- ✓ New call: **cudaMallocManaged(pointer, size, flag)**
 - Drop-in replacement for **cudaMalloc(pointer, size)**
 - The flag indicates who shares the pointer with the device:
 - **cudaMemAttachHost**: Only the CPU
 - **cudaMemAttachGlobal**: Any other GPU too.
 - All operations valid on device mem. are also ok on managed mem.
- ✓ New keyword: **managed**
 - Global variable annotation combines with **device**
 - Declares global-scope migratable device variable
 - Symbol accessible from both GPU and CPU code

Esempio

The CPU cannot access any unified memory as long as GPU is executing a `cudaDeviceSynchronize()` call is required for the CPU to be allowed to access unified memory

```
__device__ __managed__ int x, y = 2; // Unified memory

__global__ void mykernel() {           // GPU territory
    x = 10;
}

int main() {                         // CPU territory
    mykernel <<<1,1>>> ();
    y = 20;                                // ERROR: CPU access concurrent with GPU
    return 0;
}
```

The GPU has exclusive access to unified memory when any kernel is executed on the GPU, and this holds even if the kernel does not touch the unified memory

```
__device__ __managed__ int x, y = 2; // Unified memory

__global__ void mykernel() {           // GPU territory
    x = 10;
}

int main() {                         // CPU territory
    mykernel <<<1,1>>> ();
    cudaDeviceSynchronize();
    y = 20;                                // Now the GPU is idle, so access to "y" is OK
    return 0;
}
```

Uso di dati dinamici

```
void sortfile (FILE *fp, int N){  
  
    char *data;  
  
    data = (char *) malloc(N);  
  
    fread(data, 1, N, fp);  
  
    kernel(data, N, 1, compare);  
  
    use_data(data);  
  
    free(data);  
  
}
```

```
void sortfile (FILE *fp, int N){  
  
    char *data;  
  
    cudaMallocManaged(&data, N);  
  
    fread(data, 1, N, fp);  
  
    kernel<<<...>>>(data, N, 1, compare);  
  
    cudaDeviceSynchronize();  
  
    use_data(data);  
  
    free(data);  
  
}
```

Constant memory

- ✓ Il modello di programmazione CUDA consente di dichiarare variabili (non troppo grandi) nella **constant memory** che risiede nella device memory (DRAM)
- ✓ Come nella global memory queste variabili sono **visibili a tutti i thread** dei vari blocchi
- ✓ La principale differenza è che una variabile in constant memory può essere solo letta dai thread dei vari blocchi ma solo modificabile dall'host

Dichiarazione:

```
#define MAX_MASK_WIDTH 10  
__constant__ float M[MAX_MASK_WIDTH];
```

Copia:

```
cudaError_t cudaMemcpyToSymbol(const void *symbol, const void * src, size_t count,  
size_t offset, cudaMemcpyKind kind);
```

- ✓ La constant memory ha una **cache dedicata** per-SM (64 KB) on-chip
- ✓ I pattern di accesso sono differenti dalle altre memorie: il migliore approccio è quello in cui tutti i thread di un warp accedono alla **stessa locazione** simultaneamente. L'accesso a indirizzi distinti è serializzato, il costo cresce **linearmente** con i numero di **indirizzi unici** richiesti dal warp

Esercitazione (lab 6)

Convoluzione 2D di un'immagine: Scrivere un programma CUDA per la convoluzione 2D che usi la SMEM e la UNIFIED MEMORY per i dati e la constant mem per la maschera del filtro

PASSI:

1. Definire la SMEM per ogni blocco di dimensione legata alla tile size
2. Caricare la constant memory con i coefficienti del filtro da host (pre kernel)
3. Nel kernel: caricare la SMEM da global mem
4. Sincronizzare -1-
5. Effettuare localmente il prodotto di convoluzione sfruttando i dati in cache
6. Scrivere il risultato finale su vettore/matrice in global mem

Memory bandwidth

Prestazioni del kernel:

- ✓ **memory latency**, il tempo necessario a soddisfare una richiesta dati in memoria
- ✓ **memory bandwidth**, il tasso col quale la device memory viene acceduta da un SM, misurata in bytes per unità di tempo

Memory bandwidth:

