

GPU Computing

Laurea Magistrale in Informatica - AA 2019/20

Docente **G. Grossi**

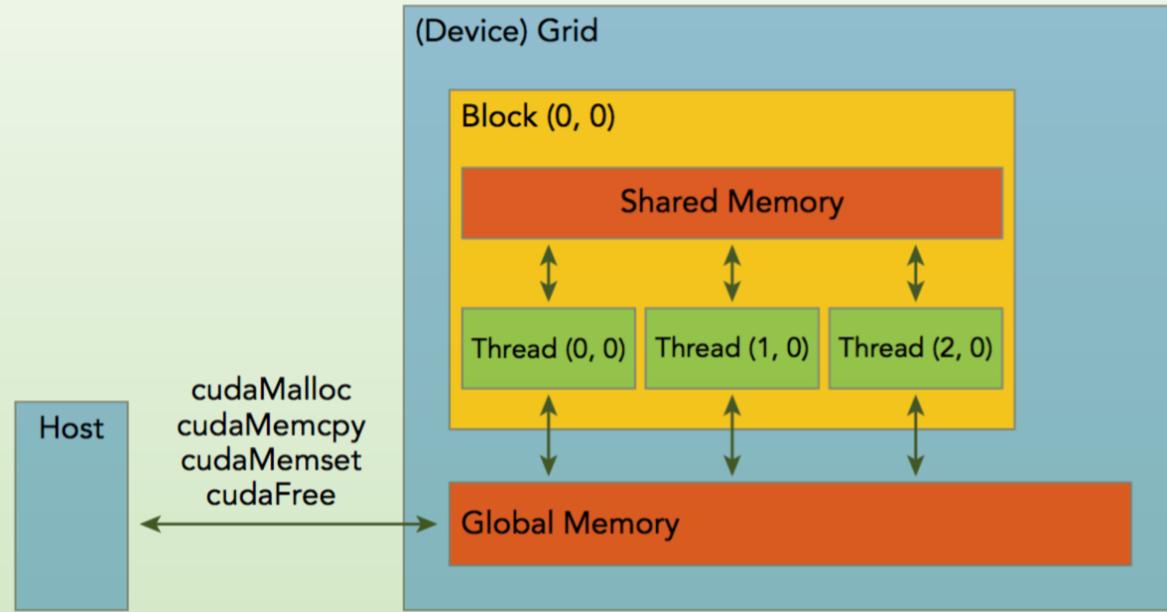
Lezione 6 – la global memory

Sommario

- ✓ Allocazione e deallocazione in global memory
- ✓ Allocazione statica e dinamica
- ✓ Pinned, unified, zero-copy memory
- ✓ Memory bandwidth
- ✓ Pattern di accesso alla global memory
- ✓ Esercitazione su convoluzione e matrice trasposta

Gestione della Memoria

- ✓ When you use **cudaMemcpy** to copy data between the host and device, **implicit synchronization** at the host side is performed and the host application must wait for the data copy to complete



- ✓ Modifiche successive del modello di indirizzamento sincrono:
 - **pinned** (page-locked) per copie asincrone
 - **zero-copy** con memoria pinned
 - **UVA** (Unified Virtual Addressing)
 - **Unified Memory**

Allocazione dinamica

prototipo-> `cudaError_t cudaMalloc (void** devPtr, size_t size)`

- `devPtr` è un puntatore a un puntatore in Global memory del device

prototipo-> `cudaError_t cudaMemcpy (void* dst, const void* src, size_t count, cudaMemcpyKind kind)`

Kind-->

`cudaMemcpyHostToHost, cudaMemcpyHostToDevice`
`cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice`

- Questa funzione esibisce un comportamento sincrono che blocca il programma host fino a che il trasferimento non viene completato
- Per capire se `destination` e `src` sono puntatori a memoria CPU o GPU si guarda la variabile `kind`

Errore--> `cudaError_t cudaFree (void* devPtr)`

- libera la global memory puntata da `devPtr`

Inizializzazione

- ✓ I valori contenuti nella memoria allocata non sono ‘azzerati’
- ✓ Si possono inizializzare con dati provenienti da host (**cudaMemcpy**)
- ✓ oppure inizializzare con un valore specifico:

prototipo -> `cudaError_t cudaMemset (void* devPtr, int value, size_t count)`

➤ Funzione che assegna il valore **value** a tutti gli indirizzi contenuti nel blocco di memoria

- ✓ La memoria allocata è **opportunamente allineata** per ogni tipo di variabile
- ✓ La **cudaMalloc** restituisce **cudaErrorMemoryAllocation** in caso di fallimento

Variabili globali statiche

✓ STATICÀ:

- 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
cudaMalloc((float**) &dev, 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**

