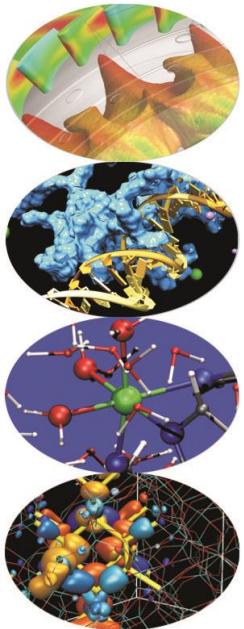




GPU (Graphics Processing Unit) Programming in CUDA

Giorno 2



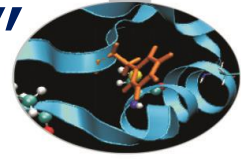


Livelli di Memoria e Coalescenza

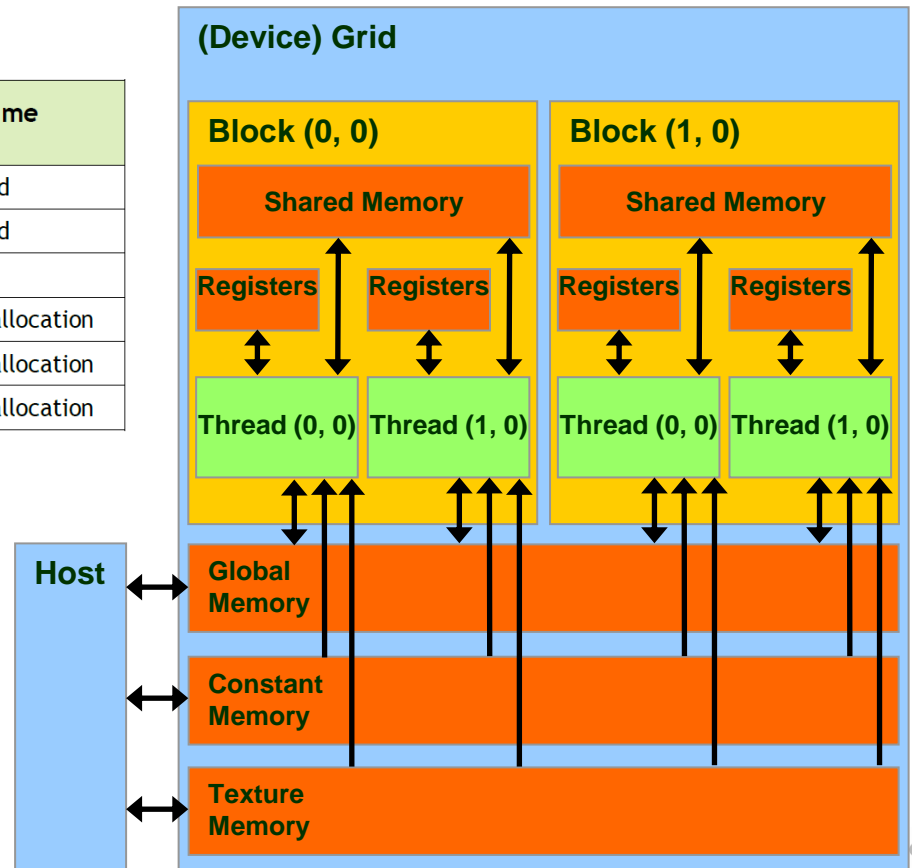
- Global Memory
 - Coalescenza
- Le altre aree di memoria:
 - *Constant*,
 - *Shared*
 - *Cache*

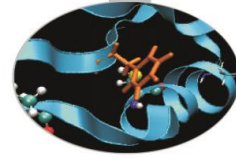


La gerarchia di memoria "classica" in CUDA



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





Definizione di variabili in CUDA

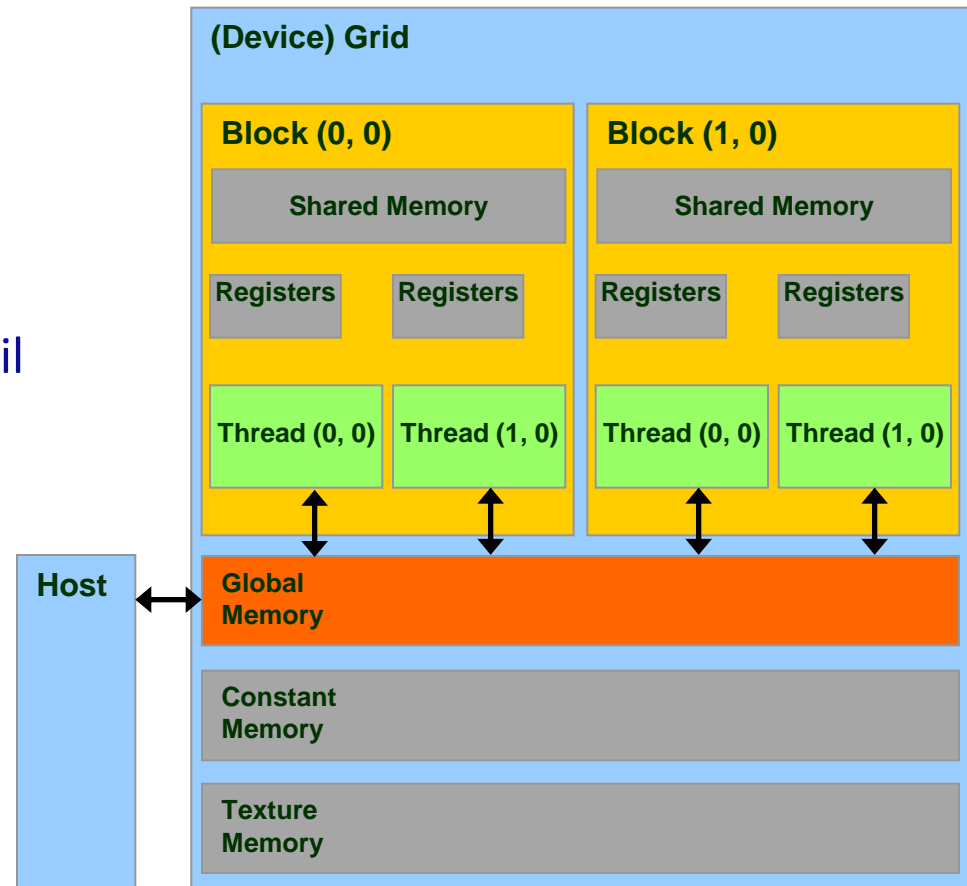
Dichiarazione di variabile	Memoria	Visibilità	Persistenza
<code>__device__ __local__ int LocalVar</code>	Local	Thread	Thread
<code>__device__ __shared__ int SharedVar</code>	Shared	Block	Block
<code>__device__ int GlobalVar</code>	Global	Grid	Application
<code>__device__ __constant__ int ConstantVar</code>	Constant	Grid	Application

- `__device__` è opzionale con `__local__`, `__shared__`, o `__constant__`
- Le variabili automatiche scalari senza qualificatori sono allocate nei registri o nella memoria locale in mancanza di registri
- Gli array la cui dimensione non è conosciuta a priori e le strutture dati di una certa complessità sono allocati nella memoria locale (lenta poiché fa parte della global memory)



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)
 - Throughput: fino a 144-177 GB/s



Allocazione della Memoria Globale



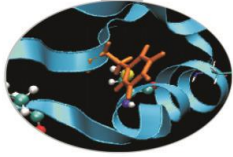
Per allocare una variabile nella memoria globale

```
__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
- E' accessibile da tutti i thread di una griglia e dall'host

Global Memory

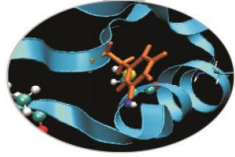


- Per sfruttare al meglio la bandwidth, tutti gli accessi alla memoria globale devono essere coalescenti, i.e gli accessi in memoria da thread differenti devono essere raggruppati e eseguiti con un'unica transizione di memoria.
- Attenzione: alcuni pattern di accesso alla memoria globale sono coalescenti altri no. (Le regole di coalescenza dipendono dalla compute capability della scheda grafica)
- L'architettura FERMI introduce dei meccanismi di caching per gli accessi alla GMEM
- L1: privata al thread, virtual cache implementata nella shared memory
- L2: 768KB, grid-coherent, 25% meglio della latenza della DRAM

