

# *Esercitazioni di Calcolo Parallelo: CUDA*

Paolo Avogadro

DISCo, Università di Milano-Bicocca  
U14, Id&aLab T36  
paolo.avogadro@unimib.it  
Aula Lezione T014, edificio U14, WEBEX  
Martedì 13:30 - 15:30  
Giovedì 8:30 - 10:30

# Outline

- 1 *Struttura Fisica*
- 2 *Kernel*
- 3 *Struttura Logica*
- 4 *Memorie*
- 5 *Prodotto Matrici*
- 6 *Stream*
- 7 *Esempi e strumenti*

# Introduzione

- testo di riferimento **CUDA By Example**, by J.Sanders and E. Kandrot, scaricabile da: <https://developer.nvidia.com/cuda-example>
- CUDA e' un acronimo per Compute Unified Device Architecture ed e' stato sviluppato da Nvidia per sfruttare al meglio le sue GPU.
- CUDA usa lo standard PTX (Parallel Thread eXecution).
- CUDA espone le GPU come una multi-core virtual machine.
- CUDA fornisce una serie di funzioni atte a fare interagire la scheda grafica con il resto del computer.

Il cuore di un codice fatto per girare su CUDA sta nell'utilizzare accuratamente le **memorie** dedicate della GPU per poter limitare il VonNeumann bottleneck e sfruttare quindi al massimo il **grande numero** di ALU= Arithmetic logic Unit. In questo modo si puo' ottenere un elevato throughput di calcolo (vicino alle prestazioni teoriche che sono  $\approx$  Frequenza  $\times$  N di core).

# Introduzione

- testo di riferimento **CUDA By Example**, by J.Sanders and E. Kandrot, scaricabile da: <https://developer.nvidia.com/cuda-example>
- CUDA e' un acronimo per Compute Unified Device Architecture ed e' stato sviluppato da Nvidia per sfruttare al meglio le sue GPU.
- CUDA usa lo standard PTX (Parallel Thread eXecution).
- CUDA espone le GPU come una multi-core virtual machine.
- CUDA fornisce una serie di funzioni atte a fare interagire la scheda grafica con il resto del computer.

Il cuore di un codice fatto per girare su CUDA sta nell'utilizzare accuratamente le **memorie** dedicate della GPU per poter limitare il VonNeumann bottleneck e sfruttare quindi al massimo il **grande numero** di ALU= Arithmetic logic Unit. In questo modo si puo' ottenere un elevato throughput di calcolo (vicino alle prestazioni teoriche che sono  $\approx$  Frequenza  $\times$  N di core).

# Introduzione

- testo di riferimento **CUDA By Example**, by J.Sanders and E. Kandrot, scaricabile da: <https://developer.nvidia.com/cuda-example>
- CUDA e' un acronimo per Compute Unified Device Architecture ed e' stato sviluppato da Nvidia per sfruttare al meglio le sue GPU.
- CUDA usa lo standard PTX (Parallel Thread eXecution).
- CUDA espone le GPU come una multi-core virtual machine.
- CUDA fornisce una serie di funzioni atte a fare interagire la scheda grafica con il resto del computer.

Il cuore di un codice fatto per girare su CUDA sta nell'utilizzare accuratamente le **memorie** dedicate della GPU per poter limitare il VonNeumann bottleneck e sfruttare quindi al massimo il **grande numero** di ALU= Arithmetic logic Unit. In questo modo si puo' ottenere un elevato throughput di calcolo (vicino alle prestazioni teoriche che sono  $\approx$  Frequenza  $\times$  N di core).

# Introduzione

- testo di riferimento **CUDA By Example**, by J.Sanders and E. Kandrot, scaricabile da: <https://developer.nvidia.com/cuda-example>
- CUDA e' un acronimo per Compute Unified Device Architecture ed e' stato sviluppato da Nvidia per sfruttare al meglio le sue GPU.
- CUDA usa lo standard PTX (Parallel Thread eXecution).
- CUDA espone le GPU come una multi-core virtual machine.
- CUDA fornisce una serie di funzioni atte a fare interagire la scheda grafica con il resto del computer.

Il cuore di un codice fatto per girare su CUDA sta nell'utilizzare accuratamente le **memorie** dedicate della GPU per poter limitare il VonNeumann bottleneck e sfruttare quindi al massimo il **grande numero** di **ALU= Arithmetic logic Unit**. In questo modo si puo' ottenere un elevato throughput di calcolo (vicino alle prestazioni teoriche che sono  $\approx$  Frequenza  $\times$  N di core).

# Introduzione

- testo di riferimento **CUDA By Example**, by J.Sanders and E. Kandrot, scaricabile da: <https://developer.nvidia.com/cuda-example>
- CUDA e' un acronimo per Compute Unified Device Architecture ed e' stato sviluppato da Nvidia per sfruttare al meglio le sue GPU.
- CUDA usa lo standard PTX (Parallel Thread eXecution).
- CUDA espone le GPU come una multi-core virtual machine.
- CUDA fornisce una serie di funzioni atte a fare interagire la scheda grafica con il resto del computer.

Il cuore di un codice fatto per girare su CUDA sta nell'utilizzare accuratamente le **memorie** dedicate della GPU per poter limitare il VonNeumann bottleneck e sfruttare quindi al massimo il **grande numero** di **ALU= Arithmetic logic Unit**. In questo modo si puo' ottenere un elevato throughput di calcolo (vicino alle prestazioni teoriche che sono  $\approx$  Frequenza  $\times$  N di core).

# Introduzione

- testo di riferimento **CUDA By Example**, by J.Sanders and E. Kandrot, scaricabile da: <https://developer.nvidia.com/cuda-example>
- CUDA e' un acronimo per Compute Unified Device Architecture ed e' stato sviluppato da Nvidia per sfruttare al meglio le sue GPU.
- CUDA usa lo standard PTX (Parallel Thread eXecution).
- CUDA espone le GPU come una multi-core virtual machine.
- CUDA fornisce una serie di funzioni atte a fare interagire la scheda grafica con il resto del computer.

Il **cuore** di un codice fatto per girare su CUDA sta nell'utilizzare accuratamente le **memorie** dedicate della GPU per poter limitare il VonNeumann bottleneck e sfruttare quindi al massimo il **grande numero** di **ALU= Arithmetic logic Unit**. In questo modo si puo' ottenere un elevato throughput di calcolo (vicino alle prestazioni teoriche che sono  $\approx$  Frequenza  $\times$  N di core).



# Von Neumann bottleneck?

## Gli informatici lo conoscono bene...

- Un'**operazione** tra 2 **float** (4 byte ognuno) puo' richiedere **1-2 cicli di clock**
- **Richiamare** (fetch) i due numeri dalla memoria principale richiede **~ 100 cicli**  
(<https://developers.redhat.com/blog/2016/03/01/reducing-memory-access-times-with-caches/>)

Il costo totale dell'esecuzione diventa 101 cicli!!!! Durante la maggior parte del tempo l'unita' di calcolo del processore (**ALU**) e' in stato di **idle**: non fa nulla!

# Von Neumann bottleneck?

## Gli informatici lo conoscono bene...

- Un'**operazione** tra 2 **float** (4 byte ognuno) puo' richiedere **1-2 cicli di clock**
- **Richiamare** (fetch) i due numeri dalla memoria principale richiede **~ 100 cicli**  
(<https://developers.redhat.com/blog/2016/03/01/reducing-memory-access-times-with-caches/>)

Il costo totale dell'esecuzione diventa 101 cicli!!!! Durante la maggior parte del tempo l'unita' di calcolo del processore (**ALU**) e' in stato di **idle**: non fa nulla!

# Von Neumann bottleneck?

Gli informatici lo conoscono bene...

- Un'**operazione** tra 2 **float** (4 byte ognuno) puo' richiedere **1-2 cicli di clock**
- **Richiamare** (fetch) i due numeri dalla memoria principale richiede **~ 100 cicli**  
(<https://developers.redhat.com/blog/2016/03/01/reducing-memory-access-times-with-caches/>)

Il costo totale dell'esecuzione diventa 101 cicli!!!! Durante la maggior parte del tempo l'unita' di calcolo del processore (**ALU**) e' in stato di **idle**: non fa nulla!

# Lingo

- 1 **host** e' il computer su cui e' installata la scheda grafica (con CPU, memoria etc...).
- 2 **device** e' il modo con cui ci si riferisce alla scheda grafica.
- 3 **compute capabilities (c.c.)** e' la *serie* a cui appartiene la scheda in questione e riassume le principali caratteristiche e limitazioni.
- 4 **kernel** e' la parte del codice CUDA definita con uno statement `__global__` e fatta per girare sul device (e' una funzione che gira sul device); **non puo' essere ricorsiva**. I kernel sono **asincroni**, ergo restituiscono **immediatamente** il controllo al programma chiamante prima del completamento...o anche prima di iniziare l'esecuzione! **Cosa significa? cominciamo a pensare...** Visto che ci sono sia una CPU che una GPU, non ha senso *bloccare* l'una mentre l'altra lavora!

# Lingo

- 1 **host** e' il computer su cui e' installata la scheda grafica (con CPU, memoria etc...).
- 2 **device** e' il modo con cui ci si riferisce alla scheda grafica.
- 3 **compute capabilities (c.c.)** e' la *serie* a cui appartiene la scheda in questione e riassume le principali caratteristiche e limitazioni.
- 4 **kernel** e' la parte del codice CUDA definita con uno statement `__global__` e fatta per girare sul **device** (e' una funzione che gira sul device); **non** puo' essere ricorsiva. I kernel sono **asincroni**, ergo restituiscono **immediatamente** il controllo al programma chiamante prima del completamento...o anche prima di iniziare l'esecuzione! **Cosa significa? cominciamo a pensare...** Visto che ci sono sia una CPU che una GPU, non ha senso *bloccare* l'una mentre l'altra lavora!

# Lingo

- 1 **host** e' il computer su cui e' installata la scheda grafica (con CPU, memoria etc...).
- 2 **device** e' il modo con cui ci si riferisce alla scheda grafica.
- 3 **compute capabilities (c.c.)** e' la *serie* a cui appartiene la scheda in questione e riassume le principali caratteristiche e limitazioni.
- 4 **kernel** e' la parte del codice CUDA definita con uno statement `__global__` e fatta per girare sul **device** (e' una funzione che gira sul device); **non** puo' essere ricorsiva. I kernel sono **asincroni**, ergo restituiscono **immediatamente** il controllo al programma chiamante prima del completamento...o anche prima di iniziare l'esecuzione! [Cosa significa? cominciamo a pensare...](#) Visto che ci sono sia una CPU che una GPU, non ha senso *bloccare* l'una mentre l'altra lavora!

- 1 **host** e' il computer su cui e' installata la scheda grafica (con CPU, memoria etc...).
- 2 **device** e' il modo con cui ci si riferisce alla scheda grafica.
- 3 **compute capabilities (c.c.)** e' la *serie* a cui appartiene la scheda in questione e riassume le principali caratteristiche e limitazioni.
- 4 **kernel** e' la parte del codice CUDA definita con uno statement `__global__` e fatta per girare sul **device** (e' una funzione che gira sul device); **non** puo' essere ricorsiva. I kernel sono **asincroni**, ergo restituiscono **immediatamente** il controllo al programma chiamante prima del completamento...o anche prima di iniziare l'esecuzione! [Cosa significa? cominciamo a pensare...](#) Visto che ci sono sia una CPU che una GPU, non ha senso *bloccare* l'una mentre l'altra lavora!

# Lingo

- 1 **host** e' il computer su cui e' installata la scheda grafica (con CPU, memoria etc...).
- 2 **device** e' il modo con cui ci si riferisce alla scheda grafica.
- 3 **compute capabilities (c.c.)** e' la *serie* a cui appartiene la scheda in questione e riassume le principali caratteristiche e limitazioni.
- 4 **kernel** e' la parte del codice CUDA definita con uno statement `__global__` e fatta per girare sul **device** (e' una funzione che gira sul device); **non** puo' essere ricorsiva. I kernel sono **asincroni**, ergo restituiscono **immediatamente** il controllo al programma chiamante prima del completamento...o anche prima di iniziare l'esecuzione! [Cosa significa? cominciamo a pensare...](#) Visto che ci sono sia una CPU che una GPU, non ha senso *bloccare* l'una mentre l'altra lavora!



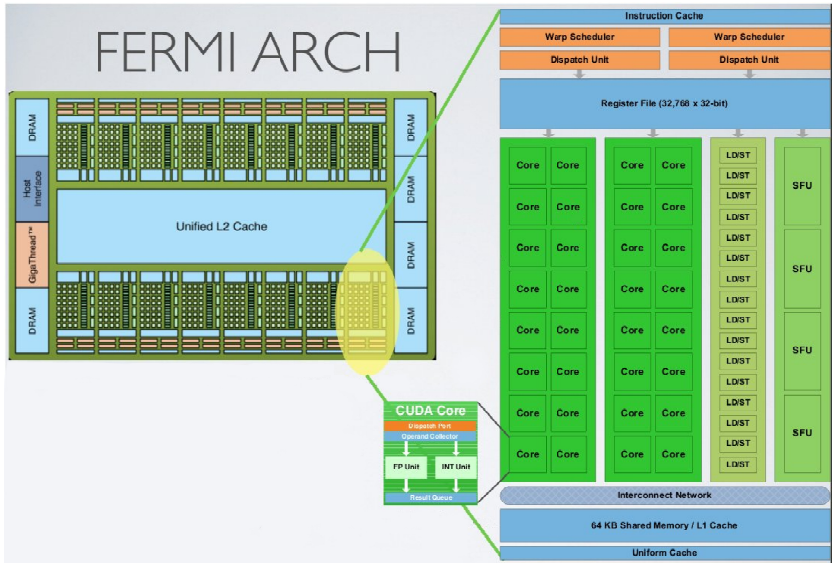
# La struttura fisica

- 1 **Cuda core** e' un termine commerciale inventato da Nvidia per definire le **ALU** che lavorano in float-int nelle schede piu' vecchie. Sinonimi: Streaming processor (**SP**), shader.
- 2 **Streaming Multiprocessor (SM)**: e' una unita' fisica di una scheda grafica, che racchiude:
  - un certo numero di **SP**
  - vari tipi di **memoria** dedicata
  - unita' speciali di calcolo
  - unita' di Load/Store
  - etc...

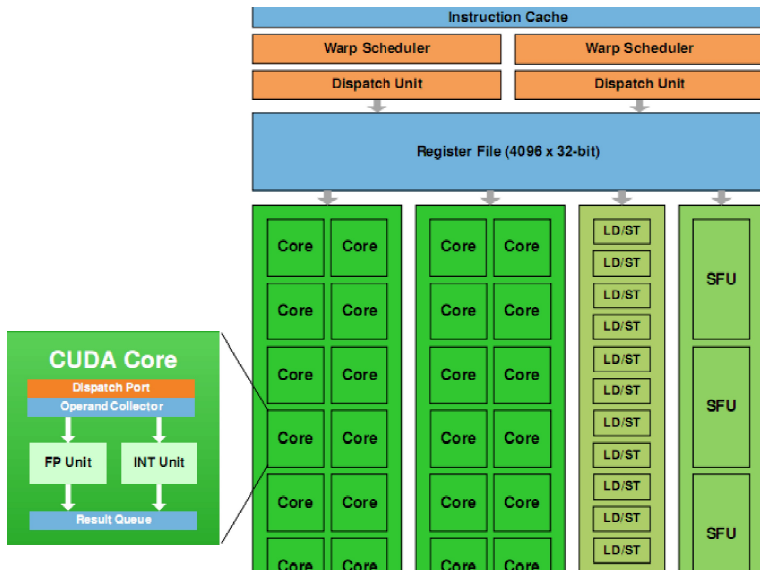
# La struttura fisica

- 1 **Cuda core** e' un termine commerciale inventato da Nvidia per definire le **ALU** che lavorano in float-int nelle schede piu' vecchie. Sinonimi: Streaming processor (**SP**), shader.
- 2 **Streaming Multiprocessor (SM)**: e' una unita' fisica di una scheda grafica, che racchiude:
  - un certo numero di **SP**
  - vari tipi di **memoria** dedicata
  - unita' speciali di calcolo
  - unita' di Load/Store
  - etc...

# Architettura Fermi (vecchia, siamo alla Turing)



# Dettaglio di uno Streaming Multiprocessor (Fermi)



# La struttura logica

- 1 **thread**: piu' piccolo flusso di esecuzione (la gerarchia e' *thread* → *block* → *grid* dal piu' piccolo al piu' grande).
  - Occhio un thread da **solo** non viene mai lanciato! i thread vengono lanciati in un insieme di 32 per volta chiamato **warp** (ne riparleremo piu' avanti)
  - In qualche modo si ha una "relazione" **thread** → **SP** (Attenzione, non e' cosi' banale!)
- 2 **block** Un *blocco* e' un insieme di thread strutturati come un array logico 1D, 2D o 3D. I thread di un blocco vengono identificati tramite delle **magic variables**, che sono degli indici univoci per ogni thread in ogni direzione del blocco: `threadIdx.x`, `threadIdx.y` e `threadIdx.z`. Lungo una direzione (x per esempio) gli *id* dei blocchi variano tra  $[0, n - 1]$ . Dal punto di vista logico, il concetto di blocco e' importante perche' i **thread in un blocco possono comunicare facilmente** tra loro tramite della memoria condivisa (spiegato poi).
  - Un **blocco** puo' essere lanciato su **un solo SM**: (**blocco** → **SM**) (notiamo che la freccia va in una sola direzione!)
- 3 **grid** (*griglia*) e' l'insieme dei blocchi (e quindi dei thread). Una *grid* e' ordinata in un array (griglia) 2D (o 3D) di *blocchi*. La struttura della griglia (e dei blocchi) viene definita **quando** si lancia un kernel.

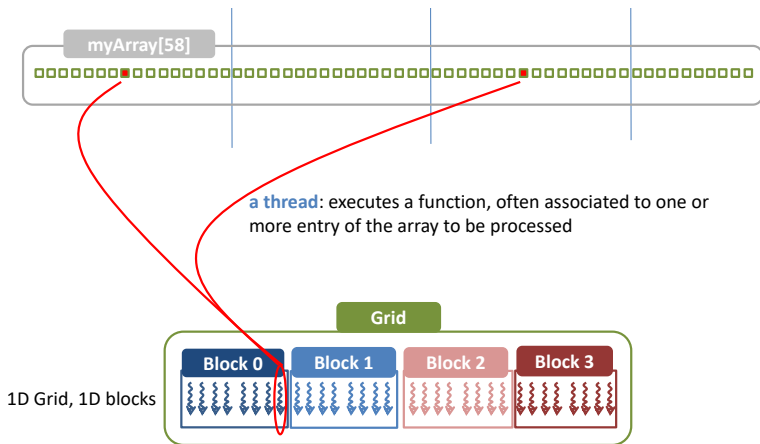
# La struttura logica

- 1 **thread**: piu' piccolo flusso di esecuzione (la gerarchia e' *thread*  $\rightarrow$  *block*  $\rightarrow$  *grid* dal piu' piccolo al piu' grande).
  - Occhio un thread da **solo** non viene mai lanciato! i thread vengono lanciati in un insieme di 32 per volta chiamato **warp** (ne ripareremo piu' avanti)
  - In qualche modo si ha una "relazione" **thread**  $\rightarrow$  **SP** (Attenzione, non e' cosi' banale!)
- 2 **block** Un *blocco* e' un insieme di thread strutturati come un array logico 1D, 2D o 3D. I thread di un blocco vengono identificati tramite delle **magic variables**, che sono degli indici univoci per ogni thread in ogni direzione del blocco: `threadIdx.x`, `threadIdx.y` e `threadIdx.z`. Lungo una direzione (x per esempio) gli *id* dei blocchi variano tra  $[0, n - 1]$ . Dal punto di vista logico, il concetto di blocco e' importante perche' **i thread in un blocco possono comunicare facilmente** tra loro tramite della memoria condivisa (spiegato poi).
  - Un **blocco** puo' essere lanciato su **un solo SM**: (**blocco**  $\rightarrow$  **SM**) (notiamo che la freccia va in una sola direzione!)
- 3 **grid** (*griglia*) e' l'insieme dei blocchi (e quindi dei thread). Una *grid* e' ordinata in un array (griglia) 2D (o 3D) di *blocchi*. La struttura della griglia (e dei blocchi) viene definita **quando** si lancia un kernel.

# La struttura logica

- 1 **thread**: piu' piccolo flusso di esecuzione (la gerarchia e' *thread*  $\rightarrow$  *block*  $\rightarrow$  *grid* dal piu' piccolo al piu' grande).
  - Occhio un thread da **solo** non viene mai lanciato! i thread vengono lanciati in un insieme di 32 per volta chiamato **warp** (ne ripareremo piu' avanti)
  - In qualche modo si ha una "relazione" **thread**  $\rightarrow$  **SP** (Attenzione, non e' cosi' banale!)
- 2 **block** Un *blocco* e' un insieme di thread strutturati come un array logico 1D, 2D o 3D. I thread di un blocco vengono identificati tramite delle **magic variables**, che sono degli indici univoci per ogni thread in ogni direzione del blocco: `threadIdx.x`, `threadIdx.y` e `threadIdx.z`. Lungo una direzione (x per esempio) gli *id* dei blocchi variano tra  $[0, n - 1]$ . Dal punto di vista logico, il concetto di blocco e' importante perche' **i thread in un blocco possono comunicare facilmente** tra loro tramite della memoria condivisa (spiegato poi).
  - Un **blocco** puo' essere lanciato su **un solo SM**: (**blocco**  $\rightarrow$  **SM**) (notiamo che la freccia va in una sola direzione!)
- 3 **grid** (*griglia*) e' l'insieme dei blocchi (e quindi dei thread). Una *grid* e' ordinata in un array (griglia) 2D (o 3D) di *blocchi*. La struttura della griglia (e dei blocchi) viene definita **quando** si lancia un kernel.

# La struttura logica visualizzata



The **grid** often times has a structure (1D, 2D, 3D) which follows the **array** to be processed: this helps the programmer to map the **threads** of the grid to the entries of the array!



# Compilare e lanciare codici sul server

- per compilare sul server: `module load cuda/7.0` (**obsoleto**, ora il compilatore e' disponibile subito)
- i codici vanno salvati con estensione `.cu`
- un codice C necessita dell'**header**: `#include <cuda.h>`
- la compilazione avviene tramite `nvcc codice.cu -o codice.x` (ci sara' una parte del codice compilata da un compilatore per la cpu e uno per la gpu)
- per indicare al compilatore che il codice prodotto deve girare su un hardware con compute capabilities maggiore di, per esempio 1.3: `nvcc -arch=sm_13`

ci sono poi una serie di flag che possono essere usate al momento della compilazione:

- `-G` compila per debuggare il codice GPU.
- `-ptxas-options=-v` mostra l'utilizzo dei registri di memoria
- `-use_fast_math` usa librerie matematiche veloci
- `-maxregcount<N>` limita il numro di registri
- `-arch sm_13` (o `sm_20`) attiva la possibilita' di doppia precisione, indicando la architettura

# Compilare e lanciare codici sul server

- per compilare sul server: `module load cuda/7.0` (**obsoleto**, ora il compilatore e' disponibile subito)
- i codici vanno salvati con estensione `.cu`
- un codice C necessita dell'**header**: `#include <cuda.h>`
- la compilazione avviene tramite `nvcc codice.cu -o codice.x` (ci sara' una parte del codice compilata da un compilatore per la cpu e uno per la gpu)
- per indicare al compilatore che il codice prodotto deve girare su un hardware con compute capabilities maggiore di, per esempio 1.3: `nvcc -arch=sm_13`

ci sono poi una serie di flag che possono essere usate al momento della compilazione:

- `-G` compila per debuggare il codice GPU.
- `-ptxas-options=-v` mostra l'utilizzo dei registri di memoria
- `-use_fast_math` usa librerie matematiche veloci
- `-maxregcount<N>` limita il numro di registri
- `-arch sm_13` (o `sm_20`) attiva la possibilita' di doppia precisione, indicando la architettura

## Interrogare i device

Può essere utile interrogare il computer in modo da ottenere informazioni sul **device** in utilizzo (e' possibile, per esempio, che sullo stesso computer ci sia piu' di un device CUDA)

- `cudaGetDeviceCount( &count )`

Esistono quindi dei tipi predefiniti di CUDA che contengono le proprietà dei device su ogni macchina, in particolare:

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

int main() {
    int nDevices; // indice delle schede sull'host

    cudaGetDeviceCount(&nDevices); //
    for (int i = 0; i < nDevices; i++) { // cicla su tutte le schede
        cudaDeviceProp prop; // definisci l'oggetto propria'
        cudaGetDeviceProperties(&prop, i); // riempi l'oggetto
        printf("Device Number: %d\n", i); // stampa a video i risultati
        printf(" Device name: %s\n", prop.name);
        printf(" Memory Clock Rate (KHz): %d\n", prop.memoryClockRate);
        printf(" Memory Bus Width (bits): %d\n", prop.memoryBusWidth);
        printf(" Peak Memory Bandwidth (GB/s): %f\n\n",
            2.0*prop.memoryClockRate*(prop.memoryBusWidth/8)/1.0e6);
    }
}
```

# Outline

- 1 *Struttura Fisica*
- 2 *Kernel***
- 3 *Struttura Logica*
- 4 *Memorie*
- 5 *Prodotto Matrici*
- 6 *Stream*
- 7 *Esempi e strumenti*

# Un codice con un kernel che non fa nulla

In questo esempio vediamo la prima definizione di un kernel (che ricordo e' una funzione che gira sulla gpu), e lanciamo il kernel stesso all'interno del main:

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

__global__ void miokernel (void) {} // definisce un esempio di kernel cuda

int main(void)
{
    miokernel<<<1,1>>>(); // lancia da host a device il kernel cuda
    printf("hello \n"); // con 1 blocco e 1 thread per blocco.
    return 0;
}
```

- si noti la scrittura `__global__` davanti al kernel. Questa avverte il compilatore che quanto segue deve essere compilato per girare su `device`
- si noti che quando e' stata chiamata la funzione `miokernel` vengono usate le cosiddette `triple angle brackets` `<<<1,1>>>`, (spiegate nel dettaglio in seguito). Dal punto di vista della notazione stanno tra il nome del kernel e le parentesi dove si mettono gli argomenti del kernel stesso.

# Un codice con un kernel che non fa nulla

In questo esempio vediamo la prima definizione di un kernel (che ricordo e' una funzione che gira sulla gpu), e lanciamo il kernel stesso all'interno del main:

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

__global__ void miokernel (void) {} // definisce un esempio di kernel cuda

int main(void)
{
    miokernel<<<1,1>>>(); // lancia da host a device il kernel cuda
    printf("hello \n"); // con 1 blocco e 1 thread per blocco.
    return 0;
}
```

- si noti la scrittura `__global__` davanti al kernel. Questa avverte il compilatore che quanto segue deve essere compilato per girare su **device**
- si noti che quando e' stata chiamata la funzione `miokernel` vengono usate le cosiddette **triple angle brackets** `<<<1,1>>>`, (spiegate nel dettaglio in seguito). Dal punto di vista della notazione stanno tra il nome del kernel e le parentesi dove si mettono gli argomenti del kernel stesso.

# Un codice con un kernel che non fa nulla

In questo esempio vediamo la prima definizione di un kernel (che ricordo e' una funzione che gira sulla gpu), e lanciamo il kernel stesso all'interno del main:

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

__global__ void miokernel (void) {} // definisce un esempio di kernel cuda

int main(void)
{
    miokernel<<<1,1>>>(); // lancia da host a device il kernel cuda
    printf("hello \n"); // con 1 blocco e 1 thread per blocco.
    return 0;
}
```

- si noti la scrittura `__global__` davanti al kernel. Questa avverte il compilatore che quanto segue deve essere compilato per girare su **device**
- si noti che quando e' stata chiamata la funzione `miokernel` vengono usate le cosiddette **triple angle brackets** `<<<1,1>>>`, (spiegate nel dettaglio in seguito). Dal punto di vista della notazione stanno tra il nome del kernel e le parentesi dove si mettono gli argomenti del kernel stesso.

## Un'altro kernel che non fa nulla...

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

__global__ void miokernel (void)
{
    printf("Ciao dal device\n"); // con c.c. >=2.0 printf funge dentro i kernel
}

int main(void)
{
    miokernel<<<1,1>>>(); // lancia da host a device il kernel cuda
    printf("Ciao dall'host \n");
    return 0;
}
```

- Qual'è il risultato dell'esecuzione di questo codice?
- Questo codice **non** dice: *Ciao dal device* ...
- I kernel sono **ASINCRONI**, ovvero restituiscono il controllo al programma chiamante **immediatamente**, che in questo caso dice *Ciao dall'host* e termina l'esecuzione...
- ...prima che il **device** abbia completato la propria esecuzione!
- Per fare funzionare il codice, bisogna mettere (per esempio), prima della fine del programma un: `cudaDeviceSynchronize()`. È una funzione che controlla che i kernel abbiano eseguito prima di ridare controllo al programma chiamante
- Attenzione, `cudaMemcpy()` non sincronizza il `printf` (testato da me)!



## Un'altro kernel che non fa nulla...

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

__global__ void miokernel (void)
{
    printf("Ciao dal device\n"); // con c.c. >=2.0 printf funge dentro i kernel
}

int main(void)
{
    miokernel<<<1,1>>>(); // lancia da host a device il kernel cuda
    printf("Ciao dall'host \n");
    return 0;
}
```

- Qual'è il risultato dell'esecuzione di questo codice?
- Questo codice **non** dice: *Ciao dal device* ...
- I kernel sono **ASINCRONI**, ovvero restituiscono il controllo al programma chiamante **immediatamente**, che in questo caso dice *Ciao dall'host* e termina l'esecuzione...
- ...prima che il **device** abbia completato la propria esecuzione!
- Per fare funzionare il codice, bisogna mettere (per esempio), prima della fine del programma un: `cudaDeviceSynchronize()`. È una funzione che controlla che i kernel abbiano eseguito prima di ridare controllo al programma chiamante
- Attenzione, `cudaMemcpy()` non sincronizza il `printf` (testato da me)!

## Un'altro kernel che non fa nulla...

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

__global__ void miokernel (void)
{
    printf("Ciao dal device\n"); // con c.c. >=2.0 printf funge dentro i kernel
}

int main(void)
{
    miokernel<<<1,1>>>(); // lancia da host a device il kernel cuda
    printf("Ciao dall'host \n");
    return 0;
}
```

- Qual'è il risultato dell'esecuzione di questo codice?
- Questo codice **non** dice: *Ciao dal device* ...
- I kernel sono **ASINCRONI**, ovvero restituiscono il controllo al programma chiamante **immediatamente**, che in questo caso dice *Ciao dall'host* e termina l'esecuzione...
- ...prima che il **device** abbia completato la propria esecuzione!
- Per fare funzionare il codice, bisogna mettere (per esempio), prima della fine del programma un: `cudaDeviceSynchronize()`. È una funzione che controlla che i kernel abbiano eseguito prima di ridare controllo al programma chiamante
- Attenzione, `cudaMemcpy()` non sincronizza il `printf` (testato da me)!

## Un'altro kernel che non fa nulla...

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

__global__ void miokernel (void)
{
    printf("Ciao dal device\n"); // con c.c. >=2.0 printf funge dentro i kernel
}

int main(void)
{
    miokernel<<<1,1>>>(); // lancia da host a device il kernel cuda
    printf("Ciao dall'host \n");
    return 0;
}
```

- Qual'è il risultato dell'esecuzione di questo codice?
- Questo codice **non** dice: *Ciao dal device* ...
- I kernel sono **ASINCRONI**, ovvero restituiscono il controllo al programma chiamante **immediatamente**, che in questo caso dice *Ciao dall'host* e termina l'esecuzione...
- ...prima che il **device** abbia completato la propria esecuzione!
- Per fare funzionare il codice, bisogna mettere (per esempio), prima della fine del programma un: `cudaDeviceSynchronize()`. È una funzione che controlla che i kernel abbiano eseguito prima di ridare controllo al programma chiamante
- Attenzione, `cudaMemcpy()` non sincronizza il `printf` (testato da me)!

## Un'altro kernel che non fa nulla...

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

__global__ void miokernel (void)
{
    printf("Ciao dal device\n"); // con c.c. >=2.0 printf funge dentro i kernel
}

int main(void)
{
    miokernel<<<1,1>>>(); // lancia da host a device il kernel cuda
    printf("Ciao dall'host \n");
    return 0;
}
```

- Qual'è il risultato dell'esecuzione di questo codice?
- Questo codice **non** dice: *Ciao dal device* ...
- I kernel sono **ASINCRONI**, ovvero restituiscono il controllo al programma chiamante **immediatamente**, che in questo caso dice *Ciao dall'host* e termina l'esecuzione...
- ...prima che il **device** abbia completato la propria esecuzione!
- Per fare funzionare il codice, bisogna mettere (per esempio), prima della fine del programma un: `cudaDeviceSynchronize()`. È una funzione che controlla che i kernel abbiano eseguito prima di ridare controllo al programma chiamante
- Attenzione, `cudaMemcpy()` non sincronizza il `printf` (testato da me)!

## Un'altro kernel che non fa nulla...

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

__global__ void miokernel (void)
{
    printf("Ciao dal device\n"); // con c.c. >=2.0 printf funge dentro i kernel
}

int main(void)
{
    miokernel<<<1,1>>>(); // lancia da host a device il kernel cuda
    printf("Ciao dall'host \n");
    return 0;
}
```

- Qual'è il risultato dell'esecuzione di questo codice?
- Questo codice **non** dice: *Ciao dal device* ...
- I kernel sono **ASINCRONI**, ovvero restituiscono il controllo al programma chiamante **immediatamente**, che in questo caso dice *Ciao dall'host* e termina l'esecuzione...
- ...prima che il **device** abbia completato la propria esecuzione!
- Per fare funzionare il codice, bisogna mettere (per esempio), prima della fine del programma un: `cudaDeviceSynchronize()`. E' una funzione che controlla che i kernel abbiano eseguito prima di ridare controllo al programma chiamante
- Attenzione, `cudaMemcpy()` non sincronizza il `printf` (testato da me)!