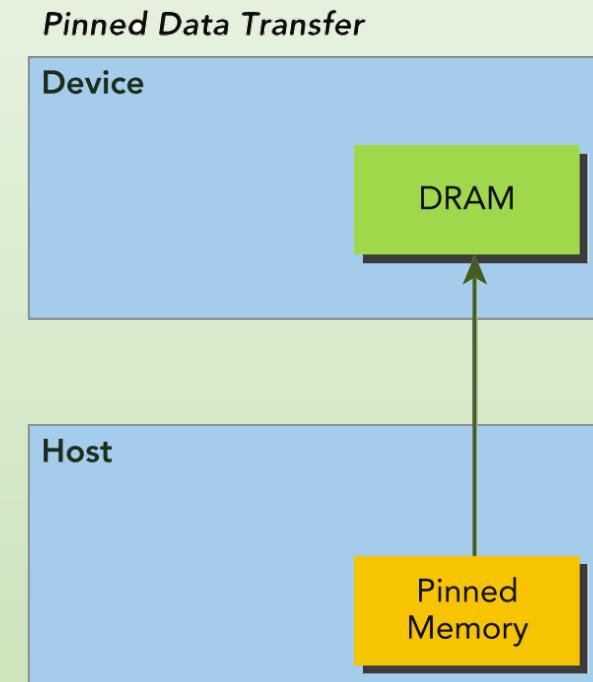
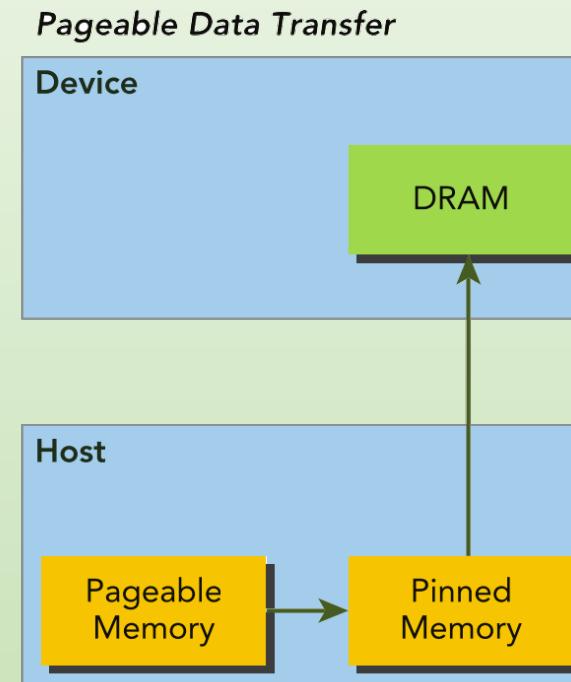
```
 . . .
// using pointer
count = 0;
int *devptr = NULL;
cudaGetSymbolAddress((void**)&devptr, devCount);
cudaMemcpy(devptr, &count, sizeof(int), cudaMemcpyHostToDevice);
// kernel launch
incGlobalVariable<<<1, 1>>>();
cudaMemcpy(&count, devptr, sizeof(int), cudaMemcpyDeviceToHost);
printf("Host: number 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 alloca 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
- ✓ La modalità di **accesso** è **asincrona!**
- ✓ 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 di memoria 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 verificato

```
// get device properties
cudaDeviceProp deviceProp;
cudaGetDeviceProperties(&deviceProp, dev);

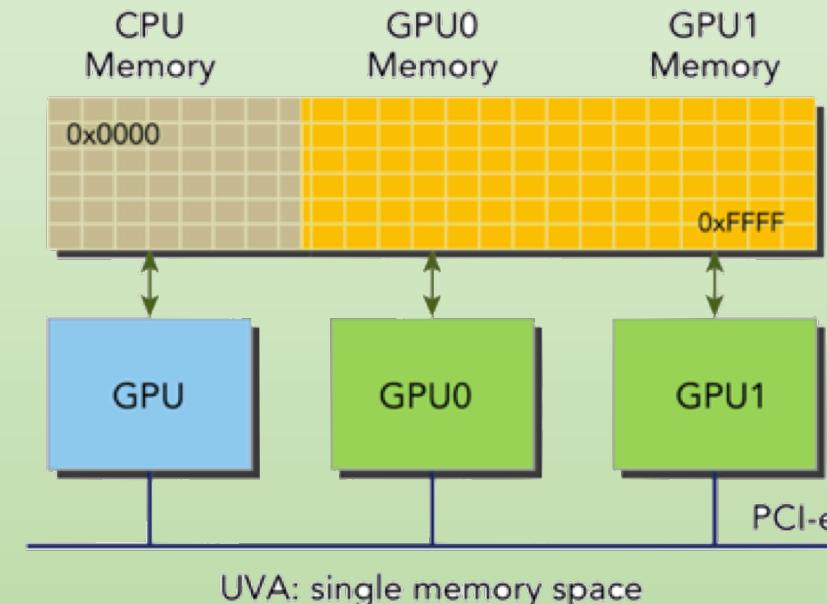
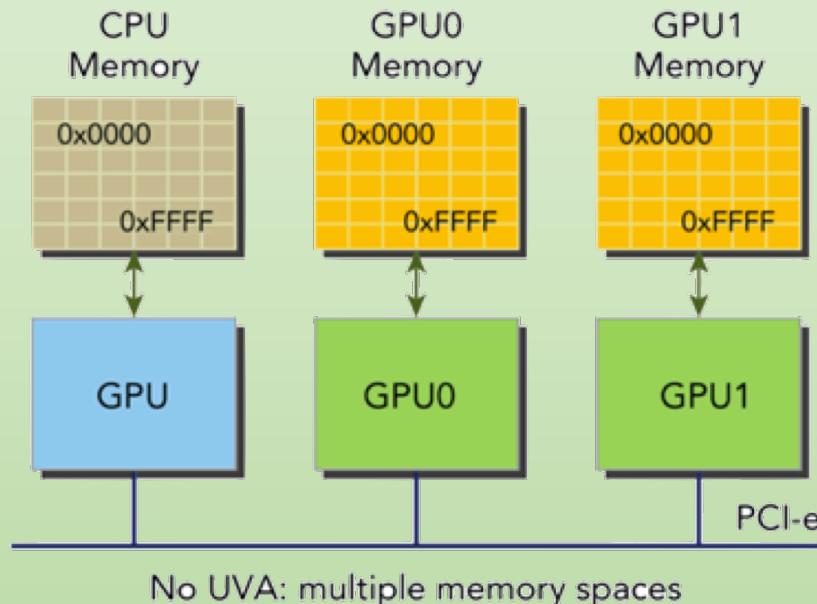
// check if support mapped memory
if (!deviceProp.canMapHostMemory) {
    printf("Device %d does not support mapping CPU host memory!\n", dev);
}
```

(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**
- ✓ Si possono avere **problemi** con le **operazioni atomiche** perché non sono atomiche per l'host

Unified Virtual Addressing (UVA)

- ✓ Dalla Compute capability 2.0 in poi viene supportato un modello di indirizzamento chiamato **Unified Virtual Addressing (UVA)**
- ✓ Introdotto in **CUDA 4.0** e supportato su architetture linux **64-bit**
- ✓ **host memory e device memory** condividono un **singolo virtual address space**



Unified Virtual Addressing (UVA)

- ✓ La memoria pinned host allocata con **cudaHostAlloc** ha stessi **puntatori host and device**
- ✓ Si passa il **puntatore** restituito **direttamente** al kernel! (zero-copy semplificata)