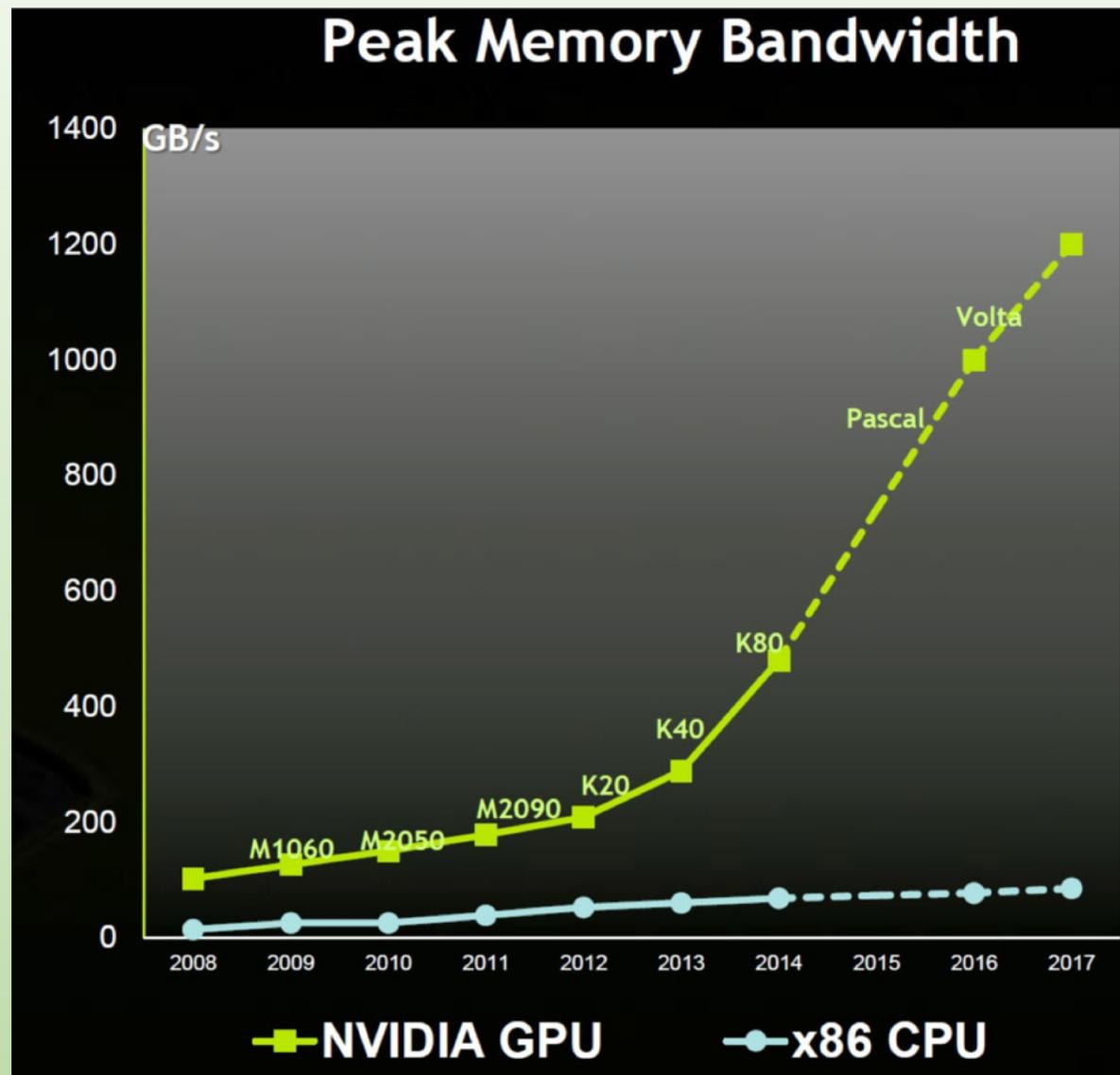
- Theoretical bandwidth (es. Fermi M2090 è pari a 177.6 GB/s)
- Effective bandwidth:

$$\text{effective bandwidth (GB/s)} = \frac{(\text{bytes read} + \text{bytes written}) \times 10^{-9}}{\text{time elapsed}}$$

Esempio: per la copia di una matrice 2048×2048 contenente interi a 4-byte si ricava

$$\text{effective bandwidth (GB/s)} = \frac{2048 \times 2048 \times 4 \times 2 \times 10^{-9}}{\text{time elapsed}}$$

GPU vS CPU (bandwidth)