```
// L1 = 48 KB  
// SH = 16 KB  
cudaFuncSetCacheConfig( kernel, cudaFuncCachePreferL1);  
// L1 = 16 KB  
// SH = 48 KB  
cudaFuncSetCacheConfig( kernel, cudaFuncCachePreferShared );
```

Kepler architecture introduced some improvements:
32 KB + 32 KB partition option

Global Memory (pre-Fermi)



Compute capability 1.0 and 1.1

- La richiesta di accesso alla GMEM per warp è spezzata in due richieste, una per ogni half-warp, che vengono eseguite indipendentemente.
- Per sfruttare al meglio la bandwidth tutti gli accessi devono essere **coalescenti** (i thread di un half-warp accedono a regioni contigue di memoria nel device)
- I threads devono accedere a words in sequenza strettamente crescente: il k-esimo thread deve accedere al k-esimo word
- Tutti e 16 i words devono risiedere nello stesso segmento di memoria.
- Un'accesso coalescente è realizzato con:
 - Una 64-byte memory transaction, per 4-byte words
 - Una 128-byte memory transaction, per 8-byte words
 - Due 128-byte memory transactions, per 16-byte words

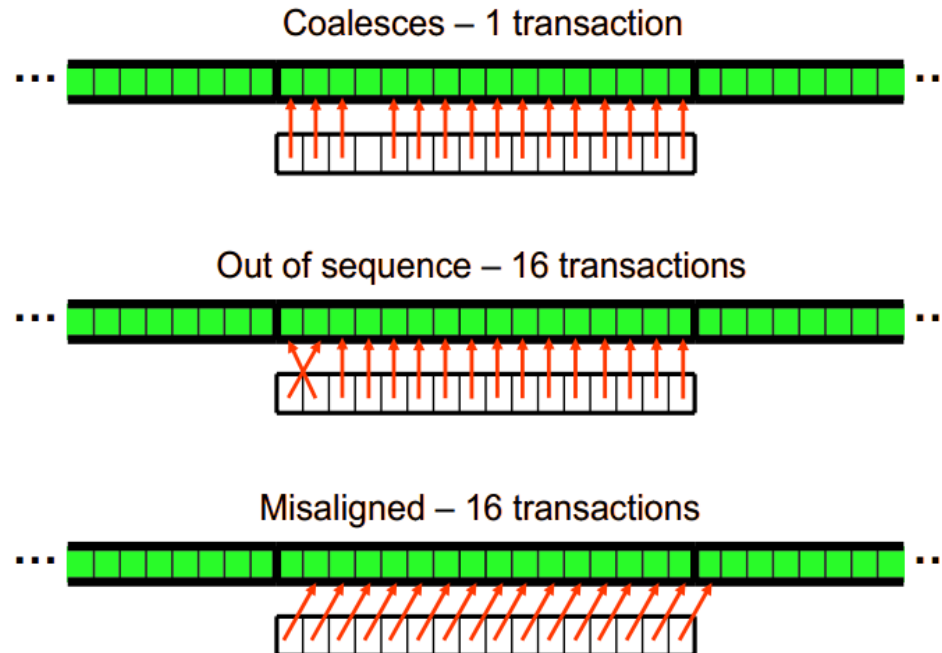


Coalescenza (pre-Fermi)

Compute capability 1.0 and 1.1

- Requisiti di accesso molto stringenti
- Accessi realizzati ad half-warp (16 threads)
- Non tutto i thread devono necessariamente partecipare ma i loro accessi devono essere allineati

k-th thread deve accedere al k-th word nel segmento



Coalescenza (pre-Fermi)



- **Compute capability 1.2 and 1.3**

Trovare il segmento dell'indirizzo di memoria richiesto dal thread con id più basso:

32B segment for 8-bit data

64B segment for 16-bit data

128B segment for 32, 64 and 128-bit data.

Trovare tutti i thread il cui indirizzo risiede nello stesso segmento.

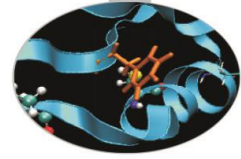
Ridurre la dimensione della transazione se possibile:

If size == 128B e se solo la prima metà o la seconda metà è utilizzata, ridurre la transazione a 64B

If size == 64B e se solo la prima metà o la seconda metà è utilizzata, ridurre la transazione a 32B

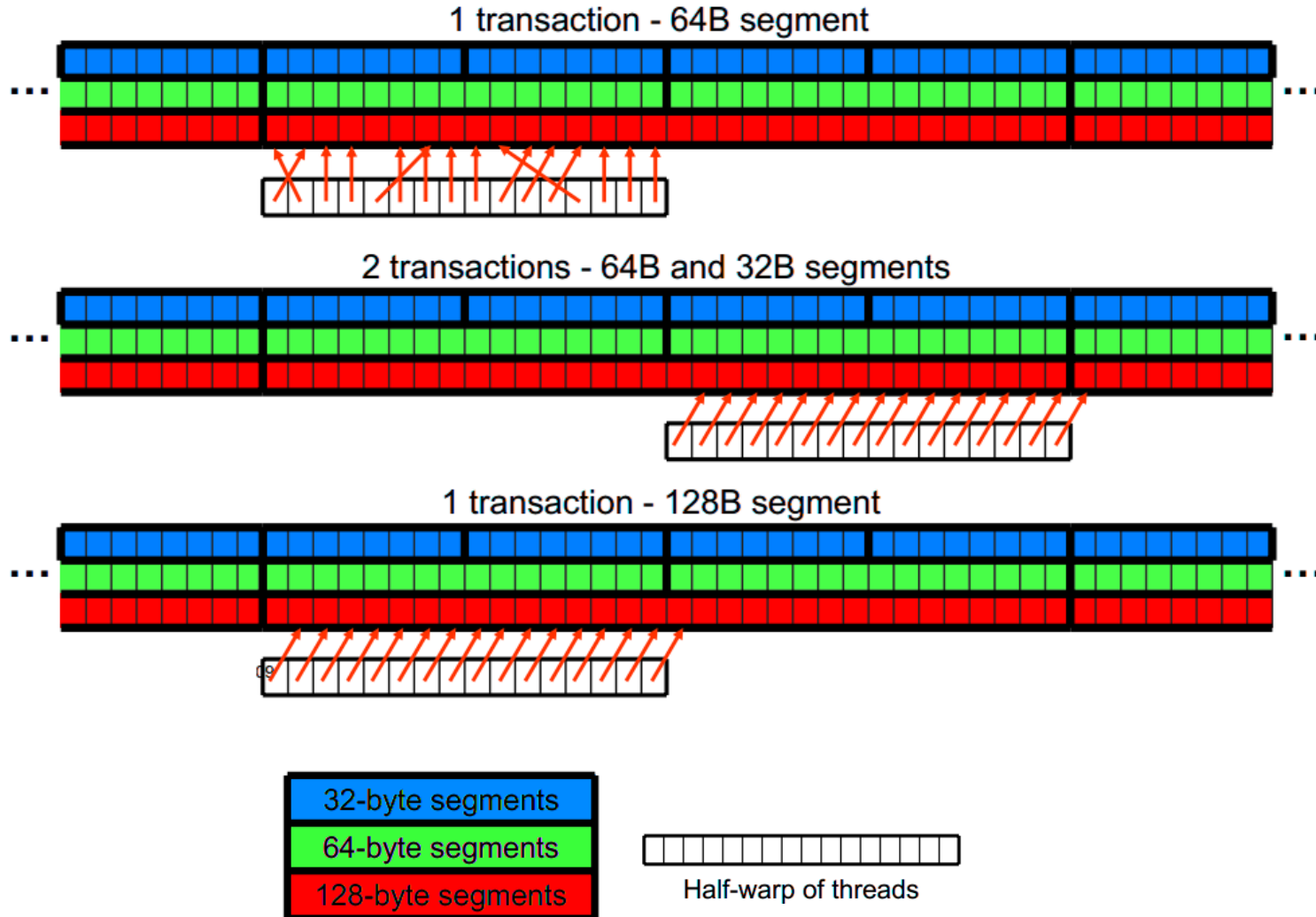
Ripetere finchè tutti i thread dell'half-warp non sono serviti.

Coalescenza (pre-Fermi)



Compute capability 1.2 and 1.3

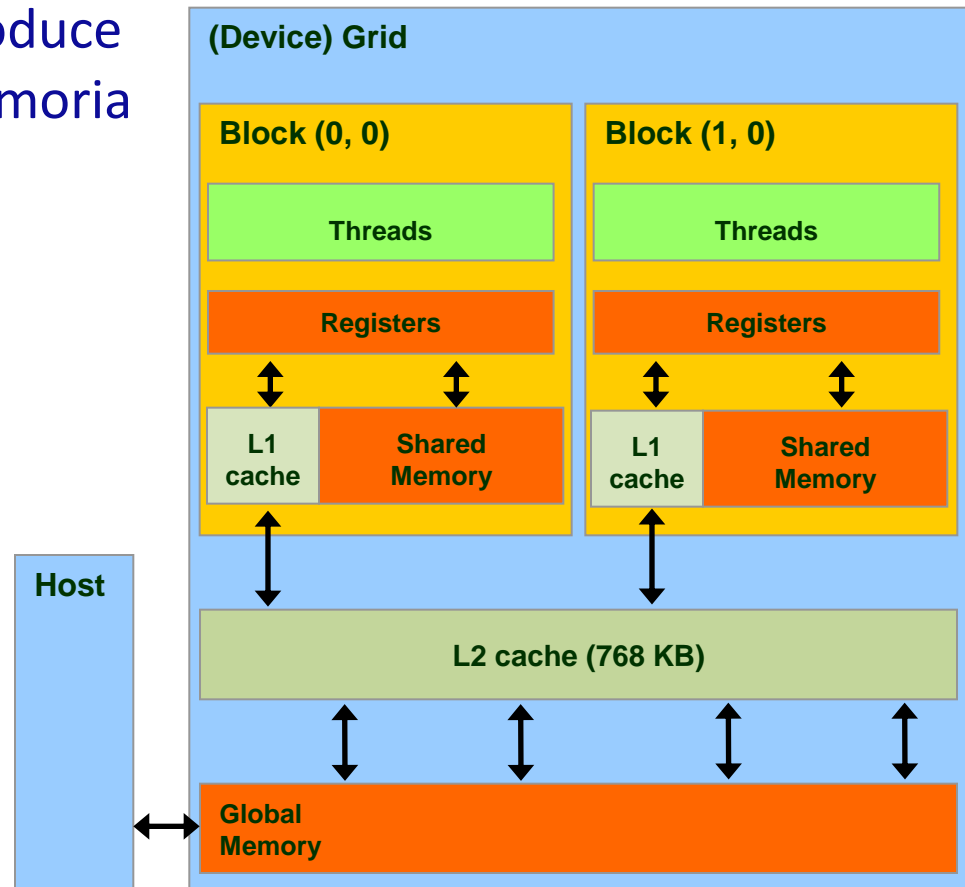
- Il memory controller è migliorato

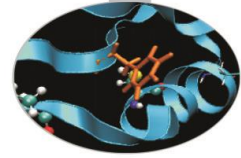




Gerarchia di cache nelle Fermi

- L'architettura Fermi (cc 2.x) introduce una gerarchia di cache tra la memoria globale e lo Streaming Multiprocessor
- due livelli di cache:
 - L2 globale a tutti gli SM
 - L1 configurabile:
 - 16KB shared / 48 KB L1
 - 48KB shared / 16KB L1
- tutti gli accessi alla memoria globale passano per la cache L2
 - includo copie da/su *host* !!!

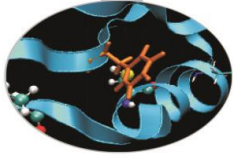




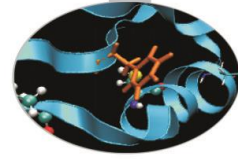
Gerarchia di cache nelle Fermi

- Una tipologia di *store*:
 - viene invalidato il dato in L1, e aggiornato direttamente la linea di cache in L2
- Due tipologie di *load*:
 - *caching (default)*
 - 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
 - *non-caching (selezionabile a compile time)*
 - disabilita la cache L1 con l'opzione di compilazione
`-Xptxas -dlcm=cg`
 - il dato viene prima cercato in L2, poi in GMEM
 - la dimensione delle linee di cache è di 32-bytes

Global Memory Load Operation (Fermi)

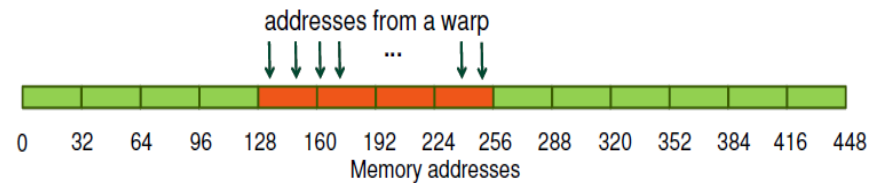
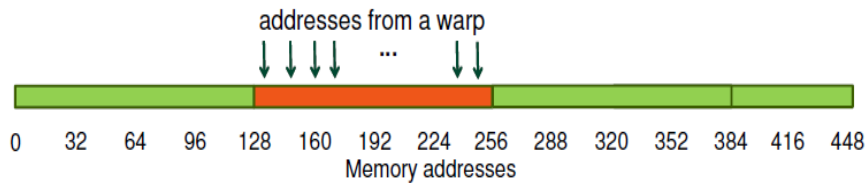


- Le operazioni di accesso alla memoria sono gestite per warp (32 threads)
- Come tutte le altre istruzioni (l'unità fondamentale di calcolo è il warp)
- Operazioni:
 - I threads in un warp calcono l'indirizzo di memoria richiesto
 - Vengono individuati i segmenti di memoria necessari
 - Parte la richiesta dei segmenti/linee di memoria necessarie

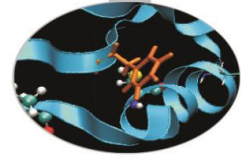


Global Memory Load Operation (Fermi)

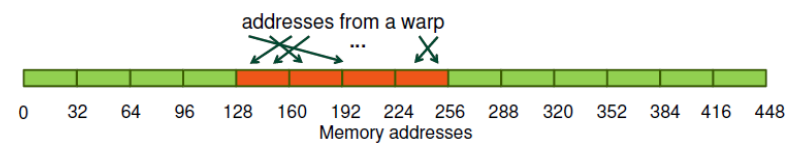
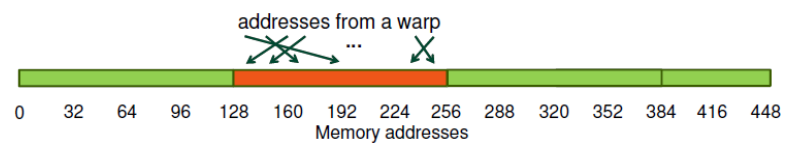
<ul style="list-style-type: none"> Warp requests 32 aligned, consecutive 4-byte words (128 bytes) 	
<ul style="list-style-type: none"> Caching Load 	<ul style="list-style-type: none"> Non-caching Load
<ul style="list-style-type: none"> Addresses fall within 1 cache-line 	<ul style="list-style-type: none"> Addresses fall within 4 segments
128 bytes move across the bus	128 bytes move across the bus
<ul style="list-style-type: none"> Bus utilization: 100% 	<ul style="list-style-type: none"> Bus utilization: 100%



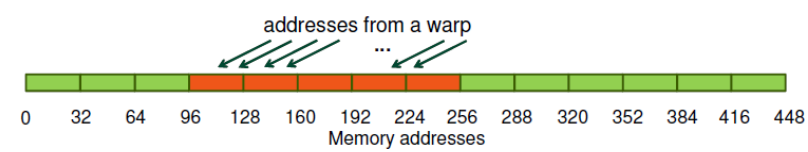
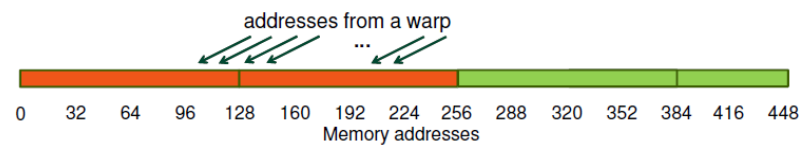
Global Memory Load Operation (Fermi)



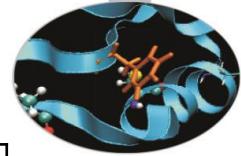
<ul style="list-style-type: none"> Warp requests 32 aligned, permuted 4-byte words (128 bytes) 	
<ul style="list-style-type: none"> Caching Load 	<ul style="list-style-type: none"> Non-caching Load
<ul style="list-style-type: none"> Addresses fall within 1 cache-line 	<ul style="list-style-type: none"> Addresses fall within 4 segments
128 bytes move across the bus	128 bytes move across the bus
<ul style="list-style-type: none"> Bus utilization: 100% 	<ul style="list-style-type: none"> Bus utilization: 100%



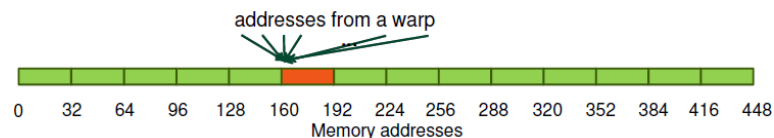
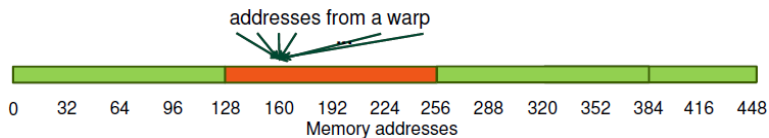
<ul style="list-style-type: none"> Warp requests 32 misaligned, consecutive 4-byte words (128 bytes) 	
<ul style="list-style-type: none"> Caching Load 	<ul style="list-style-type: none"> Non-caching Load
<ul style="list-style-type: none"> Addresses fall within 2 cache-lines 	<ul style="list-style-type: none"> Addresses fall within at most 5 segments
256 bytes move across the bus	160 bytes move across the bus
<ul style="list-style-type: none"> Bus utilization: 50% 	<ul style="list-style-type: none"> Bus utilization: at least 80%



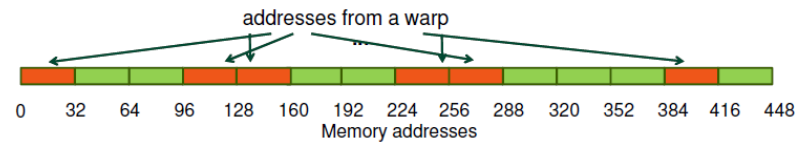
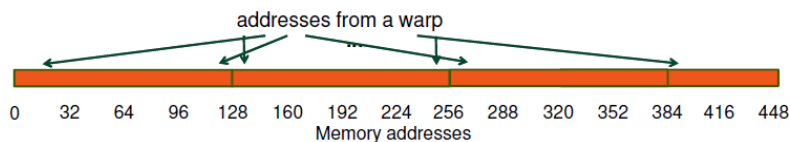
Global Memory Load Operation (Fermi)



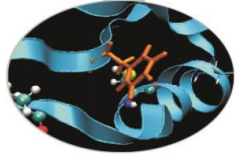
<ul style="list-style-type: none"> All threads in a warp request the same 4-byte word (4 bytes) 	
<ul style="list-style-type: none"> Caching Load 	<ul style="list-style-type: none"> Non-caching Load
<ul style="list-style-type: none"> Addresses fall within 1 cache-line 	<ul style="list-style-type: none"> Addresses fall within 1 segments
128 bytes move across the bus	32 bytes move across the bus
<ul style="list-style-type: none"> Bus utilization: 3.125% 	<ul style="list-style-type: none"> Bus utilization: 12.5%



<ul style="list-style-type: none"> Warp requests 32 scattered 4-byte words (128 bytes) 	
<ul style="list-style-type: none"> Caching Load 	<ul style="list-style-type: none"> Non-caching Load
<ul style="list-style-type: none"> Addresses fall within N cache-lines 	<ul style="list-style-type: none"> Addresses fall within N segments
$N * 128$ bytes move across the bus	$N * 32$ bytes move across the bus
<ul style="list-style-type: none"> Bus utilization: $128 / (N * 128)$ 	<ul style="list-style-type: none"> Bus utilization: $128 / (N * 32)$



Esempio

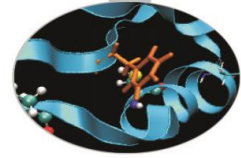