```
// allocate zero copy memory
CHECK(cudaHostAlloc((void **) &h_A, nBytes, cudaHostAllocMapped));
CHECK(cudaHostAlloc((void **) &h_B, nBytes, cudaHostAllocMapped));
// initialize data at host side
initialData(h_A, nElem);
initialData(h_B, nElem);
// execute kernel with zero copy memory
sumArraysZeroCopy<<<grid, block>>>(d_A, d_B, d_C, nElem);
```

Unified Memory

- ✓ Modello di programmazione e di memoria più semplice :
 - Introdotto da CUDA 6.0
 - **Single-pointer-to-data** accessibile ovunque da CPU e GPU
 - Elimina l'uso di **cudaMemcpy()**
 - Semplifica il porting del codice
- ✓ Performance garantite dalla località dei dati:
 - migrazione dati accedendo direttamente (trasparentemente) all'host
 - mantiene la coerenza globale
 - consente l'uso manuale di **cudaMemcpyAsync()**

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

Gestione

- ✓ **Managed memory** refers to **Unified Memory** allocations that are automatically managed by the underlying system and is interoperable with device-specific allocations, such as those created using the `cudaMalloc` routine
- ✓ You can use both types of memory in a kernel:
 - **managed** memory that is **controlled** by the **system**
 - **un-managed** memory that must be **explicitly allocated** and transferred by the application
- ✓ All CUDA operations that are valid on device memory are also valid on managed memory
- ✓ You can allocate managed memory dynamically using the following CUDA runtime function:

prototipo-> `cudaError_t cudaMallocManaged(void **devPtr, size_t size, unsigned int flags=0)`

- ✓ statically declare a device variable as a managed variable by `__managed__` annotation

referenced directly from either
host or device code:

```
__device__ __managed__ int x; // Unified memory
```

Mutua esclusione

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;
}
```

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:

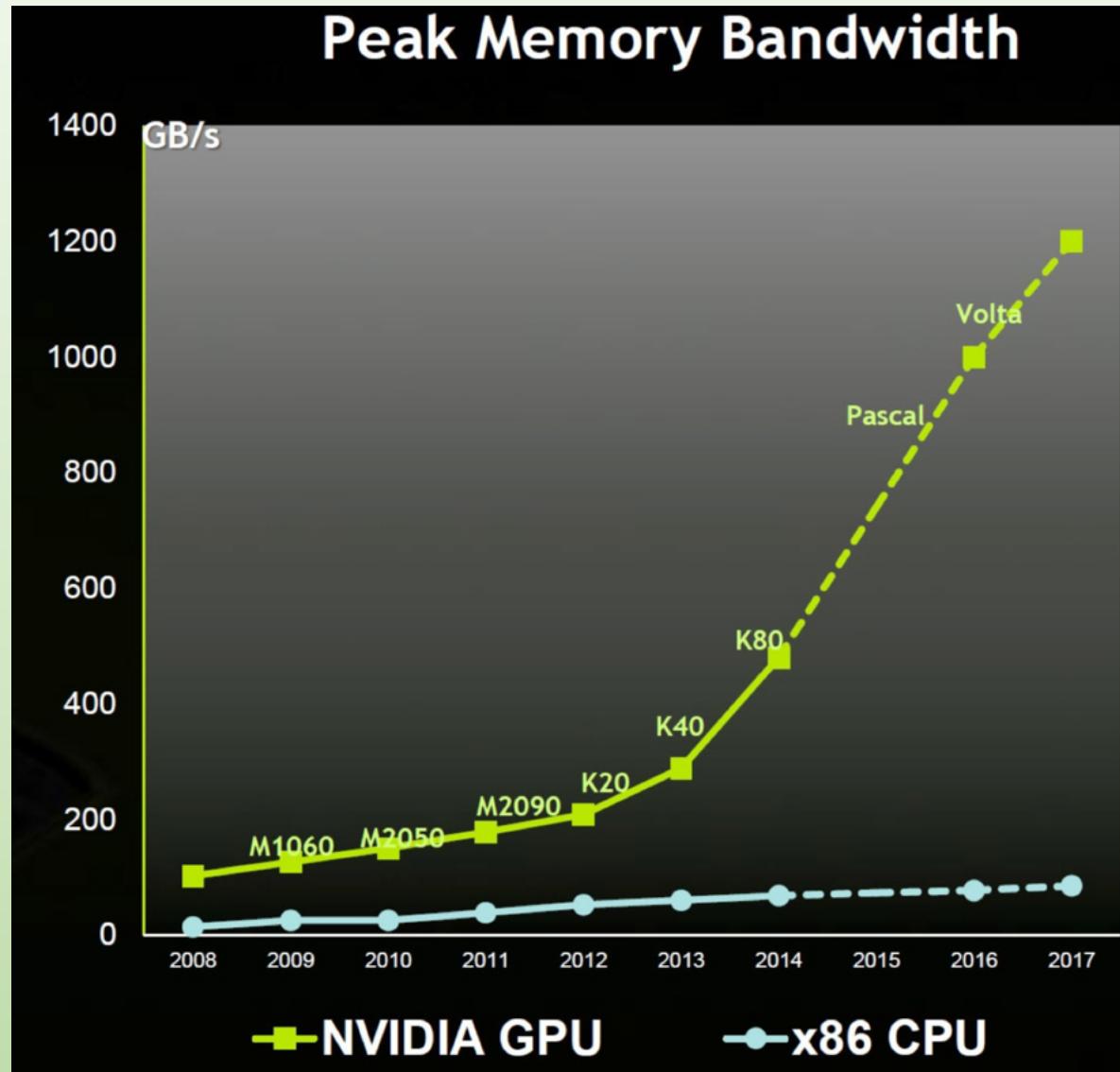
- Theoretical bandwidth (es. Fermi M2090 è pari a 177.6 GB/s)
- Effective bandwidth:

$$\text{effective bandwidth} = \frac{(\#\text{byte letti} + \#\text{byte scritti}) \times 10^9}{\text{tempo}}$$

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

$$\text{eff bandwidth} = 2048 \times 2048 \times 2 \times 4 \times 10^9 / \text{tempo}$$

GPU vS CPU (bandwidth)



Migliorare le prestazioni

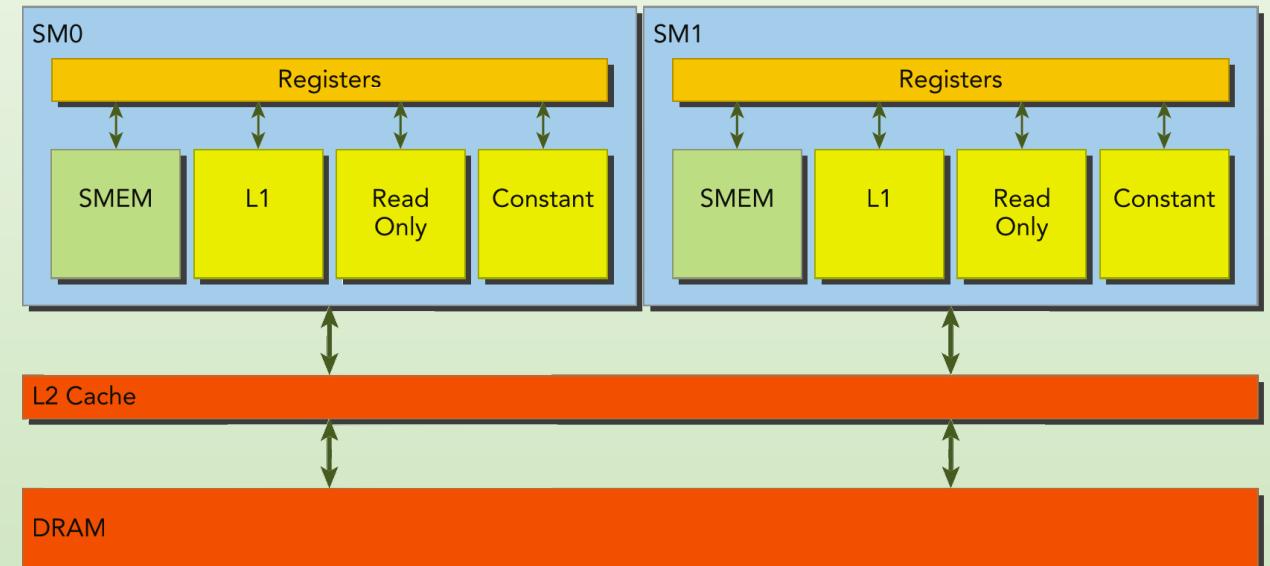
- ✓ Per migliorare prestazioni in lettura e scrittura occorre rammentare che:
 - le **istruzioni** vengono eseguite a livello di **warp** e gli **accessi in memoria** dipendono dalle operazioni svolte nel **warp**
 - per un dato indirizzo si esegue un'operazione di **loading** o **storing** (gestione 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 (throughput)** è un passo fondamentale – l'opposto potrebbe vanificare altri tipi di ottimizzazioni di prestazioni, per es. a livello kernel
- ✓ In generale per rendere **efficienti le tranzazioni** in memoria:
 - **Minimizzare il numero di transazioni** per servire il **massimo numero di accessi**
 - Il numero di **transazioni** e il livello di **throughput** che si ottiene varia con la **device compute capability**

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

Accessi allineati e coalescenti

- ✓ **Accessi a memoria efficienti:**

Combinare in una unica transazione accessi multipli a memoria allineati e coalescenti