Migliorare le prestazioni

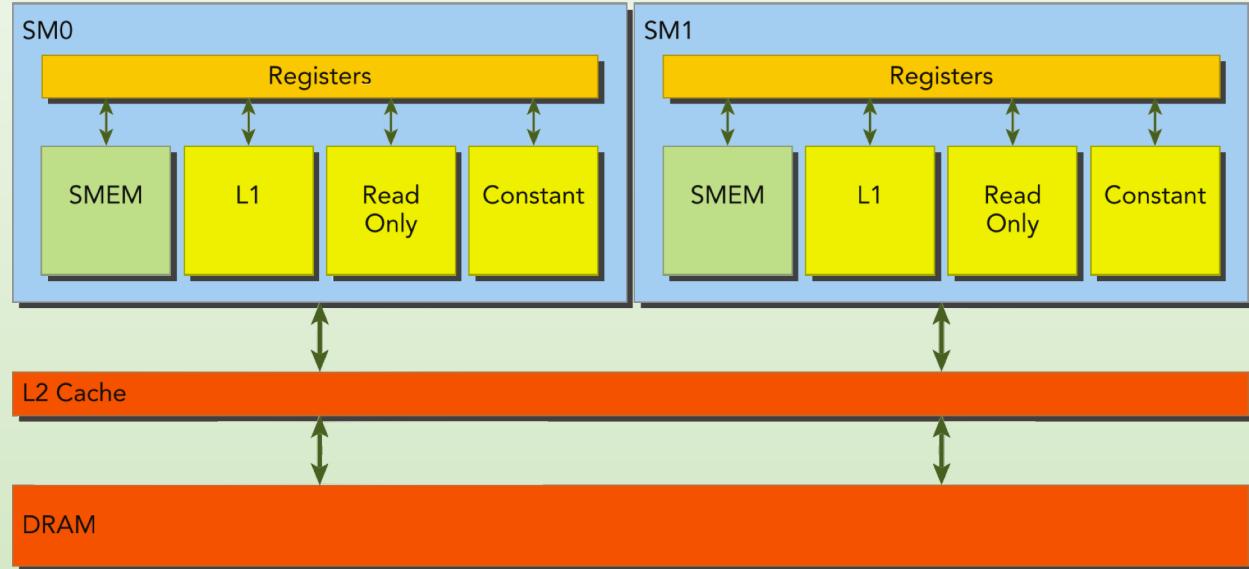
- ✓ Per migliorare prestazioni in lettura e scrittura occorre rammentare che:
 - le **istruzioni** vengono eseguite a livello di **warp** e gli **accessi in memoria** dipendono dalle operazioni svolte nel **warp**
 - per un dato indirizzo si esegue un'operazione di **loading** o **storing** (gestione diversa)
 - Cooperativamente, i 32 thread presentano una **singola** richiesta di accesso che viene servita da **una** o **più transazioni** in memoria
- ✓ Proprio in base a come sono distribuiti gli indirizzi di memoria, gli accessi alla stessa possono essere classificati in **pattern** distinti

Pattern di accesso alla global memory

- ✓ Le applicazioni GPU tendono (a volte) ad essere limitate dalla **memory bandwidth**
- ✓ **Massimizzare** l'effettiva global memory bandwidth è un passo fondamentale – l'opposto potrebbe vanificare altri tipi di ottimizzazioni di prestazioni, per es. a livello kernel

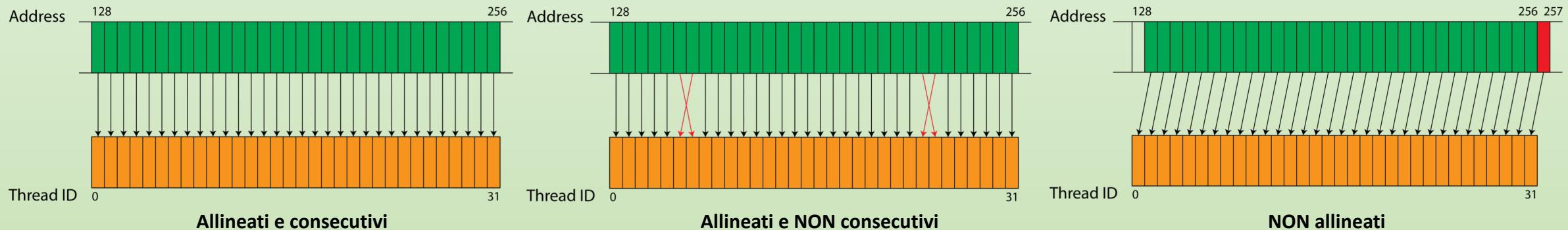
Schema delle cache

- ✓ Global memory = **spazio logico** acceduto dai kernel
- ✓ Global memory = **spazio fisico** la DRAM
- ✓ Le richieste vengono esaudite o dalla **DRAM** o dalle memorie **on-chip** degli SM
- ✓ Le transazioni sono da **32** o da **128 byte**
- ✓ Tutti gli accessi passano attraverso la **cache L2**
- ✓ Molti passano dalla **cache L1** (dipende da arch.)
- ✓ Se entrambe usate gli accessi sono a **128** se solo la L2 è usata gli accessi sono a **32** byte
- ✓ Per le architetture che usano la cache L1 per accessi alla global memory, le cache L1 possono essere esplicitamente **abilitate** o **disabilitate** a tempo di **compilazione**
- ✓ Una cache L1 ha bus di **128 linee** che si mappano su segmenti allineati di **128-byte** in device memory
- ✓ Se un thread in un warp richiede un valore allocato in 4-byte, questo comporta un accesso pieno di 128 byte di dati per ogni richiesta che mappa perfettamente con il numero di linee di cache