## Un'altro kernel che non fa nulla...

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

__global__ void miokernel (void)
{
    printf("Ciao dal device\n"); // con c.c. >=2.0 printf funge dentro i kernel
}

int main(void)
{
    miokernel<<<1,1>>>(); // lancia da host a device il kernel cuda
    printf("Ciao dall'host \n");
    return 0;
}
```

- Qual'è il risultato dell'esecuzione di questo codice?
- Questo codice **non** dice: *Ciao dal device* ...
- I kernel sono **ASINCRONI**, ovvero restituiscono il controllo al programma chiamante **immediatamente**, che in questo caso dice *Ciao dall'host* e termina l'esecuzione...
- ...prima che il **device** abbia completato la propria esecuzione!
- Per fare funzionare il codice, bisogna mettere (per esempio), prima della fine del programma un: `cudaDeviceSynchronize()`. E' una funzione che controlla che i kernel abbiano eseguito prima di ridare controllo al programma chiamante
- Attenzione, `cudaMemcpy()` non sincronizza il `printf` (testato da me)!

## Passare parametri ad un kernel: somma

```

#include<stdio.h>
#include<cuda.h>
__global__ void add( int a, int b, int *c ) {
*c = a + b;      // kernel che fa la somma di due interi e deferenzia la variabile risultato
}
int main( void )
{
    int c;        // definisce una variabile c per l'host
    int *dev_c;  // definisce un puntatore ad una variabile: servirà per il device
    cudaMalloc( (void*)&dev_c, sizeof(int) ); // alloca la memoria sul device
    add<<<1,1>>>( 2, 7, dev_c );           // lancia il kernel
    cudaMemcpy( &c, dev_c, sizeof(int),   // copia il risultato da device a host
                cudaMemcpyDeviceToHost );
    printf( "2 + 7 = %d\n", c );
    cudaFree( dev_c );                    // libera la memoria
    return 0;
}

```

- i parametri possono essere passati ad un kernel allo stesso modo in cui si passano ad una funzione (Domanda: come si passano gli argomenti ad una funzione del C?).
- bisogna allocare memoria sul device `cudaMalloc`. La prima differenza rispetto ad un normale `malloc` è nel fatto che la memoria viene allocata sul device rispetto che sull'host.
- il sistema tramite funzioni a runtime gestisce il passaggio di memoria da host a device (occhio che questo passaggio può essere molto lento)

## Passare parametri ad un kernel: somma

```

#include<stdio.h>
#include<cuda.h>
__global__ void add( int a, int b, int *c ) {
*c = a + b;      // kernel che fa la somma di due interi e deferenzia la variabile risultato
}
int main( void )
{
    int c;        // definisce una variabile c per l'host
    int *dev_c;  // definisce un puntatore ad una variabile: servirà per il device
    cudaMalloc( (void*)&dev_c, sizeof(int) ); // alloca la memoria sul device
    add<<<1,1>>>( 2, 7, dev_c );           // lancia il kernel
    cudaMemcpy( &c, dev_c, sizeof(int),   // copia il risultato da device a host
                cudaMemcpyDeviceToHost );
    printf( "2 + 7 = %d\n", c );
    cudaFree( dev_c );                    // libera la memoria
    return 0;
}

```

- i parametri possono essere passati ad un **kernel** allo stesso modo in cui si passano ad una funzione (Domanda: come si passano gli argomenti ad una funzione del C?).
- bisogna **allocare** memoria sul device `cudaMalloc`. La prima differenza rispetto ad un normale `malloc` è nel fatto che la memoria viene allocata sul **device** rispetto che sull'**host**.
- il sistema tramite funzioni a runtime gestisce il passaggio di memoria da host a device (occhio che questo passaggio può essere molto lento)



## Passare parametri ad un kernel: somma

```

#include<stdio.h>
#include<cuda.h>
__global__ void add( int a, int b, int *c ) {
*c = a + b;      // kernel che fa la somma di due interi e deferenzia la variabile risultato
}
int main( void )
{
    int c;        // definisce una variabile c per l'host
    int *dev_c;  // definisce un puntatore ad una variabile: servira' per il device
    cudaMalloc( (void*)&dev_c, sizeof(int) ); // alloca la memoria sul device
    add<<<1,1>>>( 2, 7, dev_c );           // lancia il kernel
    cudaMemcpy( &c, dev_c, sizeof(int),   // copia il risultato da device a host
                cudaMemcpyDeviceToHost );
    printf( "2 + 7 = %d\n", c );
    cudaFree( dev_c );                    // libera la memoria
    return 0;
}

```

- i parametri possono essere passati ad un **kernel** allo stesso modo in cui si passano ad una funzione (Domanda: come si passano gli argomenti ad una funzione del C?).
- bisogna **allocare** memoria sul device `cudaMalloc`. La prima differenza rispetto ad un normale `malloc` e' nel fatto che la memoria viene allocata sul **device** rispetto che sull'**host**.
- il sistema tramite funzioni a runtime gestisce il passaggio di memoria da host a device (occhio che questo passaggio puo' essere molto lento)

## Passare parametri ad un kernel: somma

```

#include<stdio.h>
#include<cuda.h>
__global__ void add( int a, int b, int *c ) {
*c = a + b;      // kernel che fa la somma di due interi e deferenzia la variabile risultato
}
int main( void )
{
    int c;        // definisce una variabile c per l'host
    int *dev_c;  // definisce un puntatore ad una variabile: servira' per il device
    cudaMalloc( (void*)&dev_c, sizeof(int) ); // alloca la memoria sul device
    add<<<1,1>>>( 2, 7, dev_c );           // lancia il kernel
    cudaMemcpy( &c, dev_c, sizeof(int),   // copia il risultato da device a host
                cudaMemcpyDeviceToHost );
    printf( "2 + 7 = %d\n", c );
    cudaFree( dev_c );                    // libera la memoria
    return 0;
}

```

- i parametri possono essere passati ad un **kernel** allo stesso modo in cui si passano ad una funzione (Domanda: come si passano gli argomenti ad una funzione del C?).
- bisogna **allocare** memoria sul device `cudaMalloc`. La prima differenza rispetto ad un normale `malloc` e' nel fatto che la memoria viene allocata sul **device** rispetto che sull'**host**.
- il sistema tramite funzioni a runtime gestisce il passaggio di memoria da host a device (occhio che questo passaggio puo' essere molto lento)

# cudaMalloc

Dato un puntatore: `int *dev_c`,  
la funzione `cudaMalloc ( (void**) &dev\_c, sizeof(int) )` ha 2 argomenti:

- 1 il puntatore del puntatore della variabile dove si vuole mettere la memoria
- 2 la grandezza in `byte` della memoria da allocare

**E' responsabilita' del programmatore non** deferenziare il puntatore restituito da `cudaMalloc` dal codice che esegue sull'**host**.  
il codice sull'**host** puo' :

- passare puntatori, allocati con `cudaMalloc`, a funzioni che eseguono sul `device`
- il codice che esegue sul `device` puo' leggere e scrivere su un puntatore che e' stato allocato con `cudaMalloc` da `host`.
- si possono passare i puntatori definiti con `cudaMalloc` a funzioni che eseguono sull'`host` (ma che non devono)
- si possono fare delle operazioni aritmetiche su di essi
- si puo' fare un cast su un tipo differente

## ATTENZIONE:

il codice dell'`host` **NON** puo' usare questo puntatore per leggere o scrivere dalla memoria: perche'? l'indirizzo di memoria definito da un `cudaMalloc` non ha corrispondenza nell'`host`.

IMPORTANTE: per **liberare** la memoria definita con `cudaMalloc()` si deve usare `cudaFree()` (non il semplice `free()` del C).

# cudaMalloc

Dato un puntatore: `int *dev_c`,  
la funzione `cudaMalloc ( (void**) &dev\_c, sizeof(int) )` ha 2 argomenti:

- 1 il puntatore del puntatore della variabile dove si vuole mettere la memoria
- 2 la grandezza in `byte` della memoria da allocare

**E' responsabilita' del programmatore non** deferenziare il puntatore restituito da `cudaMalloc` dal codice che esegue sull'**host**.  
il codice sull'**host puo'**:

- passare puntatori, allocati con `cudaMalloc`, a funzioni che eseguono sul `device`
- il codice che esegue sul `device` puo' leggere e scrivere su un puntatore che e' stato allocato con `cudaMalloc` da `host`.
- si possono passare i puntatori definiti con `cudaMalloc` a funzioni che eseguono sull'`host` (ma che non devono)
- si possono fare delle operazioni aritmetiche su di essi
- si puo' fare un cast su un tipo differente

## ATTENZIONE:

il codice dell'`host` **NON** puo' usare questo puntatore per leggere o scrivere dalla memoria: perche'? l'indirizzo di memoria definito da un `cudaMalloc` non ha corrispondenza nell'`host`.

IMPORTANTE: per **liberare** la memoria definita con `cudaMalloc()` si deve usare `cudaFree()` (non il semplice `free()` del C).

# cudaMalloc

Dato un puntatore: `int *dev_c`,  
la funzione `cudaMalloc ( (void**) &dev\_c, sizeof(int) )` ha 2 argomenti:

- 1 il puntatore del puntatore della variabile dove si vuole mettere la memoria
- 2 la grandezza in `byte` della memoria da allocare

**E' responsabilita' del programmatore non** deferenziare il puntatore restituito da `cudaMalloc` dal codice che esegue sull'**host**.  
il codice sull'**host puo'**:

- passare puntatori, allocati con `cudaMalloc`, a funzioni che eseguono sul `device`
- il codice che esegue sul `device` puo' leggere e scrivere su un puntatore che e' stato allocato con `cudaMalloc` da `host`.
- si possono passare i puntatori definiti con `cudaMalloc` a funzioni che eseguono sull'`host` (ma che non devono)
- si possono fare delle operazioni aritmetiche su di essi
- si puo' fare un cast su un tipo differente

## ATTENZIONE:

il codice dell'`host` **NON** puo' usare questo puntatore per leggere o scrivere dalla memoria: perche'? l'indirizzo di memoria definito da un `cudaMalloc` non ha corrispondenza nell'`host`.

**IMPORTANTE:** per **liberare** la memoria definita con `cudaMalloc()` si deve usare `cudaFree()` (non il semplice `free()` del C).

# cudaMalloc

Dato un puntatore: `int *dev_c`,  
la funzione `cudaMalloc ( (void**) &dev\_c, sizeof(int) )` ha 2 argomenti:

- 1 il puntatore del puntatore della variabile dove si vuole mettere la memoria
- 2 la grandezza in `byte` della memoria da allocare

**E' responsabilita' del programmatore non** deferenziare il puntatore restituito da `cudaMalloc` dal codice che esegue sull'**host**.  
il codice sull'**host puo'**:

- passare puntatori, allocati con `cudaMalloc`, a funzioni che eseguono sul `device`
- il codice che esegue sul `device` puo' leggere e scrivere su un puntatore che e' stato allocato con `cudaMalloc` da `host`.
- si possono passare i puntatori definiti con `cudaMalloc` a funzioni che eseguono sull'`host` (ma che non devono)
- si possono fare delle operazioni aritmetiche su di essi
- si puo' fare un cast su un tipo differente

## ATTENZIONE:

il codice dell'`host` **NON** puo' usare questo puntatore per leggere o scrivere dalla memoria: perche'? l'indirizzo di memoria definito da un `cudaMalloc` non ha corrispondenza nell'`host`.

IMPORTANTE: per **liberare** la memoria definita con `cudaMalloc()` si deve usare `cudaFree()` (non il semplice `free()` del C).

# cudaMalloc

Dato un puntatore: `int *dev_c`,  
la funzione `cudaMalloc ( (void**) &dev\_c, sizeof(int) )` ha 2 argomenti:

- 1 il puntatore del puntatore della variabile dove si vuole mettere la memoria
- 2 la grandezza in `byte` della memoria da allocare

**E' responsabilita' del programmatore non** deferenziare il puntatore restituito da `cudaMalloc` dal codice che esegue sull'**host**.  
il codice sull'**host** **puo'**:

- passare puntatori, allocati con `cudaMalloc`, a funzioni che eseguono sul `device`
- il codice che esegue sul `device` puo' leggere e scrivere su un puntatore che e' stato allocato con `cudaMalloc` da `host`.
- si possono passare i puntatori definiti con `cudaMalloc` a funzioni che eseguono sull'`host` (ma che non devono)
- si possono fare delle operazioni aritmetiche su di essi
- si puo' fare un cast su un tipo differente

## ATTENZIONE:

il codice dell'`host` **NON** puo' usare questo puntatore per leggere o scrivere dalla memoria: perche'? l'indirizzo di memoria definito da un `cudaMalloc` non ha corrispondenza nell'`host`.

IMPORTANTE: per **liberare** la memoria definita con `cudaMalloc()` si deve usare `cudaFree()` (non il semplice `free()` del C).

# cudaMalloc

Dato un puntatore: `int *dev_c`,  
la funzione `cudaMalloc ( (void**) &dev\_c, sizeof(int) )` ha 2 argomenti:

- 1 il puntatore del puntatore della variabile dove si vuole mettere la memoria
- 2 la grandezza in `byte` della memoria da allocare

**E' responsabilita' del programmatore non** deferenziare il puntatore restituito da `cudaMalloc` dal codice che esegue sull'**host**.  
il codice sull'**host** **puo'**:

- passare puntatori, allocati con `cudaMalloc`, a funzioni che eseguono sul `device`
- il codice che esegue sul `device` puo' leggere e scrivere su un puntatore che e' stato allocato con `cudaMalloc` da `host`.
- si possono passare i puntatori definiti con `cudaMalloc` a funzioni che eseguono sull'`host` (ma che non devono)
- si possono fare delle operazioni aritmetiche su di essi
- si puo' fare un cast su un tipo differente

## ATTENZIONE:

il codice dell'`host` **NON** puo' usare questo puntatore per leggere o scrivere dalla memoria: perche'? l'indirizzo di memoria definito da un `cudaMalloc` non ha corrispondenza nell'`host`.

IMPORTANTE: per **liberare** la memoria definita con `cudaMalloc()` si deve usare `cudaFree()` (non il semplice `free()` del C).



# cudaMalloc

Dato un puntatore: `int *dev_c`,  
la funzione `cudaMalloc ( (void**) &dev\_c, sizeof(int) )` ha 2 argomenti:

- 1 il puntatore del puntatore della variabile dove si vuole mettere la memoria
- 2 la grandezza in `byte` della memoria da allocare

**E' responsabilita' del programmatore non** deferenziare il puntatore restituito da `cudaMalloc` dal codice che esegue sull'**host**.  
il codice sull'**host puo'**:

- passare puntatori, allocati con `cudaMalloc`, a funzioni che eseguono sul `device`
- il codice che esegue sul `device` puo' leggere e scrivere su un puntatore che e' stato allocato con `cudaMalloc` da `host`.
- si possono passare i puntatori definiti con `cudaMalloc` a funzioni che eseguono sull'`host` (ma che non devono)
- si possono fare delle operazioni aritmetiche su di essi
- si puo' fare un cast su un tipo differente

## ATTENZIONE:

il codice dell'`host` **NON** puo' usare questo puntatore per leggere o scrivere dalla memoria: perche' l'indirizzo di memoria definito da un `cudaMalloc` non ha corrispondenza nell'`host`.

IMPORTANTE: per `liberare` la memoria definita con `cudaMalloc()` si deve usare `cudaFree()` (non il semplice `free()` del C).

# cudaMalloc

Dato un puntatore: `int *dev_c`,  
la funzione `cudaMalloc ( (void**) &dev\_c, sizeof(int) )` ha 2 argomenti:

- 1 il puntatore del puntatore della variabile dove si vuole mettere la memoria
- 2 la grandezza in `byte` della memoria da allocare

**E' responsabilita' del programmatore non** deferenziare il puntatore restituito da `cudaMalloc` dal codice che esegue sull'**host**.  
il codice sull'**host puo'**:

- passare puntatori, allocati con `cudaMalloc`, a funzioni che eseguono sul `device`
- il codice che esegue sul `device` puo' leggere e scrivere su un puntatore che e' stato allocato con `cudaMalloc` da `host`.
- si possono passare i puntatori definiti con `cudaMalloc` a funzioni che eseguono sull'`host` (ma che non devono)
- si possono fare delle operazioni aritmetiche su di essi
- si puo' fare un cast su un tipo differente

## ATTENZIONE:

il codice dell'`host` **NON** puo' usare questo puntatore per leggere o scrivere dalla memoria: perche'? l'indirizzo di memoria definito da un `cudaMalloc` non ha corrispondenza nell'`host`.

IMPORTANTE: per **liberare** la memoria definita con `cudaMalloc()` si deve usare `cudaFree()` (non il semplice `free()` del C).

# cudaMalloc

Dato un puntatore: `int *dev_c`,  
la funzione `cudaMalloc ( (void**) &dev\_c, sizeof(int) )` ha 2 argomenti:

- 1 il puntatore del puntatore della variabile dove si vuole mettere la memoria
- 2 la grandezza in `byte` della memoria da allocare

**E' responsabilita' del programmatore non** deferenziare il puntatore restituito da `cudaMalloc` dal codice che esegue sull'**host**.  
il codice sull'**host** **puo'**:

- passare puntatori, allocati con `cudaMalloc`, a funzioni che eseguono sul `device`
- il codice che esegue sul `device` puo' leggere e scrivere su un puntatore che e' stato allocato con `cudaMalloc` da `host`.
- si possono passare i puntatori definiti con `cudaMalloc` a funzioni che eseguono sull'`host` (ma che non devono)
- si possono fare delle operazioni aritmetiche su di essi
- si puo' fare un cast su un tipo differente

## ATTENZIONE:

il codice dell'`host` **NON** puo' usare questo puntatore per leggere o scrivere dalla memoria: perche'? l'indirizzo di memoria definito da un `cudaMalloc` non ha corrispondenza nell'`host`.

**IMPORTANTE:** per **liberare** la memoria definita con `cudaMalloc()` si deve usare `cudaFree()` (non il semplice `free()` del C).

## *cudaMemcpy: copiare la memoria*

Per copiare una variabile dal **device** all'**host** (o viceversa) bisogna utilizzare un comando del tipo:

```
cudaError_t cudaMemcpy ( void * dst,      // puntatore a destinazione
                        const void * src,  // puntatore a sorgente
                        size_t count,     // dimensione in byte
                        enum cudaMemcpyKind kind // tipo di copia
                      )
```

Nel dettaglio nell'esempio della diapositiva precedente si ha:

```
cudaMemcpy (          &c,      // puntatore destinazione
            dev_c,        // puntatore sorgente
            sizeof(int),    // dimensione in byte
            cudaMemcpyDeviceToHost // copiamo DA Device A Host
          ) ;
```

**Attenzione** `cudaMemcpy` e' una funzione **sincrona**: **non** restituisce il controllo al codice chiamante prima del completamento  
Ci sono altre possibili specifiche su come utilizzare la `cudaMemcpy`:

- `cudaMemcpyDeviceToHost`
- `cudaMemcpyHostToDevice`
- `cudaMemcpyDeviceToDevice`
- `cudaMemcpyHostToHost`

## *cudaMemcpy: copiare la memoria*

Per copiare una variabile dal **device** all'**host** (o viceversa) bisogna utilizzare un comando del tipo:

```
cudaError_t cudaMemcpy ( void * dst,      // puntatore a destinazione
                        const void * src, // puntatore a sorgente
                        size_t count,    // dimensione in byte
                        enum cudaMemcpyKind kind // tipo di copia
                      )
```

Nel dettaglio nell'esempio della diapositiva precedente si ha:

```
cudaMemcpy (          &c,      // puntatore destinazione
             dev_c,      // puntatore sorgente
             sizeof(int), // dimensione in byte
             cudaMemcpyDeviceToHost // copiamo DA Device A Host
           ) ;
```

**Attenzione** `cudaMemcpy` e' una funzione **sincrona**: **non** restituisce il controllo al codice chiamante prima del completamento  
Ci sono altre possibili specifiche su come utilizzare la `cudaMemcpy`:

- `cudaMemcpyDeviceToHost`
- `cudaMemcpyHostToDevice`
- `cudaMemcpyDeviceToDevice`
- `cudaMemcpyHostToHost`

## *cudaMemcpy: copiare la memoria*

Per copiare una variabile dal **device** all'**host** (o viceversa) bisogna utilizzare un comando del tipo:

```
cudaError_t cudaMemcpy ( void * dst,      // puntatore a destinazione
                        const void * src, // puntatore a sorgente
                        size_t count,    // dimensione in byte
                        enum cudaMemcpyKind kind // tipo di copia
                      )
```

Nel dettaglio nell'esempio della diapositiva precedente si ha:

```
cudaMemcpy (          &c,      // puntatore destinazione
             dev_c,      // puntatore sorgente
             sizeof(int), // dimensione in byte
             cudaMemcpyDeviceToHost // copiamo DA Device A Host
           ) ;
```

**Attenzione** `cudaMemcpy` e' una funzione **sincrona**: **non** restituisce il controllo al codice chiamante prima del completamento

Ci sono altre possibili specifiche su come utilizzare la `cudaMemcpy`:

- `cudaMemcpyDeviceToHost`
- `cudaMemcpyHostToDevice`
- `cudaMemcpyDeviceToDevice`
- `cudaMemcpyHostToHost`

## *cudaMemcpy: copiare la memoria*

Per copiare una variabile dal **device** all'**host** (o viceversa) bisogna utilizzare un comando del tipo:

```
cudaError_t cudaMemcpy ( void * dst,      // puntatore a destinazione
                        const void * src, // puntatore a sorgente
                        size_t count,    // dimensione in byte
                        enum cudaMemcpyKind kind // tipo di copia
                      )
```

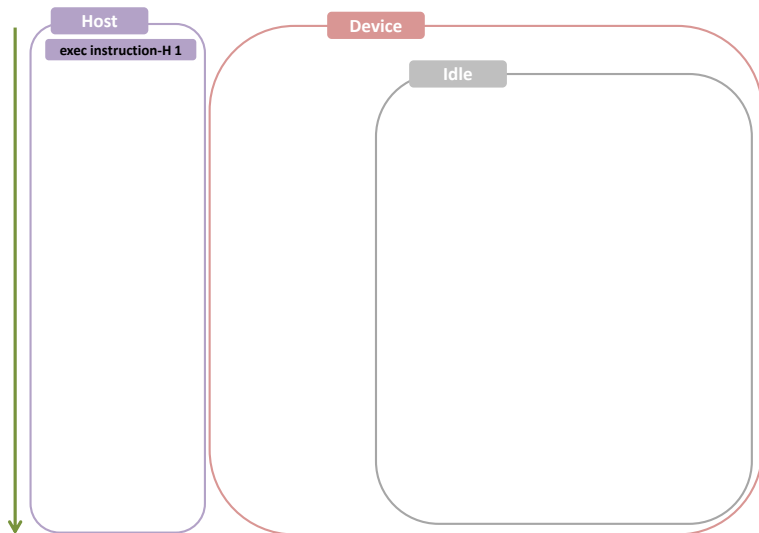
Nel dettaglio nell'esempio della diapositiva precedente si ha:

```
cudaMemcpy (          &c,      // puntatore destinazione
             dev_c,      // puntatore sorgente
             sizeof(int), // dimensione in byte
             cudaMemcpyDeviceToHost // copiamo DA Device A Host
           ) ;
```

**Attenzione** `cudaMemcpy` e' una funzione **sincrona**: **non** restituisce il controllo al codice chiamante prima del completamento  
 Ci sono altre possibili specifiche su come utilizzare la `cudaMemcpy`:

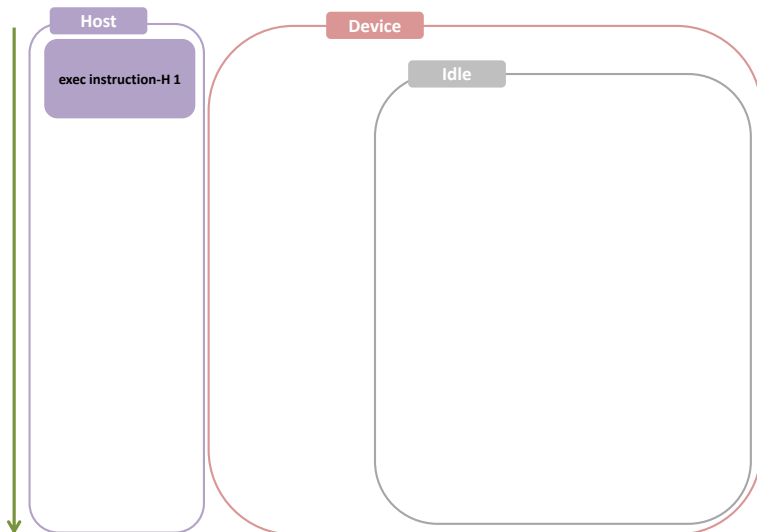
- `cudaMemcpyDeviceToHost`
- `cudaMemcpyHostToDevice`
- `cudaMemcpyDeviceToDevice`
- `cudaMemcpyHostToHost`

# Sincronizzazione: un esempio

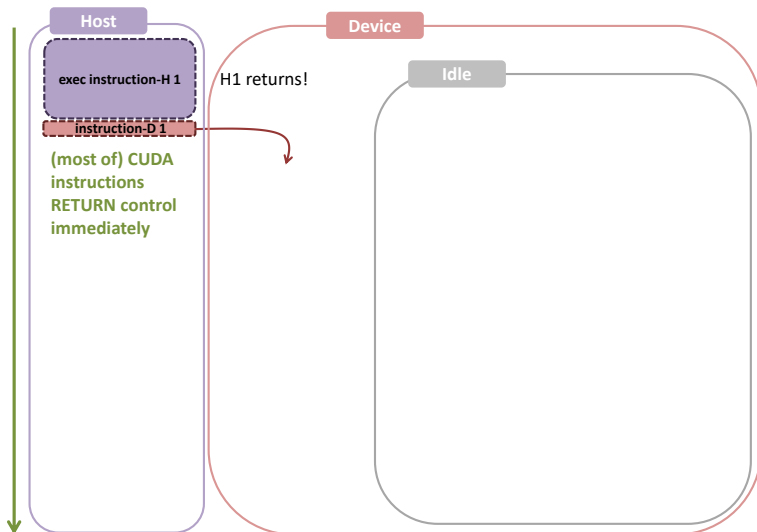




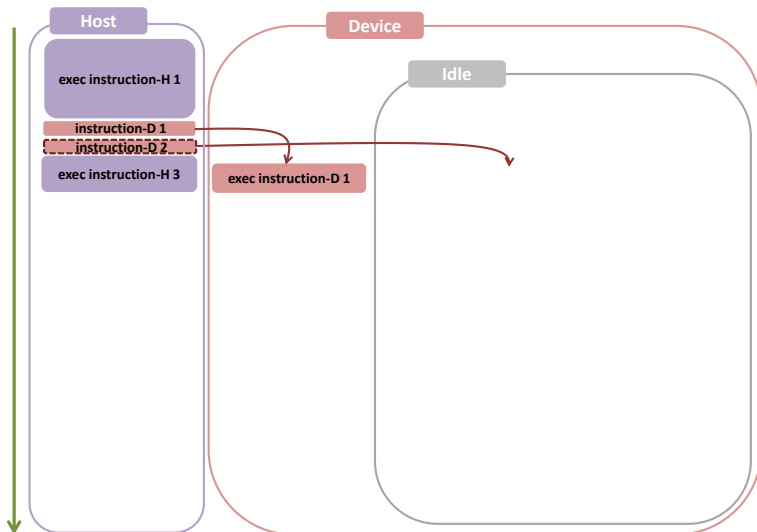
# Sincronizzazione: un esempio



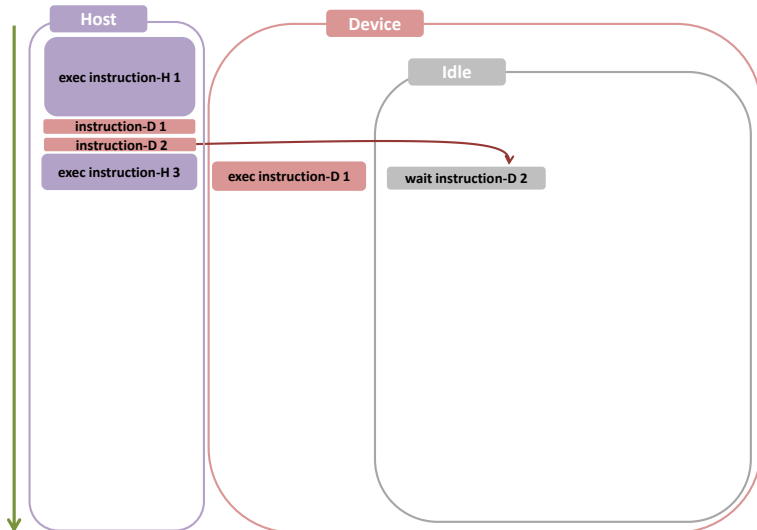
# Sincronizzazione: un esempio



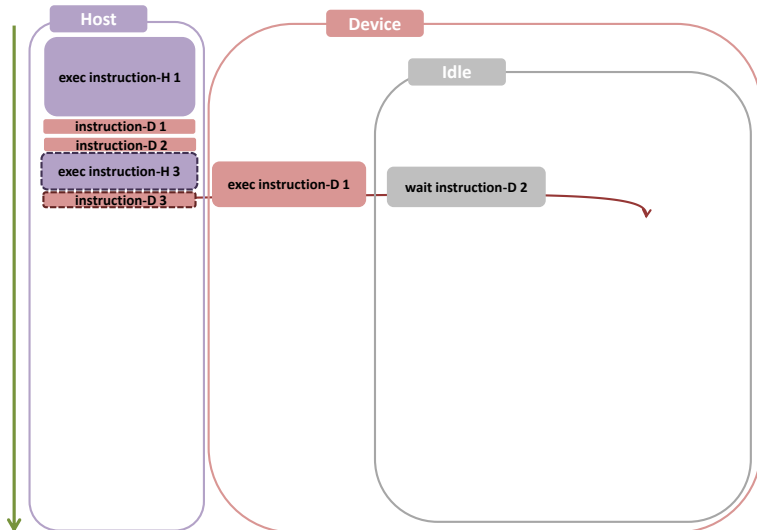
# Sincronizzazione: un esempio



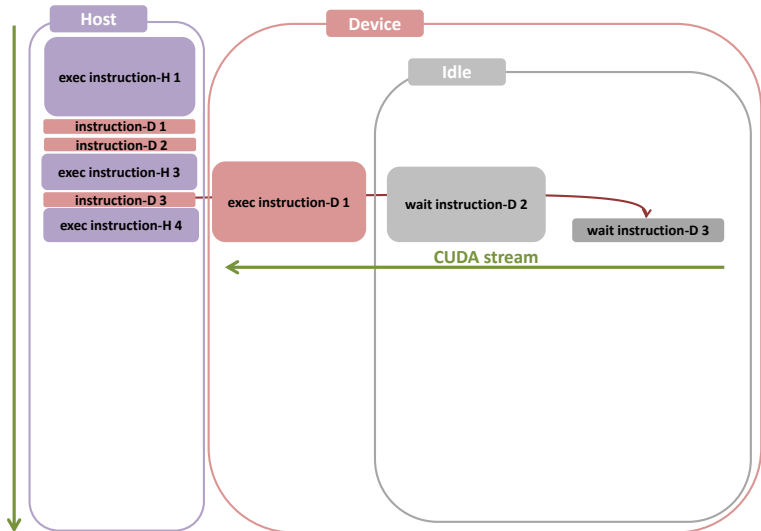
# Sincronizzazione: un esempio



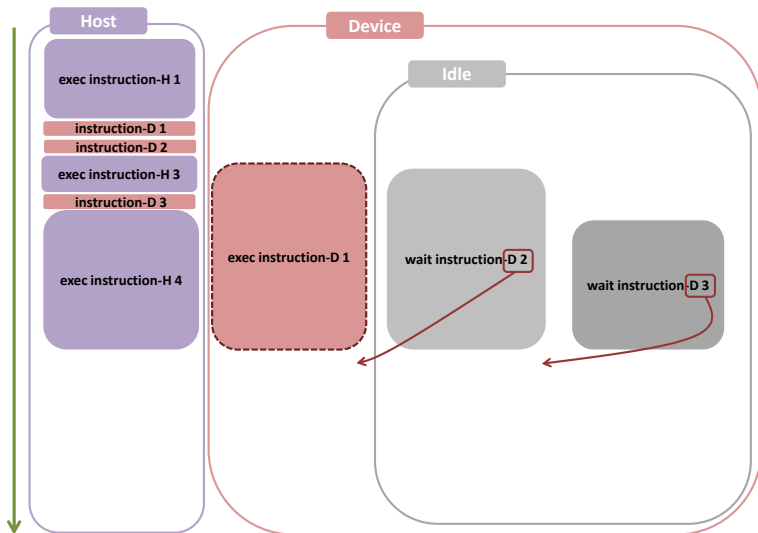
# Sincronizzazione: un esempio



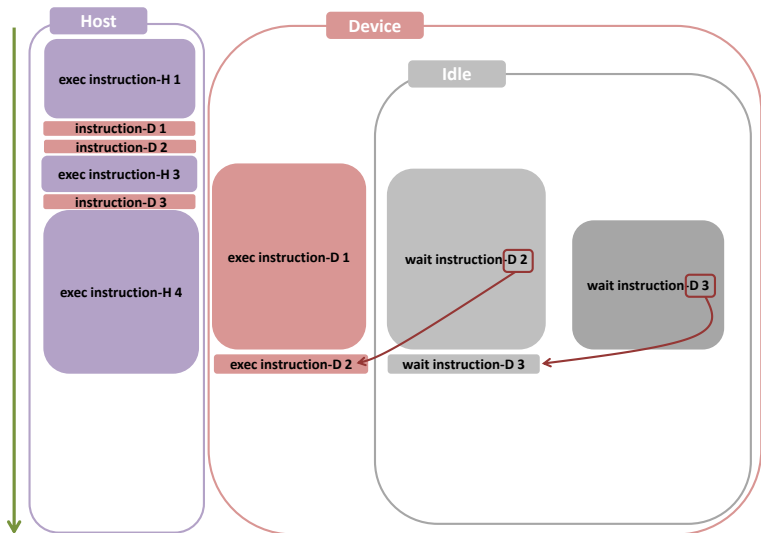
# Sincronizzazione: un esempio



# Sincronizzazione: un esempio

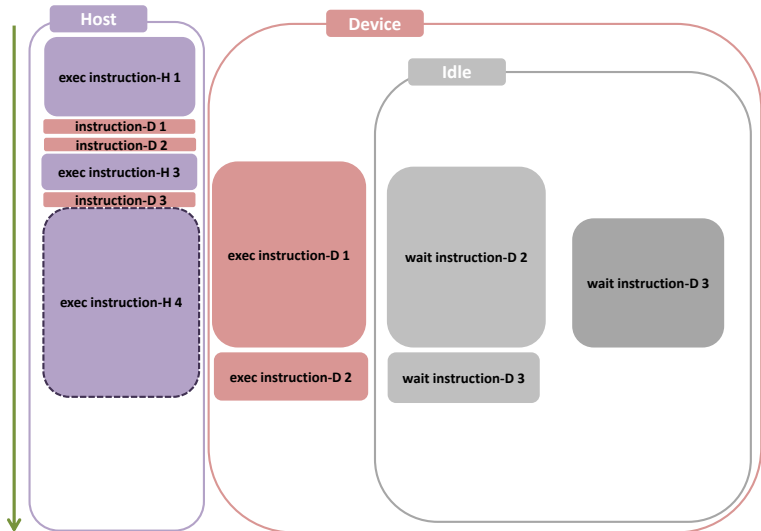


# Sincronizzazione: un esempio

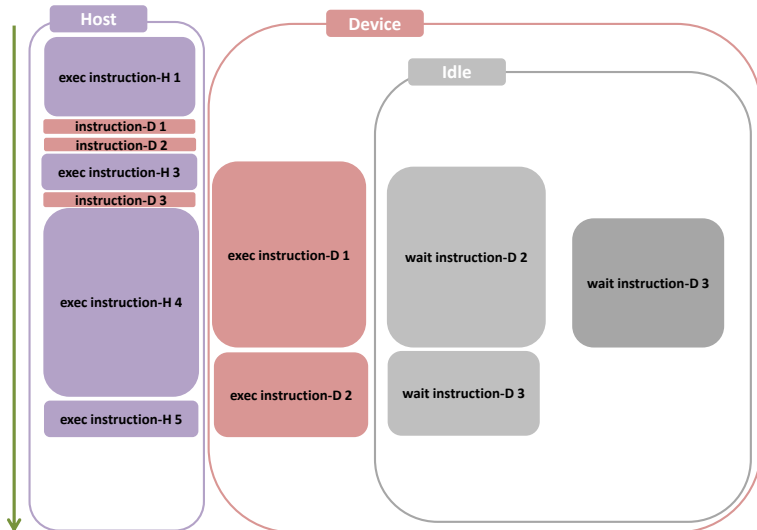




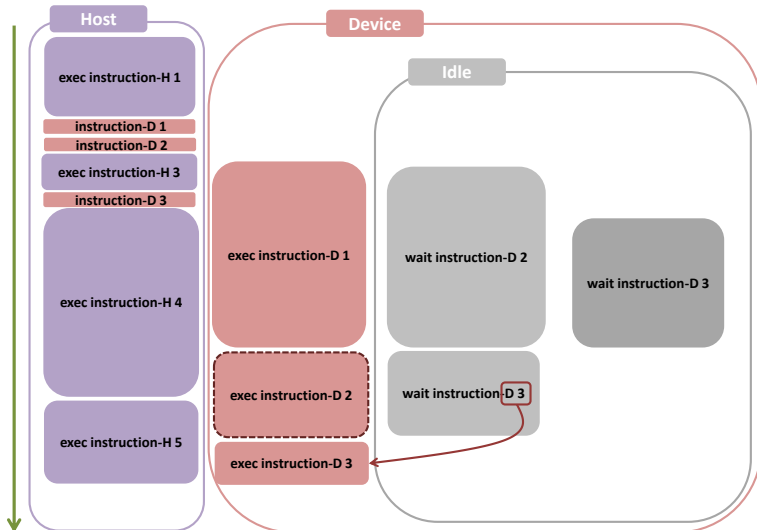
# Sincronizzazione: un esempio



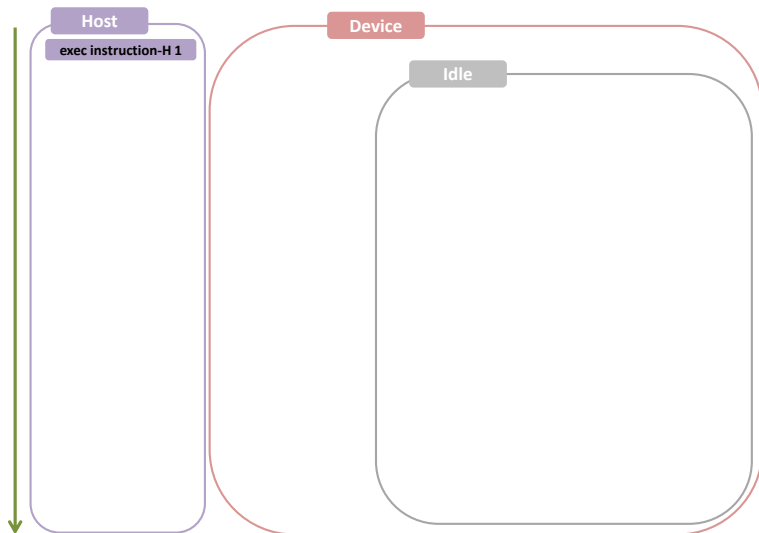
# Sincronizzazione: un esempio



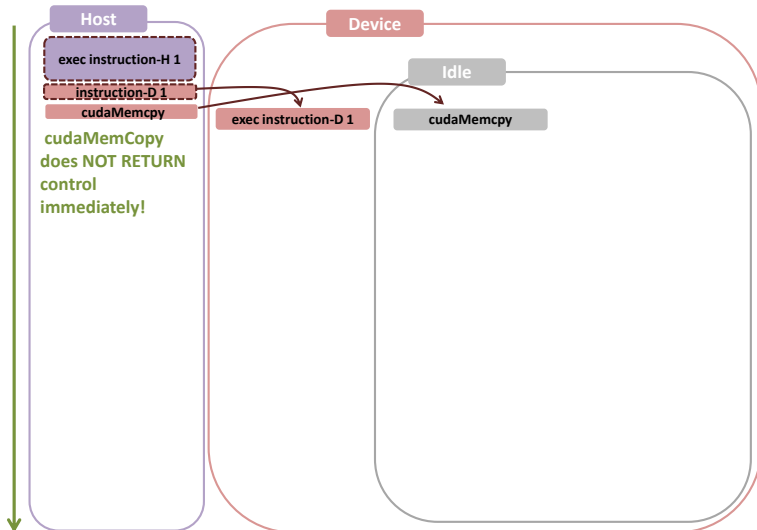
# Sincronizzazione: un esempio



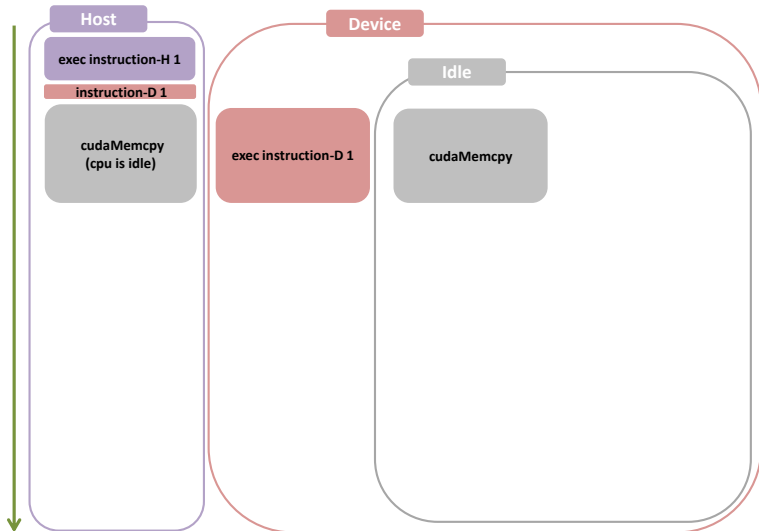
# Sincronizzazione: *cudaMemcpy*



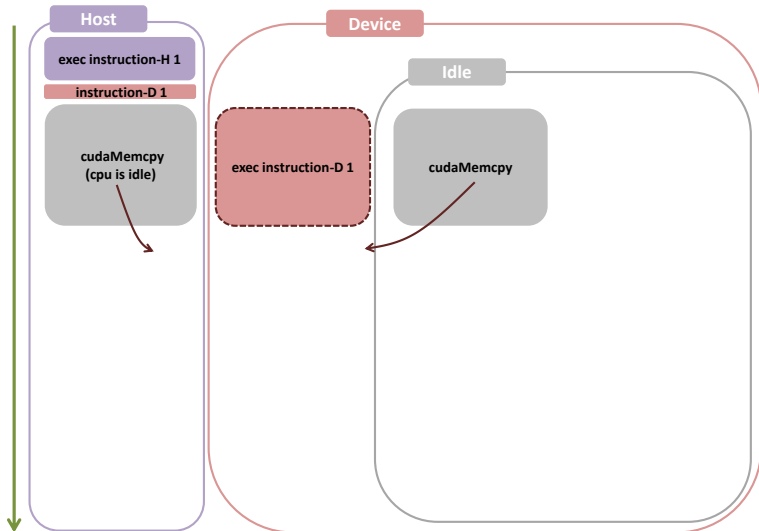
# Sincronizzazione: *cudaMemcpy*



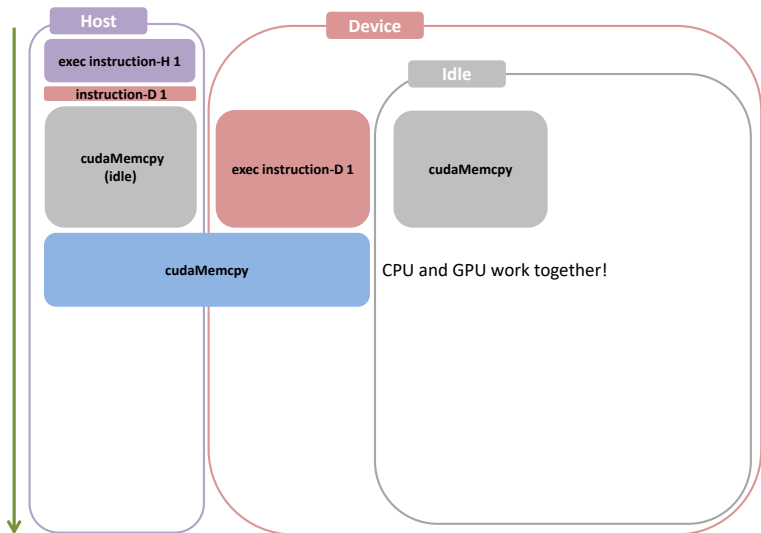
# Sincronizzazione: *cudaMemcpy*



# Sincronizzazione: *cudaMemcpy*

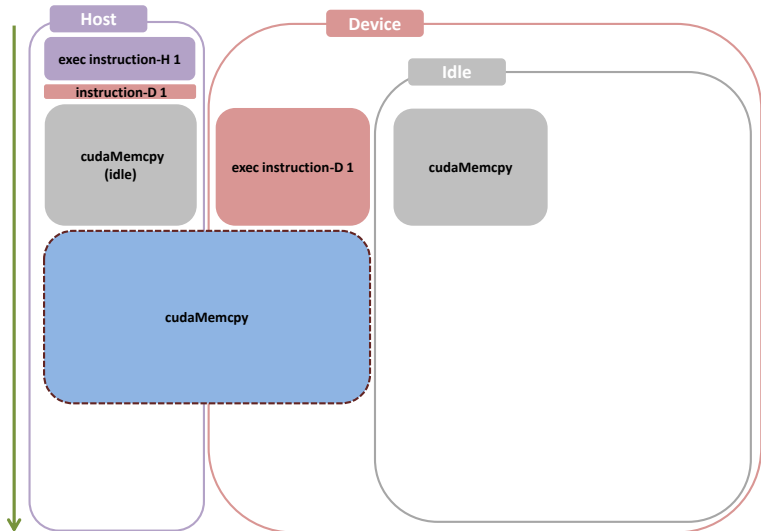


# Sincronizzazione: *cudaMemcpy*

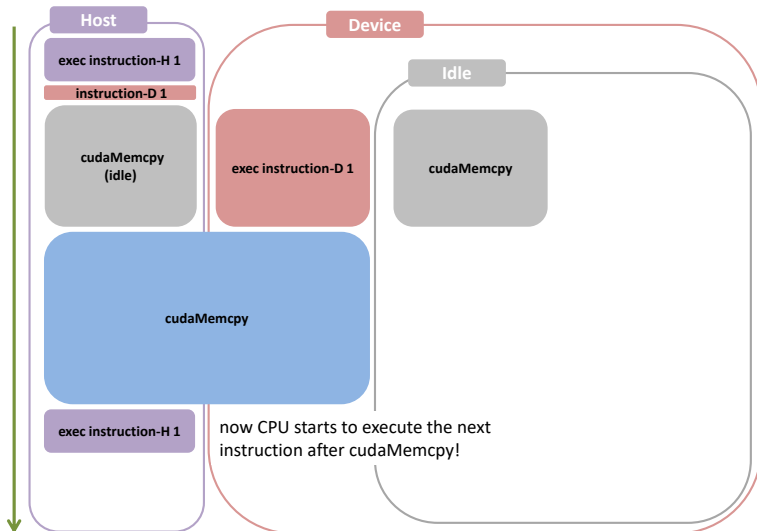




# Sincronizzazione: *cudaMemcpy*



# Sincronizzazione: *cudaMemcpy*



## Riassunto della struttura di un main

Puo' avere senso prendere un codice seriale e modificarlo per sfruttare la GPU. In questo caso, la prima cosa da fare e' **identificare** quali parti del codice sarebbero avvantaggiate da un **massiccio parallelismo** (**non** ha senso usare la GPU per 10 thread, perche' anche solo l'overhead di copiare i dati dall'host al device renderebbe il calcolo piu' lento!).

Struttura:

- 1 si dichiarano i **puntatori** a delle variabili da usare sul **device** (dichiarare vs definire?). Nota: al livello della **dichiarazione** i puntatori non sono ancora assegnati ne all'host ne al device. **Ha senso** usare dei nomi che ci ricordino quali puntatori devono essere usati dal device!
- 2 con `cudaMalloc` si allocca la memoria sul **device**. Solo in questo momento i puntatori dichiarati precedentemente vengono **assegnati al device**.
- 3 con `cudaMemcpy` si copia la memoria necessaria per il kernel da **host** a **device**
- 4 con le **triple angle brackets** viene gestito il lancio dei **kernel**.
- 5 con `cudaMemcpy` si copia da **device** a **host** il risultato ottenuto
- 6 si libera la memoria dal **device** con `cudaFree`
- 7 **Ovviamente**, bisogna avere modificato una *normale funzione* in modo che diventi il **kernel** da fare girare sul **device** (perlomeno serve il decoratore `__global__`)

## Riassunto della struttura di un main

Puo' avere senso prendere un codice seriale e modificarlo per sfruttare la GPU. In questo caso, la prima cosa da fare e' **identificare** quali parti del codice sarebbero avvantaggiate da un **massiccio parallelismo** (**non** ha senso usare la GPU per 10 thread, perche' anche solo l'overhead di copiare i dati dall'host al device renderebbe il calcolo piu' lento!).

Struttura:

- 1 si dichiarano i **puntatori** a delle variabili da usare sul **device** (dichiarare vs definire?). Nota: al livello della **dichiarazione** i puntatori non sono ancora assegnati ne all'host ne al device. **Ha senso** usare dei nomi che ci ricordino quali puntatori devono essere usati dal device!
- 2 con `cudaMalloc` si alloca la memoria sul **device**. Solo in questo momento i puntatori dichiarati precedentemente vengono **assegnati al device**.
- 3 con `cudaMemcpy` si copia la memoria necessaria per il kernel da **host** a **device**
- 4 con le **triple angle brackets** viene gestito il lancio dei **kernel**.
- 5 con `cudaMemcpy` si copia da **device** a **host** il risultato ottenuto
- 6 si libera la memoria dal **device** con `cudaFree`
- 7 **Ovviamente**, bisogna avere modificato una *normale funzione* in modo che diventi il **kernel** da fare girare sul **device** (perlomeno serve il decoratore `__global__`)

## Riassunto della struttura di un main

Puo' avere senso prendere un codice seriale e modificarlo per sfruttare la GPU. In questo caso, la prima cosa da fare e' **identificare** quali parti del codice sarebbero avvantaggiate da un **massiccio parallelismo** (**non** ha senso usare la GPU per 10 thread, perche' anche solo l'overhead di copiare i dati dall'host al device renderebbe il calcolo piu' lento!).

Struttura:

- 1 si dichiarano i **puntatori** a delle variabili da usare sul **device** (dichiarare vs definire?). Nota: al livello della **dichiarazione** i puntatori non sono ancora assegnati ne all'host ne al device. **Ha senso** usare dei nomi che ci ricordino quali puntatori devono essere usati dal device!
- 2 con `cudaMalloc` si alloca la memoria sul **device**. Solo in questo momento i puntatori dichiarati precedentemente vengono **assegnati al device**.
- 3 con `cudaMemcpy` si copia la memoria necessaria per il kernel da **host** a **device**
- 4 con le **triple angle brackets** viene gestito il lancio dei **kernel**.
- 5 con `cudaMemcpy` si copia da **device** a **host** il risultato ottenuto
- 6 si libera la memoria dal **device** con `cudaFree`
- 7 **Ovviamente**, bisogna avere modificato una *normale funzione* in modo che diventi il kernel da fare girare sul **device** (perlomeno serve il decoratore `__global__`)

## Riassunto della struttura di un main

Puo' avere senso prendere un codice seriale e modificarlo per sfruttare la GPU. In questo caso, la prima cosa da fare e' **identificare** quali parti del codice sarebbero avvantaggiate da un **massiccio parallelismo** (**non** ha senso usare la GPU per 10 thread, perche' anche solo l'overhead di copiare i dati dall'host al device renderebbe il calcolo piu' lento!).

Struttura:

- 1 si dichiarano i **puntatori** a delle variabili da usare sul **device** (dichiarare vs definire?). Nota: al livello della **dichiarazione** i puntatori non sono ancora assegnati ne all'host ne al device. **Ha senso** usare dei nomi che ci ricordino quali puntatori devono essere usati dal device!
- 2 con `cudaMalloc` si allocca la memoria sul **device**. Solo in questo momento i puntatori dichiarati precedentemente vengono **assegnati al device**.
- 3 con `cudaMemcpy` si copia la memoria necessaria per il kernel da **host** a **device**
- 4 con le **triple angle brackets** viene gestito il lancio dei **kernel**.
- 5 con `cudaMemcpy` si copia da **device** a **host** il risultato ottenuto
- 6 si libera la memoria dal **device** con `cudaFree`
- 7 **Ovviamente**, bisogna avere modificato una *normale funzione* in modo che diventi il kernel da fare girare sul **device** (perlomeno serve il decoratore `__global__`)

## Riassunto della struttura di un main

Puo' avere senso prendere un codice seriale e modificarlo per sfruttare la GPU. In questo caso, la prima cosa da fare e' **identificare** quali parti del codice sarebbero avvantaggiate da un **massiccio parallelismo** (**non** ha senso usare la GPU per 10 thread, perche' anche solo l'overhead di copiare i dati dall'host al device renderebbe il calcolo piu' lento!).

Struttura:

- 1 si dichiarano i **puntatori** a delle variabili da usare sul **device** (dichiarare vs definire?). Nota: al livello della **dichiarazione** i puntatori non sono ancora assegnati ne all'host ne al device. **Ha senso** usare dei nomi che ci ricordino quali puntatori devono essere usati dal device!
- 2 con `cudaMalloc` si allocca la memoria sul **device**. Solo in questo momento i puntatori dichiarati precedentemente vengono **assegnati al device**.
- 3 con `cudaMemcpy` si copia la memoria necessaria per il kernel da **host** a **device**
- 4 con le **triple angle brackets** viene gestito il lancio dei **kernel**.
- 5 con `cudaMemcpy` si copia da **device** a **host** il risultato ottenuto
- 6 si libera la memoria dal **device** con `cudaFree`
- 7 **Ovviamente**, bisogna avere modificato una *normale funzione* in modo che diventi il kernel da fare girare sul **device** (perlomeno serve il decoratore `__global__`)

## Resumo della struttura di un main

Può avere senso prendere un codice seriale e modificarlo per sfruttare la GPU. In questo caso, la prima cosa da fare è **identificare** quali parti del codice sarebbero avvantaggiate da un **massiccio parallelismo** (**non** ha senso usare la GPU per 10 thread, perché anche solo l'overhead di copiare i dati dall'host al device renderebbe il calcolo più lento!).

Struttura:

- 1 si dichiarano i **puntatori** a delle variabili da usare sul **device** (dichiarare vs definire?). Nota: al livello della **dichiarazione** i puntatori non sono ancora assegnati né all'host né al device. **Ha senso** usare dei nomi che ci ricordino quali puntatori devono essere usati dal device!
- 2 con `cudaMalloc` si alloca la memoria sul **device**. Solo in questo momento i puntatori dichiarati precedentemente vengono **assegnati al device**.
- 3 con `cudaMemcpy` si copia la memoria necessaria per il kernel da **host** a **device**
- 4 con le **triple angle brackets** viene gestito il lancio dei **kernel**.
- 5 con `cudaMemcpy` si copia da **device** a **host** il risultato ottenuto
- 6 si libera la memoria dal **device** con `cudaFree`
- 7 Ovviamente, bisogna avere modificato una *normale funzione* in modo che diventi il kernel da fare girare sul **device** (perlomeno serve il decoratore `__global__`)



## Riassunto della struttura di un main

Puo' avere senso prendere un codice seriale e modificarlo per sfruttare la GPU. In questo caso, la prima cosa da fare e' **identificare** quali parti del codice sarebbero avvantaggiate da un **massiccio parallelismo** (**non** ha senso usare la GPU per 10 thread, perche' anche solo l'overhead di copiare i dati dall'host al device renderebbe il calcolo piu' lento!).

Struttura:

- 1 si dichiarano i **puntatori** a delle variabili da usare sul **device** (dichiarare vs definire?). Nota: al livello della **dichiarazione** i puntatori non sono ancora assegnati ne all'host ne al device. **Ha senso** usare dei nomi che ci ricordino quali puntatori devono essere usati dal device!
- 2 con `cudaMalloc` si allocca la memoria sul **device**. Solo in questo momento i puntatori dichiarati precedentemente vengono **assegnati al device**.
- 3 con `cudaMemcpy` si copia la memoria necessaria per il kernel da **host** a **device**
- 4 con le **triple angle brackets** viene gestito il lancio dei **kernel**.
- 5 con `cudaMemcpy` si copia da **device** a **host** il risultato ottenuto
- 6 si libera la memoria dal **device** con `cudaFree`
- 7 **Ovviamente**, bisogna avere modificato una *normale funzione* in modo che diventi il **kernel** da fare girare sul **device** (perlomeno serve il decoratore `__global__`)

## Riassunto della struttura di un main

Puo' avere senso prendere un codice seriale e modificarlo per sfruttare la GPU. In questo caso, la prima cosa da fare e' **identificare** quali parti del codice sarebbero avvantaggiate da un **massiccio parallelismo** (**non** ha senso usare la GPU per 10 thread, perche' anche solo l'overhead di copiare i dati dall'host al device renderebbe il calcolo piu' lento!).

Struttura:

- 1 si dichiarano i **puntatori** a delle variabili da usare sul **device** (dichiarare vs definire?). Nota: al livello della **dichiarazione** i puntatori non sono ancora assegnati ne all'host ne al device. **Ha senso** usare dei nomi che ci ricordino quali puntatori devono essere usati dal device!
- 2 con `cudaMalloc` si alloca la memoria sul **device**. Solo in questo momento i puntatori dichiarati precedentemente vengono **assegnati al device**.
- 3 con `cudaMemcpy` si copia la memoria necessaria per il kernel da **host** a **device**
- 4 con le **triple angle brackets** viene gestito il lancio dei **kernel**.
- 5 con `cudaMemcpy` si copia da **device** a **host** il risultato ottenuto
- 6 si libera la memoria dal **device** con `cudaFree`
- 7 *Ovviamente, bisogna avere modificato una normale funzione in modo che diventi il kernel da fare girare sul device (perlomeno serve il decoratore `__global__`)*

## Riassunto della struttura di un main

Puo' avere senso prendere un codice seriale e modificarlo per sfruttare la GPU. In questo caso, la prima cosa da fare e' **identificare** quali parti del codice sarebbero avvantaggiate da un **massiccio parallelismo** (**non** ha senso usare la GPU per 10 thread, perche' anche solo l'overhead di copiare i dati dall'host al device renderebbe il calcolo piu' lento!).

Struttura:

- 1 si dichiarano i **puntatori** a delle variabili da usare sul **device** (dichiarare vs definire?). Nota: al livello della **dichiarazione** i puntatori non sono ancora assegnati ne all'host ne al device. **Ha senso** usare dei nomi che ci ricordino quali puntatori devono essere usati dal device!
- 2 con `cudaMalloc` si allocca la memoria sul **device**. Solo in questo momento i puntatori dichiarati precedentemente vengono **assegnati al device**.
- 3 con `cudaMemcpy` si copia la memoria necessaria per il kernel da **host** a **device**
- 4 con le **triple angle brackets** viene gestito il lancio dei **kernel**.
- 5 con `cudaMemcpy` si copia da **device** a **host** il risultato ottenuto
- 6 si libera la memoria dal **device** con `cudaFree`
- 7 **Ovviamente**, bisogna avere modificato una *normale funzione* in modo che diventi il **kernel** da fare girare sul **device** (perlomeno serve il decoratore `__global__`)

## Verso il parallelismo: un main di un codice parallelo

```

int main( void ) {
    int a[N], b[N], c[N];           // gli array con cui voglio lavorare
    int *dev_a, *dev_b, *dev_c;    // puntatori che servono per la GPU (device)
    int i;

    cudaMalloc( (void**)&dev_a, N * sizeof(int) ); //alloca GPU
    cudaMalloc( (void**)&dev_b, N * sizeof(int) ); //alloca GPU
    cudaMalloc( (void**)&dev_c, N * sizeof(int) ); //alloca GPU
    for (i=0; i<N; i++) {
        a[i] = -i;                  // mi invento un array sulla CPU
        b[i] = i * i;              // me ne invento un altro
    }

    cudaMemcpy(dev_a, a, N *sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(dev_b, b, N *sizeof(int), cudaMemcpyHostToDevice);

    add <<< N, 1 >>> (dev_a, dev_b, dev_c); // lanciamo il kernel

    cudaMemcpy(c, dev_c, N *sizeof(int), cudaMemcpyDeviceToHost);

    printf(" a + b = c \n");
    printf(" ----- \n");

    for (i=0; i<N; i++){
        printf(" %d + %d = %d\n", a[i],b[i],c[i]);
    }

    cudaFree (dev_a); // libero memoria sul device
    cudaFree (dev_b); //
    cudaFree (dev_c); //
    return 0;
}

```

# Triple angle brackets

Nel codice appena mostrato il kernel viene lanciato tramite:

```
add <<< N, 1 >>> (dev_a, dev_b, dev_c);
```

Tra le **triple parentesi angolari** ci sono 2 numeri (vedremo poi che possono essere degli oggetti un po' piu' complicati)

- il primo ingresso,  $N$ , e' il **numero di blocchi** su cui il **kernel** viene lanciato
  - quanti blocchi si possono lanciare contemporaneamente? dipende dalla **compute capabilities** della scheda utilizzata (per esempio nel testo *Cuda by example* si ha come limite 65 535 blocchi).
  - il secondo ingresso (in questo caso abbiamo messo 1) e' il numero di **thread per blocco** su cui viene lanciato il **kernel** (quindi in questo esempio ci sono  $N$  copie del kernel che girano in parallelo sul **device**, una per ogni blocco)
  - a questo punto tra le parentesi tonde ci sono gli argomenti del kernel (che quindi e' come una normale funzione...)  
*Attenzione* che le variabili passate siano state allocate correttamente sul **device**.

# Triple angle brackets

Nel codice appena mostrato il kernel viene lanciato tramite:

```
add <<< N, 1 >>> (dev_a, dev_b, dev_c);
```

Tra le **triple parentesi angolari** ci sono 2 numeri (vedremo poi che possono essere degli oggetti un po' piu' complicati)

- il primo ingresso,  $N$ , e' il **numero di blocchi** su cui il **kernel** viene lanciato
- quanti blocchi si possono lanciare contemporaneamente? dipende dalla **compute capabilities** della scheda utilizzata (per esempio nel testo *Cuda by example* si ha come limite 65 535 blocchi).
- il secondo ingresso (in questo caso abbiamo messo 1) e' il numero di **thread per blocco** su cui viene lanciato il **kernel** (quindi in questo esempio ci sono  $N$  copie del kernel che girano in parallelo sul **device**, una per ogni blocco)
- a questo punto tra le parentesi tonde ci sono gli argomenti del kernel (che quindi e' come una normale funzione...) **Attenzione** che le variabili passate siano state allocate correttamente sul **device**.

# Triple angle brackets

Nel codice appena mostrato il kernel viene lanciato tramite:

```
add <<< N, 1 >>> (dev_a, dev_b, dev_c);
```

Tra le [triple parentesi angolari](#) ci sono 2 numeri (vedremo poi che possono essere degli oggetti un po' piu' complicati)

- il primo ingresso,  $N$ , e' il **numero di blocchi** su cui il **kernel** viene lanciato
- quanti blocchi si possono lanciare contemporaneamente? dipende dalla **compute capabilities** della scheda utilizzata (per esempio nel testo *Cuda by example* si ha come limite 65 535 blocchi).
- il secondo ingresso (in questo caso abbiamo messo 1) e' il numero di **thread per blocco** su cui viene lanciato il **kernel** (quindi in questo esempio ci sono  $N$  copie del kernel che girano in parallelo sul **device**, una per ogni blocco)
- a questo punto tra le parentesi tonde ci sono gli argomenti del kernel (che quindi e' come una normale funzione...) **Attenzione** che le variabili passate siano state allocate correttamente sul **device**.

# Triple angle brackets

Nel codice appena mostrato il kernel viene lanciato tramite:

```
add <<< N, 1 >>> (dev_a, dev_b, dev_c);
```

Tra le [triple parentesi angolari](#) ci sono 2 numeri (vedremo poi che possono essere degli oggetti un po' piu' complicati)

- il primo ingresso,  $N$ , e' il **numero di blocchi** su cui il **kernel** viene lanciato
- quanti blocchi si possono lanciare contemporaneamente? dipende dalla **compute capabilities** della scheda utilizzata (per esempio nel testo *Cuda by example* si ha come limite 65 535 blocchi).
- il secondo ingresso (in questo caso abbiamo messo 1) e' il numero di **thread per blocco** su cui viene lanciato il **kernel** (quindi in questo esempio ci sono  $N$  copie del kernel che girano in parallelo sul **device**, una per ogni blocco)
- a questo punto tra le parentesi tonde ci sono gli argomenti del kernel (che quindi e' come una normale funzione...) **Attenzione** che le variabili passate siano state allocate correttamente sul **device**.



# Verso il parallelismo: un kernel, da usare col main precedente



Nell'esempio fatto fino ad ora si è vista la somma di due interi, ha senso cercare di spingere un po' più in là il parallelismo.

```
__global__ void add(int *a, int *b, int *c)
{
    int tid = blockIdx.x; // uso magic variable: identifico il blocco
    if (tid < N) { // limite l'operazione agli ingressi dell'array
        c[tid] = a[tid] + b[tid]; // l'elemento "tid-esimo" e' sommato dal
    } // thread "tid-esimo"
```

- si noti il decoratore (*decorator*) `__global__` che dice al compilatore che la funzione in questione è un `kernel` che va compilato sul device
- **Domanda:** quante operazioni esegue il singolo *thread*? In questo caso una.
- *chi* (quale thread) fa girare il kernel identificato dalla variabile `tid`
- la **magic variable** `blockIdx.x` è una variabile definita all'interno di CUDA che identifica i blocchi nella griglia. Si noti il `.x` indica che in qualche modo possiamo pensare i blocchi come assegnati in una griglia con più di una dimensione (la `x` e la `y` e la `z` per esempio...). A differenza di MPI o openMP non bisogna chiamare una funzione, è già una variabile!
- per evitare di andare oltre i limiti di definizione ha senso mettere un `if` (**Attenzione:** il computer non sa quale sia la dimensione dei miei array e potrei avere dei blocchi più grandi degli array... con rischio di scrivere in aree di memoria non allocate).

# Verso il parallelismo: un kernel, da usare col main precedente

Nell'esempio fatto fino ad ora si è vista la somma di due interi, ha senso cercare di spingere un po' più in là il parallelismo.

```
__global__ void add(int *a, int *b, int *c)
{
    int tid = blockIdx.x; // uso magic variable: identifico il blocco
    if (tid < N) { // limite l'operazione agli ingressi dell'array
        c[tid] = a[tid] + b[tid]; // l'elemento "tid-esimo" è sommato dal
    } // thread "tid-esimo"
}
```

- si noti il decoratore (*decorator*) `__global__` che dice al compilatore che la funzione in questione è un **kernel** che va compilato sul device
- **Domanda:** quante operazioni esegue il singolo *thread*? In questo caso una.
- *chi* (quale thread) fa girare il kernel identificato dalla variabile `tid`
- la **magic variable** `blockIdx.x` è una variabile definita all'interno di CUDA che identifica i blocchi nella griglia. Si noti il `.x` indica che in qualche modo possiamo pensare i blocchi come assegnati in una griglia con più di una dimensione (la *x* e la *y* e la *z* per esempio...). A differenza di MPI o openMP non bisogna chiamare una funzione, è già una variabile!
- per evitare di andare oltre i limiti di definizione ha senso mettere un `if` (**Attenzione:** il computer non sa quale sia la dimensione dei miei array e potrei avere dei blocchi più grandi degli array... con rischio di scrivere in aree di memoria non allocate).

# Verso il parallelismo: un kernel, da usare col main precedente

Nell'esempio fatto fino ad ora si è vista la somma di due interi, ha senso cercare di spingere un po' più in là il parallelismo.

```
__global__ void add(int *a, int *b, int *c)
{
    int tid = blockIdx.x; // uso magic variable: identifico il blocco
    if (tid < N) { // limite l'operazione agli ingressi dell'array
        c[tid] = a[tid] + b[tid]; // l'elemento "tid-esimo" e' sommato dal
    } // thread "tid-esimo"
}
```

- si noti il decoratore (*decorator*) `__global__` che dice al compilatore che la funzione in questione è un **kernel** che va compilato sul device
- **Domanda:** quante operazioni esegue il singolo *thread*? In questo caso una.
- *chi* (quale thread) fa girare il kernel identificato dalla variabile `tid`
- la *magic variable* `blockIdx.x` è una variabile definita all'interno di CUDA che identifica i blocchi nella griglia. Si noti il `.x` indica che in qualche modo possiamo pensare i blocchi come assegnati in una griglia con più di una dimensione (la `x` e la `z` per esempio...). A differenza di MPI o openMP non bisogna chiamare una funzione, è già una variabile!
- per evitare di andare oltre i limiti di definizione ha senso mettere un `if` (**Attenzione:** il computer non sa quale sia la dimensione dei miei array e potrei avere dei blocchi più grandi degli array... con rischio di scrivere in aree di memoria non allocate).

# Verso il parallelismo: un kernel, da usare col main precedente

Nell'esempio fatto fino ad ora si è vista la somma di due interi, ha senso cercare di spingere un po' più in là il parallelismo.

```
__global__ void add(int *a, int *b, int *c)
{
    int tid = blockIdx.x; // uso magic variable: identifico il blocco
    if (tid < N) { // limite l'operazione agli ingressi dell'array
        c[tid] = a[tid] + b[tid]; // l'elemento "tid-esimo" e' sommato dal
    } // thread "tid-esimo"
}
```

- si noti il decoratore (*decorator*) `__global__` che dice al compilatore che la funzione in questione è un **kernel** che va compilato sul device
- **Domanda:** quante operazioni esegue il singolo *thread*? In questo caso una.
  - *chi* (quale thread) fa girare il kernel identificato dalla variabile `tid`
  - la **magic variable** `blockIdx.x` è una variabile definita all'interno di CUDA che identifica i blocchi nella griglia. Si noti il `.x` indica che in qualche modo possiamo pensare i blocchi come assegnati in una griglia con più di una dimensione (la `y` e la `z` per esempio...). A differenza di MPI o openMP non bisogna chiamare una funzione, è già una variabile!
  - per evitare di andare oltre i limiti di definizione ha senso mettere un `if` (**Attenzione:** il computer non sa quale sia la dimensione dei miei array e potrei avere dei blocchi più grandi degli array... con rischio di scrivere in aree di memoria non allocate).

# Verso il parallelismo: un kernel, da usare col main precedente

Nell'esempio fatto fino ad ora si è vista la somma di due interi, ha senso cercare di spingere un po' più in là il parallelismo.

```
__global__ void add(int *a, int *b, int *c)
{
    int tid = blockIdx.x; // uso magic variable: identifico il blocco
    if (tid < N) { // limite l'operazione agli ingressi dell'array
        c[tid] = a[tid] + b[tid]; // l'elemento "tid-esimo" e' sommato dal
    } // thread "tid-esimo"
}
```

- si noti il decoratore (*decorator*) `__global__` che dice al compilatore che la funzione in questione è un **kernel** che va compilato sul device
- **Domanda:** quante operazioni esegue il singolo *thread*? In questo caso una.
- *chi* (quale thread) fa girare il kernel identificato dalla variabile `tid`
- la **magic variable** `blockIdx.x` è una variabile definita all'interno di CUDA che identifica i blocchi nella griglia. Si noti il `.x` indica che in qualche modo possiamo pensare i blocchi come assegnati in una griglia con più di una dimensione (la *x* e la *y* e la *z* per esempio...). A differenza di MPI o openMP non bisogna chiamare una funzione, è già una variabile!
- per evitare di andare oltre i limiti di definizione ha senso mettere un `if` (**Attenzione:** il computer non sa quale sia la dimensione dei miei array e potrei avere dei blocchi più grandi degli array... con rischio di scrivere in aree di memoria non allocate).

# Verso il parallelismo: un kernel, da usare col main precedente

Nell'esempio fatto fino ad ora si è vista la somma di due interi, ha senso cercare di spingere un po' più in là il parallelismo.

```
__global__ void add(int *a, int *b, int *c)
{
    int tid = blockIdx.x; // uso magic variable: identifico il blocco
    if (tid < N) { // limite l'operazione agli ingressi dell'array
        c[tid] = a[tid] + b[tid]; // l'elemento "tid-esimo" è sommato dal
    } // thread "tid-esimo"
}
```

- si noti il decoratore (*decorator*) `__global__` che dice al compilatore che la funzione in questione è un **kernel** che va compilato sul device
- **Domanda:** quante operazioni esegue il singolo *thread*? In questo caso una.
- *chi* (quale thread) fa girare il kernel identificato dalla variabile `tid`
- la **magic variable** `blockIdx.x` è una variabile definita all'interno di CUDA che identifica i blocchi nella griglia. Si noti il `.x` indica che in qualche modo possiamo pensare i blocchi come assegnati in una griglia con più di una dimensione (la *y* e la *z* per esempio...). A differenza di MPI o openMP non bisogna chiamare una funzione, è già una variabile!
- per evitare di andare oltre i limiti di definizione ha senso mettere un `if` (**Attenzione:** il computer non sa quale sia la dimensione dei miei array e potrei avere dei blocchi più grandi degli array... con rischio di scrivere in aree di memoria non allocate).

# Verso il parallelismo: un kernel, da usare col main precedente

Nell'esempio fatto fino ad ora si è vista la somma di due interi, ha senso cercare di spingere un po' più in là il parallelismo.