- ✓ **Accesso allineato (in CUDA):**

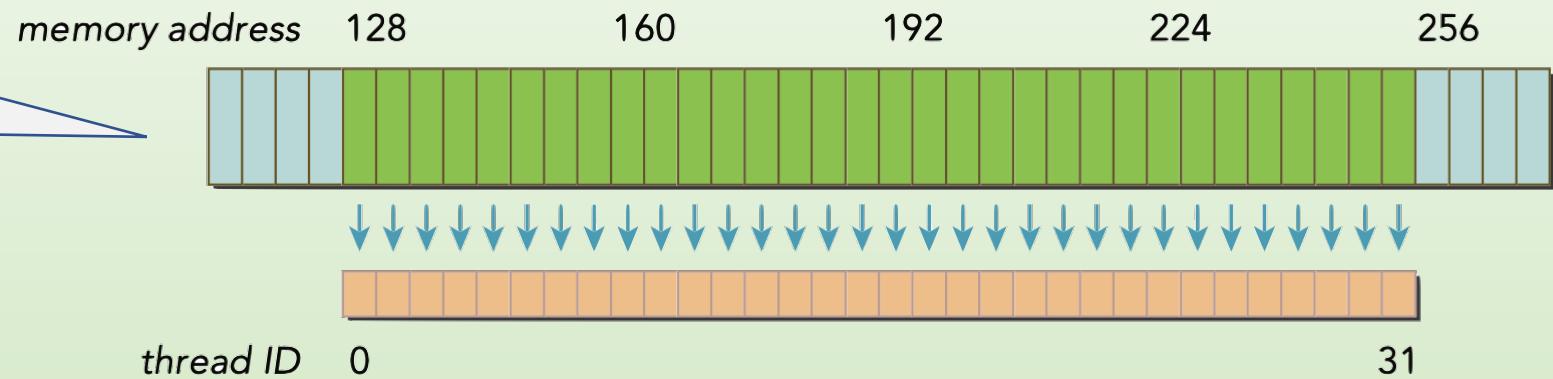
Quando il primo indirizzo della transazione è un **multiplo pari** della **granularità** della cache che viene usata per servire la transazione (32 byte per la cache L2 o 128 byte per la cache L1)

- ✓ **Accesso coalescente (in CUDA):**

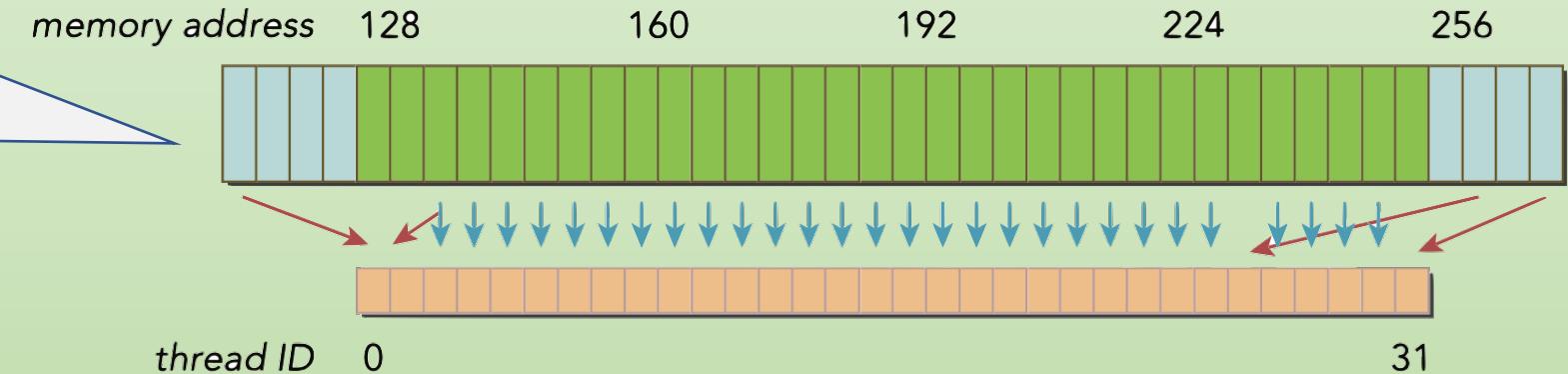
Quando tutti i 32 thread in un warp accedono a un **blocco contiguo** di memoria

Esempio

Allineato e
coalescente
(una sola
transazione di
mem)



NON allineato e
NON coalescente
(necessarie 3
transazioni di
mem)



Global memory reads

- ✓ In un SM i dati seguono **pipeline** attraverso i seguenti **tre cache/buffer paths** dipendentemente da quale tipo di device memory si accede:
 - L1/L2 cache - Constant cache - Read-only cache
- ✓ **L1/L2** cache rappresenta il **default path**... il fatto che un'operazione di load in global memory passi attraverso la cache L1 cache dipende da due:
 - **Device compute capability**
 - **Compiler options**

Esempi

https://docs.nvidia.com/pdf/CUDA_C_Programming_Guide.pdf

FERMI E KEPLER:

On Fermi GPUs (compute capability 2.x) and Kepler K40 or later GPUs (compute capability 3.5 and up) L1 caching of global memory loads can be either enabled or disabled with compiler flags.

By default, the L1 cache is enabled for global memory loads on Fermi devices and disabled on K40 and later GPUs. The following flags inform the compiler to disable the L1 cache:

```
-Xptxas -dlcm=cg
```

With the L1 cache disabled, all load requests to global memory go directly to the L2 cache; when an L2 miss occurs, the requests are serviced by DRAM. Each memory transaction may be conducted by one, two, or four segments, where one segment is 32 bytes. The L1 cache can also be explicitly enabled with the following flag:

```
-Xptxas -dlcm=ca
```

With this flag set, global memory load requests first attempt to hit in L1 cache. On an L1 miss, the requests go to L2. On an L2 miss, the requests are serviced by DRAM. In this mode, a load memory request is serviced by a 128-byte device memory transaction.

Riassumendo...