Accessi allineati e coalescenti

- ✓ **Accessi a memoria efficienti:**
Combinare in una unica transazione accessi multipli a memoria allineati e coalescenti
- ✓ **Accesso allineato (in CUDA):**
Quando il primo indirizzo della transazione è un **multiplo pari** della **granularità** della cache che viene usata per servire la transazione (32 byte per la cache L2 o 128 byte per la cache L1)
- ✓ **Accesso coalescente (in CUDA):**
Quando tutti i 32 thread in un warp accedono a un **blocco contiguo** di memoria



Esempio:

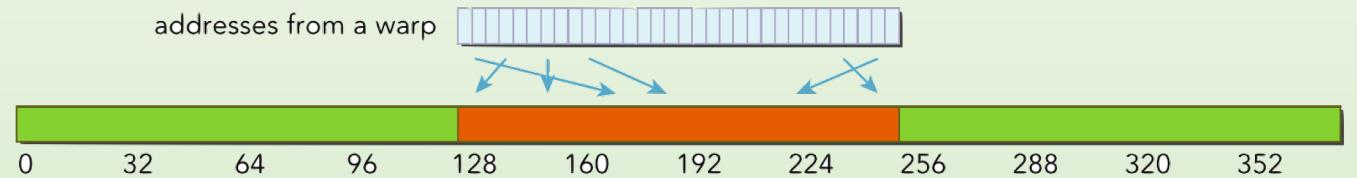
Se ogni thread accede a un **float** in singola precisione o a un **int** da 4 byte di memoria, accesso coalescente significa che il warp ottiene $32 \times 4 = 128$ byte in una sola transazione

Chached loads (L1)

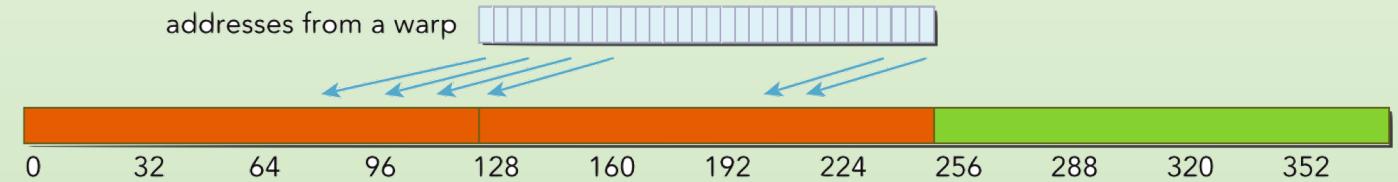
Accessi allineati e coalescenti (100% eff)



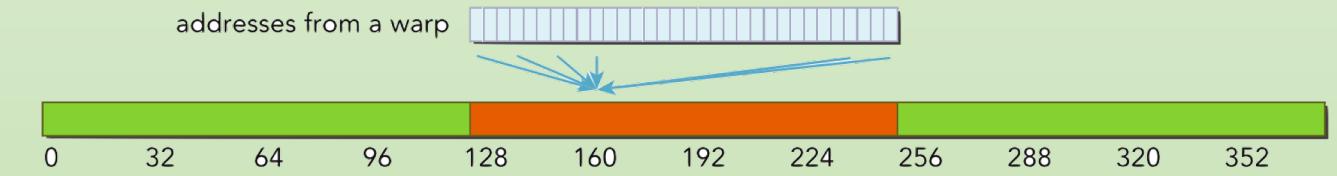
Accessi allineati (100% eff)



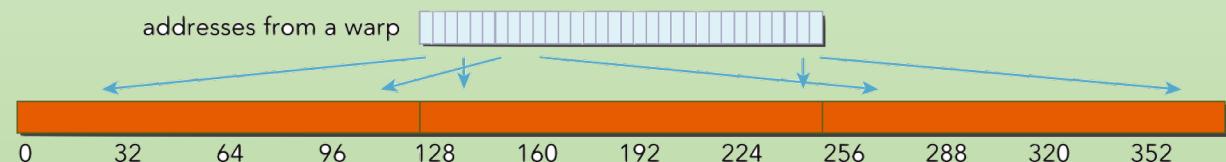
Non allineati 2 transizioni (50% eff)



Tutti chiedono un solo ind 1 transizione (100% eff)