```
#include <stdio.h>

template
__global__ void offset(T* a, int s)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x + s;
    a[i] = a[i] + 1;
}

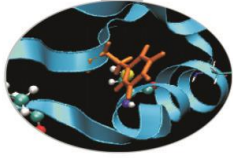
template
__global__ void stride(T* a, int s)
{
    int i = (blockDim.x * blockIdx.x + threadIdx.x) * s;
    a[i] = a[i] + 1;
}
```

Esempio



```
template
void runTest(int deviceId, int nMB)
{
    int blockSize = 256;
    float ms;
    T *d_a;
    cudaEvent_t startEvent, stopEvent;
    int n = nMB*1024*1024/sizeof(T);
    cudaMalloc(&d_a, n * 33 * sizeof(T)) ;
    cudaEventCreate(&startEvent) ;
    cudaEventCreate(&stopEvent);
    printf("Offset, Bandwidth (GB/s):\n");
    for (int i = 0; i <= 32; i++) {
        cudaMemset(d_a, 0.0, n * sizeof(T));
        cudaEventRecord(startEvent,0) ;
        offset<<<n/blockSize, blockSize>>>(d_a, i);
        cudaEventRecord(stopEvent,0);
        cudaEventSynchronize(stopEvent));
        cudaEventElapsedTime(&ms, startEvent, stopEvent) ;
        printf("%d, %f\n", i, 2*nMB/ms);
    }
    printf("\n");
}
```

Esempio

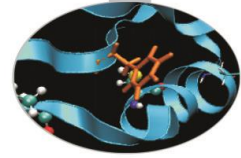