MEMORY LOAD ACCESS PATTERNS

There are two types of memory loads:

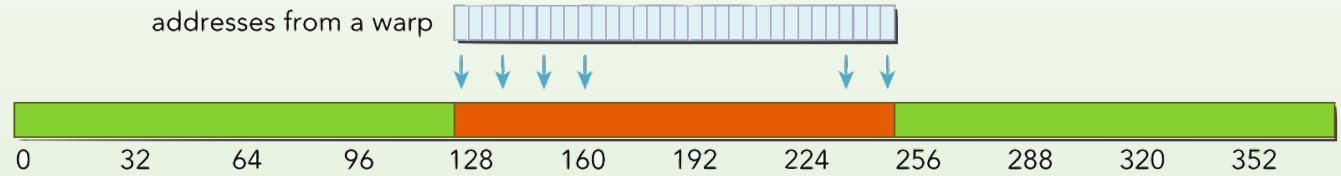
- Cached load (L1 cache enabled)
- Uncached load (L1 cache disabled)

The access pattern for memory loads can be characterized by the following combinations:

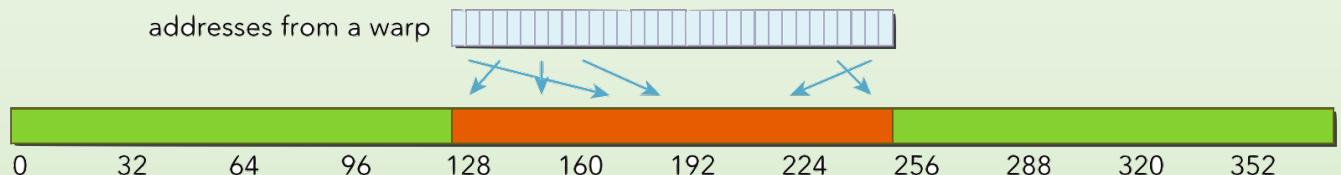
- Cached versus uncached: The load is cached if L1 cache is enabled
- Aligned versus misaligned: The load is aligned if the first address of a memory access is a multiple of 32 bytes
- Coalesced versus uncoalesced: The load is coalesced if a warp accesses a contiguous chunk of data

Chached loads (L1 – transaz. 128 byte)

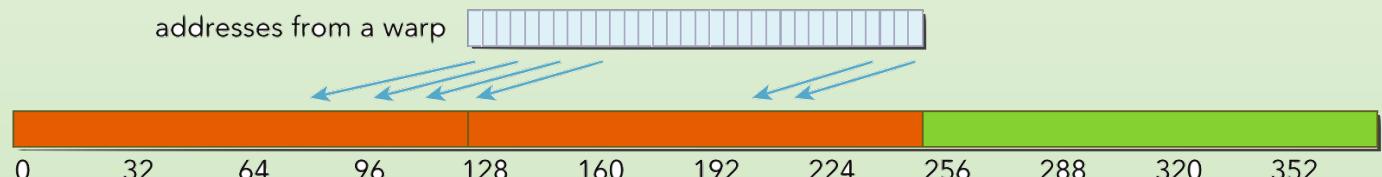
- ✓ Accessi allineati e coalescenti (100% eff. uso del bus)



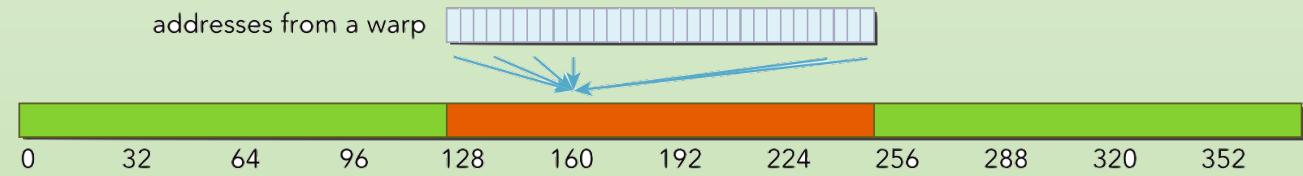
- ✓ Accessi allineati (100% eff del bus) non coalescenti



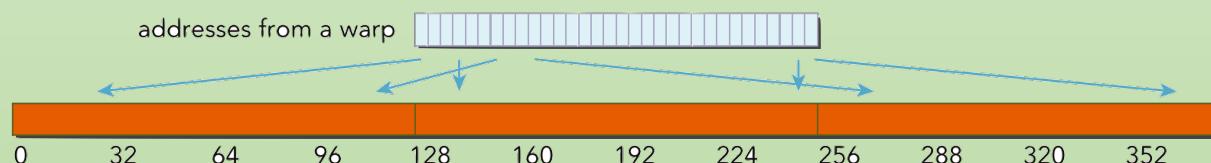
- ✓ Non allineati 2 transizioni (50% eff)



- ✓ Tutti chiedono un solo ind 1 transizione (3,125% eff uso del bus)

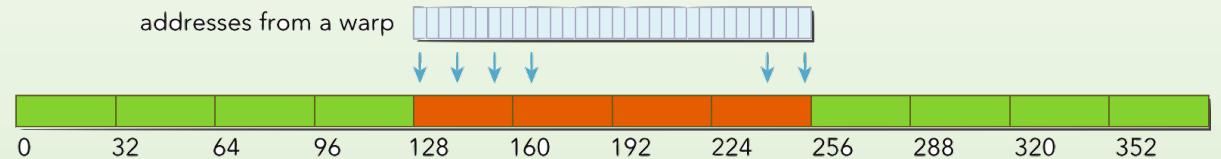


- ✓ Caso peggiore N <= 32 transazioni

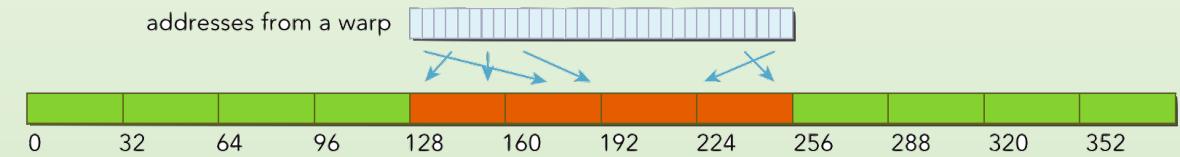


Unchached loads (no L1 – transaz. 32 byte)