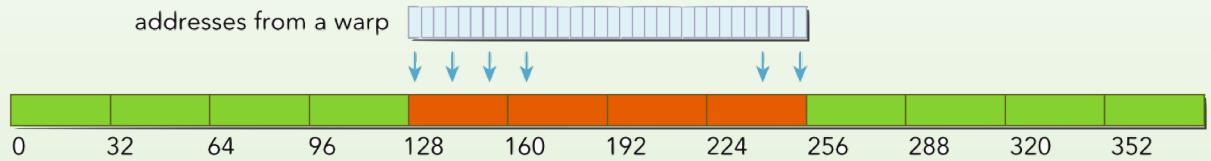


Caso peggiore N <= 32 transazioni

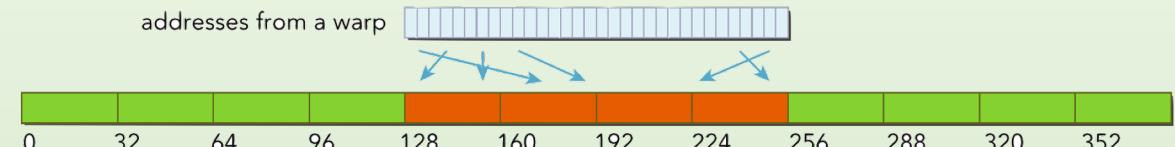


Unchached loads (no L1)

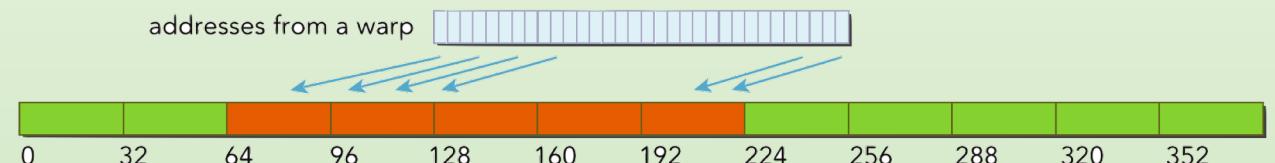
Accessi allineati e coalescenti (100% eff)



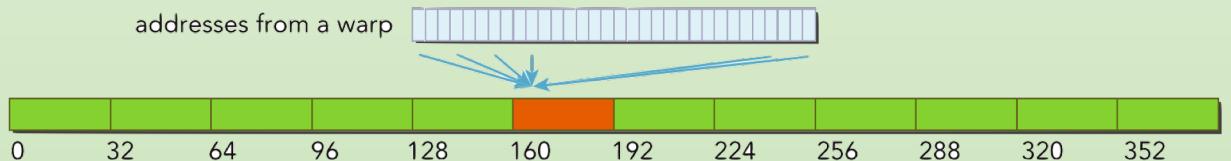
Accessi allineati no spreco del bus (100% eff)



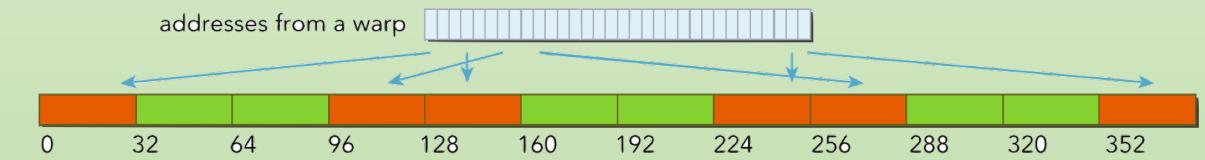
Non allineati sono richiesti 6 blocchi 2 transizioni (80% eff)



Situazione ancora efficiente (sottosfruttato i bus)