```
__global__ void add(int *a, int *b, int *c)
{
    int tid = blockIdx.x; // uso magic variable: identifico il blocco
    if (tid < N) { // limite l'operazione agli ingressi dell'array
        c[tid] = a[tid] + b[tid]; // l'elemento "tid-esimo" e' sommato dal
    } // thread "tid-esimo"
}
```

- si noti il decoratore (*decorator*) `__global__` che dice al compilatore che la funzione in questione è un **kernel** che va compilato sul device
- **Domanda:** quante operazioni esegue il singolo *thread*? In questo caso una.
- *chi* (quale thread) fa girare il kernel identificato dalla variabile `tid`
- la **magic variable** `blockIdx.x` è una variabile definita all'interno di CUDA che identifica i blocchi nella griglia. Si noti il `.x` indica che in qualche modo possiamo pensare i blocchi come assegnati in una griglia con più di una dimensione (la *y* e la *z* per esempio...). A differenza di MPI o openMP non bisogna chiamare una funzione, è già una variabile!
- per evitare di andare oltre i limiti di definizione ha senso mettere un `if` (**Attenzione:** il computer non sa quale sia la dimensione dei miei array e potrei avere dei blocchi più grandi degli array... con rischio di scrivere in aree di memoria non allocate).

# Outline

1 *Struttura Fisica*

2 *Kernel*

3 *Struttura Logica*

4 *Memorie*

5 *Prodotto Matrici*

6 *Stream*

7 *Esempi e strumenti*



# Blocchi e thread

Abbiamo visto come lanciare un kernel su molti **blocchi**. I blocchi pero' non sono il livello piu' fine di granularita' su CUDA, essi infatti possono essere suddivisi in **thread**.

```
add <<< N, m >>> (dev_a, dev_b, dev_c);
```

qui sopra i kernel vengono lanciati tra  $N$  blocchi e  $m$  thread **per** blocco.

Quindi in totale  $N \times m$  thread eseguiranno il kernel!

## Esercizio

Prendere il codice che fa la somma di vettori e invece di usare molti blocchi, ognuno con un singolo thread, usare un **singolo** blocco con molti thread ( la **magic variabile** che identifica i thread all'interno di un blocco e' `threadIdx.x`.)

```
__global__ void add( int *a, int *b, int *c ) {
int tid = threadIdx.x;
if (tid < N)
c[tid] = a[tid] + b[tid];
}
```

**Domanda:** anche il main va cambiato?

**Come?** Si devono modificare gli argomenti all'interno delle triple angle brackets!

```
add <<<1,N >>> (dev_a,dev_b,dev_c)
```

# Blocchi e thread

Abbiamo visto come lanciare un kernel su molti **blocchi**. I blocchi pero' non sono il livello piu' fine di granularita' su CUDA, essi infatti possono essere suddivisi in **thread**.

```
add <<< N, m >>> (dev_a, dev_b, dev_c);
```

qui sopra i kernel vengono lanciati tra  $N$  blocchi e  $m$  thread **per** blocco.

Quindi in totale  $N \times m$  thread eseguiranno il kernel!

## Esercizio

Prendere il codice che fa la somma di vettori e invece di usare molti blocchi, ognuno con un singolo thread, usare un **singolo** blocco con molti thread ( la **magic variabile** che identifica i thread all'interno di un blocco e' `threadIdx.x`)

```
__global__ void add( int *a, int *b, int *c ) {
int tid = threadIdx.x;
if (tid < N)
c[tid] = a[tid] + b[tid];
}
```

**Domanda:** anche il main va cambiato?

**Come?** Si devono modificare gli argomenti all'interno delle triple angle brackets!

```
add <<<1, N >>> (dev_a, dev_b, dev_c)
```

# Blocchi e thread

Abbiamo visto come lanciare un kernel su molti **blocchi**. I blocchi pero' non sono il livello piu' fine di granularita' su CUDA, essi infatti possono essere suddivisi in **thread**.

```
add <<< N, m >>> (dev_a, dev_b, dev_c);
```

qui sopra i kernel vengono lanciati tra  $N$  blocchi e  $m$  thread **per** blocco.

Quindi in totale  $N \times m$  thread eseguiranno il kernel!

## Esercizio

Prendere il codice che fa la somma di vettori e invece di usare molti blocchi, ognuno con un singolo thread, usare un **singolo** blocco con molti thread ( la **magic variabile** che identifica i thread all'interno di un blocco e' `threadIdx.x`.)

```
__global__ void add( int *a, int *b, int *c ) {
int tid = threadIdx.x;
if (tid < N)
c[tid] = a[tid] + b[tid];
}
```

**Domanda:** anche il main va cambiato?

**Come?** Si devono modificare gli argomenti all'interno delle triple angle brackets!

```
add <<<1, N >>> (dev_a, dev_b, dev_c)
```

# Blocchi e thread

Abbiamo visto come lanciare un kernel su molti **blocchi**. I blocchi pero' non sono il livello piu' fine di granularita' su CUDA, essi infatti possono essere suddivisi in **thread**.

```
add <<< N, m >>> (dev_a, dev_b, dev_c);
```

qui sopra i kernel vengono lanciati tra  $N$  blocchi e  $m$  thread **per** blocco.

Quindi in totale  $N \times m$  thread eseguiranno il kernel!

## Esercizio

Prendere il codice che fa la somma di vettori e invece di usare molti blocchi, ognuno con un singolo thread, usare un **singolo** blocco con molti thread ( la **magic variabile** che identifica i thread all'interno di un blocco e' `threadIdx.x`)

```
__global__ void add( int *a, int *b, int *c ) {
int tid = threadIdx.x;
if (tid < N)
c[tid] = a[tid] + b[tid];
}
```

**Domanda:** anche il main va cambiato?

**Come?** Si devono modificare gli argomenti all'interno delle triple angle brackets!

```
add <<<1,N >>> (dev_a,dev_b,dev_c)
```

# Blocchi e thread

Abbiamo visto come lanciare un kernel su molti **blocchi**. I blocchi pero' non sono il livello piu' fine di granularita' su CUDA, essi infatti possono essere suddivisi in **thread**.

```
add <<< N, m >>> (dev_a, dev_b, dev_c);
```

qui sopra i kernel vengono lanciati tra  $N$  blocchi e  $m$  thread **per** blocco.

Quindi in totale  $N \times m$  thread eseguiranno il kernel!

## Esercizio

Prendere il codice che fa la somma di vettori e invece di usare molti blocchi, ognuno con un singolo thread, usare un **singolo** blocco con molti thread ( la **magic variabile** che identifica i thread all'interno di un blocco e' `threadIdx.x`)

```
__global__ void add( int *a, int *b, int *c ) {
int tid = threadIdx.x;
if (tid < N)
c[tid] = a[tid] + b[tid];
}
```

**Domanda:** anche il main va cambiato?

**Come?** Si devono modificare gli argomenti all'interno delle triple angle brackets!

```
add <<<1,N >>> (dev_a,dev_b,dev_c)
```

# Blocchi e thread

Abbiamo visto come lanciare un kernel su molti **blocchi**. I blocchi pero' non sono il livello piu' fine di granularita' su CUDA, essi infatti possono essere suddivisi in **thread**.

```
add <<< N, m >>> (dev_a, dev_b, dev_c);
```

qui sopra i kernel vengono lanciati tra  $N$  blocchi e  $m$  thread **per** blocco.

Quindi in totale  $N \times m$  thread eseguiranno il kernel!

## Esercizio

Prendere il codice che fa la somma di vettori e invece di usare molti blocchi, ognuno con un singolo thread, usare un **singolo** blocco con molti thread ( la **magic variabile** che identifica i thread all'interno di un blocco e' `threadIdx.x`)

```
__global__ void add( int *a, int *b, int *c ) {
int tid = threadIdx.x;
if (tid < N)
c[tid] = a[tid] + b[tid];
}
```

**Domanda:** anche il main va cambiato?

**Come?** Si devono modificare gli argomenti all'interno delle triple angle brackets!

```
add <<<1, N >>> (dev_a, dev_b, dev_c)
```

## Esempio di una griglia 1D, con blocchi 1D

Supponiamo di:

- dover lavorare su un **array** di 16 punti, 1D.
- supponiamo di avere deciso che anche la griglia sia 1D (per esempio stiamo simulando una linea divisa in pezzi).
- vogliamo una funzione che associ ogni **thread** ad un singolo **punto** dell'array
- per esempio, suddividiamo la griglia in 4 blocchi 1D, ognuno di 4 thread.

0				1				2				3			
0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	16

- `blockIdx.x = 2`
- `threadIdx.x = 1`
- `blockDim.x = 4`
- `indice = 2 × 4 + 1 = 9`

## Esempio di una griglia 1D, con blocchi 1D

Supponiamo di:

- dover lavorare su un **array** di 16 punti, 1D.
- supponiamo di avere deciso che anche la **griglia** sia 1D (per esempio stiamo simulando una linea divisa in pezzi).
- vogliamo una funzione che associ ogni **thread** ad un singolo **punto** dell'array
- per esempio, suddividiamo la griglia in 4 blocchi 1D, ognuno di 4 thread.

0				1				2				3			
0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	16

- `blockIdx.x = 2`
- `threadIdx.x = 1`
- `blockDim.x = 4`
- `indice = 2 × 4 + 1 = 9`



## Esempio di una griglia 1D, con blocchi 1D

Supponiamo di:

- dover lavorare su un **array** di 16 punti, 1D.
- supponiamo di avere deciso che anche la **griglia** sia 1D (per esempio stiamo simulando una linea divisa in pezzi).
- vogliamo una funzione che associ ogni **thread** ad un singolo **punto** dell'array
- per esempio, suddividiamo la griglia in 4 blocchi 1D, ognuno di 4 thread.

0				1				2				3			
0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	16

- `blockIdx.x = 2`
- `threadIdx.x = 1`
- `blockDim.x = 4`
- `indice = 2 × 4 + 1 = 9`

## Esempio di una griglia 1D, con blocchi 1D

Supponiamo di:

- dover lavorare su un **array** di 16 punti, 1D.
- supponiamo di avere deciso che anche la **griglia** sia 1D (per esempio stiamo simulando una linea divisa in pezzi).
- vogliamo una funzione che associ ogni **thread** ad un singolo **punto** dell'array
- per esempio, suddividiamo la griglia in 4 blocchi 1D, ognuno di 4 thread.

0				1				2				3			
0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	16

- `blockIdx.x = 2`
- `threadIdx.x = 1`
- `blockDim.x = 4`
- `indice = 2 × 4 + 1 = 9`

## Esempio di una griglia 1D, con blocchi 1D

Supponiamo di:

- dover lavorare su un **array** di 16 punti, 1D.
- supponiamo di avere deciso che anche la **griglia** sia 1D (per esempio stiamo simulando una linea divisa in pezzi).
- vogliamo una funzione che associ ogni **thread** ad un singolo **punto** dell'array
- per esempio, suddividiamo la griglia in 4 blocchi 1D, ognuno di 4 thread.

0				1				2				3			
0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	16

- `blockIdx.x = 2`
- `threadIdx.x = 1`
- `blockDim.x = 4`
- `indice = 2 × 4 + 1 = 9`

## Esempio di una griglia 1D, con blocchi 1D

Supponiamo di:

- dover lavorare su un **array** di 16 punti, 1D.
- supponiamo di avere deciso che anche la **griglia** sia 1D (per esempio stiamo simulando una linea divisa in pezzi).
- vogliamo una funzione che associ ogni **thread** ad un singolo **punto** dell'array
- per esempio, suddividiamo la griglia in 4 blocchi 1D, ognuno di 4 thread.

0				1				2				3			
0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	16

- `blockIdx.x = 2`
- `threadIdx.x = 1`
- `blockDim.x = 4`
- `indice = 2 × 4 + 1 = 9`

## Esempio di una griglia 1D, con blocchi 1D

Supponiamo di:

- dover lavorare su un **array** di 16 punti, 1D.
- supponiamo di avere deciso che anche la **griglia** sia 1D (per esempio stiamo simulando una linea divisa in pezzi).
- vogliamo una funzione che associ ogni **thread** ad un singolo **punto** dell'array
- per esempio, suddividiamo la griglia in 4 blocchi 1D, ognuno di 4 thread.

0				1				2				3			
0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	16

- `blockIdx.x = 2`
- `threadIdx.x = 1`
- `blockDim.x = 4`
- `indice = 2 × 4 + 1 = 9`

## Identificare un thread tra blocchi e griglie:

le variabili **magiche** per i **thread** e i **blocchi**:

- `threadIdx.x` indice dei **thread** all'interno di un **blocco**, i valori vanno  $[0, \text{blockDim.x}-1]$  (ci sono anche `threadId.y` e `threadId.z`)
- `blockIdx` indice del **blocco** all'interno della **grid**  $[0, \text{gridDim.x}-1]$  (ci sono anche `blockIdx.x`, `blockIdx.y` e `blockIdx.z`)
- `blockDim.x` e' la grandezza dei blocchi lungo la dimensione x (ovviamente c'e' la magic variable anche per la y e la z).
- per esempio, se vogliamo sapere l'indice globale di un thread in un sistema dove ci sono molti blocchi (1D) (nota che `idx` non e' una magic variable, ma e' un nome scelto da me):  

$$\text{idx} = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x}$$

Block 0	Thread 0	Thread 1	Thread 2	Thread 3
Block 1	Thread 0	Thread 1	Thread 2	Thread 3
Block 2	Thread 0	Thread 1	Thread 2	Thread 3
Block 3	Thread 0	Thread 1	Thread 2	Thread 3

*Figure:* struttura dei thread e blocchi.

## Lanciare un kernel con una griglia strutturata

Abbiamo visto che i **thread** in un **blocco** possono formare una griglia 3D, lo stesso i **blocchi** nella **grid**.... come si fa ad usare le **triple angle brackets** per sfruttare questa granularità dei thread?

- per prima cosa si usa un nuovo *tipo* (un vettore 3D intero), definito in CUDA, per esempio:  

```
dim3 miaGrid (10,10,2);
```

 In questo esempio abbiamo creato un oggetto che contiene 3 interi e che definisce la struttura della griglia in funzione dei blocchi (da metter nel primo ingresso delle *triple angle brackets*).
- a questo punto si definisce la struttura del singolo blocco (con il medesimo *tipo*):  

```
dim3 mioBlocco (32,4);
```

 In questo caso abbiamo deciso che un blocco ha una struttura 2D in cui il primo ingresso ha lunghezza 32, il secondo 4. La variabile `mioBlocco` andrà quindi inserita nel secondo ingresso delle *triple angle brackets*. Questo tipo di blocchi avrà 128 thread.

**Osservazione:** Quando non si specifica una delle dimensioni di un oggetto `dim3`, questa ha lunghezza 1.

**Attenzione** dato che decido io come sono fatti questi oggetti, nel momento in cui li inserisco nelle triple angle brackets definisco la **struttura** dei thread e dei blocchi. Nell'esempio la grid è 3D e ci sono 200 blocchi (10 x 10 x 2), mentre i blocchi sono 2D (32 x 4).

```
dim3 miaGrid (10,10,2); // grid di 10 x 10 x 2 = 200 blocchi nella griglia
dim3 mioBlocco (32,4); // blocco di 32 x 4 x 1 = 128 thread nel singolo blocco
```

**Esempio di chiamata:** supponiamo di avere definito un kernel chiamato `add`, questo può essere lanciato in questo modo:

```
add <<< miaGrid, mioBlocco >>> (argomenti);
```

## Lanciare un kernel con una griglia strutturata

Abbiamo visto che i **thread** in un **blocco** possono formare una griglia 3D, lo stesso i **blocchi** nella **grid**.... come si fa ad usare le *triple angle brackets* per sfruttare questa granularità dei thread?

- per prima cosa si usa un nuovo *tipo* (un vettore 3D intero), definito in CUDA, per esempio:

```
dim3 miaGrid (10,10,2);
```

In questo esempio abbiamo creato un oggetto che contiene 3 interi e che definisce la struttura della griglia in funzione dei blocchi (da metter nel primo ingresso delle *triple angle brackets*).

- a questo punto si definisce la struttura del singolo blocco (con il medesimo *tipo*):

```
dim3 mioBlocco (32,4);
```

In questo caso abbiamo deciso che un blocco ha una struttura 2D in cui il primo ingresso ha lunghezza 32, il secondo 4. La variabile `mioBlocco` andrà quindi inserita nel secondo ingresso delle *triple angle brackets*. Questo tipo di blocchi avrà 128 thread.

**Osservazione:** Quando non si specifica una delle dimensioni di un oggetto `dim3`, questa ha lunghezza 1.

**Attenzione** dato che decido io come sono fatti questi oggetti, nel momento in cui li inserisco nelle *triple angle brackets* definisco la **struttura** dei thread e dei blocchi. Nell'esempio la **grid** è **3D** e ci sono 200 blocchi (10 x 10 x 2), mentre i blocchi sono **2D** (32 x 4).

```
dim3 miaGrid (10,10,2); // grid di 10 x 10 x 2 = 200 blocchi nella griglia
dim3 mioBlocco (32,4); // blocco di 32 x 4 x 1 = 128 thread nel singolo blocco
```

**Esempio di chiamata:** supponiamo di avere definito un kernel chiamato `add`, questo può essere lanciato in questo modo:

```
add <<< miaGrid, mioBlocco >>> (argomenti);
```



## Lanciare un kernel con una griglia strutturata

Abbiamo visto che i **thread** in un **blocco** possono formare una griglia 3D, lo stesso i **blocchi** nella **grid**.... come si fa ad usare le **triple angle brackets** per sfruttare questa granularità dei thread?

- per prima cosa si usa un nuovo **tipo** (un vettore 3D intero), definito in CUDA, per esempio:  
`dim3 miaGrid (10,10,2);`  
 In questo esempio abbiamo creato un oggetto che contiene 3 interi e che definisce la struttura della griglia in funzione dei blocchi (da metter nel primo ingresso delle **triple angle brackets**).
- a questo punto si definisce la struttura del singolo blocco (con il medesimo **tipo**):  
`dim3 mioBlocco (32,4);`  
 In questo caso abbiamo deciso che un blocco ha una struttura 2D in cui il primo ingresso ha lunghezza 32, il secondo 4. La variabile `mioBlocco` andrà quindi inserita nel secondo ingresso delle **triple angle brackets**. Questo tipo di blocchi avrà 128 thread.

**Osservazione:** Quando non si specifica una delle dimensioni di un oggetto `dim3`, questa ha lunghezza 1.

**Attenzione** dato che decido io come sono fatti questi oggetti, nel momento in cui li inserisco nelle triple angle brackets definisco la **struttura** dei thread e dei blocchi. Nell'esempio la grid è **3D** e ci sono 200 blocchi (10 x 10 x 2), mentre i blocchi sono **2D** (32 x 4).

```
dim3 miaGrid (10,10,2); // grid di 10 x 10 x 2 = 200 blocchi nella griglia
dim3 mioBlocco (32,4); // blocco di 32 x 4 x 1 = 128 thread nel singolo blocco
```

**Esempio di chiamata:** supponiamo di avere definito un kernel chiamato `add`, questo può essere lanciato in questo modo:

```
add <<< miaGrid, mioBlocco >>> (argomenti);
```

## Lanciare un kernel con una griglia strutturata

Abbiamo visto che i **thread** in un **blocco** possono formare una griglia 3D, lo stesso i **blocchi** nella **grid**.... come si fa ad usare le **triple angle brackets** per sfruttare questa granularità dei thread?

- per prima cosa si usa un nuovo *tipo* (un vettore 3D intero), definito in CUDA, per esempio:  
`dim3 miaGrid (10,10,2);`  
 In questo esempio abbiamo creato un oggetto che contiene 3 interi e che definisce la struttura della griglia in funzione dei blocchi (da metter nel primo ingresso delle *triple angle brackets*).
- a questo punto si definisce la struttura del singolo blocco (con il medesimo *tipo*):  
`dim3 mioBlocco (32,4);`  
 In questo caso abbiamo deciso che un blocco ha una struttura 2D in cui il primo ingresso ha lunghezza 32, il secondo 4. La variabile `mioBlocco` andrà quindi inserita nel secondo ingresso delle *triple angle brackets*. Questo tipo di blocchi avrà 128 thread.

**Osservazione:** Quando non si specifica una delle dimensioni di un oggetto `dim3`, questa ha lunghezza 1.

**Attenzione** dato che decido io come sono fatti questi oggetti, nel momento in cui li inserisco nelle triple angle brackets definisco la **struttura** dei thread e dei blocchi. Nell'esempio la grid è **3D** e ci sono 200 blocchi (10 x 10 x 2), mentre i blocchi sono **2D** (32 x 4).

```
dim3 miaGrid (10,10,2); // grid di 10 x 10 x 2 = 200 blocchi nella griglia
dim3 mioBlocco (32,4); // blocco di 32 x 4 x 1 = 128 thread nel singolo blocco
```

**Esempio di chiamata:** supponiamo di avere definito un kernel chiamato `add`, questo può essere lanciato in questo modo:

```
add <<< miaGrid, mioBlocco >>> (argomenti);
```

## Lanciare un kernel con una griglia strutturata

Abbiamo visto che i **thread** in un **blocco** possono formare una griglia 3D, lo stesso i **blocchi** nella **grid**.... come si fa ad usare le *triple angle brackets* per sfruttare questa granularità dei thread?

- per prima cosa si usa un nuovo *tipo* (un vettore 3D intero), definito in CUDA, per esempio:  
`dim3 miaGrid (10,10,2);`  
 In questo esempio abbiamo creato un oggetto che contiene 3 interi e che definisce la struttura della griglia in funzione dei blocchi (da metter nel primo ingresso delle *triple angle brackets*).
- a questo punto si definisce la struttura del singolo blocco (con il medesimo *tipo*):  
`dim3 mioBlocco (32,4);`  
 In questo caso abbiamo deciso che un blocco ha una struttura 2D in cui il primo ingresso ha lunghezza 32, il secondo 4. La variabile `mioBlocco` andrà quindi inserita nel secondo ingresso delle *triple angle brackets*. Questo tipo di blocchi avrà 128 thread.

**Osservazione:** Quando non si specifica una delle dimensioni di un oggetto `dim3`, questa ha lunghezza 1.

**Attenzione** dato che decido io come sono fatti questi oggetti, nel momento in cui li inserisco nelle triple angle brackets definisco la **struttura** dei thread e dei blocchi. Nell'esempio la grid è **3D** e ci sono 200 blocchi (10 x 10 x 2), mentre i blocchi sono **2D** (32 x 4).

```
dim3 miaGrid (10,10,2); // grid di 10 x 10 x 2 = 200 blocchi nella griglia
dim3 mioBlocco (32,4); // blocco di 32 x 4 x 1 = 128 thread nel singolo blocco
```

**Esempio di chiamata:** supponiamo di avere definito un kernel chiamato `add`, questo può essere lanciato in questo modo:

```
add <<< miaGrid, mioBlocco >>> (argomenti);
```

## Lanciare un kernel con una griglia strutturata

Abbiamo visto che i **thread** in un **blocco** possono formare una griglia 3D, lo stesso i **blocchi** nella **grid**.... come si fa ad usare le *triple angle brackets* per sfruttare questa granularità dei thread?

- per prima cosa si usa un nuovo *tipo* (un vettore 3D intero), definito in CUDA, per esempio:  

```
dim3 miaGrid (10,10,2);
```

 In questo esempio abbiamo creato un oggetto che contiene 3 interi e che definisce la struttura della griglia in funzione dei blocchi (da metter nel primo ingresso delle *triple angle brackets*).
- a questo punto si definisce la struttura del singolo blocco (con il medesimo *tipo*):  

```
dim3 mioBlocco (32,4);
```

 In questo caso abbiamo deciso che un blocco ha una struttura 2D in cui il primo ingresso ha lunghezza 32, il secondo 4. La variabile `mioBlocco` andrà quindi inserita nel secondo ingresso delle *triple angle brackets*. Questo tipo di blocchi avrà 128 thread.

**Osservazione:** Quando non si specifica una delle dimensioni di un oggetto `dim3`, questa ha lunghezza 1.

**Attenzione** dato che decido io come sono fatti questi oggetti, nel momento in cui li inserisco nelle triple angle brackets definisco la **struttura** dei thread e dei blocchi. Nell'esempio la grid è **3D** e ci sono 200 blocchi (10 x 10 x 2), mentre i blocchi sono **2D** (32 x 4).

```
dim3 miaGrid (10,10,2); // grid di 10 x 10 x 2 = 200 blocchi nella griglia
dim3 mioBlocco (32,4); // blocco di 32 x 4 x 1 = 128 thread nel singolo blocco
```

**Esempio di chiamata:** supponiamo di avere definito un kernel chiamato `add`, questo può essere lanciato in questo modo:

```
add <<< miaGrid, mioBlocco>>> (argomenti);
```

# Compute capabilities da Wikipedia

Technical specifications	Compute capability (version)														
	1.0	1.1	1.2	1.3	2.x	3.0	3.2	3.5	3.7	5.0	5.2	5.3	6.0	6.1	6.2
Maximum number of resident grids per device (Concurrent Kernel Execution)	t.b.d.				16		4		32			16	128	32	16
Maximum dimensionality of grid of thread blocks	2				3										
Maximum x-dimension of a grid of thread blocks	65535					2 <sup>31</sup> - 1									
Maximum y-, or z-dimension of a grid of thread blocks	65535														
Maximum dimensionality of thread block	3														
Maximum x- or y-dimension of a block	512				1024										
Maximum z-dimension of a block	64														
Maximum number of threads per block	512				1024										
Warp size	32														
Maximum number of resident blocks per multiprocessor	8					16					32				
Maximum number of resident warps per multiprocessor	24	32		48		64									

## Da una funzione ad un kernel

Un kernel non e' altro che una funzione che gira sul **device**. Vediamo qual'e' l'evoluzione che possiamo pensare per una funzione che fa la somma degli elementi di due array (componente per componente).

1) Prima vediamo una normale funzione che gira sull'**host**, che si occupa di tutti gli ingressi dell'array:

```
void add( int *a, int *b, int *c ) {
    int id = 0;           // qui id e' una variabile sugli elementi dell'array
    while (id < N) {     // loop che gira su tutti gli elementi dell'array
        c[id] = a[id] + b[id];
        id += 1;        // prossimo giro, prossimo elemento dell'array!
    }
}
```

2) Trasformiamo la funzione sopra in un **kernel** che giri su un **device**, (pensato per molti blocchi, ciascuno con un unico thread):

```
__global__ void add( int *a, int *b, int *c ) {
    int id = blockIdx.x; // qui id identifica un blocco
    //----- tolto il while, ogni blocco processa 1 solo ingresso
    if(id<N)             c[id] = a[id] + b[id]; // un elemento differente
    //----- tolto l'aggiornamento al prossimo elemento
}
```

Supponiamo ora che il numero di elementi nell'array sia piu' grande di tutti i possibili thread che possiamo lanciare ( $\text{numMaxBlocchi} \times \text{numMaxThreadPerBlocco}$ ), cosa facciamo?

- **Semplice**, ogni thread dovra' processare **piu' di un elemento** dell'array!
- **Attenzione**: thread diversi devono essere sicuri di processare elementi diversi!
- **Necessita'**: tutti gli ingressi dell'array devono essere riempiti

# Da una funzione ad un kernel

Un kernel non e' altro che una funzione che gira sul **device**. Vediamo qual'e' l'evoluzione che possiamo pensare per una funzione che fa la somma degli elementi di due array (componente per componente).

1) Prima vediamo una normale funzione che gira sull'**host**, che si occupa di tutti gli ingressi dell'array:

```
void add( int *a, int *b, int *c ) {
    int id = 0;           // qui id e' una variabile sugli elementi dell'array
    while (id < N) {     // loop che gira su tutti gli elementi dell'array
        c[id] = a[id] + b[id];
        id += 1;        // prossimo giro, prossimo elemento dell'array!
    }
}
```

2) Trasformiamo la funzione sopra in un **kernel** che giri su un **device**, (pensato per molti blocchi, ciascuno con un unico thread):

```
__global__ void add( int *a, int *b, int *c ) {
    int id = blockIdx.x; // qui id identifica un blocco
    //===== tolto il while, ogni blocco processa 1 solo ingresso
    if(id<N)             c[id] = a[id] + b[id]; // un elemento differente
    //===== tolto l'aggiornamento al prossimo elemento
}
```

Supponiamo ora che il numero di elementi nell'array sia piu' grande di tutti i possibili thread che possiamo lanciare ( $\text{numMaxBlocchi} \times \text{numMaxThreadPerBlocco}$ ), cosa facciamo?

- **Semplice**, ogni thread dovra' processare **piu' di un elemento** dell'array!
- **Attenzione**: thread diversi devono essere sicuri di processare elementi diversi!
- **Necessita'**: tutti gli ingressi dell'array devono essere riempiti

## Da una funzione ad un kernel

Un kernel non e' altro che una funzione che gira sul **device**. Vediamo qual'e' l'evoluzione che possiamo pensare per una funzione che fa la somma degli elementi di due array (componente per componente).

1) Prima vediamo una normale funzione che gira sull'**host**, che si occupa di tutti gli ingressi dell'array:

```
void add( int *a, int *b, int *c ) {
    int id = 0;           // qui id e' una variabile sugli elementi dell'array
    while (id < N) {     // loop che gira su tutti gli elementi dell'array
        c[id] = a[id] + b[id];
        id += 1;        // prossimo giro, prossimo elemento dell'array!
    }
}
```

2) Trasformiamo la funzione sopra in un **kernel** che giri su un **device**, (pensato per molti blocchi, ciascuno con un unico thread):

```
__global__ void add( int *a, int *b, int *c ) {
    int id = blockIdx.x; // qui id identifica un blocco
    //===== tolto il while, ogni blocco processa 1 solo ingresso
    if(id<N)             c[id] = a[id] + b[id]; // un elemento differente
    //===== tolto l'aggiornamento al prossimo elemento
}
```

Supponiamo ora che il numero di elementi nell'array sia piu' grande di tutti i possibili thread che possiamo lanciare ( $\text{numMaxBlocchi} \times \text{numMaxThreadPerBloco}$ ), cosa facciamo?

- **Semplice**, ogni thread dovra' processare **piu' di un elemento** dell'array!
- **Attenzione**: thread diversi devono essere sicuri di processare elementi diversi!
- **Necessita'**: tutti gli ingressi dell'array devono essere riempiti



# Come scegliere il numero di blocchi: passo 0

In base alle **compute capabilities** ogni GPU ha dei limiti, per esempio:

- il numero di blocchi (per esempio 65535)
- il numero di thread per blocco (per esempio 1024)
- grandezza di **ogni** dimensione di un blocco (per esempio 512 x 512 x 64)

E' importante quando si lanciano i **kernel** che non si vada oltre questi limiti (altrimenti si ha errore).

Problema:

- Supponiamo di voler lanciare  $N$  thread, come li dividiamo tra i blocchi? (1 blocco con  $N$  thread? 2 blocchi con  $N/2$  thread, 4 blocchi con  $N/4$  thread???)
- Oppure potremmo avere **deciso** che ogni blocco deve avere 128 thread: quanti blocchi e' bene usare? **Nota** esistono dei motivi per limitare il numero di thread per blocco legati alla **memoria shared** (per esempio limitando il numero di thread per blocco ognuno ha a disposizione un quantitativo di memoria shared, che e' piuttosto limitata).

Esempio (**problematico**) di un lancio di kernel con blocchi, ciascuno da 128 thread:

```
add<<< (N+127)/128, 128 >>>( dev_a, dev_b, dev_c );
```

# Come scegliere il numero di blocchi: passo 0

In base alle **compute capabilities** ogni GPU ha dei limiti, per esempio:

- il numero di blocchi (per esempio 65535)
- il numero di thread per blocco (per esempio 1024)
- grandezza di **ogni** dimensione di un blocco (per esempio 512 x 512 x 64)

E' importante quando si lanciano i **kernel** che non si vada oltre questi limiti (altrimenti si ha errore).

**Problema:**

- Supponiamo di voler lanciare  $N$  thread, come li dividiamo tra i blocchi? (1 blocco con  $N$  thread? 2 blocchi con  $N/2$  thread, 4 blocchi con  $N/4$  thread???)
- Oppure potremmo avere **deciso** che ogni blocco deve avere 128 thread: quanti blocchi e' bene usare? **Nota** esistono dei motivi per limitare il numero di thread per blocco legati alla **memoria shared** (per esempio limitando il numero di thread per blocco ognuno ha a disposizione un quantitativo di memoria shared, che e' piuttosto limitata).

Esempio (**problematico**) di un lancio di kernel con blocchi, ciascuno da 128 thread:

```
add<<< (N+127)/128, 128 >>>( dev_a, dev_b, dev_c );
```

# Come scegliere il numero di blocchi: passo 0

In base alle **compute capabilities** ogni GPU ha dei limiti, per esempio:

- il numero di blocchi (per esempio 65535)
- il numero di thread per blocco (per esempio 1024)
- grandezza di **ogni** dimensione di un blocco (per esempio 512 x 512 x 64)

E' importante quando si lanciano i **kernel** che non si vada oltre questi limiti (altrimenti si ha errore).

**Problema:**

- Supponiamo di voler lanciare  $N$  thread, come li dividiamo tra i blocchi? (1 blocco con  $N$  thread? 2 blocchi con  $N/2$  thread, 4 blocchi con  $N/4$  thread???)
- Oppure potremmo avere **deciso** che ogni blocco deve avere 128 thread: quanti blocchi e' bene usare? **Nota** esistono dei motivi per limitare il numero di thread per blocco legati alla **memoria shared** (per esempio limitando il numero di thread per blocco ognuno ha a disposizione un quantitativo di memoria shared, che e' piuttosto limitata).

Esempio (**problematico**) di un lancio di kernel con blocchi, ciascuno da 128 thread:

```
add<<< (N+127)/128, 128 >>>( dev_a, dev_b, dev_c );
```

# Dimensionare i blocchi passo 1

Supponiamo che il numero di punti totali da calcolare sia  $N$  (variabile), e che si vogliano blocchi di 128 thread (in cui ogni thread processa un solo punto).

La formula  $N/128 = 0$  e' pericolosa, per esempio con  $N = 127$  si ottengono 0 blocchi! **Dobbiamo essere sicuri di processare tutti i punti!**

Per questa ragione ha senso definire il numero di blocchi in funzione del numero di punti da calcolare ( $n_{tXb}^N$ ) e del numero di thread per blocco ( $tXb$ ), come:

- $N$  = numero di punti
- $tXb$  = thread per blocco
- $n_{tXb}^N$  = numero di blocchi, in funzione del numero di punti e del numero di thread per blocco

$$n_{tXb}^N = \frac{N + (tXb - 1)}{tXb} \quad (1)$$

In questo modo se:

$$N = 127, tXb = 128 \rightarrow n_{128}^{127} = \frac{127 + 127}{128} = 1 \quad (1 \times 128 = 128) \quad (2)$$

se invece:

$$N = 10000, tXb = 128 \rightarrow n_{128}^{10000} = \frac{10000 + 127}{128} = 79 \quad (79 \times 128 = 10112) \quad (3)$$

In questo modo abbiamo dimensionato il numero di blocchi in modo che tutti i punti vengano processati!

**Problema**, esistono dei limiti:

- un numero massimo di blocchi per griglia
- un numero massimo di thread per blocco
- un numero massimo di thread per griglia (kernel)

# Dimensionare i blocchi passo 1

Supponiamo che il numero di punti totali da calcolare sia  $N$  (variabile), e che si vogliano blocchi di 128 thread (in cui ogni thread processa un solo punto).

La formula  $N/128 = 0$  e' pericolosa, per esempio con  $N = 127$  si ottengono 0 blocchi! **Dobbiamo essere sicuri di processare tutti i punti!**

Per questa ragione ha senso definire il numero di blocchi in funzione del numero di punti da calcolare ( $n_{tXb}^N$ ) e del numero di thread per blocco ( $tXb$ ), come:

- $N$  = numero di punti
- $tXb$  = thread per blocco
- $n_{tXb}^N$  = numero di blocchi, in funzione del numero di punti e del numero di thread per blocco

$$n_{tXb}^N = \frac{N + (tXb - 1)}{tXb} \quad (1)$$

In questo modo se:

$$N = 127, tXb = 128 \rightarrow n_{128}^{127} = \frac{127 + 127}{128} = 1 \quad (1 \times 128 = 128) \quad (2)$$

se invece:

$$N = 10000, tXb = 128 \rightarrow n_{128}^{10000} = \frac{10000 + 127}{128} = 79 \quad (79 \times 128 = 10112) \quad (3)$$

In questo modo abbiamo dimensionato il numero di blocchi in modo che tutti i punti vengano processati!

**Problema**, esistono dei limiti:

- un numero massimo di **blocchi** per griglia
- un numero massimo di **thread** per **blocco**
- un numero massimo di **thread** per griglia (kernel)

## Dimensionare i blocchi passo 1

Supponiamo che il numero di punti totali da calcolare sia  $N$  (variabile), e che si vogliano blocchi di 128 thread (in cui ogni thread processa un solo punto).

La formula  $N/128 = 0$  e' pericolosa, per esempio con  $N = 127$  si ottengono 0 blocchi! **Dobbiamo essere sicuri di processare tutti i punti!**

Per questa ragione ha senso definire il numero di blocchi in funzione del numero di punti da calcolare ( $n_{tXb}^N$ ) e del numero di thread per blocco ( $tXb$ ), come:

- $N$  = numero di punti
- $tXb$  = thread per blocco
- $n_{tXb}^N$  = numero di blocchi, in funzione del numero di punti e del numero di thread per blocco

$$n_{tXb}^N = \frac{N + (tXb - 1)}{tXb} \quad (1)$$

In questo modo se:

$$N = 127, tXb = 128 \rightarrow n_{128}^{127} = \frac{127 + 127}{128} = 1 \quad (1 \times 128 = 128) \quad (2)$$

se invece:

$$N = 10000, tXb = 128 \rightarrow n_{128}^{10000} = \frac{10000 + 127}{128} = 79 \quad (79 \times 128 = 10112) \quad (3)$$

In questo modo abbiamo dimensionato il numero di blocchi in modo che tutti i punti vengano processati!

**Problema**, esistono dei limiti:

- un numero massimo di **blocchi** per griglia
- un numero massimo di **thread** per **blocco**
- un numero massimo di **thread** per griglia (kernel)

# Dimensionare i blocchi passo 1

Supponiamo che il numero di punti totali da calcolare sia  $N$  (variabile), e che si vogliano blocchi di 128 thread (in cui ogni thread processa un solo punto).

La formula  $N/128 = 0$  e' pericolosa, per esempio con  $N = 127$  si ottengono 0 blocchi! **Dobbiamo essere sicuri di processare tutti i punti!**

Per questa ragione ha senso definire il numero di blocchi in funzione del numero di punti da calcolare ( $n_{tXb}^N$ ) e del numero di thread per blocco ( $tXb$ ), come:

- $N$  = numero di punti
- $tXb$  = thread per blocco
- $n_{tXb}^N$  = numero di blocchi, in funzione del numero di punti e del numero di thread per blocco

$$n_{tXb}^N = \frac{N + (tXb - 1)}{tXb} \quad (1)$$

In questo modo se:

$$N = 127, tXb = 128 \rightarrow n_{128}^{127} = \frac{127 + 127}{128} = 1 \quad (1 \times 128 = 128) \quad (2)$$

se invece:

$$N = 10000, tXb = 128 \rightarrow n_{128}^{10000} = \frac{10000 + 127}{128} = 79 \quad (79 \times 128 = 10112) \quad (3)$$

In questo modo abbiamo dimensionato il numero di blocchi in modo che tutti i punti vengano processati!

**Problema**, esistono dei limiti:

- un numero massimo di **blocchi** per **griglia**
- un numero massimo di **thread** per **blocco**
- un numero massimo di **thread** per **griglia** (kernel)

## Evoluzione di un Kernel bis

Se il numero **totale** di thread disponibili (ricordiamo che le c.c. limitano il numero max di thread per blocco e il numero max di blocchi), non e' sufficiente per coprire tutti gli ingressi dell'array da sommare, cosa succede?

- ogni thread deve processare piu' di un ingresso!
- vogliamo che **tutti** gli ingressi dell'array `c[]` vengano presi (non ci devono essere dei **buchi**!)
- non vogliamo che un ingresso di `c[]` venga processato **piu' di una volta** (sarebbe uno spreco)

```
void add( int *a, int *b, int *c ) {
    int i = threadIdx.x + blockIdx.x * blockDim.x; // identificativo univoco per ogni thread
    while ( i < N ) {                               // evita di uscire dall'array...
        c[i] = a[i] + b[i];
        i += blockDim.x * gridDim.x; // salta di un valore uguale al numero
                                     // di tutti i thread del kernel
    }
}
```

Supponiamo che:

- numero totale di thread lanciati sul kernel  $n_{TotThread}=10000$
- numero totale di ingressi dell'array  $N=3\ 000\ 000$

allora:

- all'inizio riempiamo tutti gli ingressi di `c[]` corrispondenti all'identificativo del thread che sta eseguendo (in questo modo sono sicuro che ho riempito tutti i valori di `c[]` dallo 0 a 1 numero di thread lanciati con il kernel (=  $n_{TotThread}$ ), in questo esempio quindi [0,9999])
- a questo punto si devono riempire gli ingressi dopo il 9999.
  - il thread con identificativo 0, riempira' gli ingressi 10000, 20000,.... e tutti i multipli  $< N$
  - il thread con identificativo 1, riempira' gli ingressi 10001, 20001,....
  - il thread con identificativo 2, riempira' gli ingressi 10002, 20002,....
  - ...



## Evoluzione di un Kernel bis

Se il numero **totale** di thread disponibili (ricordiamo che le c.c. limitano il numero max di thread per blocco e il numero max di blocchi), non e' sufficiente per coprire tutti gli ingressi dell'array da sommare, cosa succede?

- ogni thread deve processare piu' di un ingresso!
- vogliamo che **tutti** gli ingressi dell'array `c[]` vengano presi (non ci devono essere dei **buchi**!)
- non vogliamo che un ingresso di `c[]` venga processato **piu' di una volta** (sarebbe uno spreco)

```
void add( int *a, int *b, int *c ) {
    int i = threadIdx.x + blockIdx.x * blockDim.x; // identificativo univoco per ogni thread
    while ( i < N ) {                               // evita di uscire dall'array...
        c[i] = a[i] + b[i];
        i += blockDim.x * gridDim.x; // salta di un valore uguale al numero
                                     // di tutti i thread del kernel
    }
}
```

Supponiamo che:

- numero totale di thread lanciati sul kernel  $n_{TotThread}=10000$
- numero totale di ingressi dell'array  $N=3\ 000\ 000$

allora:

- all'inizio riempiamo tutti gli ingressi di `c[]` corrispondenti all'identificativo del thread che sta eseguendo (in questo modo sono sicuro che ho riempito tutti i valori di `c[]` dallo 0 a 1 numero di thread lanciati con il kernel (=  $n_{TotThread}$ ), in questo esempio quindi [0,9999])
- a questo punto si devono riempire gli ingressi dopo il 9999.
  - il thread con identificativo 0, riempira' gli ingressi 10000, 20000,.... e tutti i multipli <  $N$
  - il thread con identificativo 1, riempira' gli ingressi 10001, 20001,....
  - il thread con identificativo 2, riempira' gli ingressi 10002, 20002,....
  - ...

## Evoluzione di un Kernel bis

Se il numero **totale** di thread disponibili (ricordiamo che le c.c. limitano il numero max di thread per blocco e il numero max di blocchi), non e' sufficiente per coprire tutti gli ingressi dell'array da sommare, cosa succede?

- ogni thread deve processare piu' di un ingresso!
- vogliamo che **tutti** gli ingressi dell'array `c[]` vengano presi (non ci devono essere dei **buchi!**)
- non vogliamo che un ingresso di `c[]` venga processato **piu' di una volta** (sarebbe uno spreco)

```
void add( int *a, int *b, int *c ) {
    int i = threadIdx.x + blockIdx.x * blockDim.x; // identificativo univoco per ogni thread
    while ( i < N ) { // evita di uscire dall'array...
        c[i] = a[i] + b[i];
        i += blockDim.x * gridDim.x; // salta di un valore uguale al numero
                                    // di tutti i thread del kernel
    }
}
```

Supponiamo che:

- numero totale di thread lanciati sul kernel  $n_{TotThread}=10000$
- numero totale di ingressi dell'array  $N=3\ 000\ 000$

allora:

- all'inizio riempiamo tutti gli ingressi di `c[]` corrispondenti all'identificativo del thread che sta eseguendo (in questo modo sono sicuro che ho riempito tutti i valori di `c[]` dallo 0 a 1 numero di thread lanciati con il kernel (=  $n_{TotThread}$ ), in questo esempio quindi [0,9999])
- a questo punto si devono riempire gli ingressi dopo il 9999.
  - il thread con identificativo 0, riempira' gli ingressi 10000, 20000,.... e tutti i multipli <  $N$
  - il thread con identificativo 1, riempira' gli ingressi 10001, 20001,....
  - il thread con identificativo 2, riempira' gli ingressi 10002, 20002,....
  - ...

## Evoluzione di un Kernel bis

Se il numero **totale** di thread disponibili (ricordiamo che le c.c. limitano il numero max di thread per blocco e il numero max di blocchi), non e' sufficiente per coprire tutti gli ingressi dell'array da sommare, cosa succede?

- ogni thread deve processare piu' di un ingresso!
- vogliamo che **tutti** gli ingressi dell'array `c[]` vengano presi (non ci devono essere dei **buchi!**)
- **non** vogliamo che un ingresso di `c[]` venga processato **piu' di una volta** (sarebbe uno spreco)

```
void add( int *a, int *b, int *c ) {
    int i = threadIdx.x + blockIdx.x * blockDim.x; // identificativo univoco per ogni thread
    while ( i < N ) { // evita di uscire dall'array...
        c[i] = a[i] + b[i];
        i += blockDim.x * gridDim.x; // salta di un valore uguale al numero
        // di tutti i thread del kernel
    }
}
```

Supponiamo che:

- numero totale di thread lanciati sul kernel  $n_{TotThread}=10000$
- numero totale di ingressi dell'array  $N=3\ 000\ 000$

allora:

- all'inizio riempiamo tutti gli ingressi di `c[]` corrispondenti all'identificativo del thread che sta eseguendo (in questo modo sono sicuro che ho riempito tutti i valori di `c[]` dallo 0 a 1 numero di thread lanciati con il kernel (=  $n_{TotThread}$ ), in questo esempio quindi [0,9999])
- a questo punto si devono riempire gli ingressi dopo il 9999.
  - Il thread con identificativo 0, riempira' gli ingressi 10000, 20000,.... e tutti i multipli <  $N$
  - il thread con identificativo 1, riempira' gli ingressi 10001, 20001,....
  - il thread con identificativo 2, riempira' gli ingressi 10002, 20002,....
  - ...

## Evoluzione di un Kernel bis

Se il numero **totale** di thread disponibili (ricordiamo che le c.c. limitano il numero max di thread per blocco e il numero max di blocchi), non e' sufficiente per coprire tutti gli ingressi dell'array da sommare, cosa succede?

- ogni thread deve processare piu' di un ingresso!
- vogliamo che **tutti** gli ingressi dell'array `c[]` vengano presi (non ci devono essere dei **buchi!**)
- **non** vogliamo che un ingresso di `c[]` venga processato **piu' di una volta** (sarebbe uno spreco)

```
void add( int *a, int *b, int *c ) {
    int i = threadIdx.x + blockIdx.x * blockDim.x; // identificativo univoco per ogni thread
    while ( i < N ) { // evita di uscire dall'array...
        c[i] = a[i] + b[i];
        i += blockDim.x * gridDim.x; // salta di un valore uguale al numero
        // di tutti i thread del kernel
    }
}
```

Supponiamo che:

- numero totale di thread lanciati sul kernel  $n_{TotThread}=10000$
- numero totale di ingressi dell'array  $N=3\ 000\ 000$

allora:

- all'inizio riempiamo tutti gli ingressi di `c[]` corrispondenti all'identificativo del thread che sta eseguendo (in questo modo sono sicuro che ho riempito tutti i valori di `c[]` dallo 0 a 1 numero di thread lanciati con il kernel (=  $n_{TotThread}$ ), in questo esempio quindi [0,9999])
- a questo punto si devono riempire gli ingressi dopo il 9999.
  - Il thread con identificativo 0, riempira' gli ingressi 10000, 20000,.... e tutti i multipli  $< N$
  - il thread con identificativo 1, riempira' gli ingressi 10001, 20001,....
  - il thread con identificativo 2, riempira' gli ingressi 10002, 20002,....
  - ...

## Evoluzione di un Kernel bis

Se il numero **totale** di thread disponibili (ricordiamo che le c.c. limitano il numero max di thread per blocco e il numero max di blocchi), non e' sufficiente per coprire tutti gli ingressi dell'array da sommare, cosa succede?

- ogni thread deve processare piu' di un ingresso!
- vogliamo che **tutti** gli ingressi dell'array `c[]` vengano presi (non ci devono essere dei **buchi!**)
- **non** vogliamo che un ingresso di `c[]` venga processato **piu' di una volta** (sarebbe uno spreco)

```
void add( int *a, int *b, int *c ) {
    int i = threadIdx.x + blockIdx.x * blockDim.x; // identificativo univoco per ogni thread
    while ( i < N ) { // evita di uscire dall'array...
        c[i] = a[i] + b[i];
        i += blockDim.x * gridDim.x; // salta di un valore uguale al numero
        // di tutti i thread del kernel
    }
}
```

Supponiamo che:

- numero totale di thread lanciati sul kernel  $n_{TotThread}=10000$
- numero totale di ingressi dell'array  $N=3\ 000\ 000$

allora:

- all'inizio riempiamo tutti gli ingressi di `c[]` corrispondenti all'identificativo del thread che sta eseguendo (in questo modo sono sicuro che ho riempito tutti i valori di `c[]` dallo 0 a 1 numero di thread lanciati con il kernel (=  $n_{TotThread}$ ), in questo esempio quindi [0,9999])
  - a questo punto si devono riempire gli ingressi dopo il 9999.
    - Il thread con identificativo 0, riempira' gli ingressi 10000, 20000,.... e tutti i multipli <  $N$
    - il thread con identificativo 1, riempira' gli ingressi 10001, 20001,....
    - il thread con identificativo 2, riempira' gli ingressi 10002, 20002,....
    - ...

## Evoluzione di un Kernel bis

Se il numero **totale** di thread disponibili (ricordiamo che le c.c. limitano il numero max di thread per blocco e il numero max di blocchi), non e' sufficiente per coprire tutti gli ingressi dell'array da sommare, cosa succede?

- ogni thread deve processare piu' di un ingresso!
- vogliamo che **tutti** gli ingressi dell'array `c[]` vengano presi (non ci devono essere dei **buchi!**)
- **non** vogliamo che un ingresso di `c[]` venga processato **piu' di una volta** (sarebbe uno spreco)

```
void add( int *a, int *b, int *c ) {
    int i = threadIdx.x + blockIdx.x * blockDim.x; // identificativo univoco per ogni thread
    while ( i < N ) { // evita di uscire dall'array...
        c[i] = a[i] + b[i];
        i += blockDim.x * gridDim.x; // salta di un valore uguale al numero
        // di tutti i thread del kernel
    }
}
```

Supponiamo che:

- numero totale di thread lanciati sul kernel  $n_{TotThread}=10000$
- numero totale di ingressi dell'array  $N=3\ 000\ 000$

allora:

- all'inizio riempiamo tutti gli ingressi di `c[]` corrispondenti all'identificativo del thread che sta eseguendo (in questo modo sono sicuro che ho riempito tutti i valori di `c[]` dallo 0 a 1 numero di thread lanciati con il kernel (=  $n_{TotThread}$ ), in questo esempio quindi [0,9999])
- a questo punto si devono riempire gli ingressi dopo il 9999.
  - Il thread con identificativo 0, riempira' gli ingressi 10000, 20000,.... e tutti i multipli  $< N$
  - il thread con identificativo 1, riempira' gli ingressi 10001, 20001,....
  - il thread con identificativo 2, riempira' gli ingressi 10002, 20002,....
  - ...

## Evoluzione di un Kernel bis

Se il numero **totale** di thread disponibili (ricordiamo che le c.c. limitano il numero max di thread per blocco e il numero max di blocchi), non e' sufficiente per coprire tutti gli ingressi dell'array da sommare, cosa succede?

- ogni thread deve processare piu' di un ingresso!
- vogliamo che **tutti** gli ingressi dell'array `c[]` vengano presi (non ci devono essere dei **buchi!**)
- **non** vogliamo che un ingresso di `c[]` venga processato **piu' di una volta** (sarebbe uno spreco)

```
void add( int *a, int *b, int *c ) {
    int i = threadIdx.x + blockIdx.x * blockDim.x; // identificativo univoco per ogni thread
    while ( i < N ) { // evita di uscire dall'array...
        c[i] = a[i] + b[i];
        i += blockDim.x * gridDim.x; // salta di un valore uguale al numero
        // di tutti i thread del kernel
    }
}
```

Supponiamo che:

- numero totale di thread lanciati sul kernel  $n_{TotThread}=10000$
- numero totale di ingressi dell'array  $N=3\ 000\ 000$

allora:

- all'inizio riempiamo tutti gli ingressi di `c[]` corrispondenti all'identificativo del thread che sta eseguendo (in questo modo sono sicuro che ho riempito tutti i valori di `c[]` dallo 0 a 1 numero di thread lanciati con il kernel (=  $n_{TotThread}$ ), in questo esempio quindi [0,9999])
- a questo punto si devono riempire gli ingressi dopo il 9999.
  - Il thread con identificativo 0, riempira' gli ingressi 10000, 20000,.... e tutti i multipli  $< N$
  - il thread con identificativo 1, riempira' gli ingressi 10001, 20001,....
  - il thread con identificativo 2, riempira' gli ingressi 10002, 20002,....
  - ...

## Evoluzione di un Kernel bis

Se il numero **totale** di thread disponibili (ricordiamo che le c.c. limitano il numero max di thread per blocco e il numero max di blocchi), non e' sufficiente per coprire tutti gli ingressi dell'array da sommare, cosa succede?

- ogni thread deve processare piu' di un ingresso!
- vogliamo che **tutti** gli ingressi dell'array `c[]` vengano presi (non ci devono essere dei **buchi!**)
- **non** vogliamo che un ingresso di `c[]` venga processato **piu' di una volta** (sarebbe uno spreco)

```
void add( int *a, int *b, int *c ) {
    int i = threadIdx.x + blockIdx.x * blockDim.x; // identificativo univoco per ogni thread
    while ( i < N ) { // evita di uscire dall'array...
        c[i] = a[i] + b[i];
        i += blockDim.x * gridDim.x; // salta di un valore uguale al numero
        // di tutti i thread del kernel
    }
}
```

Supponiamo che:

- numero totale di thread lanciati sul kernel  $n_{TotThread}=10000$
- numero totale di ingressi dell'array  $N=3\ 000\ 000$

allora:

- all'inizio riempiamo tutti gli ingressi di `c[]` corrispondenti all'identificativo del thread che sta eseguendo (in questo modo sono sicuro che ho riempito tutti i valori di `c[]` dallo 0 a 1 numero di thread lanciati con il kernel (=  $n_{TotThread}$ ), in questo esempio quindi [0,9999])
- a questo punto si devono riempire gli ingressi dopo il 9999.
  - Il thread con identificativo 0, riempira' gli ingressi 10000, 20000,.... e tutti i multipli  $< N$
  - il thread con identificativo 1, riempira' gli ingressi 10001, 20001,....
  - il thread con identificativo 2, riempira' gli ingressi 10002, 20002,....



## Evoluzione di un Kernel bis

Se il numero **totale** di thread disponibili (ricordiamo che le c.c. limitano il numero max di thread per blocco e il numero max di blocchi), non e' sufficiente per coprire tutti gli ingressi dell'array da sommare, cosa succede?

- ogni thread deve processare piu' di un ingresso!
- vogliamo che **tutti** gli ingressi dell'array `c[]` vengano presi (non ci devono essere dei **buchi!**)
- **non** vogliamo che un ingresso di `c[]` venga processato **piu' di una volta** (sarebbe uno spreco)

```
void add( int *a, int *b, int *c ) {
    int i = threadIdx.x + blockIdx.x * blockDim.x; // identificativo univoco per ogni thread
    while ( i < N ) { // evita di uscire dall'array...
        c[i] = a[i] + b[i];
        i += blockDim.x * gridDim.x; // salta di un valore uguale al numero
        // di tutti i thread del kernel
    }
}
```

Supponiamo che:

- numero totale di thread lanciati sul kernel  $n_{TotThread}=10000$
- numero totale di ingressi dell'array  $N=3\ 000\ 000$

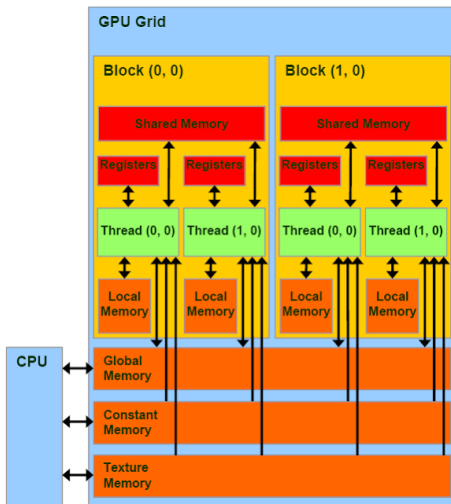
allora:

- all'inizio riempiamo tutti gli ingressi di `c[]` corrispondenti all'identificativo del thread che sta eseguendo (in questo modo sono sicuro che ho riempito tutti i valori di `c[]` dallo 0 a 1 numero di thread lanciati con il kernel (=  $n_{TotThread}$ ), in questo esempio quindi [0,9999])
- a questo punto si devono riempire gli ingressi dopo il 9999.
  - Il thread con identificativo 0, riempira' gli ingressi 10000, 20000,.... e tutti i multipli  $< N$
  - il thread con identificativo 1, riempira' gli ingressi 10001, 20001,....
  - il thread con identificativo 2, riempira' gli ingressi 10002, 20002,....
  - ...

# Outline

- 1 *Struttura Fisica*
- 2 *Kernel*
- 3 *Struttura Logica*
- 4 *Memorie***
- 5 *Prodotto Matrici*
- 6 *Stream*
- 7 *Esempi e strumenti*

# Struttura delle memorie: overview



- i **registri** (Kepler) sono 255 per thread (64kb). Velocità di accesso  $\approx 1$  ciclo.
- la **shared** memory ( $\approx 64$  kb per **SM**) e' condivisa da tutti i thread nel blocco: e' **veloce**. I thread di un blocco non accedono alla shared di un altro blocco. Velocità di accesso  $\approx 5$  cicli.
- la **constant** memory ( $\approx 64$  kb), velocità di accesso  $\approx 5$  cicli (sulla cache).
- la **local** memory ( $\approx 512$  kb) e' assegnata ad ogni singolo thread, e' solo un sottoinsieme della global: e' **lenta** come la global  $\approx 500$  cicli.
- la **global** memory (DRAM  $\approx 4$ Gb) e' **lenta** ( $\approx 500$  cicli). Bandwidth circa 200 Gb/s. Vista da tutti i thread.
- la **texture** memory ( $\approx 12/48$  kb)

I valori qui riportati sono solo **indicativi**, e servono solo per avere un'idea di quanto sono gli **ordini di grandezza** delle varie memorie associate (ovviamente dipendono dalle **compute capabilities** di ogni scheda).

*Figure:* da <http://cuda-programming.blogspot.it/2013/01/what-is-constant-memory-in-cuda.html>.

# Shared Memory

- `__shared__` e' il comando per dichiarare la memoria **condivisa** dai **thread** in un **blocco**.
- si dichiara **dentro** il kernel
- esempio di una dichiarazione: `__shared__ int a[100];`
- la *shared* e' **veloce**. Il buffer della shared memory risiede fisicamente sulla GPU invece che essere altrove sulla scheda grafica, come la DRAM. Per questo la **latency** (latenza) e' molto bassa  $\approx 10$  cicli
- la *shared* e' **poca**  $\approx 64$  kb/SM (dipende dalla GPU): **Attenzione**: ogni SM ha il proprio blocco di **memoria shared**
- visto che thread diversi possono accedere alla stessa area di memoria e' importante definire dei metodi di sincronizzazione per evitare **race conditions**.
- non c'e' bisogno di indicare a quale blocco questa memoria viene assegnata, in quanto **OGNI** blocco ha la sua copia della variabile/array *shared* creata automaticamente dal compilatore.
- supponiamo di lanciare un kernel su vari blocchi... come si distribuisce la memoria *shared*? Semplice, appena lanciato **ogni blocco** avra' una **propria copia** della memoria *shared* che potra' quindi gestire. **Attenzione** Dal momento del lancio in poi le memorie *shared* di blocchi diversi potranno divergere!
- La memoria *shared* e' accessibile **velocemente** da **tutti thread di un blocco**, in questo modo i thread di un blocco possono comunicare in modo naturale (pensiamo ad openMP)!
- un thread puo' accedere solo alla memoria `__shared__` del **proprio blocco** (non a quella degli altri blocchi).
- la memoria `__shared__` **non e' persistente**, ovvero il suo status non e' mantenuto tra differenti chiamate di kernel

# Shared Memory

- `__shared__` e' il comando per dichiarare la memoria **condivisa** dai **thread** in un **blocco**.
- si dichiara **dentro** il kernel
- esempio di una dichiarazione: `__shared__ int a[100];`
- la *shared* e' **veloce**. Il buffer della shared memory risiede fisicamente sulla GPU invece che essere altrove sulla scheda grafica, come la DRAM. Per questo la **latency** (latenza) e' molto bassa  $\approx 10$  cicli
- la *shared* e' **poca**  $\approx 64$  kb/SM (dipende dalla GPU): Attenzione: ogni SM ha il proprio blocco di **memoria shared**
- visto che thread diversi possono accedere alla stessa area di memoria e' importante definire dei metodi di sincronizzazione per evitare **race conditions**.
- non c'e' bisogno di indicare a quale blocco questa memoria viene assegnata, in quanto **OGNI** blocco ha la sua copia della variabile/array *shared* creata automaticamente dal compilatore.
- supponiamo di lanciare un kernel su vari blocchi... come si distribuisce la memoria *shared*? Semplice, appena lanciato **ogni blocco** avra' una **propria copia** della memoria *shared* che potra' quindi gestire. **Attenzione** Dal momento del lancio in poi le memorie *shared* di blocchi diversi potranno divergere!
- La memoria *shared* e' accessibile **velocemente** da **tutti thread di un blocco**, in questo modo i thread di un blocco possono comunicare in modo naturale (pensiamo ad openMP)!
- un thread puo' accedere solo alla memoria `__shared__` del **proprio blocco** (non a quella degli altri blocchi).
- la memoria `__shared__` **non e' persistente**, ovvero il suo status non e' mantenuto tra differenti chiamate di kernel

# Shared Memory

- `__shared__` e' il comando per dichiarare la memoria **condivisa** dai **thread** in un **blocco**.
- si dichiara **dentro** il kernel
- esempio di una dichiarazione: `__shared__ int a[100];`
- la *shared* e' **veloce**. Il buffer della shared memory risiede fisicamente sulla GPU invece che essere altrove sulla scheda grafica, come la DRAM. Per questo la **latency** (latenza) e' molto bassa  $\approx 10$  cicli
- la *shared* e' **poca**  $\approx 64$  kb/SM (dipende dalla GPU): **Attenzione**: ogni **SM** ha il proprio blocco di **memoria shared**
- visto che thread diversi possono accedere alla stessa area di memoria e' importante definire dei metodi di sincronizzazione per evitare **race conditions**.
- non c'e' bisogno di indicare a quale blocco questa memoria viene assegnata, in quanto **OGNI** blocco ha la sua copia della variabile/array *shared* creata automaticamente dal compilatore.
- supponiamo di lanciare un kernel su vari blocchi... come si distribuisce la memoria *shared*? Semplice, appena lanciato **ogni blocco** avra' una **propria copia** della memoria *shared* che potra' quindi gestire. **Attenzione** Dal momento del lancio in poi le memorie *shared* di blocchi diversi potranno divergere!
- La memoria *shared* e' accessibile **velocemente** da **tutti thread di un blocco**, in questo modo i thread di un blocco possono comunicare in modo naturale (pensiamo ad openMP)!
- un thread puo' accedere solo alla memoria `__shared__` del **proprio blocco** (non a quella degli altri blocchi).
- la memoria `__shared__` **non e' persistente**, ovvero il suo status non e' mantenuto tra differenti chiamate di kernel

# Shared Memory

- `__shared__` e' il comando per dichiarare la memoria **condivisa** dai **thread** in un **blocco**.
- si dichiara **dentro** il kernel
- esempio di una dichiarazione: `__shared__ int a[100];`
- la *shared* e' **veloce**. Il buffer della shared memory risiede fisicamente sulla GPU invece che essere altrove sulla scheda grafica, come la DRAM. Per questo la **latency** (latenza) e' molto bassa  $\approx 10$  cicli
- la *shared* e' **poca**  $\approx 64$  kb/SM (dipende dalla GPU): Attenzione: ogni SM ha il proprio blocco di **memoria shared**
- visto che thread diversi possono accedere alla stessa area di memoria e' importante definire dei metodi di sincronizzazione per evitare **race conditions**.
- non c'e' bisogno di indicare a quale blocco questa memoria viene assegnata, in quanto **OGNI** blocco ha la sua copia della variabile/array *shared* creata automaticamente dal compilatore.
- supponiamo di lanciare un kernel su vari blocchi... come si distribuisce la memoria *shared*? Semplice, appena lanciato **ogni blocco** avra' una **propria copia** della memoria *shared* che potra' quindi gestire. **Attenzione** Dal momento del lancio in poi le memorie *shared* di blocchi diversi potranno divergere!
- La memoria *shared* e' accessibile **velocemente** da **tutti thread di un blocco**, in questo modo i thread di un blocco possono comunicare in modo naturale (pensiamo ad openMP)!
- un thread puo' accedere solo alla memoria `__shared__` del **proprio blocco** (non a quella degli altri blocchi).
- la memoria `__shared__` **non e' persistente**, ovvero il suo status non e' mantenuto tra differenti chiamate di kernel

# Shared Memory

- `__shared__` e' il comando per dichiarare la memoria **condivisa** dai **thread** in un **blocco**.
- si dichiara **dentro** il kernel
- esempio di una dichiarazione: `__shared__ int a[100];`
- la *shared* e' **veloce**. Il buffer della shared memory risiede fisicamente sulla GPU invece che essere altrove sulla scheda grafica, come la DRAM. Per questo la **latency** (latenza) e' molto bassa  $\approx 10$  cicli
- la *shared* e' **poca**  $\approx 64$  kb/SM (dipende dalla GPU): Attenzione: ogni **SM** ha il proprio blocco di **memoria shared**
- visto che thread diversi possono accedere alla stessa area di memoria e' importante definire dei metodi di sincronizzazione per evitare **race conditions**.
- non c'e' bisogno di indicare a quale blocco questa memoria viene assegnata, in quanto **OGNI** blocco ha la sua copia della variabile/array *shared* creata automaticamente dal compilatore.
- supponiamo di lanciare un kernel su vari blocchi... come si distribuisce la memoria *shared*? Semplice, appena lanciato **ogni blocco** avra' una **propria copia** della memoria *shared* che potra' quindi gestire. **Attenzione** Dal momento del lancio in poi le memorie *shared* di blocchi diversi potranno divergere!
- La memoria *shared* e' accessibile **velocemente** da **tutti thread** di un blocco, in questo modo i thread di un blocco possono comunicare in modo naturale (pensiamo ad openMP)
- un thread puo' accedere solo alla memoria `__shared__` del **proprio blocco** (non a quella degli altri blocchi).
- la memoria `__shared__` **non e' persistente**, ovvero il suo status non e' mantenuto tra differenti chiamate di kernel



# Shared Memory

- `__shared__` e' il comando per dichiarare la memoria **condivisa** dai **thread** in un **blocco**.
- si dichiara **dentro** il kernel
- esempio di una dichiarazione: `__shared__ int a[100];`
- la *shared* e' **veloce**. Il buffer della shared memory risiede fisicamente sulla GPU invece che essere altrove sulla scheda grafica, come la DRAM. Per questo la **latency** (latenza) e' molto bassa  $\approx 10$  cicli
- la *shared* e' **poca**  $\approx 64$  kb/SM (dipende dalla GPU): Attenzione: ogni **SM** ha il proprio blocco di **memoria shared**
- visto che thread diversi possono accedere alla stessa area di memoria e' importante definire dei metodi di sincronizzazione per evitare **race conditions**.
- non c'e' bisogno di indicare a quale blocco questa memoria viene assegnata, in quanto **OGNI** blocco ha la sua copia della variabile/array *shared* creata automaticamente dal compilatore.
- supponiamo di lanciare un kernel su vari blocchi... come si distribuisce la memoria *shared*? Semplice, appena lanciato **ogni blocco** avra' una **propria copia** della memoria *shared* che potra' quindi gestire. **Attenzione** Dal momento del lancio in poi le memorie *shared* di blocchi diversi potranno divergere!
- La memoria *shared* e' accessibile **velocemente** da **tutti thread** di un blocco, in questo modo i thread di un blocco possono comunicare in modo naturale (pensiamo ad openMP)
- un thread puo' accedere solo alla memoria `__shared__` del **proprio blocco** (non a quella degli altri blocchi).
- la memoria `__shared__` **non e' persistente**, ovvero il suo status non e' mantenuto tra differenti chiamate di kernel

# Shared Memory

- `__shared__` e' il comando per dichiarare la memoria **condivisa** dai **thread** in un **blocco**.
- si dichiara **dentro** il kernel
- esempio di una dichiarazione: `__shared__ int a[100];`
- la *shared* e' **veloce**. Il buffer della shared memory risiede fisicamente sulla GPU invece che essere altrove sulla scheda grafica, come la DRAM. Per questo la **latency** (latenza) e' molto bassa  $\approx 10$  cicli
- la *shared* e' **poca**  $\approx 64$  kb/SM (dipende dalla GPU): Attenzione: ogni **SM** ha il proprio blocco di **memoria shared**
- visto che thread diversi possono accedere alla stessa area di memoria e' importante definire dei metodi di sincronizzazione per evitare **race conditions**.
- non c'e' bisogno di indicare a quale blocco questa memoria viene assegnata, in quanto **OGNI** blocco ha la sua copia della variabile/array `shared` creata automaticamente dal compilatore.
- supponiamo di lanciare un kernel su vari blocchi... come si distribuisce la memoria *shared*? Semplice, appena lanciato **ogni blocco** avra' una **propria copia** della memoria *shared* che potra' quindi gestire. **Attenzione** Dal momento del lancio in poi le memorie *shared* di blocchi diversi potranno divergere!
- La memoria *shared* e' accessibile **velocemente** da **tutti thread di un blocco**, in questo modo i thread di un blocco possono comunicare in modo naturale (pensiamo ad openMP)!
- un thread puo' accedere solo alla memoria `__shared__` del **proprio blocco** (non a quella degli altri blocchi).
- la memoria `__shared__` **non e' persistente**, ovvero il suo status non e' mantenuto tra differenti chiamate di kernel

# Shared Memory

- `__shared__` e' il comando per dichiarare la memoria **condivisa** dai **thread** in un **blocco**.
- si dichiara **dentro** il kernel
- esempio di una dichiarazione: `__shared__ int a[100];`
- la *shared* e' **veloce**. Il buffer della shared memory risiede fisicamente sulla GPU invece che essere altrove sulla scheda grafica, come la DRAM. Per questo la **latency** (latenza) e' molto bassa  $\approx 10$  cicli
- la *shared* e' **poca**  $\approx 64$  kb/SM (dipende dalla GPU): Attenzione: ogni **SM** ha il proprio blocco di **memoria shared**
- visto che thread diversi possono accedere alla stessa area di memoria e' importante definire dei metodi di sincronizzazione per evitare **race conditions**.
- non c'e' bisogno di indicare a quale blocco questa memoria viene assegnata, in quanto **OGNI** blocco ha la sua copia della variabile/array *shared* creata automaticamente dal compilatore.
- supponiamo di lanciare un kernel su vari blocchi... come si distribuisce la memoria *shared*? Semplice, appena lanciato **ogni blocco** avra' una **propria copia** della memoria *shared* che potra' quindi gestire. **Attenzione** Dal momento del lancio in poi le memorie *shared* di blocchi diversi potranno divergere!
- La memoria *shared* e' accessibile **velocemente** da **tutti thread di un blocco**, in questo modo i thread di un blocco possono comunicare in modo naturale (pensiamo ad openMP)!
- un thread puo' accedere solo alla memoria `__shared__` del **proprio blocco** (non a quella degli altri blocchi).
- la memoria `__shared__` **non e' persistente**, ovvero il suo status non e' mantenuto tra differenti chiamate di kernel

# Shared Memory

- `__shared__` e' il comando per dichiarare la memoria **condivisa** dai **thread** in un **blocco**.
- si dichiara **dentro** il kernel
- esempio di una dichiarazione: `__shared__ int a[100];`
- la *shared* e' **veloce**. Il buffer della shared memory risiede fisicamente sulla GPU invece che essere altrove sulla scheda grafica, come la DRAM. Per questo la **latency** (latenza) e' molto bassa  $\approx 10$  cicli
- la *shared* e' **poca**  $\approx 64$  kb/SM (dipende dalla GPU): Attenzione: ogni **SM** ha il proprio blocco di **memoria shared**
- visto che thread diversi possono accedere alla stessa area di memoria e' importante definire dei metodi di sincronizzazione per evitare **race conditions**.
- non c'e' bisogno di indicare a quale blocco questa memoria viene assegnata, in quanto **OGNI** blocco ha la sua copia della variabile/array `shared` creata automaticamente dal compilatore.
- supponiamo di lanciare un kernel su vari blocchi... come si distribuisce la memoria `shared`? Semplice, appena lanciato **ogni blocco** avra' una **propria copia** della memoria `shared` che potra' quindi gestire. **Attenzione** Dal momento del lancio in poi le memorie `shared` di blocchi diversi potranno divergere!
- La memoria `shared` e' accessibile **velocemente** da **tutti thread di un blocco**, in questo modo i thread di un blocco possono comunicare in modo naturale (pensiamo ad openMP)!
  - un thread puo' accedere solo alla memoria `__shared__` del **proprio blocco** (non a quella degli altri blocchi).
  - la memoria `__shared__` **non e' persistente**, ovvero il suo status non e' mantenuto tra differenti chiamate di kernel

# Shared Memory

- `__shared__` e' il comando per dichiarare la memoria **condivisa** dai **thread** in un **blocco**.
- si dichiara **dentro** il kernel
- esempio di una dichiarazione: `__shared__ int a[100];`
- la *shared* e' **veloce**. Il buffer della shared memory risiede fisicamente sulla GPU invece che essere altrove sulla scheda grafica, come la DRAM. Per questo la **latency** (latenza) e' molto bassa  $\approx 10$  cicli
- la *shared* e' **poca**  $\approx 64$  kb/SM (dipende dalla GPU): Attenzione: ogni **SM** ha il proprio blocco di **memoria shared**
- visto che thread diversi possono accedere alla stessa area di memoria e' importante definire dei metodi di sincronizzazione per evitare **race conditions**.
- non c'e' bisogno di indicare a quale blocco questa memoria viene assegnata, in quanto **OGNI** blocco ha la sua copia della variabile/array `shared` creata automaticamente dal compilatore.
- supponiamo di lanciare un kernel su vari blocchi... come si distribuisce la memoria `shared`? Semplice, appena lanciato **ogni blocco** avra' una **propria copia** della memoria `shared` che potra' quindi gestire. **Attenzione** Dal momento del lancio in poi le memorie `shared` di blocchi diversi potranno divergere!
- La memoria `shared` e' accessibile **velocemente** da **tutti thread di un blocco**, in questo modo i thread di un blocco possono comunicare in modo naturale (pensiamo ad openMP)!
- un thread puo' accedere solo alla memoria `__shared__` del **proprio blocco** (non a quella degli altri blocchi).
- la memoria `__shared__` **non e' persistente**, ovvero il suo status non e' mantenuto tra differenti chiamate di kernel

# Shared Memory

- `__shared__` e' il comando per dichiarare la memoria **condivisa** dai **thread** in un **blocco**.
- si dichiara **dentro** il kernel
- esempio di una dichiarazione: `__shared__ int a[100];`
- la *shared* e' **veloce**. Il buffer della shared memory risiede fisicamente sulla GPU invece che essere altrove sulla scheda grafica, come la DRAM. Per questo la **latency** (latenza) e' molto bassa  $\approx 10$  cicli
- la *shared* e' **poca**  $\approx 64$  kb/SM (dipende dalla GPU): Attenzione: ogni **SM** ha il proprio blocco di **memoria shared**
- visto che thread diversi possono accedere alla stessa area di memoria e' importante definire dei metodi di sincronizzazione per evitare **race conditions**.
- non c'e' bisogno di indicare a quale blocco questa memoria viene assegnata, in quanto **OGNI** blocco ha la sua copia della variabile/array `shared` creata automaticamente dal compilatore.
- supponiamo di lanciare un kernel su vari blocchi... come si distribuisce la memoria `shared`? Semplice, appena lanciato **ogni blocco** avra' una **propria copia** della memoria `shared` che potra' quindi gestire. **Attenzione** Dal momento del lancio in poi le memorie `shared` di blocchi diversi potranno divergere!
- La memoria `shared` e' accessibile **velocemente** da **tutti thread di un blocco**, in questo modo i thread di un blocco possono comunicare in modo naturale (pensiamo ad openMP)!
- un thread puo' accedere solo alla memoria `__shared__` del **proprio blocco** (non a quella degli altri blocchi).
- la memoria `__shared__` **non e' persistente**, ovvero il suo status non e' mantenuto tra differenti chiamate di kernel

## Prodotto Scalare: un ingresso per thread

Consideriamo il prodotto scalare (spesso chiamato *dot product* dagli amici anglofoni) tra due vettori:

$$c = \mathbf{x} \cdot \mathbf{y} = (x_1, x_2, x_3, x_4) \cdot (y_1, y_2, y_3, y_4) = x_1y_1 + x_2y_2 + x_3y_3 + x_4y_4$$

- Creiamo un array *shared*, con  $n$  componenti, (chiamato *cache*) uno per ogni ingresso dell'array (tutti i thread di un blocco possono accedervi)
- facciamo sì che il thread  $n$ -esimo calcoli il prodotto delle  $n$ -esime componenti  $x_n y_n$ . Questo valore va inserito poi nella  $n$ -esima componente dell'array *shared*.
- Per ottenere il valore del prodotto scalare, dobbiamo poi sommare tutti questi valori.
- Proviamo a scrivere una implementazione *semplice* di un kernel dove un singolo thread esegue la somma di tutti gli ingressi del vettore *shared*:

```
__global__ void dot( int *a, int *b, int *c ) {
  __shared__ int cache[N]; // <- questo viene messo nella memoria shared
  cache[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x]; // ogni th riempie un ingresso
  __syncthreads(); // aspettiamo che tutti gli ingressi siano riempiti
  if( 0 == threadIdx.x ) { // facciamo fare la somma totale al thread 0
    int sum = 0;
    for( int i = 0; i < N; i++ ) sum += cache[i];
    *c = sum; // dereferenzio il risultato
  }
}
```

- **Peccato** in questo caso il processo di riduzione non e' parallelo (un solo thread lavora)
- la funzione `__syncthreads()` **sincronizza** i thread di un **blocco**, questi non possono continuare finche' tutti i thread (del blocco) hanno chiamato `__syncthreads()` (e' come una barriera di MPI)
- Se non ci fosse `__syncthreads()` ci sarebbe una condizione di **race condition**. Il thread 0 che deve sommare tutti gli ingressi del vettore *cache* non **sa** se questi ingressi sono gia' stati riempiti dagli altri thread o sono ancora inizializzati a zero!
- **Attenzione** `__syncthreads()` e' equivalente alle **collective calls** di MPI se solo una parte dei thread del blocco lo esegue il sistema si ferma, si va incontro ad un **deadlock!**

## Prodotto Scalare: un ingresso per thread

Consideriamo il prodotto scalare (spesso chiamato *dot product* dagli amici anglofoni) tra due vettori:

$$c = \mathbf{x} \cdot \mathbf{y} = (x_1, x_2, x_3, x_4) \cdot (y_1, y_2, y_3, y_4) = x_1y_1 + x_2y_2 + x_3y_3 + x_4y_4$$

- Creiamo un array *shared*, con  $n$  componenti, (chiamato `cache`) uno per ogni ingresso dell'array (tutti i thread di un blocco possono accedervi)
- facciamo sì che il thread  $n$ -esimo calcoli il prodotto delle  $n$ -esime componenti  $x_n y_n$ . Questo valore va inserito poi nella  $n$ -esima componente dell'array `shared`.
- Per ottenere il valore del prodotto scalare, dobbiamo poi sommare tutti questi valori.
- Proviamo a scrivere una implementazione *semplice* di un kernel dove un singolo thread esegue la somma di tutti gli ingressi del vettore `shared`:

```
__global__ void dot( int *a, int *b, int *c ) {
    __shared__ int cache[N]; // <- questo viene messo nella memoria shared
    cache[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x]; // ogni th riempie un ingresso
    __syncthreads(); // aspettiamo che tutti gli ingressi siano riempiti
    if( 0 == threadIdx.x ) { // facciamo fare la somma totale al thread 0
        int sum = 0;
        for( int i = 0; i < N; i++ ) sum += cache[i];
        *c = sum; // dereferenzio il risultato
    }
}
```

- **Peccato** in questo caso il processo di riduzione non e' parallelo (un solo thread lavora)
- la funzione `__syncthreads()` **sincronizza** i thread di un **blocco**, questi non possono continuare finche' tutti i thread (del blocco) hanno chiamato `__syncthreads()` (e' come una barriera di MPI)
- Se non ci fosse `__syncthreads()` ci sarebbe una condizione di **race condition**. Il thread 0 che deve sommare tutti gli ingressi del vettore `cache` non **sa** se questi ingressi sono gia' stati riempiti dagli altri thread o sono ancora inizializzati a zero!
- **Attenzione** `__syncthreads()` e' equivalente alle **collective calls** di MPI se solo una parte dei thread del blocco lo esegue il sistema si ferma, si va incontro ad un **deadlock!**



## Prodotto Scalare: un ingresso per thread

Consideriamo il prodotto scalare (spesso chiamato *dot product* dagli amici anglofoni) tra due vettori:

$$c = \mathbf{x} \cdot \mathbf{y} = (x_1, x_2, x_3, x_4) \cdot (y_1, y_2, y_3, y_4) = x_1y_1 + x_2y_2 + x_3y_3 + x_4y_4$$

- Creiamo un array *shared*, con  $n$  componenti, (chiamato `cache`) uno per ogni ingresso dell'array (tutti i thread di un blocco possono accedervi)
- facciamo sì che il thread  $n$ -esimo calcoli il prodotto delle  $n$ -esime componenti  $x_n y_n$ . Questo valore va inserito poi nella  $n$ -esima componente dell'array `shared`.
- Per ottenere il valore del prodotto scalare, dobbiamo poi sommare tutti questi valori.
- Proviamo a scrivere una implementazione *semplice* di un kernel dove un singolo thread esegue la somma di tutti gli ingressi del vettore `shared`:

```
__global__ void dot( int *a, int *b, int *c ) {
    __shared__ int cache[N]; // <- questo viene messo nella memoria shared
    cache[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x]; // ogni th riempie un ingresso
    __syncthreads(); // aspettiamo che tutti gli ingressi siano riempiti
    if( 0 == threadIdx.x ) { // facciamo fare la somma totale al thread 0
        int sum = 0;
        for( int i = 0; i < N; i++ ) sum += cache[i];
        *c = sum; // dereferenzio il risultato
    }
}
```

- **Peccato** in questo caso il processo di riduzione non è parallelo (un solo thread lavora)
- la funzione `__syncthreads()` **sincronizza** i thread di un blocco, questi non possono continuare finché tutti i thread (del blocco) hanno chiamato `__syncthreads()` (è come una barriera di MPI)
- Se non ci fosse `__syncthreads()` ci sarebbe una condizione di **race condition**. Il thread 0 che deve sommare tutti gli ingressi del vettore `cache` non sa se questi ingressi sono già stati riempiti dagli altri thread o sono ancora inizializzati a zero!
- **Attenzione** `__syncthreads()` è equivalente alle **collective calls** di MPI se solo una parte dei thread del blocco lo esegue il sistema si ferma, si va incontro ad un **deadlock!**

## Prodotto Scalare: un ingresso per thread

Consideriamo il prodotto scalare (spesso chiamato *dot product* dagli amici anglofoni) tra due vettori:

$$c = \mathbf{x} \cdot \mathbf{y} = (x_1, x_2, x_3, x_4) \cdot (y_1, y_2, y_3, y_4) = x_1y_1 + x_2y_2 + x_3y_3 + x_4y_4$$

- Creiamo un array *shared*, con  $n$  componenti, (chiamato `cache`) uno per ogni ingresso dell'array (tutti i thread di un blocco possono accedervi)
- facciamo sì che il thread  $n$ -esimo calcoli il prodotto delle  $n$ -esime componenti  $x_n y_n$ . Questo valore va inserito poi nella  $n$ -esima componente dell'array `shared`.
- Per ottenere il valore del prodotto scalare, dobbiamo poi sommare tutti questi valori.
- Proviamo a scrivere una implementazione *semplice* di un kernel dove un singolo thread esegue la somma di tutti gli ingressi del vettore `shared`:

```
__global__ void dot( int *a, int *b, int *c ) {
    __shared__ int cache[N]; // <= questo viene messo nella memoria shared
    cache[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x]; // ogni th riempie un ingresso
    __syncthreads(); // aspettiamo che tutti gli ingressi siano riempiti
    if( 0 == threadIdx.x ) { // facciamo fare la somma totale al thread 0
        int sum = 0;
        for( int i = 0; i < N; i++ ) sum += cache[i];
        *c = sum; // dereferenzio il risultato
    }
}
```

- **Peccato** in questo caso il processo di riduzione non è parallelo (un solo thread lavora)
- la funzione `__syncthreads()` **sincronizza** i thread di un blocco, questi non possono continuare finché tutti i thread (del blocco) hanno chiamato `__syncthreads()` (è come una barriera di MPI)
- Se non ci fosse `__syncthreads()` ci sarebbe una condizione di **race condition**. Il thread 0 che deve sommare tutti gli ingressi del vettore `cache` non sa se questi ingressi sono già stati riempiti dagli altri thread o sono ancora inizializzati a zero!
- **Attenzione** `__syncthreads()` è equivalente alle **collective calls** di MPI se solo una parte dei thread del blocco lo esegue il sistema si ferma, si va incontro ad un **deadlock!**

## Prodotto Scalare: un ingresso per thread

Consideriamo il prodotto scalare (spesso chiamato *dot product* dagli amici anglofoni) tra due vettori:

$$c = \mathbf{x} \cdot \mathbf{y} = (x_1, x_2, x_3, x_4) \cdot (y_1, y_2, y_3, y_4) = x_1y_1 + x_2y_2 + x_3y_3 + x_4y_4$$

- Creiamo un array *shared*, con  $n$  componenti, (chiamato `cache`) uno per ogni ingresso dell'array (tutti i thread di un blocco possono accedervi)
- facciamo sì che il thread  $n$ -esimo calcoli il prodotto delle  $n$ -esime componenti  $x_n y_n$ . Questo valore va inserito poi nella  $n$ -esima componente dell'array *shared*.
- Per ottenere il valore del prodotto scalare, dobbiamo poi sommare tutti questi valori.
- Proviamo a scrivere una implementazione [semplice](#) di un kernel dove un singolo thread esegue la somma di tutti gli ingressi del vettore *shared*:

```
__global__ void dot( int *a, int *b, int *c ) {
    __shared__ int cache[N]; // <= questo viene messo nella memoria shared
    cache[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x]; // ogni th riempie un ingresso
    __syncthreads(); // aspettiamo che tutti gli ingressi siano riempiti
    if( 0 == threadIdx.x ) { // facciamo fare la somma totale al thread 0
        int sum = 0;
        for( int i = 0; i < N; i++ ) sum += cache[i];
        *c = sum; // dereferenzio il risultato
    }
}
```

- **Peccato** in questo caso il processo di riduzione non è parallelo (un solo thread lavora!)
- la funzione `__syncthreads()` **sincronizza** i thread di un blocco, questi non possono continuare finché tutti i thread (del blocco) hanno chiamato `__syncthreads()` (è come una barriera di MPI)
- Se non ci fosse `__syncthreads()` ci sarebbe una condizione di **race condition**. Il thread 0 che deve sommare tutti gli ingressi del vettore `cache` non sa se questi ingressi sono già stati riempiti dagli altri thread o sono ancora inizializzati a zero!
- **Attenzione** `__syncthreads()` è equivalente alle **collective calls** di MPI se solo una parte dei thread del blocco lo esegue il sistema si ferma, si va incontro ad un **deadlock!**

## Prodotto Scalare: un ingresso per thread

Consideriamo il prodotto scalare (spesso chiamato *dot product* dagli amici anglofoni) tra due vettori:

$$c = \mathbf{x} \cdot \mathbf{y} = (x_1, x_2, x_3, x_4) \cdot (y_1, y_2, y_3, y_4) = x_1y_1 + x_2y_2 + x_3y_3 + x_4y_4$$

- Creiamo un array *shared*, con  $n$  componenti, (chiamato `cache`) uno per ogni ingresso dell'array (tutti i thread di un blocco possono accedervi)
- facciamo sì che il thread  $n$ -esimo calcoli il prodotto delle  $n$ -esime componenti  $x_n y_n$ . Questo valore va inserito poi nella  $n$ -esima componente dell'array *shared*.
- Per ottenere il valore del prodotto scalare, dobbiamo poi sommare tutti questi valori.
- Proviamo a scrivere una implementazione **semplice** di un kernel dove un singolo thread esegue la somma di tutti gli ingressi del vettore *shared*:

```
__global__ void dot( int *a, int *b, int *c ) {
    __shared__ int cache[N]; // <= questo viene messo nella memoria shared
    cache[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x]; // ogni th riempie un ingresso
    __syncthreads(); // aspettiamo che tutti gli ingressi siano riempiti
    if( 0 == threadIdx.x ) { // facciamo fare la somma totale al thread 0
        int sum = 0;
        for( int i = 0; i < N; i++ ) sum += cache[i];
        *c = sum; // dereferenzio il risultato
    }
}
```

- **Peccato** in questo caso il processo di riduzione non è parallelo (un solo thread lavora!)
- la funzione `__syncthreads()` **sincronizza** i thread di un **blocco**, questi non possono continuare finché tutti i thread (del blocco) hanno chiamato `__syncthreads()` (è come una **barriera** di MPI)
- Se non ci fosse `__syncthreads()` ci sarebbe una condizione di **race condition**. Il thread 0 che deve sommare tutti gli ingressi del vettore *cache* non sa se questi ingressi sono già stati riempiti dagli altri thread o sono ancora inizializzati a zero!
- **Attenzione** `__syncthreads()` è equivalente alle **collective calls** di MPI se solo una parte dei thread del blocco lo esegue il sistema si ferma, si va incontro ad un **deadlock!**

## Prodotto Scalare: un ingresso per thread

Consideriamo il prodotto scalare (spesso chiamato *dot product* dagli amici anglofoni) tra due vettori:

$$c = \mathbf{x} \cdot \mathbf{y} = (x_1, x_2, x_3, x_4) \cdot (y_1, y_2, y_3, y_4) = x_1y_1 + x_2y_2 + x_3y_3 + x_4y_4$$

- Creiamo un array *shared*, con  $n$  componenti, (chiamato `cache`) uno per ogni ingresso dell'array (tutti i thread di un blocco possono accedervi)
- facciamo sì che il thread  $n$ -esimo calcoli il prodotto delle  $n$ -esime componenti  $x_n y_n$ . Questo valore va inserito poi nella  $n$ -esima componente dell'array *shared*.
- Per ottenere il valore del prodotto scalare, dobbiamo poi sommare tutti questi valori.
- Proviamo a scrivere una implementazione **semplice** di un kernel dove un singolo thread esegue la somma di tutti gli ingressi del vettore *shared*:

```
__global__ void dot( int *a, int *b, int *c ) {
    __shared__ int cache[N]; // <= questo viene messo nella memoria shared
    cache[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x]; // ogni th riempie un ingresso
    __syncthreads(); // aspettiamo che tutti gli ingressi siano riempiti
    if( 0 == threadIdx.x ) { // facciamo fare la somma totale al thread 0
        int sum = 0;
        for( int i = 0; i < N; i++ ) sum += cache[i];
        *c = sum; // dereferenzio il risultato
    }
}
```

- **Peccato** in questo caso il processo di riduzione non è parallelo (un solo thread lavora!)
  - la funzione `__syncthreads()` **sincronizza** i thread di un blocco, questi non possono continuare finché tutti i thread (del blocco) hanno chiamato `__syncthreads()` (è come una barriera di MPI)
  - Se non ci fosse `__syncthreads()` ci sarebbe una condizione di **race condition**. Il thread 0 che deve sommare tutti gli ingressi del vettore *cache* non sa se questi ingressi sono già stati riempiti dagli altri thread o sono ancora inizializzati a zero!
  - **Attenzione** `__syncthreads()` è equivalente alle *collective calls* di MPI se solo una parte dei thread del blocco lo esegue il sistema si ferma, si va incontro ad un deadlock!

## Prodotto Scalare: un ingresso per thread

Consideriamo il prodotto scalare (spesso chiamato *dot product* dagli amici anglofoni) tra due vettori:

$$c = \mathbf{x} \cdot \mathbf{y} = (x_1, x_2, x_3, x_4) \cdot (y_1, y_2, y_3, y_4) = x_1y_1 + x_2y_2 + x_3y_3 + x_4y_4$$

- Creiamo un array *shared*, con  $n$  componenti, (chiamato `cache`) uno per ogni ingresso dell'array (tutti i thread di un blocco possono accedervi)
- facciamo sì che il thread  $n$ -esimo calcoli il prodotto delle  $n$ -esime componenti  $x_n y_n$ . Questo valore va inserito poi nella  $n$ -esima componente dell'array *shared*.
- Per ottenere il valore del prodotto scalare, dobbiamo poi sommare tutti questi valori.
- Proviamo a scrivere una implementazione **semplice** di un kernel dove un singolo thread esegue la somma di tutti gli ingressi del vettore *shared*:

```
__global__ void dot( int *a, int *b, int *c ) {
    __shared__ int cache[N]; // <= questo viene messo nella memoria shared
    cache[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x]; // ogni th riempie un ingresso
    __syncthreads(); // aspettiamo che tutti gli ingressi siano riempiti
    if( 0 == threadIdx.x ) { // facciamo fare la somma totale al thread 0
        int sum = 0;
        for( int i = 0; i < N; i++ ) sum += cache[i];
        *c = sum; // dereferenzio il risultato
    }
}
```

- **Peccato** in questo caso il processo di riduzione non è parallelo (un solo thread lavora!)
- la funzione `__syncthreads()` **sincronizza** i thread di un **blocco**, questi non possono continuare finché tutti i thread (del blocco) hanno chiamato `__syncthreads()` (è come una **barriera** di MPI)
- Se non ci fosse `__syncthreads()` ci sarebbe una condizione di **race condition**. Il thread 0 che deve sommare tutti gli ingressi del vettore *cache* non sa se questi ingressi sono già stati riempiti dagli altri thread o sono ancora inizializzati a zero!
- **Attenzione** `__syncthreads()` è equivalente alle **collective calls** di MPI se solo una parte dei thread del blocco lo esegue il sistema si ferma, si va incontro ad un **deadlock**!

## Prodotto Scalare: un ingresso per thread

Consideriamo il prodotto scalare (spesso chiamato *dot product* dagli amici anglofoni) tra due vettori:

$$\mathbf{c} = \mathbf{x} \cdot \mathbf{y} = (x_1, x_2, x_3, x_4) \cdot (y_1, y_2, y_3, y_4) = x_1y_1 + x_2y_2 + x_3y_3 + x_4y_4$$

- Creiamo un array *shared*, con  $n$  componenti, (chiamato `cache`) uno per ogni ingresso dell'array (tutti i thread di un blocco possono accedervi)
- facciamo sì che il thread  $n$ -esimo calcoli il prodotto delle  $n$ -esime componenti  $x_n y_n$ . Questo valore va inserito poi nella  $n$ -esima componente dell'array *shared*.
- Per ottenere il valore del prodotto scalare, dobbiamo poi sommare tutti questi valori.
- Proviamo a scrivere una implementazione **semplice** di un kernel dove un singolo thread esegue la somma di tutti gli ingressi del vettore *shared*:

```
__global__ void dot( int *a, int *b, int *c ) {
    __shared__ int cache[N]; // <= questo viene messo nella memoria shared
    cache[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x]; // ogni th riempie un ingresso
    __syncthreads(); // aspettiamo che tutti gli ingressi siano riempiti
    if( 0 == threadIdx.x ) { // facciamo fare la somma totale al thread 0
        int sum = 0;
        for( int i = 0; i < N; i++ ) sum += cache[i];
        *c = sum; // dereferenzio il risultato
    }
}
```

- **Peccato** in questo caso il processo di riduzione non è parallelo (un solo thread lavora!)
- la funzione `__syncthreads()` **sincronizza** i thread di un **blocco**, questi non possono continuare finché tutti i thread (del blocco) hanno chiamato `__syncthreads()` (è come una **barriera** di MPI)
- Se non ci fosse `__syncthreads()` ci sarebbe una condizione di **race condition**. Il thread 0 che deve sommare tutti gli ingressi del vettore *cache* non sa se questi ingressi sono già stati riempiti dagli altri thread o sono ancora inizializzati a zero!
- **Attenzione** `__syncthreads()` è equivalente alle **collective calls** di MPI se solo una parte dei thread del blocco lo esegue il sistema si ferma, si va incontro ad un **deadlock**!

## Prodotto Scalare: un ingresso per thread

Consideriamo il prodotto scalare (spesso chiamato *dot product* dagli amici anglofoni) tra due vettori:

$$\mathbf{c} = \mathbf{x} \cdot \mathbf{y} = (x_1, x_2, x_3, x_4) \cdot (y_1, y_2, y_3, y_4) = x_1y_1 + x_2y_2 + x_3y_3 + x_4y_4$$

- Creiamo un array *shared*, con  $n$  componenti, (chiamato `cache`) uno per ogni ingresso dell'array (tutti i thread di un blocco possono accedervi)
- facciamo sì che il thread  $n$ -esimo calcoli il prodotto delle  $n$ -esime componenti  $x_n y_n$ . Questo valore va inserito poi nella  $n$ -esima componente dell'array *shared*.
- Per ottenere il valore del prodotto scalare, dobbiamo poi sommare tutti questi valori.
- Proviamo a scrivere una implementazione **semplice** di un kernel dove un singolo thread esegue la somma di tutti gli ingressi del vettore *shared*:

```
__global__ void dot( int *a, int *b, int *c ) {
    __shared__ int cache[N]; // <= questo viene messo nella memoria shared
    cache[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x]; // ogni th riempie un ingresso
    __syncthreads(); // aspettiamo che tutti gli ingressi siano riempiti
    if( 0 == threadIdx.x ) { // facciamo fare la somma totale al thread 0
        int sum = 0;
        for( int i = 0; i < N; i++ ) sum += cache[i];
        *c = sum; // dereferenzio il risultato
    }
}
```

- **Peccato** in questo caso il processo di riduzione non è parallelo (un solo thread lavora!)
- la funzione `__syncthreads()` **sincronizza** i thread di un **blocco**, questi non possono continuare finché tutti i thread (del blocco) hanno chiamato `__syncthreads()` (è come una **barriera** di MPI)
- Se non ci fosse `__syncthreads()` ci sarebbe una condizione di **race condition**. Il thread 0 che deve sommare tutti gli ingressi del vettore *cache* non **sa** se questi ingressi sono già stati riempiti dagli altri thread o sono ancora inizializzati a zero!
- **Attenzione** `__syncthreads()` è equivalente alle **collective** calls di MPI se solo una parte dei thread del blocco lo esegue il sistema si ferma, si va incontro ad un **deadlock!**



# Prodotto scalare (multiblocco)

## Esercizio:

- Proviamo ad implementare un prodotto scalare, utilizzando blocchi e thread. Supponiamo inoltre che il numero di ingressi dell'array sia maggiore del numero totale di thread (quindi ogni thread, in generale, lavorerà con più di un ingresso).
- usiamo la memoria shared

```
#define N 1048576 // dimensione totale degli array da moltiplicare
#define threadsPerBlock 256 // dimensione del blocco
__global__ void dot( float *a, float *b, float *c ) { // kernel
    __shared__ float cache[threadsPerBlock]; // array shared
    int tid = threadIdx.x + blockIdx.x * blockDim.x; // indice thread
    int cacheIndex = threadIdx.x; // //
    float temp = 0; // valore temp =0
    while (tid < N) {
        temp += a[tid] * b[tid]; // //
        tid += blockDim.x * gridDim.x; // //
    }
    cache[cacheIndex] = temp; // inserisci i valori nell'array shared
    // cosa manca qui ? vedi diapositiva successiva
}
```

- ha senso fare notare che lo stesso thread fa più di un prodotto.
- nell'array chiamato *cache* (alloccato nella memoria *shared*) si mette la somma che viene calcolata da ogni thread.
- c'è un array *cache* per ogni blocco! (ognuno con valori differenti)
- la dimensione dell'array *shared* viene definita come *threadsPerBlock* (in questo modo l'array ha un ingresso per ognuno dei thread nel blocco, che sono appunto quelli che possono comunicare tra loro).
- se ci sono dei thread che non lavorano, la variabile *temp* viene azzerata per evitare errori.

## Prodotto scalare (multiblocco)

### Esercizio:

- Proviamo ad implementare un prodotto scalare, utilizzando blocchi e thread. Supponiamo inoltre che il numero di ingressi dell'array sia maggiore del numero totale di thread (quindi ogni thread, in generale, lavorerà con più di un ingresso).
- usiamo la memoria shared

```
#define N 1048576           // dimensione totale degli array da moltiplicare
#define threadsPerBlock 256 // dimensione del blocco
__global__ void dot( float *a, float *b, float *c ) { // kernel
    __shared__ float cache[threadsPerBlock];        // array shared
    int tid = threadIdx.x + blockIdx.x * blockDim.x; // indice thread
    int cacheIndex = threadIdx.x;                  // //
    float temp = 0;                                // valore temp =0
    while (tid < N) {
        temp += a[tid] * b[tid];                    // //
        tid += blockDim.x * gridDim.x;              // //
    }
    cache[cacheIndex] = temp; // inserisci i valori nell'array shared
    // cosa manca qui ? vedi diapositiva successiva
}
```

- ha senso fare notare che lo stesso **thread** fa più di un prodotto.
- nell'array chiamato *cache* (allocato nella memoria *shared*) si mette la somma che viene calcolata da ogni thread.
- c'è un array *cache* per ogni blocco! (ognuno con valori differenti)
- la dimensione dell'array *shared* viene definita come *threadsPerBlock* (in questo modo l'array ha un ingresso per ognuno dei thread nel blocco, che sono appunto quelli che possono comunicare tra loro).
- se ci sono dei thread che non lavorano, la variabile *temp* viene azzerata per evitare errori.

## Prodotto scalare (multiblocco)

### Esercizio:

- Proviamo ad implementare un prodotto scalare, utilizzando blocchi e thread. Supponiamo inoltre che il numero di ingressi dell'array sia maggiore del numero totale di thread (quindi ogni thread, in generale, lavorerà con più di un ingresso).
- usiamo la memoria shared

```
#define N 1048576           // dimensione totale degli array da moltiplicare
#define threadsPerBlock 256 // dimensione del blocco
__global__ void dot( float *a, float *b, float *c ) { // kernel
    __shared__ float cache[threadsPerBlock];        // array shared
    int tid = threadIdx.x + blockIdx.x * blockDim.x; // indice thread
    int cacheIndex = threadIdx.x;                  // //
    float temp = 0;                                // valore temp =0
    while (tid < N) {
        temp += a[tid] * b[tid];                    // //
        tid += blockDim.x * gridDim.x;              // //
    }
    cache[cacheIndex] = temp; // inserisci i valori nell'array shared
    // cosa manca qui ? vedi diapositiva successiva
}
```

- ha senso fare notare che lo stesso **thread** fa più di un prodotto.
- nell'array chiamato `cache` (allocato nella memoria `shared`) si mette la somma che viene calcolata da ogni thread.
- c'è un array `cache` per ogni blocco! (ognuno con valori differenti)
- la dimensione dell'array `shared` viene definita come `threadsPerBlock` (in questo modo l'array ha un ingresso per ognuno dei thread nel blocco, che sono appunto quelli che possono comunicare tra loro).
- se ci sono dei thread che non lavorano, la variabile `temp` viene azzerata per evitare errori.

## Prodotto scalare (multiblocco)

### Esercizio:

- Proviamo ad implementare un prodotto scalare, utilizzando blocchi e thread. Supponiamo inoltre che il numero di ingressi dell'array sia maggiore del numero totale di thread (quindi ogni thread, in generale, lavorerà con più di un ingresso).
- usiamo la memoria shared

```
#define N 1048576           // dimensione totale degli array da moltiplicare
#define threadsPerBlock 256 // dimensione del blocco
__global__ void dot( float *a, float *b, float *c ) { // kernel
    __shared__ float cache[threadsPerBlock]; // array shared
    int tid = threadIdx.x + blockIdx.x * blockDim.x; // indice thread
    int cacheIndex = threadIdx.x; // //
    float temp = 0; // valore temp =0
    while (tid < N) {
        temp += a[tid] * b[tid]; // //
        tid += blockDim.x * gridDim.x; // //
    }
    cache[cacheIndex] = temp; // inserisci i valori nell'array shared
    // cosa manca qui ? vedi diapositiva successiva
}
```

- ha senso fare notare che lo stesso **thread** fa più di un prodotto.
- nell'array chiamato `cache` (allocato nella memoria `shared`) si mette la somma che viene calcolata da ogni thread.
- c'è un array `cache` per ogni blocco! (ognuno con valori differenti)
- la dimensione dell'array `shared` viene definita come `threadsPerBlock` (in questo modo l'array ha un ingresso per ognuno dei thread nel blocco, che sono appunto quelli che possono comunicare tra loro).
- se ci sono dei thread che non lavorano, la variabile `temp` viene azzerata per evitare errori.

## Prodotto scalare (multiblocco)

### Esercizio:

- Proviamo ad implementare un prodotto scalare, utilizzando blocchi e thread. Supponiamo inoltre che il numero di ingressi dell'array sia maggiore del numero totale di thread (quindi ogni thread, in generale, lavorerà con più di un ingresso).
- usiamo la memoria shared

```
#define N 1048576           // dimensione totale degli array da moltiplicare
#define threadsPerBlock 256 // dimensione del blocco
__global__ void dot( float *a, float *b, float *c ) { // kernel
    __shared__ float cache[threadsPerBlock];        // array shared
    int tid = threadIdx.x + blockIdx.x * blockDim.x; // indice thread
    int cacheIndex = threadIdx.x;                  // //
    float temp = 0;                                // valore temp =0
    while (tid < N) {
        temp += a[tid] * b[tid];                    // //
        tid += blockDim.x * gridDim.x;              // //
    }
    cache[cacheIndex] = temp; // inserisci i valori nell'array shared
    // cosa manca qui ? vedi diapositiva successiva
}
```

- ha senso fare notare che lo stesso **thread** fa più di un prodotto.
- nell'array chiamato `cache` (allocato nella memoria `shared`) si mette la somma che viene calcolata da ogni thread.
- c'è un array `cache` per ogni blocco! (ognuno con valori differenti)
- la dimensione dell'array `shared` viene definita come `threadsPerBlock` (in questo modo l'array ha un ingresso per ognuno dei thread nel blocco, che sono appunto quelli che possono comunicare tra loro).
- se ci sono dei thread che non lavorano, la variabile `temp` viene azzerata per evitare errori.

## Prodotto scalare (multiblocco)

### Esercizio:

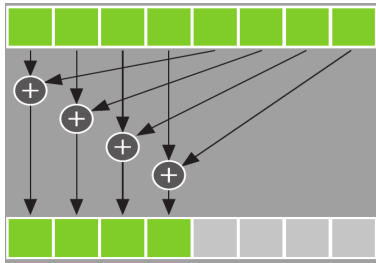
- Proviamo ad implementare un prodotto scalare, utilizzando blocchi e thread. Supponiamo inoltre che il numero di ingressi dell'array sia maggiore del numero totale di thread (quindi ogni thread, in generale, lavorerà con più di un ingresso).
- usiamo la memoria shared

```
#define N 1048576           // dimensione totale degli array da moltiplicare
#define threadsPerBlock 256 // dimensione del blocco
__global__ void dot( float *a, float *b, float *c ) { // kernel
    __shared__ float cache[threadsPerBlock];        // array shared
    int tid = threadIdx.x + blockIdx.x * blockDim.x; // indice thread
    int cacheIndex = threadIdx.x;                  // //
    float temp = 0;                                // valore temp =0
    while (tid < N) {
        temp += a[tid] * b[tid];                    // //
        tid += blockDim.x * gridDim.x;              // //
    }
    cache[cacheIndex] = temp; // inserisci i valori nell'array shared
    // cosa manca qui ? vedi diapositiva successiva
}
```

- ha senso fare notare che lo stesso **thread** fa più di un prodotto.
- nell'array chiamato `cache` (allocato nella memoria `shared`) si mette la somma che viene calcolata da ogni thread.
- c'è un array `cache` per ogni blocco! (ognuno con valori differenti)
- la dimensione dell'array `shared` viene definita come `threadsPerBlock` (in questo modo l'array ha un ingresso per ognuno dei thread nel blocco, che sono appunto quelli che possono comunicare tra loro).
- se ci sono dei thread che non lavorano, la variabile `temp` viene azzerata per evitare errori.

# Operazioni di Riduzione

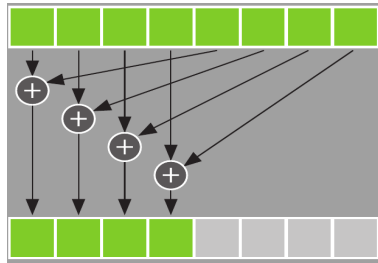
Proviamo a pensare ad un algoritmo che sfrutti piu' di un thread per operare una riduzione di una somma:



```
int i = blockDim.x/2;
while (i != 0) {
    if (cacheIndex < i) cache[cacheIndex] += cache[cacheIndex + i]; // gira sull'array shared
    __syncthreads(); // vogliamo che al prossimo giro tutti i thread abbiano lavorato
    i /= 2;
}
```

# Operazioni di Riduzione

Proviamo a pensare ad un algoritmo che sfrutti piu' di un thread per operare una riduzione di una somma:

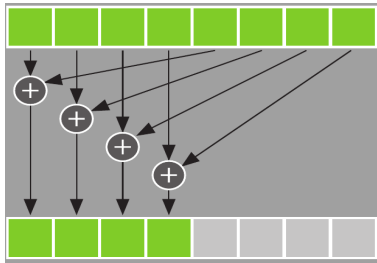


```
int i = blockDim.x/2;
while (i != 0) {
    if (cacheIndex < i) cache[cacheIndex] += cache[cacheIndex + i]; // gira sull'array shared
    __syncthreads(); // vogliamo che al prossimo giro tutti i thread abbiano lavorato
    i /= 2;
}
```



# Operazioni di Riduzione

Proviamo a pensare ad un algoritmo che sfrutti piu' di un thread per operare una riduzione di una somma:



```
int i = blockDim.x/2;
while (i != 0) {
    if (cacheIndex < i) cache[cacheIndex] += cache[cacheIndex + i]; // gira sull'array shared
    __syncthreads(); // vogliamo che al prossimo giro tutti i thread abbiano lavorato
    i /= 2;
}
```

# Inversione con shared memory: l'host

Voglio usare CUDA per invertire un array: l'ultimo elemento deve diventare il primo, ...