```
printf("Stride, Bandwidth (GB/s):\n");

for (int i = 1; i <= 32; i++) {
    cudaMemset(d_a, 0.0, n * sizeof(T)) ;
    cudaEventRecord(startEvent,0) ;
    stride<<<n/blockSize, blockSize>>>(d_a, i);
    cudaEventRecord(stopEvent,0) ;
    cudaEventSynchronize(stopEvent);
    cudaEventElapsedTime(&ms, startEvent, stopEvent);
    printf("%d, %f\n", i, 2*nMB/ms);
}

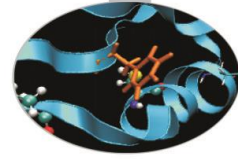
cudaEventDestroy(startEvent);
cudaEventDestroy(stopEvent);
cudaFree(d_a);
}
```

Esempio



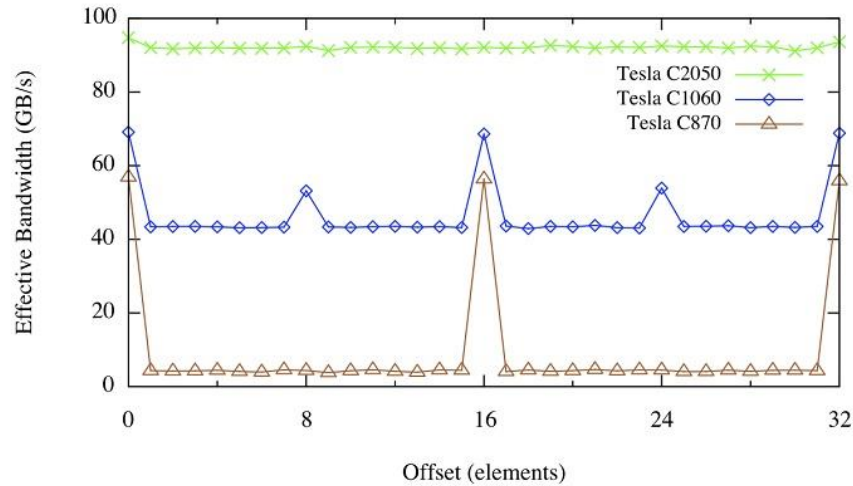
```
int main(int argc, char **argv)
{
    int nMB = 4;
    bool bFp64 = false;
    for (int i = 1; i < argc; i++) {
        if (!strcmp(argv[i], "fp64"))
            bFp64 = true;
    }

    printf("Transfer size (MB): %d\n", nMB);
    printf("%s Precision\n", bFp64 ? "Double" : "Single");
    if (bFp64) runTest<double>(deviceId, nMB);
    else      runTest<int>(deviceId, nMB);
}
```

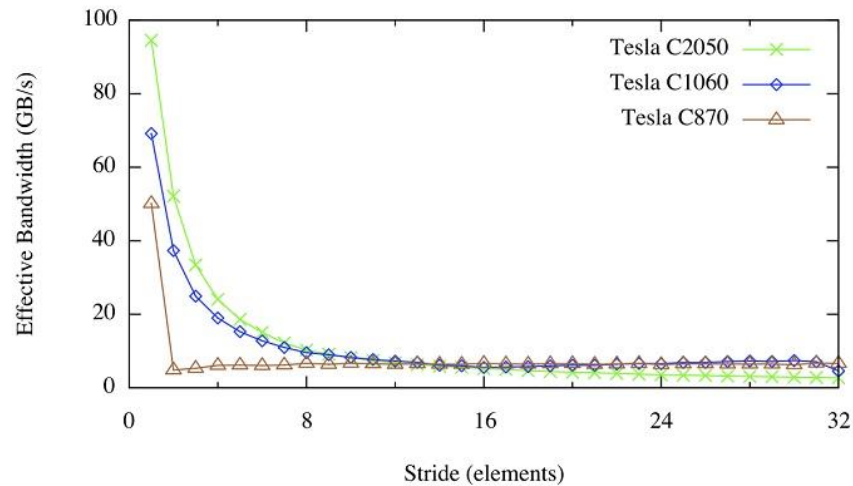


Esempio

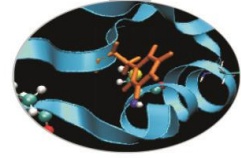
Effective Bandwidth vs. Offset for Single Precision



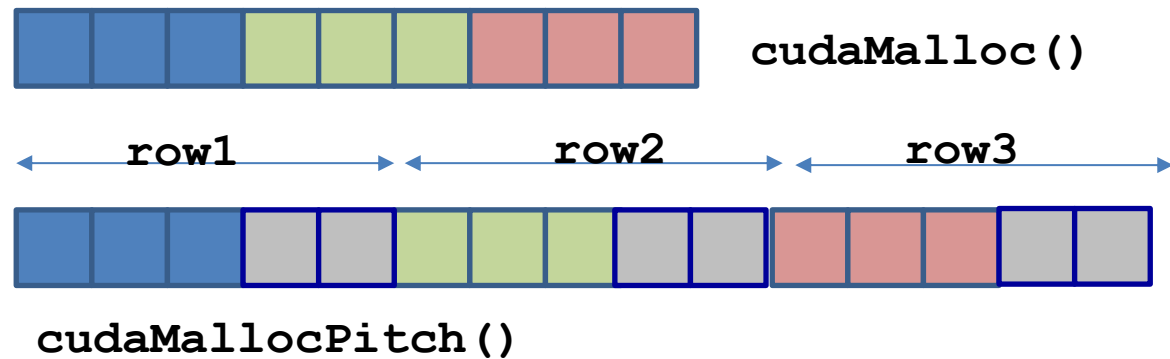
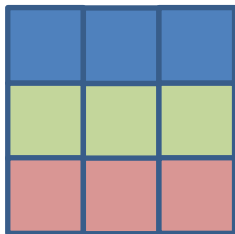
Effective Bandwidth vs. Stride for Single Precision



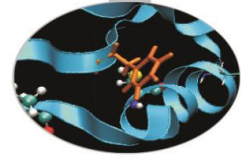
Garantire l'allineamento dei dati in memoria



- 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



Garantire l'allineamento dei dati in memoria

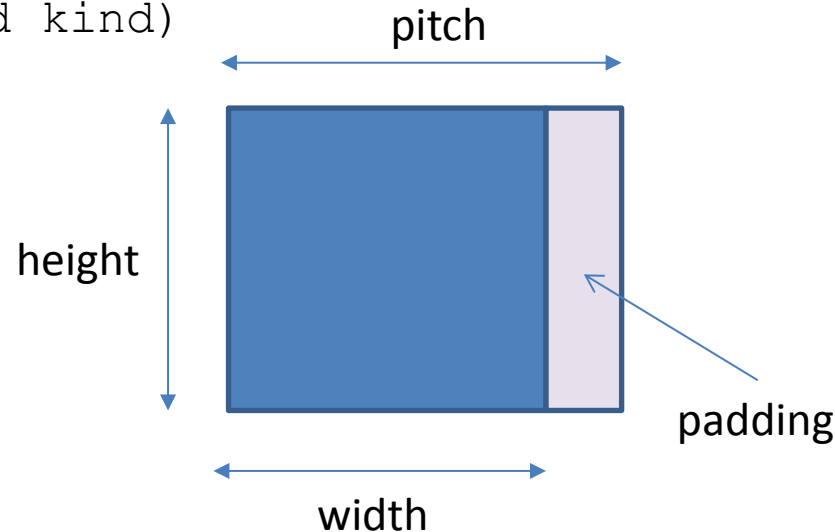


Allocazione di array 2D

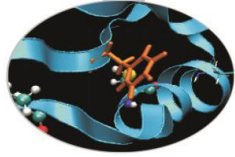
```
cudaError_t cudaMallocPitch (void **devPtr, size_t *pitch,  
size_t width, int height)
```

Copia di array 2D

```
cudaError_t cudaMemcpy2D (void *dst, size_t dpitch,  
const void *src, size_t spitch, size_t width, size_t  
height, cudaMemcpyKind kind)
```



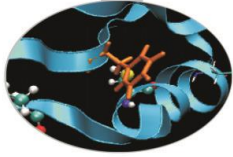
Garantire l'allineamento dei dati in memoria



Il valore del pitch ritornato deve essere poi utilizzato per gestire correttamente l'accesso agli elementi dell'array 2D . Dato una Row e una Column l'indirizzo di un elemento di tipo T viene calcolato come:

$$T^* \text{ pElement} = (T^*) (\text{ArrayAddress} + \text{Row} * \text{pitch}) + \text{Column};$$

```
// host code
int width = 64, height = 64;
float *devPtr;
float *hPtr;
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];
    }
}
```



Race condition e operazioni atomiche

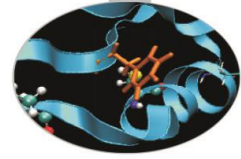
- Una race condition è una situazione comune a molte applicazioni multithread.
- L'ordine di esecuzione delle istruzioni tra più threads è arbitrario.
- Operazioni di read-modify-write eseguite parallelamente da più thread nella stessa area di memoria possono portare a race condition.

L'operazione $*x+3$ viene eseguita in 3 step:

- Read: lettura del valore $*x$ nei registri
- Modify: modifica del valore letto +3
- Write: scrittura del risultato finale

Se l'operazione viene eseguita parallelamente da più threads il valore finale della variabile x è imprevedibile.

Esempio

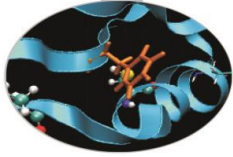


```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>

__global__ void add(int *a_d)
{
    *a_d+=1;
}

int main()
{
    int a=0;
    int* a_d;
    cudaMalloc((void**) &a_d, sizeof(int));
    cudaMemcpy(a_d, &a, sizeof(int), cudaMemcpyHost
ToDevice);
    float elapsedTime;
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
```

Esempio