Caso peggiore ma meno del caso cached

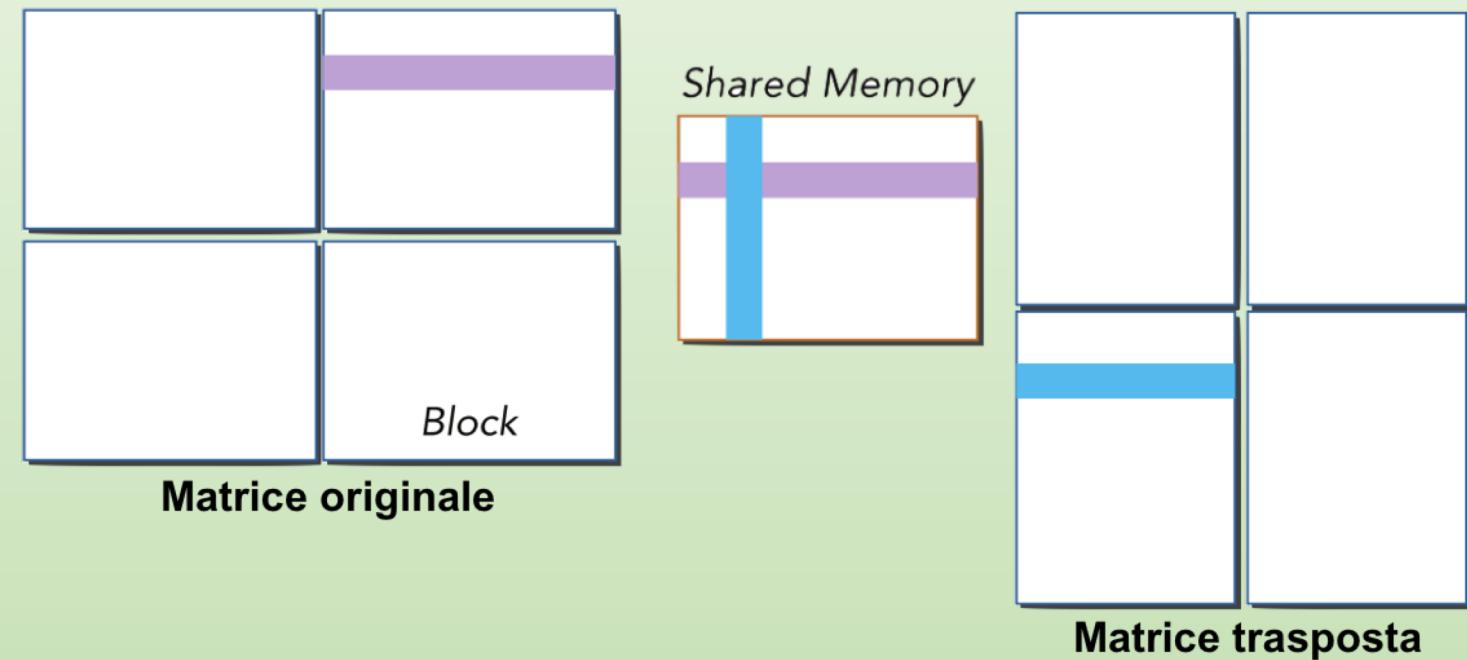


Matrice trasposta con shared memory

La shared memory qui viene usata per **evitare accessi non-coalescenti** alla global memory
(normalmente coalescente in lettura –reads – non coalescente in scrittura – stores - no)

PASSI

1. Leggere una riga blocco x blocco
(suddivisa in warp) dalla global
memory
2. Salvare la riga in una riga di shared
memory
3. leggere una colonna in shared
memory e scrivere una riga in
global memory



Esempio: matrice trasposta

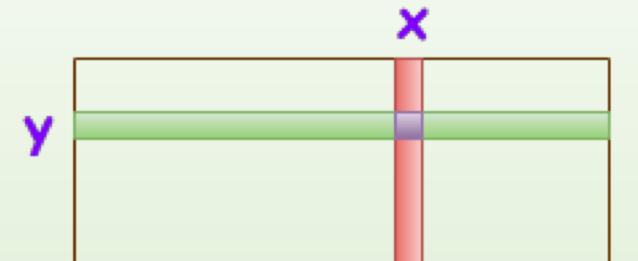
- ✓ Algoritmo naïve:

```
// macro x conversione indici lineari
#define INDEX(n, m, stride) (n * stride + m)

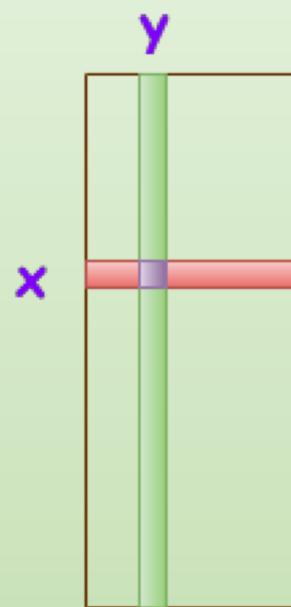
// kernel naive che legge dati e scrive dati dalla global memory
__global__ void naiveGmem(float *out, float *in, const int nrows, const int ncols) {
    // coordinate matrice (x,y)
    unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;
    unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
    // transpose with boundary test
    if (y < nrows && col < ncols)
        out[INDEX(x, y, nrows)] = in[INDEX(y, x, ncols)];
}
```

- ✓ Come utilizzare la shared memory?