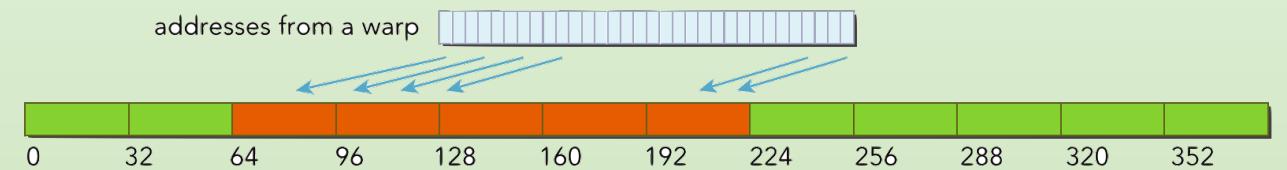
- ✓ Accessi allineati e coalescenti (100% eff)



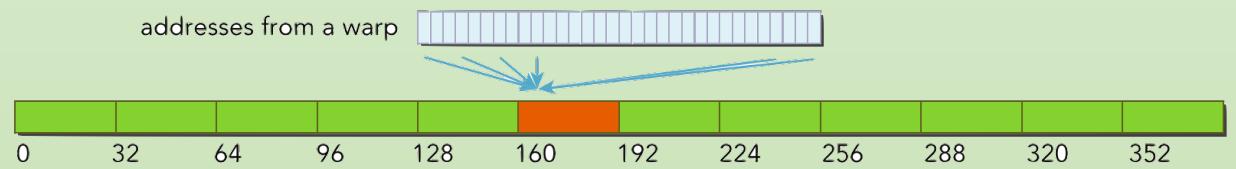
- ✓ Accessi allineati no spreco del bus (100% eff)



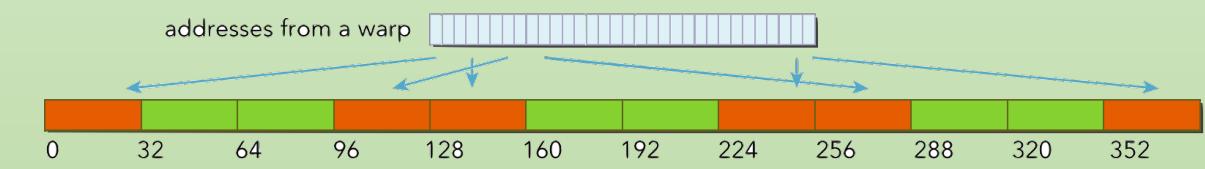
- ✓ Non allineati sono richiesti 5 blocchi (80% eff)



- ✓ Situazione più efficiente che con cahe L1 (12,5 % eff uso del bus)

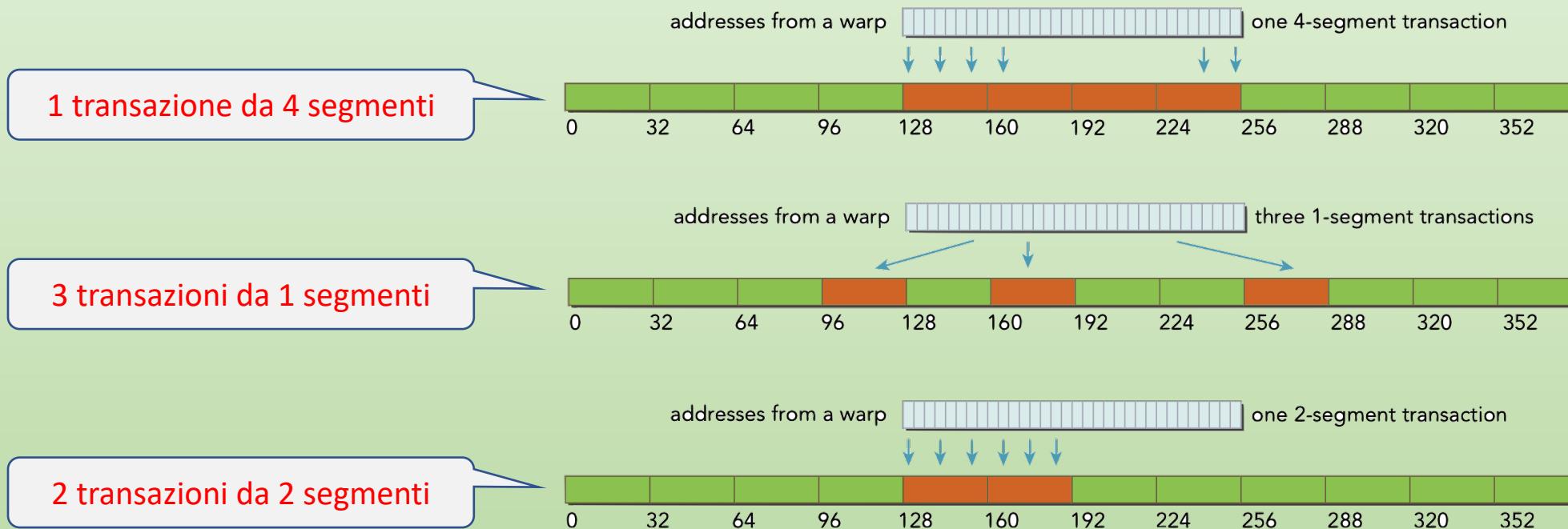


- ✓ Caso peggiore ma meno del caso cached (al più N 32-byte segmenti invece di N 128-byte cache lines)



Global Memory Writes

- ✓ Memory store operations are relatively simple
- ✓ The **L1 cache** is **not used** for store operations
- ✓ Store operations are only **cached** in the **L2 cache** before being sent to device memory
- ✓ **Stores** are performed at a **32-byte segment granularity**
- ✓ Memory transactions can be **one, two, or four segments** at a time



CUDA Zone...