```
cudaEventRecord(start, 0);  
add<<<100, 100>>>(a_d);  
cudaEventRecord(stop, 0);  
cudaEventSynchronize(stop);  
cudaEventElapsedTime(&elapsedTime, start, stop);  
cudaEventDestroy(start);  
cudaEventDestroy(stop);  
printf("Elapsed Time %f\n", elapsedTime);  
cudaMemcpy(&a, a_d, sizeof(int), cudaMemcpyDeviceToHost);  
printf("a=%d\n", a);  
cudaFree(a_d);  
}
```

Elapsed Time 0.023296

a=1



Atomic operations

- CUDA fornisce un modo semplice per evitare race condition.
- Le operazioni atomiche vengono utilizzate per gestire problemi di sincronizzazione e coordinamento tra threads.
- Garantiscono che le operazioni siano eseguite senza che i threads interferiscano tra di loro.
 - `atomicAdd()`
 - `atomicSub()`
 - `atomicMin()`
 - `atomicMax()`
 - `atomicInc()`
 - `atomicDec()`
 - ...



Atomic Operation

```

__global__ void add(int *a_d)
{
  atomicAdd(a_d,1);
}
  
```

Elapsed Time 0.146048

a=10000

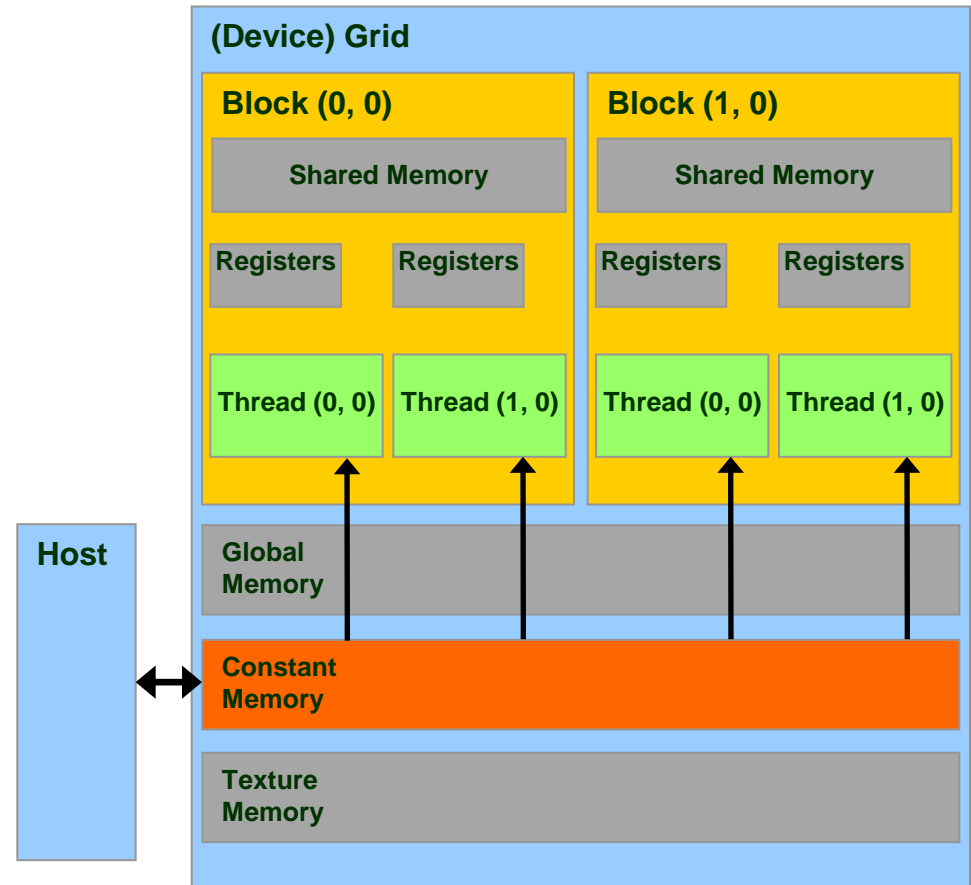
Le operazioni atomiche agiscono sia in global che in local memory. Le operazioni atomiche supportate dipendono dalla compute capability.

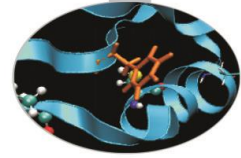
Feature Support	Compute Capability					
(Unlisted features are supported for all compute capabilities)	1.0	1.1	1.2	1.3	2.x, 3.0	3.5
Atomic functions operating on 32-bit integer values in global memory (Atomic Functions)	No	Yes				
atomicExch() operating on 32-bit floating point values in global memory (atomicExch())						
Atomic functions operating on 32-bit integer values in shared memory (Atomic Functions)	No		Yes			
atomicExch() operating on 32-bit floating point values in shared memory (atomicExch())						
Atomic functions operating on 64-bit integer values in global memory (Atomic Functions)						
Warp vote functions (Warp Vote Functions)						
Double-precision floating-point numbers	No			Yes		



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
 - dimensione massima: 64 Kbyte
 - throughput: 32 bits per warp ogni 2 clocks



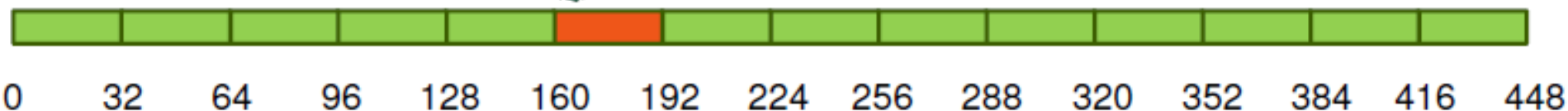


La *Constant Memory* in CUDA

Supponiamo che un kernel esegua 10.000 *thread* (320 *warps*) per SM

- tutti i *thread* accedono a una 4B word:
- usando la memoria globale:
 - tutti i *warp* accedono la memoria globale
 - si genera traffico sul BUS per 320 volte per il numero di accessi
- 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

addresses from a warp





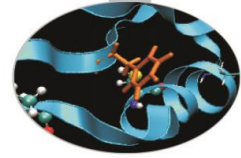
Allocazione della *Constant Memory*

Per allocare una variabile nella constant memory

```
__constant__ type var_name; // statica  
  
cudaMemcpyToSymbol(var_name, &host_src, sizeof(type), cudaMemcpyHostToDevice);  
// attenzione  
// non può essere allocata dinamicamente  
// attenzione
```

- `type, constant :: var_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

Esercitazione



Esercitazione

- pi Approximation



Esercizi

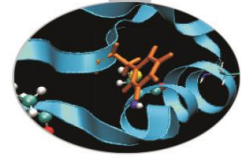


- **piApproximation**

- Implementare un kernel per l'approssimazione di PI usando il seguente algoritmo:

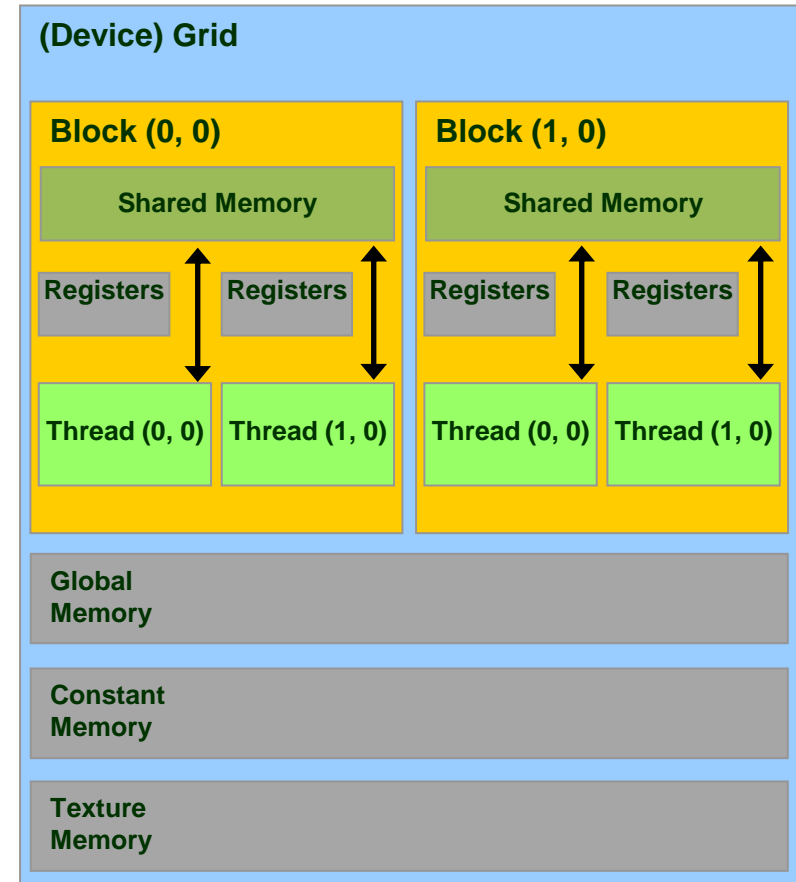
```
count = 0;
for(j=0, j<npoints; j++) {
x = random();
y = random();
if (inCircle(x, y))
count++;
}
PI = 4.0*count/npoints
```

usando la global memory e generando i numeri casuali su host.

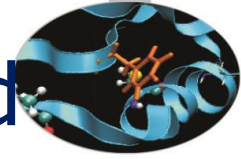


La *Shared Memory* in CUDA

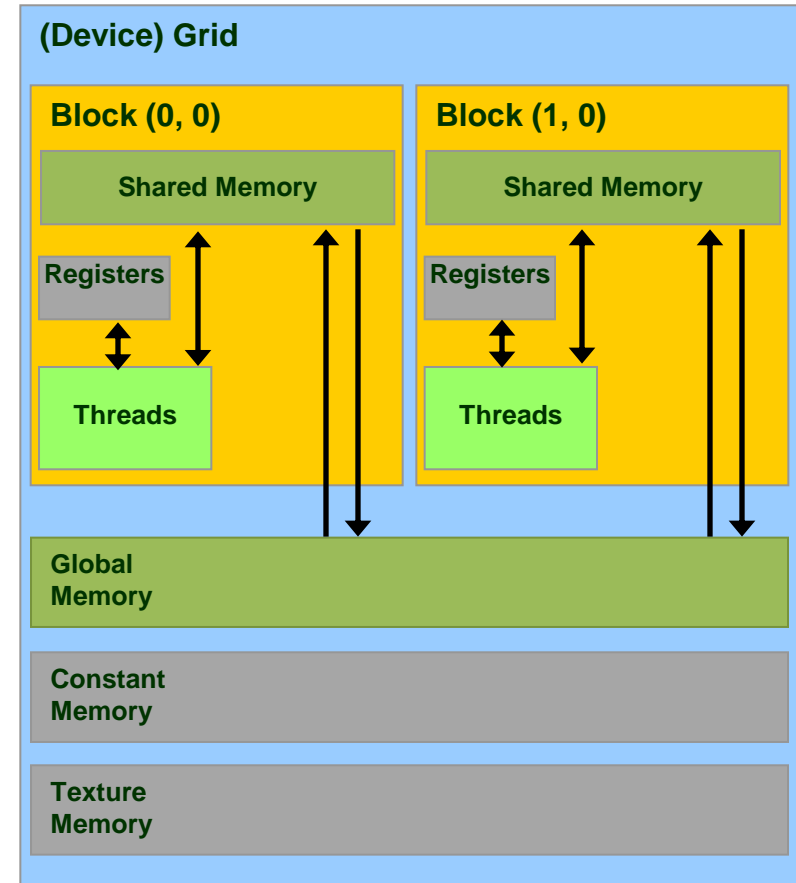
- La memoria condivisa (Shared Memory) è una memoria veloce residente su ciascun Streaming Multiprocessor
- accessibile in lettura e scrittura dai soli thread del blocco
- non mantiene il suo stato tra il lancio di un kernel e l'altro
- bassa latenza: 2 cicli di clock
- tempo di vita pari a quello del blocco
- accessibile solo dai thread dello stesso blocco



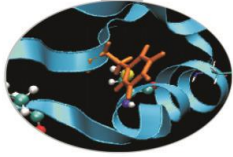
Tipico utilizzo della memoria shared



- La memoria shared è spesso utilizzata come una cache a rapido accesso gestita dal programmatore
 - 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



Allocazione della Memoria Shared



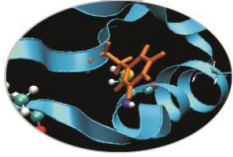
```
// Staticamente all'interno del kernel