- Quali sono i vantaggi?
- Come può favorire letture e scritture coalescenti in global memory?



$$\text{idx}_O = y \times \text{ncols} + x$$



$$\text{idx}_T = x \times \text{nrows} + y$$

Codice CUDA x caricare la SMEM₍₁₎

Dimensione
2D di blocco

Dichiarazione mem
shared di taglia pari al
blocco thread

Il warp scrive i dati nella shared
memory in row-major ordering
evitando bank conflict sulle scritture
Ogni warp fa una lettura coalescente
dei dati in global memory

Sincronizzazione
dei thread

```
// Dimensione del blocco
#define BDIMX 16
#define BDIMY 16

// macro x conversione indici lineari
#define INDEX(row, col, stride) (row * stride + col)

__global__ void transposeSmem(float *out, float *in, int nrows, int ncols) {
    // shared memory statica
    __shared__ float tile[BDIMY][BDIMX];

    // coordinate matrice originale: elem. Arow,col
    unsigned int row = blockDim.y * blockIdx.y + threadIdx.y;
    unsigned int col = blockDim.x * blockIdx.x + threadIdx.x;

    // indice lineare della mat nella global memory
    unsigned int offset = INDEX(row, col, ncols);

    // trasferimento dati dalla global memory alla shared memory
    if (row < nrows && col < ncols)
        tile[threadIdx.y][threadIdx.x] = in[offset];
    // thread synchronization
    __syncthreads();
}
```

Dettagli nella trasposizione

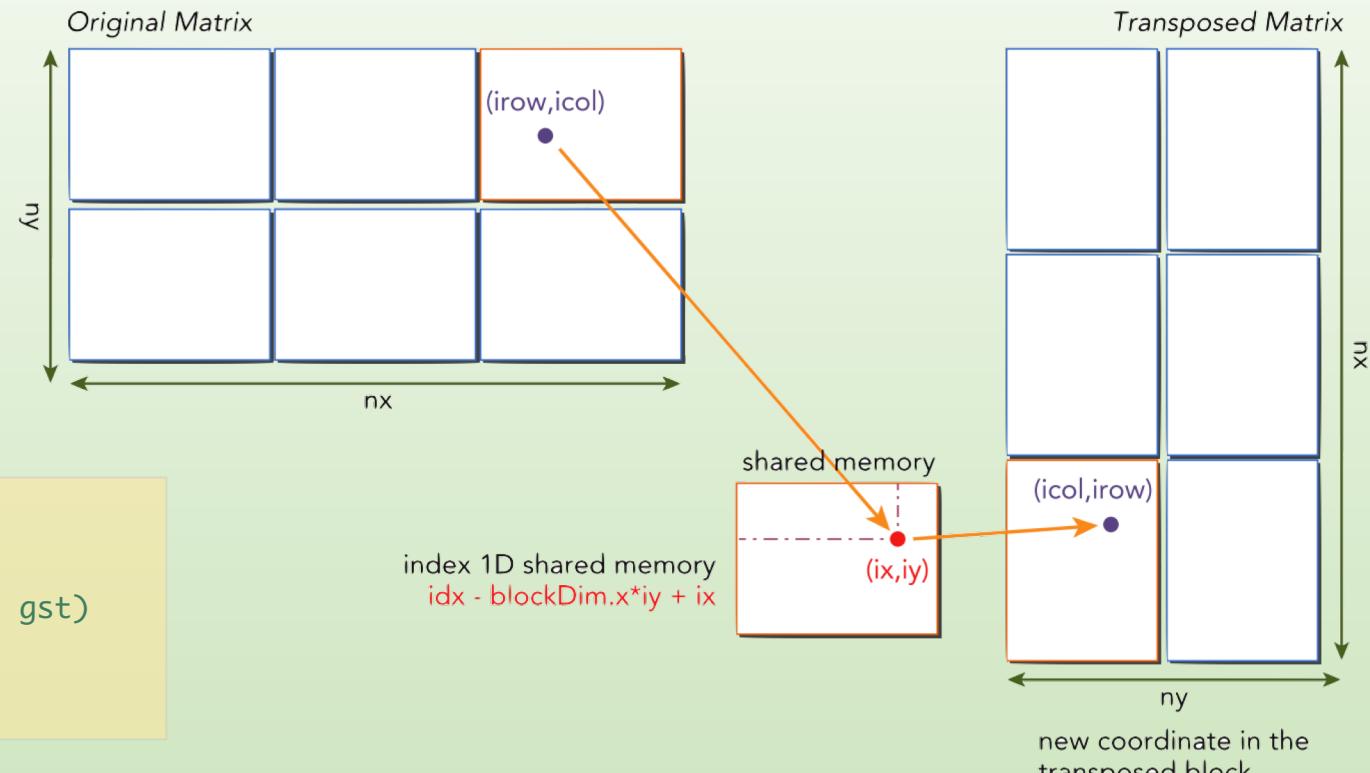
4. Calcolo degli indici per la scrittura...

```
// thread index in transposed block  
unsigned int bidx, irow, icol;  
bidx = threadIdx.y * blockDim.x + threadIdx.x;  
irow = bidx / blockDim.y;  
icol = bidx % blockDim.y;
```

```
// 1. swap coordinate x-y  
// 2. swap thread x-y assignment  
// col ha threadIdx.x contigui (coalesced gst)  
row = blockIdx.x * blockDim.x + irow;  
col = blockIdx.y * blockDim.y + icol;
```

```
// indice lineare globale di memoria nella matrice trasposta  
unsigned int transposed_offset = INDEX(row, col, nrows);
```

```
// NOTA: controlli invertiti nelle dim riga colonna  
if (row < ncols && col < nrows)  
    out[transposed_offset] = tile[icol][irow];
```