Uso ottimale degli accessi a global memory

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

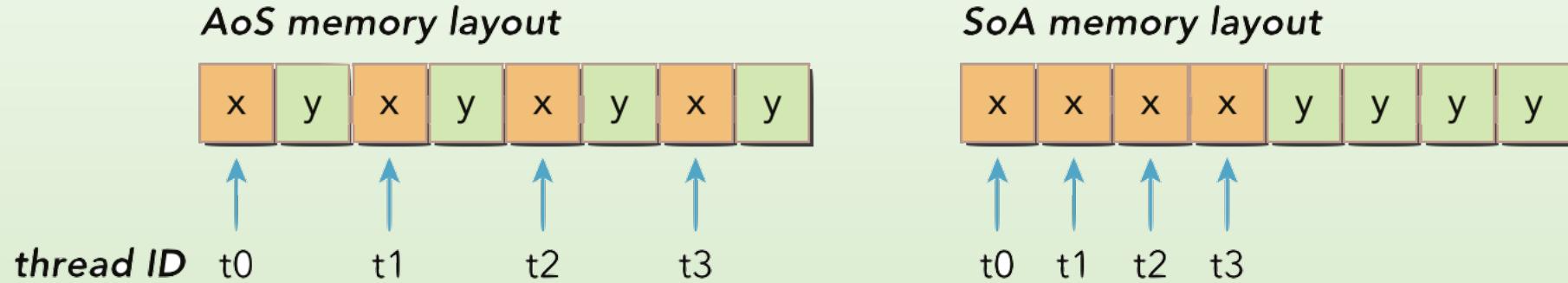
- ✓ Meglio un array di strutture (AoS) o una struttura di array (SoA)?

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

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

- ✓ Come vengono organizzati i dati in memoria fisica?
- ✓ Chi si avvantaggia della località in cache?
- ✓ Allineamento e coalescenza?

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



- ✓ Storing the example data in AoS format on the GPU and performing an operation that only requires the *x* field would result in a 50 percent loss of bandwidth as *y* values are implicitly loaded in each 32-byte segment or 128-byte cache line.
- ✓ An AoS format would also waste L2 cache space on unneeded *y* values.
- ✓ Storing the data in SoA fashion makes full use of GPU memory bandwidth
- ✓ Because there is no interleaving of elements of the same field, the SoA layout on the GPU provides coalesced memory accesses and can achieve more efficient global memory utilization

Misurare efficienza

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”

Matrice trasposta

0	1	2	3
4	5	6	7
8	9	10	11

matrix

0	4	8
1	5	9
2	6	10
3	7	11

transposed

```
void transposeHost(float *out, float *in, int nx, int ny) {  
    for (int iy = 0; iy < ny; ++iy) {  
        for (int ix = 0; ix < nx; ++ix) {  
            out[ix * ny + iy] = in[iy * nx + ix];  
        }  
    }  
}
```

- ✓ **Reads:** accesso per riga... coalescente
- ✓ **Writes:** accesso per colonna nella matrice trasposta... non coalescente

data layout of original matrix

0	1	2	3	4	5	6	7	8	9	10	11
---	---	---	---	---	---	---	---	---	---	----	----

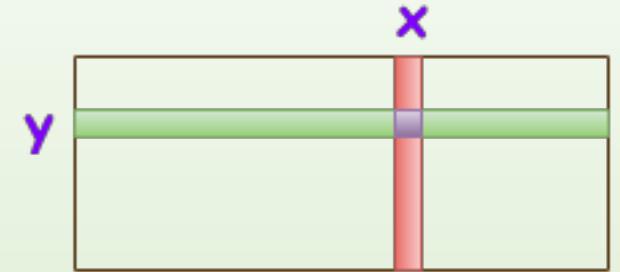
data layout of transposed matrix

0	4	8	1	5	9	2	6	10	3	7	11
---	---	---	---	---	---	---	---	----	---	---	----

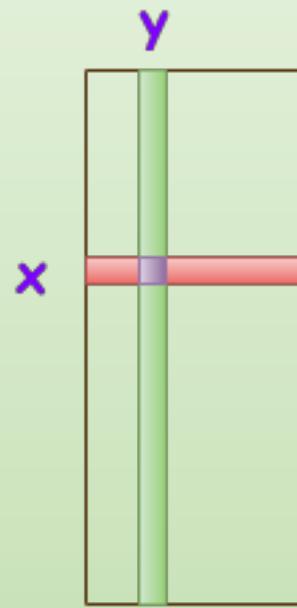
Algoritmo naïve

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

// kernel naïve 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)];
}
```



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



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

- ✓ Come può favorire letture e scritture coalescenti in global memory?
 - Può essere utile la shared memory?
 - Quali sono i vantaggi?

Matrice trasposta con shared memory

- ✓ La shared memory qui viene usata per **evitare accessi non-coalescenti** alla global memory

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