__global__ void Kernel(int count_a, int count_b)
{
    __shared__ int a[100];
    __shared__ int b[4];
}

//Dinamicamente

__global__ void Kernel(int count_a, int count_b)
{
    extern __shared__ int shared[];
    int *a = &shared[0]; //a is set at the beginning of shared
mem
    int *b = &shared[count_a]; //b is set at position count_a of
shared
}
sharedMemory = count_a*sizeof(int) + count_b*sizeof(int);
Kernel <<<numBlocks, threadsPerBlock, sharedMemory>>> (count_a,
count_b);
```

Allocazione della Memoria Shared



```
module my_kernels
  use cudafor
  implicit none
contains

  attributes(global) subroutine mykernel(N)
    ! Declare variables
    integer, intent(IN), value :: N
    real, shared, dimension(*) :: shared_array
  end subroutine mykernel
end module my_kernels

program cuda

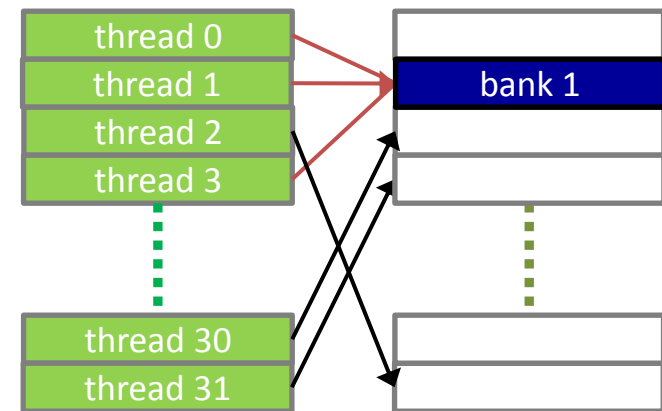
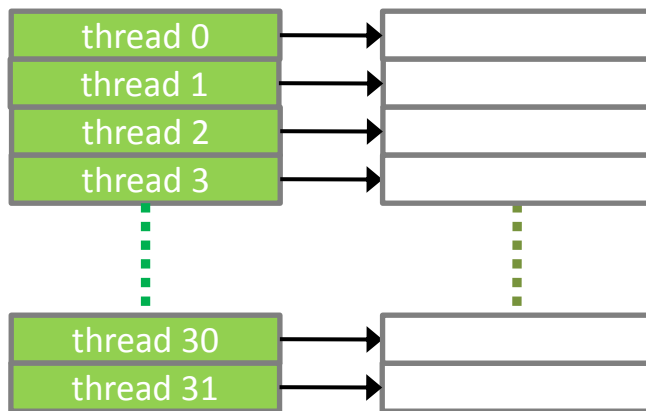
  use my_kernels
  implicit none

  integer :: N = 9
  call mykernel<<<N/3, 3, N*4>>>(N)
end program cuda
```



La Memoria Shared

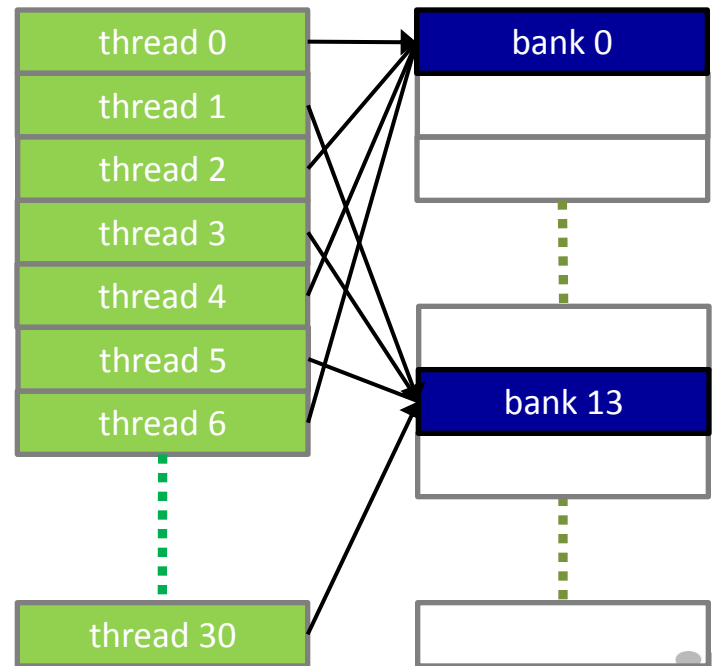
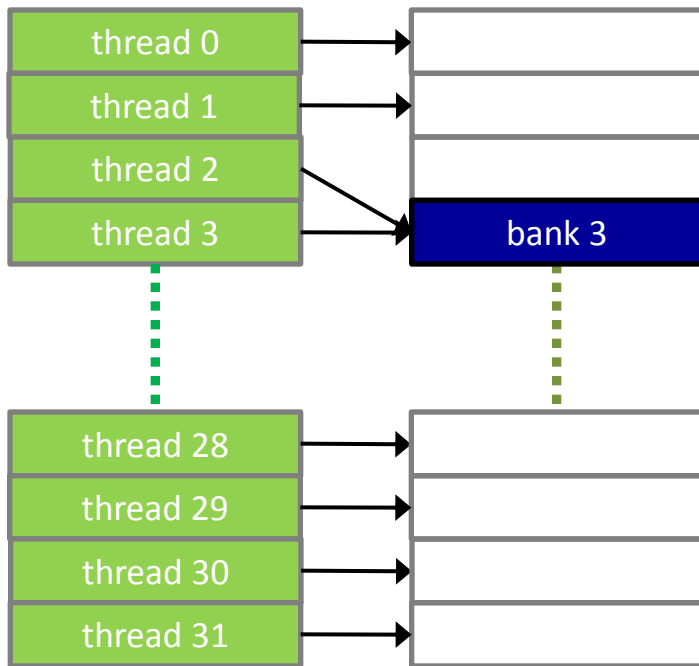
- La *shared memory* è organizzata in linee di 32 banchi, ciascuno di ampiezza di 4-byte
- i dati vengono distribuiti ciclicamente su banchi successivi ogni 4-byte
- gli accessi alla shared memory avvengono per warp
- **multicast**: se n thread accedono lo stesso elemento, l'accesso è eseguito in una singola transazione



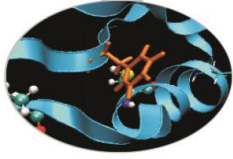


La Memoria Shared

- si ha un *bank conflict* quando *thread* differenti, **dello stesso *warp***, tentano di accedere a dati differenti, residenti sullo stesso banco
 - ogni conflitto viene servito e risolto serialmente



Scelta delle dimensioni shared/cache L1



- Le schede di cc 2.x hanno una shared-memory di 48KB configurabili a 16KB in favore della cache L1
 - selezionabile in modo differente per ogni kernel
 - il default è 48KB shared e 16KB di L1
 - un nuovo default può essere definito mediante la API `cudaThreadSetCacheConfig()`