Codice CUDA x scrivere da SMEM₍₂₎

Indice lineare del blocco della matrice trasposta e sue coordinate

Indici di colonna e riga della matrice trasposta

Indice lineare della matrice trasposta

```
// thread index in transposed block
unsigned int bidx, irow, icol;
bidx = threadIdx.y * blockDim.x + threadIdx.x;
irow = bidx / blockDim.y;
icol = bidx % blockDim.y;

// NOTE - need to transpose row and col on block and thread-block level:
// 1. swap blocks x-y
// 2. swap thread x-y assignment (irow and icol calculations above)
// note col still has continuous threadIdx.x -> coalesced gst
col = blockIdx.y * blockDim.y + icol;
row = blockIdx.x * blockDim.x + irow;

// linear global memory index for transposed matrix
// NOTE nrows is stride of result, row and col are transposed
unsigned int transposed_offset = INDEX(row, col, nrows);

// NOTE invert sizes for write check
if (row < ncols && col < nrows)
    out[transposed_offset] = tile[icol][irow];
}
```

Struct of arrays (SOA) vs array of structs (AoS)

Meglio un array di strutture (AoS) o una struttura di array (SoA)?

```
struct innerStruct {  
    float x;  
    float y;  
};  
.  
.  
.  
struct innerStruct AoS[N];
```

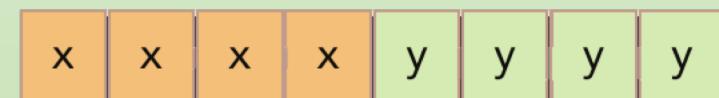
```
struct innerArray {  
    float x[N];  
    float y[N];  
};  
.  
.  
.  
struct innerArray SoA;
```

AoS memory layout



thread ID t0 t1 t2 t3

SoA memory layout



t0 t1 t2 t3

Semplice esempio

Implementazione SoA

```
--global__ void testInnerArray(InnerArray *data, InnerArray *result, const int n) {  
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;  
  
    if (i < n) {  
        float tmpx = data->x[i];  
        float tmpy = data->y[i];  
  
        tmpx += 10.f;  
        tmpy += 20.f;  
        result->x[i] = tmpx;  
        result->y[i] = tmpy;  
    }  
}
```

Implementazione AoS

```
--global__ void testInnerStruct(innerStruct *data, innerStruct * result, const int n) {  
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;  
  
    if (i < n) {  
        innerStruct tmp = data[i];  
        tmp.x += 10.f;  
        tmp.y += 20.f;  
        result[i] = tmp;  
    }  
}
```

Misurare l'efficienza in
lettura e scrittura

```
$ nvprof --metrics gld_efficiency,gst_efficiency ./app
```

***-efficiency:** Ratio of requested global memory load/store throughput to required global memory load throughput expressed as percentage”

Esercitazione

Elaborazione immagini:

Scrivere un programma CUDA per i kernel sotto definiti mettendo in evidenza vantaggi e svantaggi nell'organizzare i dati nel formato AoS o SoA .

Usare nell'analisi di prestazioni:

1. i tempi di esecuzione
2. Profili di storage con le metriche **global load efficiency** e **globalstore efficiency**

```
/*
 * Riscalza l'immagine al valore massimo [max] fissato
 */
__global__ void rescaleImg<tipo immagine> *img, const int max, const int n)
```

```
/*
 * cancella un piano dell'immagine [plane = 'r' o 'g' o 'b'] fissato
 */
__global__ void deletePlane(<tipo immagine> *img, const char plane, const int n)
```

Esercitazione

ESERCITAZIONE:

Usando il codice CUDA per il prodotto di matrici stimare su lagrange il vantaggio dell'uso della pinned memory per il trasferimento delle 3 matrici coinvolte nella moltiplicazione.

Usare varie taglie della matrice:

- Matrici di 10 MByte
- Matrici di 100 MByte
- Matrici di 1000 MByte (1GB)