```
cudaFuncSetCacheConfig(firstKernel, cudaFuncCachePreferShared);  
cudaFuncSetCacheConfig(secondKernel, cudaFuncCachePreferL1);  
...  
// 48KB shmem - 16KB L1  
firstKernel<<<numBlocks, numThreads>>>(...);  
// 16KB shmem - 48KB L1  
secondKernel<<<numBlocks, numThreads>>>(...);  
// default cache configuration  
thirdKernel<<<numBlocks, numThreads>>>(...);
```

Esempi

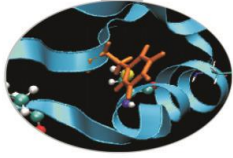


- Alcuni esempi
 - Matrix Transpose
 - accessi coalesced
 - uso shared memory
 - avoiding bank-conflicts
 - misura performance
 - Matrix-Matrix Product
 - sfruttare la shared memory



Esempio: Matrix Transpose

- Implementiamo un kernel GPU che:
 - esegua la trasposizione di una matrice (out-of-place)
 - usi matrici quadrate con lati multipli di 32 elementi
- Si tratta di un kernel memory-bounded
 - non ci sono calcoli da eseguire
 - si tratta solamente di copiare dati
- Metrica per misurare le performance:
 - effective bandwidth (GB/s)
 - un kernel di pura copia può farci da riferimento



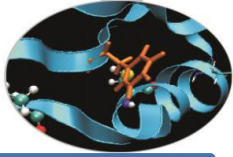
Simple Copy Kernel

```
__global__ void copy (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 = index_in;

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

- gli elementi vengono processati a gruppi:
 - TILE_DIM è la dimensione del tile quadrato
 - il blocco dei thread ha dimensioni: TILE_DIMxTILE_DIM



Naive 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];
}
```

```
attributes(global) subroutine transposeNaive (idata, odata, width,
height)
    integer, intent(in), value :: width, height
    real, intent(in) :: idata(width,height)
    real, intent(out) :: odata(height,width)
    i = ( blockIdx%x - 1 ) * TILE_DIM + threadIdx%x
    j = ( blockIdx%y - 1 ) * TILE_DIM + threadIdx%y
    odata(j,i) = idata(i,j)
end subroutine
```



Measuring Performance

```
// take measurements for loop over kernel launches
cudaEventRecord(start);
for (int i=0; i < NUM_REPS; i++)
    kernel<<<grid, threads>>>(
        d_idata, d_odata, width, height);
cudaEventRecord(stop);
cudaEventSynchronize(stop);
float outerTime;
cudaEventElapsedTime(&outerTime, start, stop);

// take measurements for loop inside kernel
cudaEventRecord(start);
kernel<<<grid, threads>>>(
    d_idata, d_odata, width, height, NUM_REPS);
cudaEventRecord(stop);
cudaEventSynchronize(stop);
float innerTime;
cudaEventElapsedTime(&innerTime, start, stop);
```

Effective Bandwidth (GB/s) on 2048x2048 C1060

<i>kernel</i>	<i>Loop over</i>	<i>Loop inside</i>
Copy	100.2	325.2
Naive	28.9	33.0

... cosa succede?



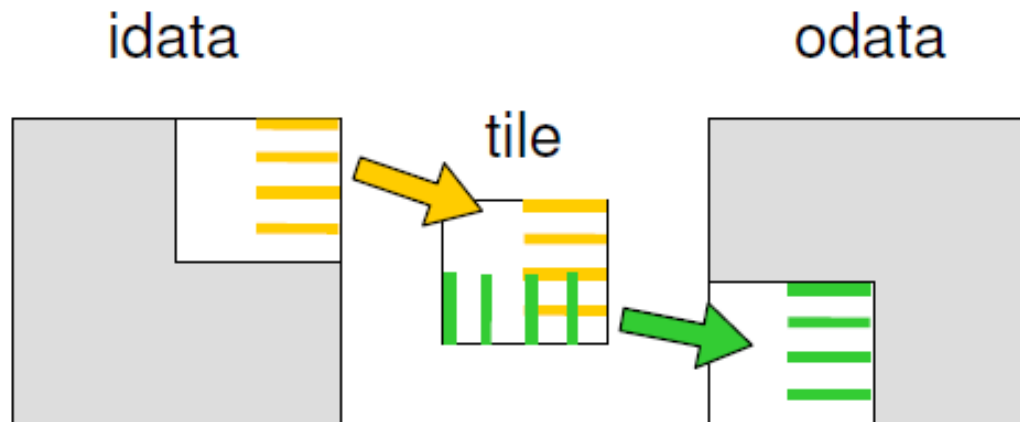
Accessi non coalesced

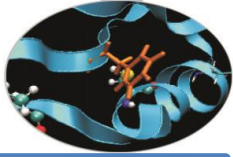
- Tutti gli accessi in lettura sono coalesced:
 - ogni *warp* legge una riga di elementi contigui in memoria
 - 32 float risiedono sullo stesso segmento
- Gli accessi in scrittura differiscono:
 - il kernel copy scrive per righe
 - il kernel naive transpose scrive per colonne
 - ogni thread del warp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



Coalesced Transpose

- 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





Coalesced Transpose

```
__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;

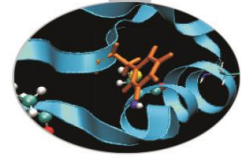
    tile[threadIdx.y][threadIdx.x] = idata[index_in];
    __syncthreads();

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



More on 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, altrimenti l'esecuzione del codice potrebbe andare in stallo o produrre effetti non prevedibili.



Coalesced Matrix Transpose

Effective Bandwidth (GB/s) on 2048x2048 C1060

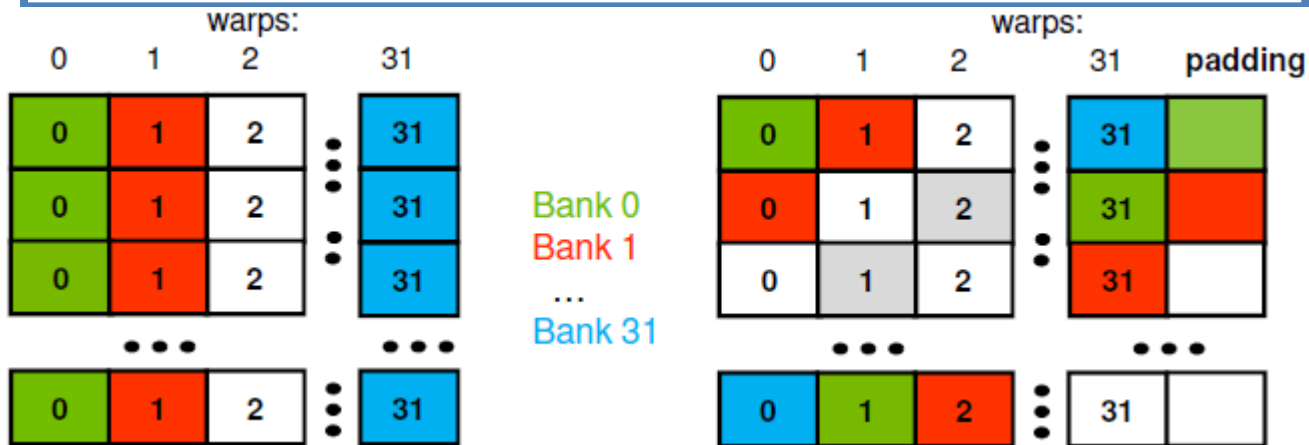
<i>kernel</i>	<i>Loop over</i>	<i>Loop inside</i>
Copy	100.2	325.2
Naive	28.9	33.0
Coalesced	43.7	53.5



Avoiding 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];
```

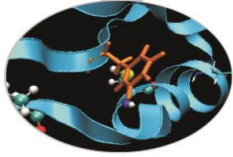




Avoiding Bank Conflict

Effective Bandwidth (GB/s) on 2048x2048 C1060		
<i>kernel</i>	<i>Loop over</i>	<i>Loop inside</i>
Copy	100.2	325.2
Naive	28.9	33.0
Coalesced	43.7	53.5
no Bank Conflicts	96.2	306.3

Esercizi



- Esercitazione:
 - implementazione di:
 - Pi approximation
 - Dot product
 - Matrix matrix product





Esercizi

- **piApproximation**
 - Reimplementare l'algoritmo di approssimazione del π sfruttando opportunamente la shared memory.
- **dotProduct**
 - Implementare un kernel CUDA per il prodotto tra due vettori **a**, **b** sfruttando opportunamente la shared memory.

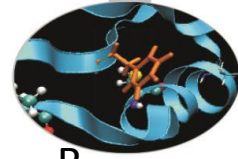
$$c = \sum_{i=0}^N a_i b_i$$

Prodotto matrice-matrice usando memoria shared

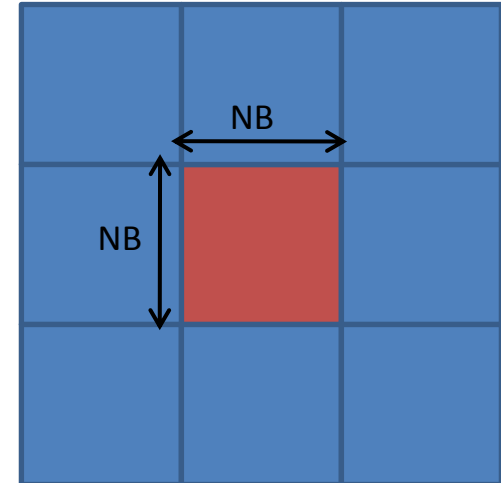
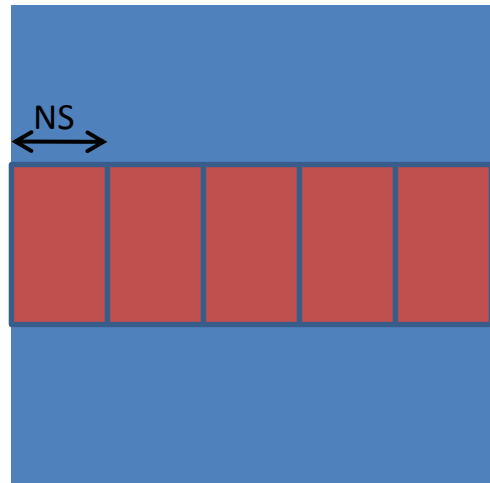
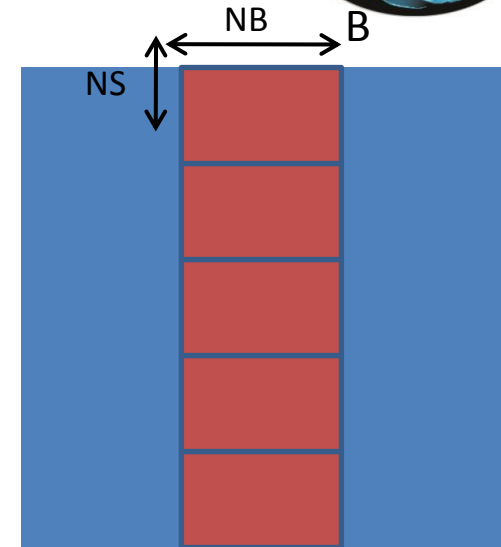


- ▶ Usando solo la memoria globale, ogni thread calcola un elemento di C accedendo a $2N$ elementi (N di A e N di B) e svolgendo $2N$ operazioni floating-point (N *add* e N *mul*)
- ▶ Poichè gli accessi alla memoria globale sono lenti, consideriamo di usare la memoria condivisa all'interno dei blocchi di thread (*shared*)
- ▶ Il kernel dovrà contenere due macro-fasi:
 - ▶ I fase: i thread caricano gli elementi di A e B dalla memoria globale alla memoria shared
 - ▶ II fase: i thread calcolano i prodotti necessari per ottenere C estraendo i valori di A e B dalla memoria shared
- ▶ Il miglioramento di performance è dovuto al fatto che 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

Prodotto matrice-matrice usando memoria shared

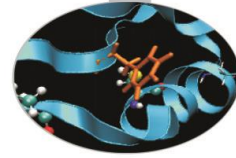


- ▶ Siano per semplicità matrici quadrate di ordine N e blocchi di thread quadrati di ordine NB con N multiplo di NB . Ogni thread calcola un elemento di C . Un blocco di thread calcola una sottomatrice di C di ordine NB
- ▶ Un blocco di thread ha bisogno di una sottomatrice di A di ordine $NB \times N$ e di una sottomatrice di B di ordine $N \times NB$
- ▶ Tali sottomatrici non sono caricate in memoria shared tutte insieme perchè la memoria shared può non essere abbastanza grande. Le sottomatrici sono quindi ulteriormente suddivise in blocchi $A_s(k)$ e $B_s(k)$ di dimensioni $NB \times NS$ e $NS \times NB$
- ▶ La sottomatrice C da calcolare risulta dall'accumulo dei prodotti tra i blocchi di A e B moltiplicati tra di loro



$$C_S = \sum_i A_S(k) \cdot B_S(k)$$

Prodotto matrice-matrice usando memoria shared

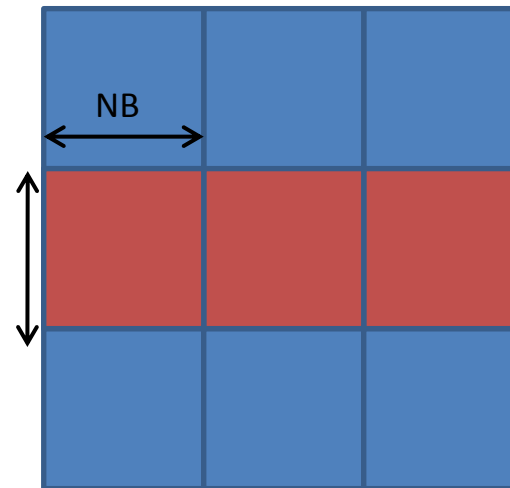
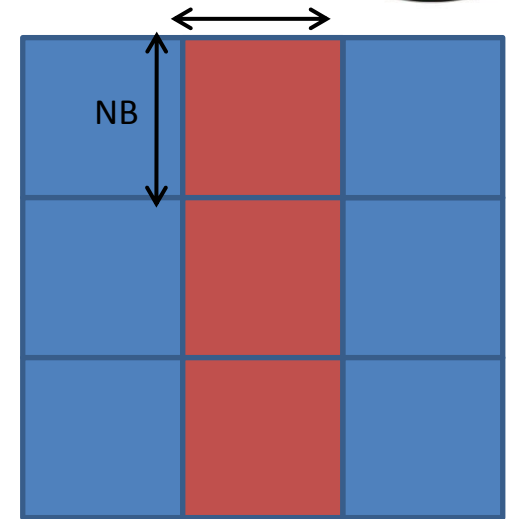


La scelta più semplice è prendere $NS=NB$. In questo modo ogni thread carica in memoria shared un elemento di A e un elemento di B.

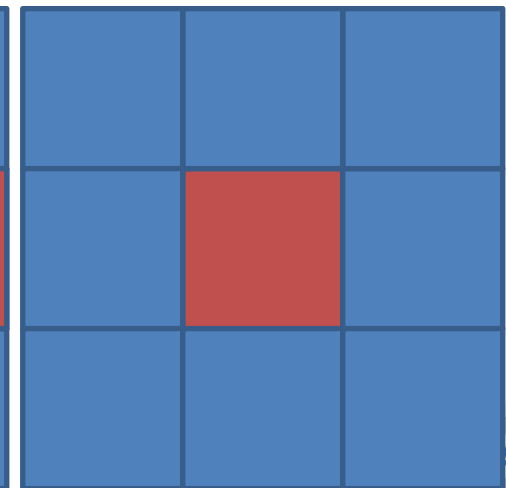
Sono necessarie sincronizzazioni:

- ▶ 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

▶ Gli elementi di C non vengono accumulati direttamente nella memoria globale ma in variabili di appoggio, messe su registri, e alla fine copiate su C

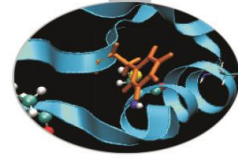


A



C

Matrice-matrice con shared: algoritmo del kernel



$C_{ij}=0.$

Ciclo sui blocchi
 $kb=1, N/NB$

$A_s(tx,ty)=A(i,(kb-1) \cdot NB+ty)$
 $B_s(tx,ty)=B((kb-1) \cdot NB+tx,j)$

Sincronizzazione

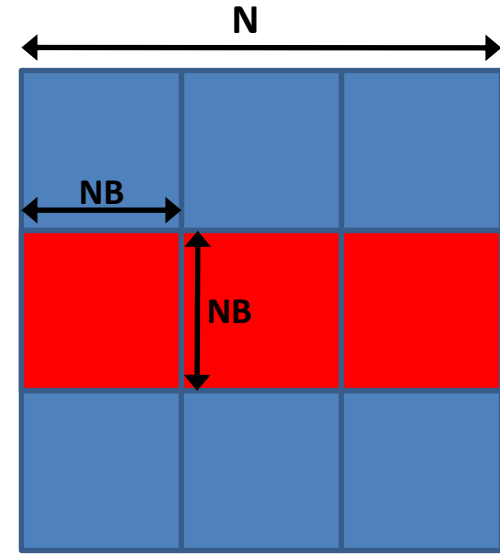
Ciclo nel blocco: $k=1, NB$

$C_{ij}=C_{ij}+A_s(tx,k) \cdot B_s(k,ty)$

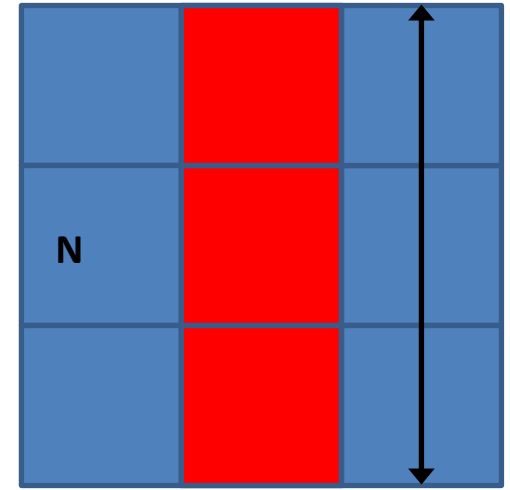
Sincronizzazione

$C(i,j)=C_{ij}$

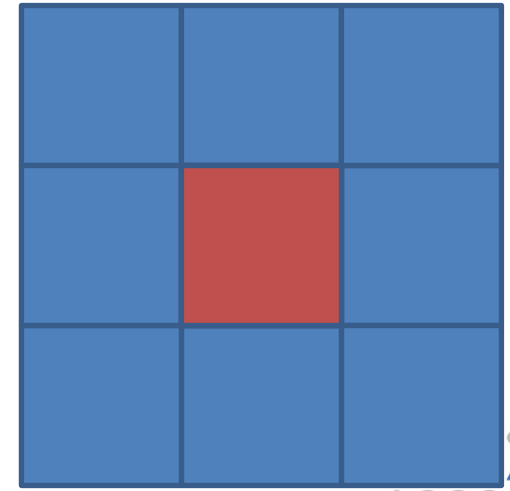
*thread (tx,ty)
 che calcola il
 nodo C(i,j)*



A



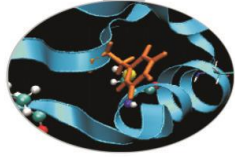
B



C



Matrice-matrice con shared: codice kernel



- ▶ Per rendere il codice più pulito può essere opportuno usare:
 - ▶ strutture di dati definite dall'utente: e.g., in C la struct Matrix

```
typedef struct {  
    int width;    int height;  
    int stride;   float* elements;  
} Matrix
```

- ▶ *device* functions (in C) o *device* subroutines (in fortran): e.g. in C

```
__device__ float GetElement(const Matrix A, int row, int col)  
{    return A.elements[row*A.stride+col];    }  
__device__ float SetElement(const Matrix A, int row, int col)  
{    A.elements[row*A.stride+col] = value ;    }  
__device__ Matrix GetSubMatrix(Matrix A, int row, int col)  
{    Matrix Asub ;  
    Asub.width = NB; Asub.height = NB ; Asub.stride = A.stride;  
    Asub.elements = &A.elements[A.stride * NB * row + NB * col]    }
```

- ▶ direttive di pre-processing per definire ad esempio la dimensione dei blocchi

```
#define NB 32
```

```
#define NB 32
```