```
int main(void)
{
    const int n = 64;
    int a[n], r[n], d[n];

    for (int i = 0; i < n; i++) {
        a[i] = i;                // vettore da riordinare
        r[i] = n-i-1;           // vettore già riordinato nell' HOST per controllo
        d[i] = 0;               // vettore che contiene il risultato
    }

    int *device_d;              // vettore che deve essere usato nel device
    cudaMalloc(&device_d, n * sizeof(int));

    cudaMemcpy(device_d, a, n*sizeof(int), cudaMemcpyHostToDevice);
    staticReverse<<<1,n>>>(device_d, n);
    cudaMemcpy(d, device_d, n*sizeof(int), cudaMemcpyDeviceToHost);

    for (int i = 0; i < n; i++)
        if (d[i] != r[i]) printf("Error: d[%d]!=r[%d] (%d, %d)n", i, i, d[i], r[i]);
}
```

# Kernel dell'inversione con uso della Shared Memory

<https://devblogs.nvidia.com/using-shared-memory-cuda-cc/>

- Cerchiamo di scrivere un kernel che "inverte" un array (l'elemento 1 va nella posizione  $n$ , l'elemento 2 nella  $n-1$ , ...)
- usiamo la memoria *shared*
- supponiamo di voler lanciare 1 blocco con  $n=64$  thread (per semplicità)
- definiamo l'array con il decoratore `__shared__`
- creiamo un indice invertito (in modo da scambiare la posizione degli elementi)

```
__global__ void staticReverse(int *d, int n)
{
    __shared__ int s[64];           // definisco un array statico nella memoria shared
    int t = threadIdx.x;           // definisco un indice del thread che giri sugli indici dell'array
    int tr = n-t-1;                // creo un indice invertito: 1->N, 2->N-1, ...
    s[t] = d[t];                   // copio l'array dalla Global alla Shared
    __syncthreads();              // sincronizzo
    d[t] = s[tr];                  // costruisco il vettore invertito
}
```

# Kernel dell'inversione con uso della Shared Memory

<https://devblogs.nvidia.com/using-shared-memory-cuda-cc/>

- Cerchiamo di scrivere un kernel che "inverte" un array (l'elemento 1 va nella posizione  $n$ , l'elemento 2 nella  $n-1$ , ...)
- usiamo la memoria *shared*
- supponiamo di voler lanciare 1 blocco con  $n=64$  thread (per semplicità')
- definiamo l'array con il decoratore `__shared__`
- creiamo un indice invertito (in modo da scambiare la posizione degli elementi)

```
__global__ void staticReverse(int *d, int n)
{
    __shared__ int s[64];           // definisco un array statico nella memoria shared
    int t = threadIdx.x;           // definisco un indice del thread che giri sugli indici dell'array
    int tr = n-t-1;                // creo un indice invertito: 1->N, 2->N-1, ...
    s[t] = d[t];                   // copio l'array dalla Global alla Shared
    __syncthreads();               // sincronizzo
    d[t] = s[tr];                  // costruisco il vettore invertito
}
```

# Kernel dell'inversione con uso della Shared Memory

<https://devblogs.nvidia.com/using-shared-memory-cuda-cc/>

- Cerchiamo di scrivere un kernel che "inverte" un array (l'elemento 1 va nella posizione  $n$ , l'elemento 2 nella  $n-1$ , ...)
- usiamo la memoria *shared*
- supponiamo di voler lanciare 1 blocco con  $n=64$  thread (per semplicità')
- definiamo l'array con il decoratore `__shared__`
- creiamo un indice invertito (in modo da scambiare la posizione degli elementi)

```

__global__ void staticReverse(int *d, int n)
{
    __shared__ int s[64];           // definisco un array statico nella memoria shared
    int t = threadIdx.x;           // definisco un indice del thread che giri sugli indici dell'array
    int tr = n-t-1;                // creo un indice invertito: 1->N, 2->N-1, ...
    s[t] = d[t];                   // copio l'array dalla Global alla Shared
    __syncthreads();               // sincronizzo
    d[t] = s[tr];                  // costruisco il vettore invertito
}

```

# Kernel dell'inversione con uso della Shared Memory

<https://devblogs.nvidia.com/using-shared-memory-cuda-cc/>

- Cerchiamo di scrivere un kernel che "inverte" un array (l'elemento 1 va nella posizione  $n$ , l'elemento 2 nella  $n-1$ , ...)
- usiamo la memoria *shared*
- supponiamo di voler lanciare 1 blocco con  $n=64$  thread (per semplicità')
- definiamo l'array con il decoratore `__shared__`
- creiamo un indice invertito (in modo da scambiare la posizione degli elementi)

```
__global__ void staticReverse(int *d, int n)
{
    __shared__ int s[64];           // definisco un array statico nella memoria shared
    int t = threadIdx.x;           // definisco un indice del thread che giri sugli indici dell'array
    int tr = n-t-1;                // creo un indice invertito: 1->N, 2->N-1, ...
    s[t] = d[t];                   // copio l'array dalla Global alla Shared
    __syncthreads();               // sincronizzo
    d[t] = s[tr];                  // costruisco il vettore invertito
}
```

## Kernel dell'inversione con uso della Shared Memory

<https://devblogs.nvidia.com/using-shared-memory-cuda-cc/>

- Cerchiamo di scrivere un kernel che "inverte" un array (l'elemento 1 va nella posizione  $n$ , l'elemento 2 nella  $n-1$ , ...)
- usiamo la memoria *shared*
- supponiamo di voler lanciare 1 blocco con  $n=64$  thread (per semplicità')
- definiamo l'array con il decoratore `__shared__`
- creiamo un indice invertito (in modo da scambiare la posizione degli elementi)

```
__global__ void staticReverse(int *d, int n)
{
    __shared__ int s[64];           // definisco un array statico nella memoria shared
    int t = threadIdx.x;           // definisco un indice del thread che giri sugli indici dell'array
    int tr = n-t-1;                // creo un indice invertito: 1->N, 2->N-1, ...
    s[t] = d[t];                   // copio l'array dalla Global alla Shared
    __syncthreads();              // sincronizzo
    d[t] = s[tr];                  // costruisco il vettore invertito
}
```

# Kernel dell'inversione con uso della Shared Memory

<https://devblogs.nvidia.com/using-shared-memory-cuda-cc/>

- Cerchiamo di scrivere un kernel che "inverte" un array (l'elemento 1 va nella posizione  $n$ , l'elemento 2 nella  $n-1$ , ...)
- usiamo la memoria *shared*
- supponiamo di voler lanciare 1 blocco con  $n=64$  thread (per semplicità')
- definiamo l'array con il decoratore `__shared__`
- creiamo un indice invertito (in modo da scambiare la posizione degli elementi)

```
__global__ void staticReverse(int *d, int n)
{
    __shared__ int s[64];           // definisco un array statico nella memoria shared
    int t = threadIdx.x;           // definisco un indice del thread che giri sugli indici dell'array
    int tr = n-t-1;                // creo un indice invertito: 1->N, 2->N-1, ...
    s[t] = d[t];                   // copio l'array dalla Global alla Shared
    __syncthreads();              // sincronizzo
    d[t] = s[tr];                  // costruisco il vettore invertito
}
```



# Kernel dell'inversione con uso della Shared Memory

<https://devblogs.nvidia.com/using-shared-memory-cuda-cc/>

- Cerchiamo di scrivere un kernel che "inverte" un array (l'elemento 1 va nella posizione  $n$ , l'elemento 2 nella  $n-1$ , ...)
- usiamo la memoria *shared*
- supponiamo di voler lanciare 1 blocco con  $n=64$  thread (per semplicità')
- definiamo l'array con il decoratore `__shared__`
- creiamo un indice invertito (in modo da scambiare la posizione degli elementi)

```
__global__ void staticReverse(int *d, int n)
{
    __shared__ int s[64];           // definisco un array statico nella memoria shared
    int t = threadIdx.x;           // definisco un indice del thread che giri sugli indici dell'array
    int tr = n-t-1;                // creo un indice invertito: 1->N, 2->N-1, ...
    s[t] = d[t];                   // copio l'array dalla Global alla Shared
    __syncthreads();               // sincronizzo
    d[t] = s[tr];                  // costruisco il vettore invertito
}
```

## Memoria dinamica shared

Fino ad ora abbiamo visto un uso della memoria *shared* in cui la dimensione dell'array assegnato e' nota al momento della scrittura.

E' pero' possibile allocare un array della memoria shared, proviamo a vedere come si trasforma il kernel precedente, nel caso di memoria allocata dinamicamente:

```
__global__ void dynamicReverse(int *d, int n)
{
    extern __shared__ int s[]; // differences: "extern" and s[]
    int t = threadIdx.x;
    int tr = n-t-1;
    s[t] = d[t];
    __syncthreads(); // syncs all the threads of a BLOCK
    d[t] = s[tr];
}
```

Ok ma chiaramente manca qualcosa... dove e' indicata la dimensione dell'array *shared*?

## Memoria dinamica shared

Fino ad ora abbiamo visto un uso della memoria *shared* in cui la dimensione dell'array assegnato e' nota al momento della scrittura.

E' pero' possibile allocare un array della memoria shared, proviamo a vedere come si trasforma il kernel precedente, nel caso di memoria allocata dinamicamente:

```
__global__ void dynamicReverse(int *d, int n)
{
  extern __shared__ int s[]; // differences: "extern" and s[]
  int t = threadIdx.x;
  int tr = n-t-1;
  s[t] = d[t];
  __syncthreads(); // syncs all the threads of a BLOCK
  d[t] = s[tr];
}
```

Ok ma chiaramente manca qualcosa... dove e' indicata la dimensione dell'array *shared*?

# Memoria dinamica shared

Fino ad ora abbiamo visto un uso della memoria *shared* in cui la dimensione dell'array assegnato e' nota al momento della scrittura.

E' pero' possibile allocare un array della memoria shared, proviamo a vedere come si trasforma il kernel precedente, nel caso di memoria allocata dinamicamente:

```
__global__ void dynamicReverse(int *d, int n)
{
    extern __shared__ int s[]; // differences: "extern" and s[]
    int t = threadIdx.x;
    int tr = n-t-1;
    s[t] = d[t];
    __syncthreads(); // syncs all the threads of a BLOCK
    d[t] = s[tr];
}
```

Ok ma chiaramente manca qualcosa... dove e' indicata la dimensione dell'array *shared*?

# Main della memoria shared dinamica

```
int main(void)
{
    const int n = 64;
    int a[n], r[n], d[n];

    for (int i = 0; i < n; i++) {
        a[i] = i;
        r[i] = n-i-1;
        d[i] = 0;
    }
    int *device_d;
    cudaMalloc(&device_d, n * sizeof(int));

    cudaMemcpy(device_d, a, n*sizeof(int), cudaMemcpyHostToDevice);

    dynamicReverse<<<1,n,n*sizeof(int)>>>(device_d, n); // <==== parametro aggiuntivo

    cudaMemcpy(d, device_d, n * sizeof(int), cudaMemcpyDeviceToHost);
    for (int i = 0; i < n; i++)
        if (d[i] != r[i]) printf("Error: d[%d]!=r[%d] (%d, %d)n", i, i, d[i], r[i]);
}
```

- Va inserito un parametro aggiuntivo, come terzo ingresso delle **triple angle brackets**
- questo parametro indica la dimensione in **byte** dell'array assegnato alla memoria *shared*

# Main della memoria shared dinamica

```

int main(void)
{
    const int n = 64;
    int a[n], r[n], d[n];

    for (int i = 0; i < n; i++) {
        a[i] = i;
        r[i] = n-i-1;
        d[i] = 0;
    }
    int *device_d;
    cudaMalloc(&device_d, n * sizeof(int));

    cudaMemcpy(device_d, a, n*sizeof(int), cudaMemcpyHostToDevice);

    dynamicReverse<<<1,n,n*sizeof(int)>>>(device_d, n); // <==== parametro aggiuntivo

    cudaMemcpy(d, device_d, n * sizeof(int), cudaMemcpyDeviceToHost);
    for (int i = 0; i < n; i++)
        if (d[i] != r[i]) printf("Error: d[%d]!=r[%d] (%d, %d)n", i, i, d[i], r[i]);
}

```

- Va inserito un parametro aggiuntivo, come terzo ingresso delle **triple angle brackets**
- questo parametro indica la dimensione in **byte** dell'array assegnato alla memoria *shared*

# Main della memoria shared dinamica

```

int main(void)
{
    const int n = 64;
    int a[n], r[n], d[n];

    for (int i = 0; i < n; i++) {
        a[i] = i;
        r[i] = n-i-1;
        d[i] = 0;
    }
    int *device_d;
    cudaMalloc(&device_d, n * sizeof(int));

    cudaMemcpy(device_d, a, n*sizeof(int), cudaMemcpyHostToDevice);

    dynamicReverse<<<1,n,n*sizeof(int)>>>(device_d, n); // <==== parametro aggiuntivo

    cudaMemcpy(d, device_d, n * sizeof(int), cudaMemcpyDeviceToHost);
    for (int i = 0; i < n; i++)
        if (d[i] != r[i]) printf("Error: d[%d]!=r[%d] (%d, %d)n", i, i, d[i], r[i]);
}

```

- Va inserito un parametro aggiuntivo, come terzo ingresso delle **triple angle brackets**
- questo parametro indica la dimensione in **byte** dell'array assegnato alla memoria *shared*

# Main della memoria shared dinamica

```

int main(void)
{
    const int n = 64;
    int a[n], r[n], d[n];

    for (int i = 0; i < n; i++) {
        a[i] = i;
        r[i] = n-i-1;
        d[i] = 0;
    }
    int *device_d;
    cudaMalloc(&device_d, n * sizeof(int));

    cudaMemcpy(device_d, a, n*sizeof(int), cudaMemcpyHostToDevice);

    dynamicReverse<<<1,n,n*sizeof(int)>>>(device_d, n); // <==== parametro aggiuntivo

    cudaMemcpy(d, device_d, n * sizeof(int), cudaMemcpyDeviceToHost);
    for (int i = 0; i < n; i++)
        if (d[i] != r[i]) printf("Error: d[%d]!=r[%d] (%d, %d)n", i, i, d[i], r[i]);
}

```

- Va inserito un parametro aggiuntivo, come terzo ingresso delle **triple angle brackets**
- questo parametro indica la dimensione in **byte** dell'array assegnato alla memoria *shared*



## Riassunto dell'utilizzo della Shared dinamica:

- all'interno del Kernel, davanti al decoratore `__shared__` si usa il comando `extern` (questo assicura che l'oggetto sia solo **dichiarato**):

```
extern __shared__ int s[]; // inoltre non si mette quanto e' grande l'array
```

- dove si lancia il kernel (dall'host), si passa un **terzo** parametro alle triple parentesi angolari: la **dimensione in byte** associata all'array shared:

```
dynamicReverse<<<1,n,n*sizeof(int)>>>(device_d, n); // <----- parametro aggiuntivo
```

- a questo punto il compilatore CUDA assegna un'area di memoria **shared**, puntata dal puntatore `s` (dichiarato all'interno del kernel). La memoria shared e' ora **definita**

## Riassunto dell'utilizzo della Shared dinamica:

- all'interno del Kernel, davanti al decoratore `__shared__` si usa il comando `extern` (questo assicura che l'oggetto sia solo **dichiarato**):

```
extern __shared__ int s[]; // inoltre non si mette quanto e' grande l'array
```

- dove si lancia il kernel (dall'host), si passa un **terzo** parametro alle triple parentesi angolari: la **dimensione in byte** associata all'array shared:

```
dynamicReverse<<<1,n,n*sizeof(int)>>>(device_d, n); // <==== parametro aggiuntivo
```

- a questo punto il compilatore CUDA assegna un'area di memoria `shared`, puntata dal puntatore `s` (dichiarato all'interno del kernel). La memoria `shared` e' ora **definita**

## Riassunto dell'utilizzo della Shared dinamica:

- all'interno del Kernel, davanti al decoratore `__shared__` si usa il comando `extern` (questo assicura che l'oggetto sia solo **dichiarato**):

```
extern __shared__ int s[]; // inoltre non si mette quanto e' grande l'array
```

- dove si lancia il kernel (dall'host), si passa un **terzo** parametro alle triple parentesi angolari: la **dimensione in byte** associata all'array shared:

```
dynamicReverse<<<1, n, n*sizeof(int)>>>(device_d, n); // <==== parametro aggiuntivo
```

- a questo punto il compilatore CUDA assegna un'area di memoria **shared**, puntata dal puntatore `s` (dichiarato all'interno del kernel). La memoria shared e' ora **definita**

## Piu' di un array shared dinamico?

- Per la memoria shared allocata dinamicamente esiste **un solo spazio** disponibile. Detto in altri termini ci puo' essere un solo nome che viene dichiarato come `extern __shared__`, un caso come il seguente e' sbagliato!

```
extern __shared__ int *primoShared[]
extern __shared__ int *secondoShared[] // <= ERRORE! solo un array dinamico shared
```

- nelle triple angle brackets puo' e deve essere inserito **un'unico quantitativo** di byte da allocare alla memoria shared
- nel caso in cui si abbia bisogno di suddividere la memoria shared allocata dinamicamente si devono usare dei **trucchi**.
- per esempio creare altri puntatori all'interno del kernel, in modo che corrispondano agli array voluti.
- e' necessario **allinare** questi puntatori uno dietro l'altro, in modo da evitare di sprecare memoria shared, p. es:

```
__global__ void Kernel(int count_a, int count_b)
{
    extern __shared__ int *shared; // puntatore alla memoria shared
    int *a = &shared[0]; // "a" messo manualmente all'inizio dello spazio shared
    int *b = &shared[count_a]; // "b" messo manualmente alla fine di "a"
    ... // use array a and b
}
```

il kernel viene invocato nel main con:

```
sharedMemory = count_a*size(int) + count_b*size(int); // dimensione della mem shared
Kernel <<<numBlocks, threadsPerBlock, sharedMemory>>> (count_a, count_b);
```

## Piu' di un array shared dinamico?

- Per la memoria shared allocata dinamicamente esiste **un solo spazio** disponibile. Detto in altri termini ci puo' essere un solo nome che viene dichiarato come `extern __shared__`, un caso come il seguente e' sbagliato!

```
extern __shared__ int *primoShared[]
extern __shared__ int *secondoShared[] // <= ERRORE! solo un array dinamico shared
```

- nelle triple angle brackets puo' e deve essere inserito **un'unico quantitativo** di byte da allocare alla memoria shared
- nel caso in cui si abbia bisogno di suddividere la memoria shared allocata dinamicamente si devono usare dei trucchi.
- per esempio creare altri puntatori all'interno del kernel, in modo che corrispondano agli array voluti.
- e' necessario allineare questi puntatori uno dietro l'altro, in modo da evitare di sprecare memoria shared, p. es:

```
__global__ void Kernel(int count_a, int count_b)
{
    extern __shared__ int *shared; // puntatore alla memoria shared
    int *a = &shared[0]; // "a" messo manualmente all'inizio dello spazio shared
    int *b = &shared[count_a]; // "b" messo manualmente alla fine di "a"
    ... // use array a and b
}
```

il kernel viene invocato nel main con:

```
sharedMemory = count_a*size(int) + count_b*size(int); // dimensione della mem shared
Kernel <<<numBlocks, threadsPerBlock, sharedMemory>>> (count_a, count_b);
```

## Piu' di un array shared dinamico?

- Per la memoria shared allocata dinamicamente esiste **un solo spazio** disponibile. Detto in altri termini ci puo' essere un solo nome che viene dichiarato come `extern __shared__`, un caso come il seguente e' sbagliato!

```
extern __shared__ int *primoShared[]
extern __shared__ int *secondoShared[] // <= ERRORE! solo un array dinamico shared
```

- nelle triple angle brackets puo' e deve essere inserito **un'unico quantitativo** di byte da allocare alla memoria shared
- nel caso in cui si abbia bisogno di suddividere la memoria shared allocata dinamicamente si devono usare dei **trucchi**.
- per esempio creare altri puntatori all'interno del kernel, in modo che corrispondano agli array voluti.
- e' necessario **allinare** questi puntatori uno dietro l'altro, in modo da evitare di sprecare memoria shared, p. es:

```
__global__ void Kernel(int count_a, int count_b)
{
    extern __shared__ int *shared; // puntatore alla memoria shared
    int *a = &shared[0]; // "a" messo manualmente all'inizio dello spazio shared
    int *b = &shared[count_a]; // "b" messo manualmente alla fine di "a"
    ... // use array a and b
}
```

il kernel viene invocato nel main con:

```
sharedMemory = count_a*size(int) + count_b*size(int); // dimensione della mem shared
Kernel <<<numBlocks, threadsPerBlock, sharedMemory>>> (count_a, count_b);
```

## Piu' di un array shared dinamico?

- Per la memoria shared allocata dinamicamente esiste **un solo spazio** disponibile. Detto in altri termini ci puo' essere un solo nome che viene dichiarato come `extern __shared__`, un caso come il seguente e' sbagliato!

```
extern __shared__ int *primoShared[]
extern __shared__ int *secondoShared[] // <= ERRORE! solo un array dinamico shared
```

- nelle triple angle brackets puo' e deve essere inserito **un'unico quantitativo** di byte da allocare alla memoria shared
- nel caso in cui si abbia bisogno di suddividere la memoria shared allocata dinamicamente si devono usare dei **trucchi**.
- per esempio creare altri puntatori all'interno del kernel, in modo che corrispondano agli array voluti.
- e' necessario **allinare** questi puntatori uno dietro l'altro, in modo da evitare di sprecare memoria shared, p. es:

```
__global__ void Kernel(int count_a, int count_b)
{
    extern __shared__ int *shared; // puntatore alla memoria shared
    int *a = &shared[0]; // "a" messo manualmente all'inizio dello spazio shared
    int *b = &shared[count_a]; // "b" messo manualmente alla fine di "a"
    ... // use array a and b
}
```

il kernel viene invocato nel main con:

```
sharedMemory = count_a*size(int) + count_b*size(int); // dimensione della mem shared
Kernel <<<numBlocks, threadsPerBlock, sharedMemory>>> (count_a, count_b);
```

## Piu' di un array shared dinamico?

- Per la memoria shared allocata dinamicamente esiste **un solo spazio** disponibile. Detto in altri termini ci puo' essere un solo nome che viene dichiarato come `extern __shared__`, un caso come il seguente e' sbagliato!

```
extern __shared__ int *primoShared[]
extern __shared__ int *secondoShared[] // <= ERRORE! solo un array dinamico shared
```

- nelle triple angle brackets puo' e deve essere inserito **un'unico quantitativo** di byte da allocare alla memoria shared
- nel caso in cui si abbia bisogno di suddividere la memoria shared allocata dinamicamente si devono usare dei **trucchi**.
- per esempio creare altri puntatori all'interno del kernel, in modo che corrispondano agli array voluti.
- e' necessario **allinare** questi puntatori uno dietro l'altro, in modo da evitare di sprecare memoria shared, p. es:

```
__global__ void Kernel(int count_a, int count_b)
{
    extern __shared__ int *shared; // puntatore alla memoria shared
    int *a = &shared[0]; // "a" messo manualmente all'inizio dello spazio shared
    int *b = &shared[count_a]; // "b" messo manualmente alla fine di "a"
    ... // use array a and b
}
```

il kernel viene invocato nel main con:

```
sharedMemory = count_a*size(int) + count_b*size(int); // dimensione della mem shared
Kernel <<<numBlocks, threadsPerBlock, sharedMemory>>> (count_a, count_b);
```



## Piu' di un array shared dinamico?

- Per la memoria shared allocata dinamicamente esiste **un solo spazio** disponibile. Detto in altri termini ci puo' essere un solo nome che viene dichiarato come `extern __shared__`, un caso come il seguente e' sbagliato!

```
extern __shared__ int *primoShared[]
extern __shared__ int *secondoShared[] // <= ERRORE! solo un array dinamico shared
```

- nelle triple angle brackets puo' e deve essere inserito **un'unico quantitativo** di byte da allocare alla memoria shared
- nel caso in cui si abbia bisogno di suddividere la memoria shared allocata dinamicamente si devono usare dei **trucchi**.
- per esempio creare altri puntatori all'interno del kernel, in modo che corrispondano agli array voluti.
- e' necessario **allinare** questi puntatori uno dietro l'altro, in modo da evitare di sprecare memoria shared, p. es:

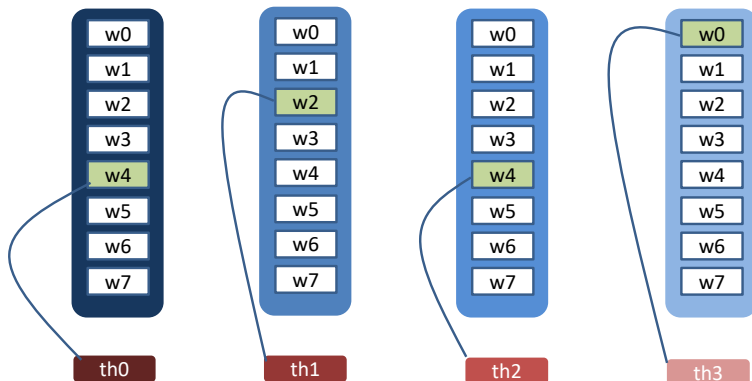
```
__global__ void Kernel(int count_a, int count_b)
{
  extern __shared__ int *shared; // puntatore alla memoria shared
  int *a = &shared[0]; // "a" messo manualmente all'inizio dello spazio shared
  int *b = &shared[count_a]; // "b" messo manualmente alla fine di "a"
  ... // use array a and b
}
```

il kernel viene invocato nel main con:

```
sharedMemory = count_a*size(int) + count_b*size(int); // dimensione della mem shared
Kernel <<<numBlocks, threadsPerBlock, sharedMemory>>> (count_a, count_b);
```

# Bank Conflict

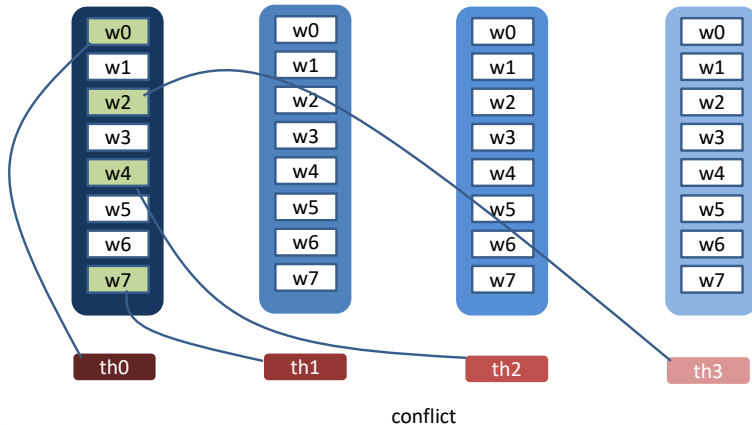
the Shared memory is divided in 32 BANKS (here we show only 4)



No conflict

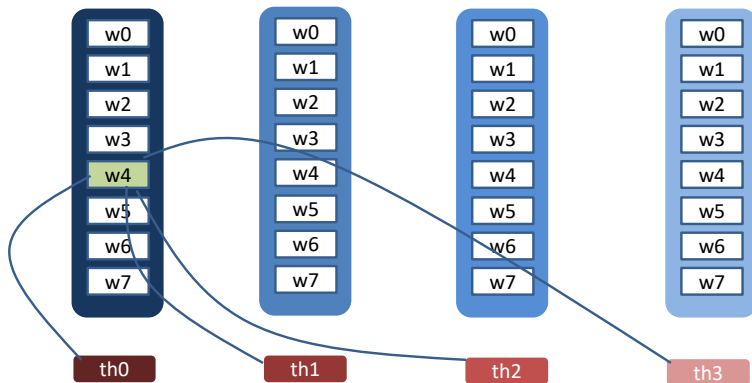
# Bank Conflict

the Shared memory is divided in 32 BANKS (here we show only 4)



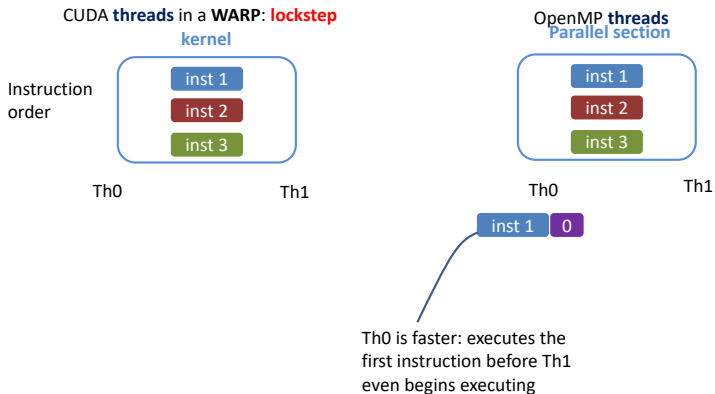
# Bank Conflict

the Shared memory is divided in 32 BANKS (here we show only 4)

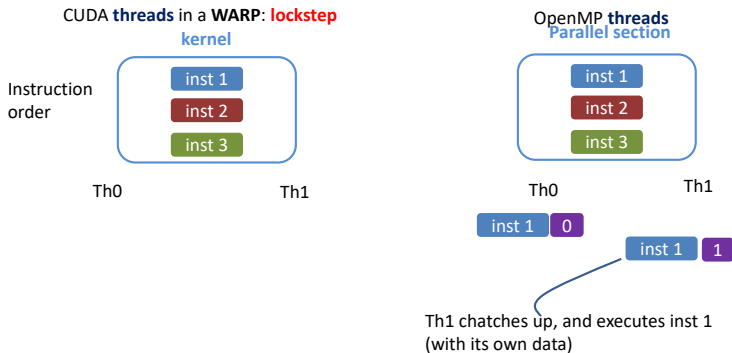


No conflict

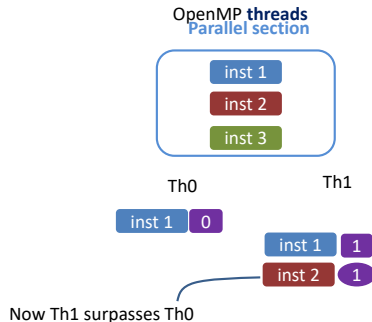
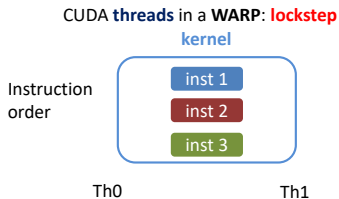
# Warp: Lockstep



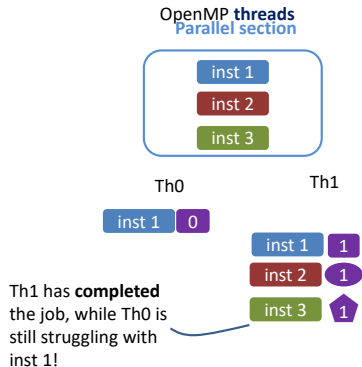
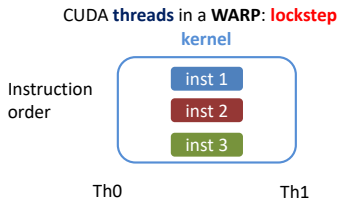
# Warp: Lockstep



# Warp: Lockstep

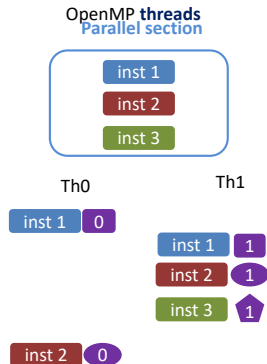
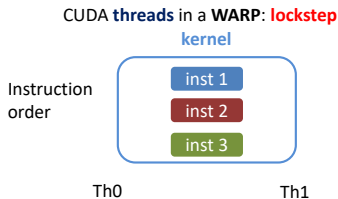


# Warp: Lockstep

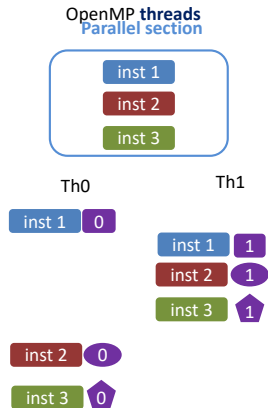
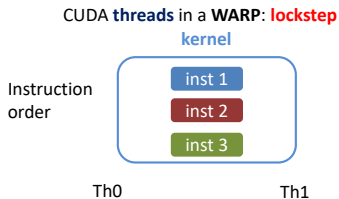




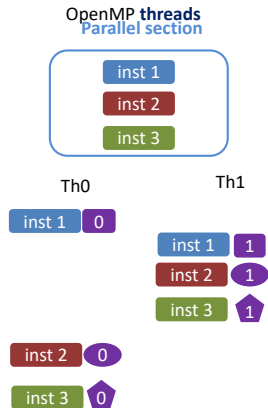
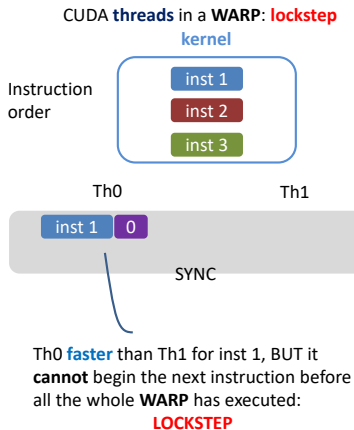
# Warp: Lockstep



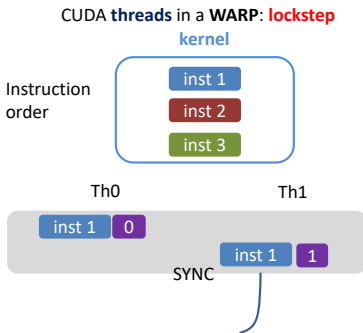
# Warp: Lockstep



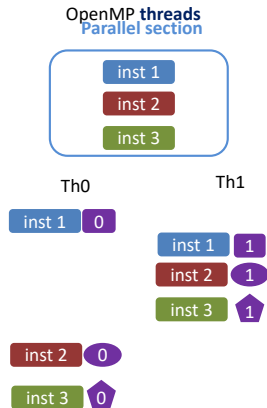
# Warp: Lockstep



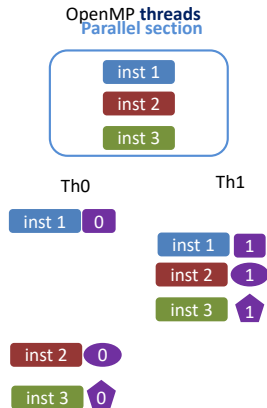
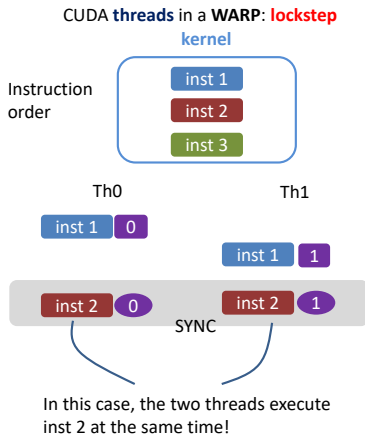
# Warp: Lockstep



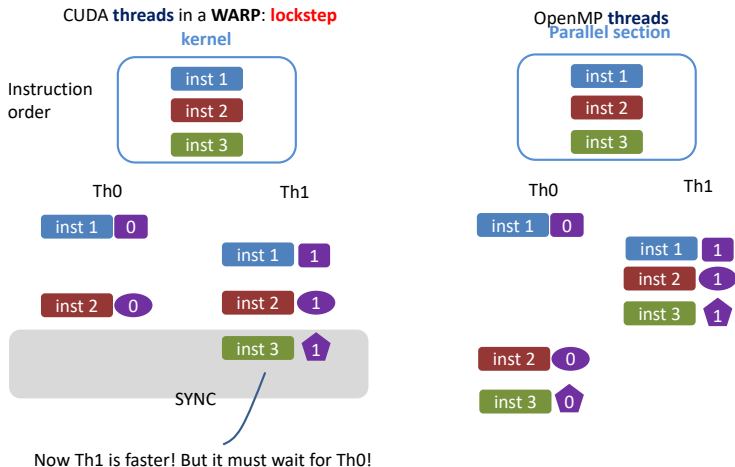
The whole **WARP** (2 threads in this example) has executed! It can move to the next instruction.



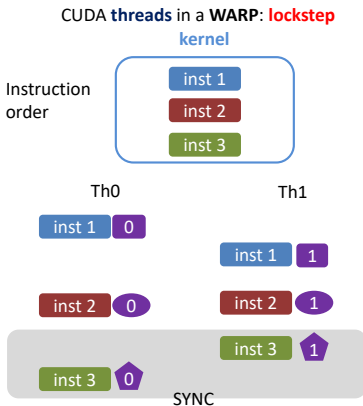
# Warp: Lockstep



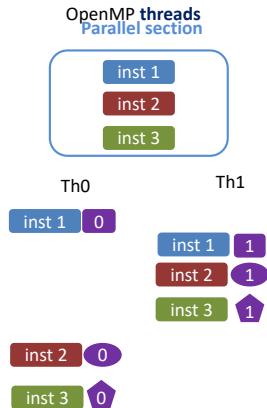
# Warp: Lockstep



# Warp: Lockstep



The warp has terminated the execution of this kernel!



# *Warp: come progettare un codice CUDA*

Per avere dei consigli su come scrivere un codice CUDA efficace, facciamo riferimento alle seguenti slide:

[https://mc.stanford.edu/cgi-bin/images/3/34/Darve\\_cme343\\_cuda\\_3.pdf](https://mc.stanford.edu/cgi-bin/images/3/34/Darve_cme343_cuda_3.pdf)



# Operazioni Atomiche

Quando un thread legge-modifica-scrive una variabile si dice che c'e' stato un **read-modify-write**. Una operazione di tipo **read-modify-write** e' problematica nel senso che puo' essere soggetta a **race conditions**. Per scongiurare le race-condition, si possono usare le funzioni **atomic**.

- un Read-modify-write non puo' essere interrotto durante operazioni di tipo **atomic**
- le atomiche sono implementate dalle c.c. 1.1.
- esistono molte operazioni **atomic** a disposizione del CUDA C
  - `atomicAdd()`
  - `atomicInc()`
  - `atomicSub()`
  - `atomicDec()`
  - `atomicMin()`
  - `atomicExch()`
  - `atomicMax()`
  - `atomicCAS()`
- il risultato diventa predicibile (invece che non-deterministico) anche quando ci sono molti accessi alla stessa area di memoria
- le **atomic** generalmente funzionano sia quando si usa memoria **global** che **shared**.

# Operazioni Atomiche

Quando un thread legge-modifica-scrive una variabile si dice che c'è stato un **read-modify-write**. Una operazione di tipo **read-modify-write** è problematica nel senso che può essere soggetta a **race conditions**. Per scongiurare le race-condition, si possono usare le funzioni **atomic**.

- un Read-modify-write non può essere interrotto durante operazioni di tipo **atomic**
- le atomiche sono implementate dalle **c.c. 1.1**.
- esistono molte operazioni **atomic** a disposizione del CUDA C
  - `atomicAdd()`
  - `atomicInc()`
  - `atomicSub()`
  - `atomicDec()`
  - `atomicMin()`
  - `atomicExch()`
  - `atomicMax()`
  - `atomicCAS()`
- il risultato diventa predicibile (invece che non-deterministico) anche quando ci sono molti accessi alla stessa area di memoria
- le **atomic** generalmente funzionano sia quando si usa memoria **global** che **shared**.

# Operazioni Atomiche

Quando un thread legge-modifica-scrive una variabile si dice che c'è stato un **read-modify-write**. Una operazione di tipo **read-modify-write** è problematica nel senso che può essere soggetta a **race conditions**. Per scongiurare le race-condition, si possono usare le funzioni **atomic**.

- un Read-modify-write non può essere interrotto durante operazioni di tipo **atomic**
- le atomiche sono implementate dalle **c.c.** 1.1.
- esistono molte operazioni **atomic** a disposizione del CUDA C
  - `atomicAdd()`
  - `atomicInc()`
  - `atomicSub()`
  - `atomicDec()`
  - `atomicMin()`
  - `atomicExch()`
  - `atomicMax()`
  - `atomicCAS()`
- il risultato diventa predicibile (invece che non-deterministico) anche quando ci sono molti accessi alla stessa area di memoria
- le **atomic** generalmente funzionano sia quando si usa memoria **global** che **shared**.

# Operazioni Atomiche

Quando un thread legge-modifica-scrive una variabile si dice che c'è stato un **read-modify-write**. Una operazione di tipo **read-modify-write** è problematica nel senso che può essere soggetta a **race conditions**. Per scongiurare le race-condition, si possono usare le funzioni **atomic**.

- un Read-modify-write non può essere interrotto durante operazioni di tipo **atomic**
- le atomiche sono implementate dalle **c.c.** 1.1.
- esistono molte operazioni **atomic** a disposizione del CUDA C
  - `atomicAdd()`
  - `atomicInc()`
  - `atomicSub()`
  - `atomicDec()`
  - `atomicMin()`
  - `atomicExch()`
  - `atomicMax()`
  - `atomicCAS()`
- il risultato diventa predicibile (invece che non-deterministico) anche quando ci sono molti accessi alla stessa area di memoria
- le **atomic** generalmente funzionano sia quando si usa memoria **global** che **shared**.

# Operazioni Atomiche

Quando un thread legge-modifica-scrive una variabile si dice che c'è stato un **read-modify-write**. Una operazione di tipo **read-modify-write** è problematica nel senso che può essere soggetta a **race conditions**. Per scongiurare le race-condition, si possono usare le funzioni **atomic**.

- un Read-modify-write non può essere interrotto durante operazioni di tipo **atomic**
- le atomiche sono implementate dalle **c.c.** 1.1.
- esistono molte operazioni **atomic** a disposizione del CUDA C
  - `atomicAdd()`
  - `atomicInc()`
  - `atomicSub()`
  - `atomicDec()`
  - `atomicMin()`
  - `atomicExch()`
  - `atomicMax()`
  - `atomicCAS()`
- il risultato diventa predicibile (invece che non-deterministico) anche quando ci sono molti accessi alla stessa area di memoria
- le **atomic** generalmente funzionano sia quando si usa memoria **global** che **shared**.

# Operazioni Atomiche

Quando un thread legge-modifica-scrive una variabile si dice che c'è stato un **read-modify-write**. Una operazione di tipo **read-modify-write** è problematica nel senso che può essere soggetta a **race conditions**. Per scongiurare le race-condition, si possono usare le funzioni **atomic**.

- un Read-modify-write non può essere interrotto durante operazioni di tipo **atomic**
- le atomiche sono implementate dalle **c.c.** 1.1.
- esistono molte operazioni **atomic** a disposizione del CUDA C
  - `atomicAdd()`
  - `atomicInc()`
  - `atomicSub()`
  - `atomicDec()`
  - `atomicMin()`
  - `atomicExch()`
  - `atomicMax()`
  - `atomicCAS()`
- il risultato diventa predicibile (invece che non-deterministico) anche quando ci sono molti accessi alla stessa area di memoria
- le **atomic** generalmente funzionano sia quando si usa memoria **global** che **shared**.

# atomicAdd

Esempio:

```
int atomicAdd(int* address, int val);
```

- Il thread legge la memoria puntata da `address` e a questa aggiunge la quantità `val`: il risultato e' scritto nell'area di memoria puntata da `address`.
- **ATTENZIONE** come risultato la funzione restituisce il valore dell'area di memoria puntata da `address` prima che fosse fatta l'accumulazione. Per esempio:

```
int valore= 1;
int daAggiungere =3;
int vecchio;
vecchio = atomicAdd(&valore, daAggiungere);
```

Dopo queste linee di codice si ottiene: `valore= 4` **ma**: `vecchio=1`

- **Subdolo**: `vecchio` NON viene aggiornato in modo atomic, e ci puo' essere race condition!
- Quindi se io voglio sommare una quantità ad un'altra devo semplicemente fare:

```
int valore= 1;
int daAggiungere =3;
atomicAdd(&valore, daAggiungere); // senza ASSEGNARE nulla alla funzione!
```

- in base alle c.c. esistono degli **overload** della funzione che lavorano con `float`, `double`, ecc.

# atomicAdd

Esempio:

```
int atomicAdd(int* address, int val);
```

- Il thread legge la memoria puntata da `address` e a questa aggiunge la quantità `val`: il risultato e' scritto nell'area di memoria puntata da `address`.
- **ATTENZIONE** come risultato la funzione **restituisce** il **valore** dell'area di memoria puntata da `address` **prima** che fosse fatta l'accumulazione. Per esempio:

```
int valore= 1;  
int daAggiungere =3;  
int vecchio;  
vecchio = atomicAdd(&valore, daAggiungere);
```

Dopo queste linee di codice si ottiene: `valore= 4` **ma**: `vecchio=1`

- **Subdolo**: `vecchio` NON viene aggiornato in modo atomic, e ci puo' essere race condition!
- Quindi se io voglio sommare una quantità ad un'altra devo semplicemente fare:

```
int valore= 1;  
int daAggiungere =3;  
atomicAdd(&valore, daAggiungere); // senza ASSEGNARE nulla alla funzione!
```

- in base alle c.c. esistono degli **overload** della funzione che lavorano con `float`, `double`, ecc.



# atomicAdd

Esempio:

```
int atomicAdd(int* address, int val);
```

- Il thread legge la memoria puntata da `address` e a questa aggiunge la quantità `val`: il risultato è scritto nell'area di memoria puntata da `address`.
- **ATTENZIONE** come risultato la funzione restituisce il **valore** dell'area di memoria puntata da `address` **prima** che fosse fatta l'accumulazione. Per esempio:

```
int valore= 1;  
int daAggiungere =3;  
int vecchio;  
vecchio = atomicAdd(&valore, daAggiungere);
```

Dopo queste linee di codice si ottiene: `valore= 4` **ma**: `vecchio=1`

- **Subdolo**: `vecchio` NON viene aggiornato in modo atomic, e ci può essere race condition!
- Quindi se io voglio sommare una quantità ad un'altra devo semplicemente fare:

```
int valore= 1;  
int daAggiungere =3;  
atomicAdd(&valore, daAggiungere); // senza ASSEGNARE nulla alla funzione!
```

- in base alle c.c. esistono degli **overload** della funzione che lavorano con `float`, `double`, ecc.

# atomicAdd

Esempio:

```
int atomicAdd(int* address, int val);
```

- Il thread legge la memoria puntata da `address` e a questa aggiunge la quantità `val`: il risultato è scritto nell'area di memoria puntata da `address`.
- **ATTENZIONE** come risultato la funzione restituisce il **valore** dell'area di memoria puntata da `address` **prima** che fosse fatta l'accumulazione. Per esempio:

```
int valore= 1;  
int daAggiungere =3;  
int vecchio;  
vecchio = atomicAdd(&valore, daAggiungere);
```

Dopo queste linee di codice si ottiene: `valore= 4` **ma**: `vecchio=1`

- **Subdolo**: `vecchio` NON viene aggiornato in modo atomic, e ci può essere race condition!
- Quindi se io voglio sommare una quantità ad un'altra devo semplicemente fare:

```
int valore= 1;  
int daAggiungere =3;  
atomicAdd(&valore, daAggiungere); // senza ASSEGNARE nulla alla funzione!
```

- in base alle c.c. esistono degli **overload** della funzione che lavorano con float, double, ecc.

# atomicAdd

Esempio:

```
int atomicAdd(int* address, int val);
```

- Il thread legge la memoria puntata da `address` e a questa aggiunge la quantità `val`: il risultato è scritto nell'area di memoria puntata da `address`.
- **ATTENZIONE** come risultato la funzione restituisce il **valore** dell'area di memoria puntata da `address` **prima** che fosse fatta l'accumulazione. Per esempio:

```
int valore= 1;
int daAggiungere =3;
int vecchio;
vecchio = atomicAdd(&valore, daAggiungere);
```

Dopo queste linee di codice si ottiene: `valore= 4` **ma**: `vecchio=1`

- **Subdolo**: `vecchio` NON viene aggiornato in modo atomic, e ci può essere race condition!
- Quindi se io voglio sommare una quantità ad un'altra devo semplicemente fare:

```
int valore= 1;
int daAggiungere =3;
atomicAdd(&valore, daAggiungere); // senza ASSEGNARE nulla alla funzione!
```

- in base alle c.c. esistono degli **overload** della funzione che lavorano con `float`, `double`, ecc.

# atomicAdd

Esempio:

```
int atomicAdd(int* address, int val);
```

- Il thread legge la memoria puntata da `address` e a questa aggiunge la quantità `val`: il risultato è scritto nell'area di memoria puntata da `address`.
- **ATTENZIONE** come risultato la funzione restituisce il **valore** dell'area di memoria puntata da `address` **prima** che fosse fatta l'accumulazione. Per esempio:

```
int valore= 1;
int daAggiungere =3;
int vecchio;
vecchio = atomicAdd(&valore, daAggiungere);
```

Dopo queste linee di codice si ottiene: `valore= 4` **ma**: `vecchio=1`

- **Subdolo**: `vecchio` NON viene aggiornato in modo atomic, e ci può essere race condition!
- Quindi se io voglio sommare una quantità ad un'altra devo semplicemente fare:

```
int valore= 1;
int daAggiungere =3;
atomicAdd(&valore, daAggiungere); // senza ASSEGNARE nulla alla funzione!
```

- in base alle c.c. esistono degli **overload** della funzione che lavorano con float, double, ecc.

# Atomic Attention

Un'operazione *atomic*, protegge **solo** da altri accessi atomici... Se accedo alla variabile anche in modo **non** atomico si puo' avere **race condition**. Per esempio:

```
__global__ void myRace(int *x) //  
{  
    int i = threadIdx.x + blockDim.x * blockIdx.x;  
  
    if (i== 0 | i==32) // il primo thread del warp 0 e del warp 1 aggiornano in modo non Atomico  
    {  
        *x = *x + 1; //  
    }  
    else // tutti gli altri aggiornano in modo Atomico  
    {  
        atomicAdd(x, 1); //  
    }  
}
```

- Supponiamo di chiamare il kernel su 2 warp
- Se il thread 32 (che e' il primo thread del secondo warp) si impossessa della x, prima che tutti gli altri abbiano fatto la atomic, il risultato diventa non predicibile!
- Occhio che nelle diapositiva viste ieri, il codice a questo riguardo ha degli errori.

# Atomic Attention

Un'operazione *atomic*, protegge **solo** da altri accessi atomici... Se accedo alla variabile anche in modo **non** atomico si puo' avere **race condition**. Per esempio:

```
__global__ void myRace(int *x)  //
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;

  if (i== 0 | i==32)  // il primo thread del warp 0 e del warp 1 aggiornano in modo non Atomico
  {
    *x = *x + 1;  //
  }
  else  // tutti gli altri aggiornano in modo Atomico
  {
    atomicAdd(x, 1);  //
  }
}
```

- Supponiamo di chiamare il kernel su 2 warp
- Se il thread 32 (che e' il primo thread del secondo warp) si impossessa della x, prima che tutti gli altri abbiano fatto la atomic, il risultato diventa non predicibile!
- Occhio che nelle diapositiva viste ieri, il codice a questo riguardo ha degli errori.

# Atomic Attention

Un'operazione *atomic*, protegge **solo** da altri accessi atomici... Se accedo alla variabile anche in modo **non** atomico si puo' avere **race condition**. Per esempio:

```
__global__ void myRace(int *x) //
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;

    if (i== 0 | i==32) // il primo thread del warp 0 e del warp 1 aggiornano in modo non Atomico
    {
        *x = *x + 1; //
    }
    else // tutti gli altri aggiornano in modo Atomico
    {
        atomicAdd(x, 1); //
    }
}
```

- Supponiamo di chiamare il kernel su 2 warp
- Se il thread 32 (che e' il primo thread del secondo warp) si impossessa della x, prima che tutti gli altri abbiano fatto la atomic, il risultato diventa non predicibile!
- Occhio che nelle diapositiva viste ieri, il codice a questo riguardo ha degli errori.

# Atomic Attention

Un'operazione *atomic*, protegge **solo** da altri accessi atomici... Se accedo alla variabile anche in modo **non** atomico si puo' avere **race condition**. Per esempio:

```
__global__ void myRace(int *x)  //
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;

  if (i== 0 | i==32)  // il primo thread del warp 0 e del warp 1 aggiornano in modo non Atomico
  {
    *x = *x + 1;  //
  }
  else  // tutti gli altri aggiornano in modo Atomico
  {
    atomicAdd(x, 1);  //
  }
}
```

- Supponiamo di chiamare il kernel su 2 warp
- Se il thread 32 (che e' il primo thread del secondo warp) si impossessa della x, prima che tutti gli altri abbiano fatto la atomic, il risultato diventa non predicibile!
- Occhio che nelle diapositiva viste ieri, il codice a questo riguardo ha degli errori.



# Atomic Attention

Un'operazione *atomic*, protegge **solo** da altri accessi atomici... Se accedo alla variabile anche in modo **non** atomico si puo' avere **race condition**. Per esempio:

```
__global__ void myRace(int *x)  //
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;

  if (i== 0 | i==32)  // il primo thread del warp 0 e del warp 1 aggiornano in modo non Atomico
  {
    *x = *x + 1;  //
  }
  else  // tutti gli altri aggiornano in modo Atomico
  {
    atomicAdd(x, 1);  //
  }
}
```

- Supponiamo di chiamare il kernel su 2 warp
- Se il thread 32 (che e' il primo thread del secondo warp) si impossessa della x, prima che tutti gli altri abbiano fatto la atomic, il risultato diventa non predicibile!
- Occhio che nelle diapositiva viste ieri, il codice a questo riguardo ha degli errori.

# Esempio Atomico

```

#include <stdio.h>
#include <cuda.h>
__global__ void Somma(float *a, int nDim){
    for (int i = 0; i < nDim ; i++){ // per ognuno degli ingressi
        atomicAdd(&a[i], 1.0f); // il singolo thread aggiunge 1 ad ogni ingresso
    }
}

void lanciaKernel(float *a, int nDim){  Somma<<<1, 10>>>(a,nDim); }

int main(){
    int i, nDim=100, tXb=32; // dim array, thread per blocco
    float *xDevice, *xHost; // puntatore che va verso il device e host

    xHost=(float *) malloc(nDim*sizeof(float)); // allocca sull'HOST
    cudaMalloc((void **)&xDevice, nDim*sizeof(float)); // allocca sul DEVICE
    cudaMemset(xDevice, 0, nDim*sizeof(float)); // azzerà array nel device

    lanciaKernel(xDevice,nDim); //

    cudaMemcpy(xHost, xDevice, nDim*sizeof(float), cudaMemcpyDeviceToHost);

    for ( i = 0; i < nDim; i++) if (xHost[i] != 1.0f*tXb){
        printf("Errore xHost[%d] e' %f ma dovrebbe essere %f\n", i, xHost[i], 1.0f*tXb);
        return 1; } printf("Successo!\n");
    return 0;
}

```

Attenzione: passo un puntatore allocato sul device ad una funzione dell'host, ma non c'è nessun problema!

# Esempio Atomico

```

#include <stdio.h>
#include <cuda.h>
__global__ void Somma(float *a, int nDim){
    for (int i = 0; i < nDim; i++){ // per ognuno degli ingressi
        atomicAdd(&a[i], 1.0f); // il singolo thread aggiunge 1 ad ogni ingresso
    }
}

void lanciaKernel(float *a, int nDim){  Somma<<<1, 10>>>(a,nDim); }

int main(){
    int i, nDim=100, tXb=32; // dim array, thread per blocco
    float *xDevice, *xHost; // puntatore che va verso il device e host

    xHost=(float *) malloc(nDim*sizeof(float)); // allocca sull'HOST
    cudaMalloc((void **)&xDevice, nDim*sizeof(float)); // allocca sul DEVICE
    cudaMemset(xDevice, 0, nDim*sizeof(float)); // azzerà array nel device

    lanciaKernel(xDevice,nDim); //

    cudaMemcpy(xHost, xDevice, nDim*sizeof(float), cudaMemcpyDeviceToHost);

    for ( i = 0; i < nDim; i++) if (xHost[i] != 1.0f*tXb){
        printf("Errore xHost[%d] e' %f ma dovrebbe essere %f\n", i, xHost[i], 1.0f*tXb);
        return 1; } printf("Successo!\n");
    return 0;
}

```

**Attenzione:** passo un puntatore allocato sul **device** ad una funzione dell'**host**, ma non c'è nessun problema!

# Altri costrutti di sincronizzazione

Oltre alle Atomic:

- `__syncthreads()` questo comando sincronizza tutti i thread di un **blocco**
- ricordiamo che kernel successivi sono automaticamente in coda uno dietro all'altro nel cuda stream, in particolare nello **stream 0**
- se vogliamo essere sicuri che tutto un cuda stream abbia finito di eseguire prima che l'**host** continui ad eseguire istruzioni, possiamo usare:  
`cudaDeviceSynchronize()`
- `cudaMemcpy()` sincronizza **host** e **device** ...
- ... ma potrebbe non essere sufficiente per i kernel che usano `printf`, per quelli si va sul sicuro se si usa `cudaDeviceSynchronize()`!
- ci sono molti altri costrutti per sincronizzare in modo granulare, e si possono usare anche dei trucchi che riguardano la misurazione delle performance di un codice CUDA: CUDA events (che vedremo al volo in seguito).

# Altri costrutti di sincronizzazione

Oltre alle Atomic:

- `__syncthreads()` questo comando sincronizza tutti i thread di un **blocco**
- ricordiamo che kernel successivi sono automaticamente in coda uno dietro all'altro nel cuda stream, in particolare nello **stream 0**
- se vogliamo essere sicuri che tutto un cuda stream abbia finito di eseguire prima che l'host continui ad eseguire istruzioni, possiamo usare:  
`cudaDeviceSynchronize()`
- `cudaMemcpy()` sincronizza **host** e **device** ...
- ... ma potrebbe non essere sufficiente per i kernel che usano `printf`, per quelli si va sul sicuro se si usa `cudaDeviceSynchronize()`!
- ci sono molti altri costrutti per sincronizzare in modo granulare, e si possono usare anche dei trucchi che riguardano la misurazione delle performance di un codice CUDA: CUDA events (che vedremo al volo in seguito).

## Altri costrutti di sincronizzazione

Oltre alle Atomic:

- `__syncthreads()` questo comando sincronizza tutti i thread di un **blocco**
- ricordiamo che kernel successivi sono automaticamente in coda uno dietro all'altro nel cuda stream, in particolare nello **stream 0**
- se vogliamo essere sicuri che tutto un cuda stream abbia finito di eseguire prima che l'**host** continui ad eseguire istruzioni, possiamo usare:  
`cudaDeviceSynchronize()`
- `cudaMemcpy()` sincronizza **host** e **device** ...
- ... ma potrebbe non essere sufficiente per i kernel che usano `printf`, per quelli si va sul sicuro se si usa `cudaDeviceSynchronize()`!
- ci sono molti altri costrutti per sincronizzare in modo granulare, e si possono usare anche dei trucchi che riguardano la misurazione delle performance di un codice CUDA: CUDA events (che vedremo al volo in seguito).

## Altri costrutti di sincronizzazione

Oltre alle Atomic:

- `__syncthreads()` questo comando sincronizza tutti i thread di un **blocco**
- ricordiamo che kernel successivi sono automaticamente in coda uno dietro all'altro nel cuda stream, in particolare nello **stream 0**
- se vogliamo essere sicuri che tutto un cuda stream abbia finito di eseguire prima che l'**host** continui ad eseguire istruzioni, possiamo usare:  
`cudaDeviceSynchronize()`
- `cudaMemcpy()` sincronizza **host** e **device** ...
- ... ma potrebbe non essere sufficiente per i kernel che usano `printf`, per quelli si va sul sicuro se si usa `cudaDeviceSynchronize()`!
- ci sono molti altri costrutti per sincronizzare in modo granulare, e si possono usare anche dei trucchi che riguardano la misurazione delle performance di un codice CUDA: CUDA events (che vedremo al volo in seguito).

## Altri costrutti di sincronizzazione

Oltre alle Atomic:

- `__syncthreads()` questo comando sincronizza tutti i thread di un **blocco**
- ricordiamo che kernel successivi sono automaticamente in coda uno dietro all'altro nel cuda stream, in particolare nello **stream 0**
- se vogliamo essere sicuri che tutto un cuda stream abbia finito di eseguire prima che l'**host** continui ad eseguire istruzioni, possiamo usare:  
`cudaDeviceSynchronize()`
- `cudaMemcpy()` sincronizza **host** e **device** ...
- ... ma potrebbe non essere sufficiente per i kernel che usano `printf`, per quelli si va sul sicuro se si usa `cudaDeviceSynchronize()`!
- ci sono molti altri costrutti per sincronizzare in modo granulare, e si possono usare anche dei trucchi che riguardano la misurazione delle performance di un codice CUDA: CUDA events (che vedremo al volo in seguito).



## Altri costrutti di sincronizzazione

Oltre alle Atomic:

- `__syncthreads()` questo comando sincronizza tutti i thread di un **blocco**
- ricordiamo che kernel successivi sono automaticamente in coda uno dietro all'altro nel cuda stream, in particolare nello **stream 0**
- se vogliamo essere sicuri che tutto un cuda stream abbia finito di eseguire prima che l'**host** continui ad eseguire istruzioni, possiamo usare:  
`cudaDeviceSynchronize()`
- `cudaMemcpy()` sincronizza **host** e **device** ...
- ... ma potrebbe non essere sufficiente per i kernel che usano `printf`, per quelli si va sul sicuro se si usa `cudaDeviceSynchronize()`!
- ci sono molti altri costrutti per sincronizzare in modo granulare, e si possono usare anche dei trucchi che riguardano la misurazione delle performance di un codice CUDA: CUDA events (che vedremo al volo in seguito).

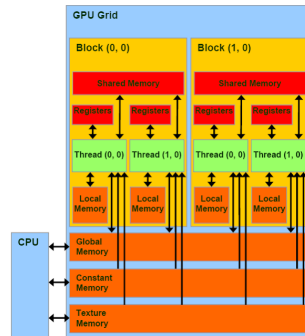
# Memoria Global

- e' la memoria piu' grande che c'e' su un *device*, per esempio 4-8 Gb
- alla Global possono accedere in scrittura e lettura **tutti** i thread (GPU) (pericolo di **race condition**)
- e' l'unica che puo' essere usata per un accesso sia di **lettura** che di **scrittura** dalla CPU
- ha una larghezza di banda molto grande: **throughput** fino a  $\approx 200$  Gb/s
- ha una **latenza molto grande**:  $\approx 400 - 800$  cicli di clock
- la global (e anche la constant e la texture) hanno una **persistent storage duration** ovvero sopravvivono al processo che le ha create (se anche il processo finisce la memoria continua ad esistere)

Puo' essere allocata sia in modo **statico** che **dinamico**:

- **statico**: `__device__ tipo_nome_variab;le;` (occhio che va scritto SOPRA il kernel che la chiama, altrimenti il compilatore da' errore).
- **dinamico**: per esempio

```
int *device_x;
cudaMalloc((void**) &device_x, dim_in_byte
);
cudaFree(device_x);
```



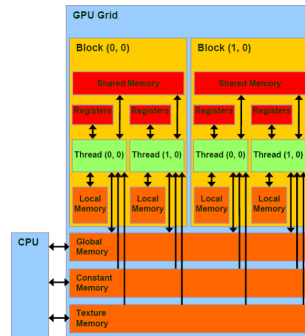
# Memoria Global

- e' la memoria piu' grande che c'e' su un *device*, per esempio 4-8 Gb
- alla Global possono accedere in scrittura e lettura **tutti** i thread (GPU) (pericolo di **race condition**)
- e' l'unica che puo' essere usata per un accesso sia di lettura che di scrittura dalla CPU
- ha una larghezza di banda molto grande: **throughput** fino a  $\approx 200$  Gb/s
- ha una **latenza molto grande**:  $\approx 400 - 800$  cicli di clock
- la global (e anche la constant e la texture) hanno una *persistent storage duration* ovvero sopravvivono al processo che le ha create (se anche il processo finisce la memoria continua ad esistere)

Puo' essere allocata sia in modo **statico** che **dinamico**:

- **statico**: `__device__ tipo nome_variab;le;` (occhio che va scritto SOPRA il kernel che la chiama, altrimenti il compilatore da' errore).
- **dinamico**: per esempio

```
int *device_x;
cudaMalloc((void**) &device_x, dim_in_byte);
};
cudaFree(var);
```



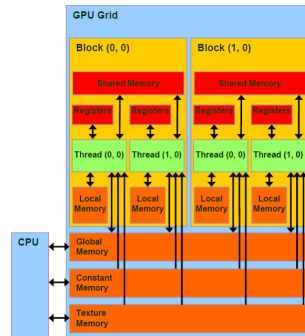
# Memoria Global

- e' la memoria piu' grande che c'e' su un *device*, per esempio 4-8 Gb
- alla Global possono accedere in scrittura e lettura **tutti** i thread (GPU) (pericolo di **race condition**)
- e' l'unica che puo' essere usata per un accesso sia di **lettura** che di **scrittura** dalla CPU
- ha una larghezza di banda molto grande: **throughput** fino a  $\approx 200$  Gb/s
- ha una **latenza molto grande**:  $\approx 400 - 800$  cicli di clock
- la global (e anche la constant e la texture) hanno una *persistent storage duration* ovvero sopravvivono al processo che le ha create (se anche il processo finisce la memoria continua ad esistere)

Puo' essere allocata sia in modo **statico** che **dinamico**:

- **statico**: `__device__ tipo_nome_variab;ile;` (occhio che va scritto SOPRA il kernel che la chiama, altrimenti il compilatore da' errore).
- **dinamico**: per esempio

```
int *device_x;
cudaMalloc((void**) &device_x, dim_in_byte
);
cudaFree(device_x);
```



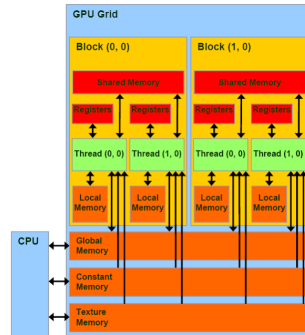
# Memoria Global

- e' la memoria piu' grande che c'e' su un *device*, per esempio 4-8 Gb
- alla Global possono accedere in scrittura e lettura **tutti** i thread (GPU) (pericolo di **race condition**)
- e' l'unica che puo' essere usata per un accesso sia di **lettura** che di **scrittura** dalla CPU
- ha una **larghezza di banda molto grande: throughput** fino a  $\approx 200$  Gb/s
- ha una **latenza molto grande**:  $\approx 400 - 800$  cicli di clock
- la global (e anche la constant e la texture) hanno una **persistent storage duration** ovvero sopravvivono al processo che le ha create (se anche il processo finisce la memoria continua ad esistere)

Puo' essere allocata sia in modo **statico** che **dinamico**:

- **statico**: `__device__ tipo nome_variab;le;` (occhio che va scritto SOPRA il kernel che la chiama, altrimenti il compilatore da' errore).
- **dinamico**: per esempio

```
int *device_x;
cudaMalloc((void**) &device_x, dim_in_byte);
};
cudaFree(var);
```



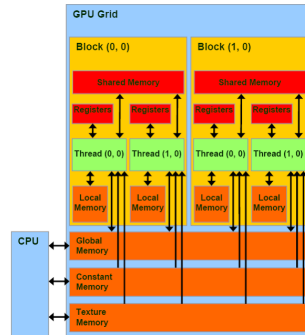
# Memoria Global

- e' la memoria piu' grande che c'e' su un *device*, per esempio 4-8 Gb
- alla Global possono accedere in scrittura e lettura **tutti** i thread (GPU) (pericolo di **race condition**)
- e' l'unica che puo' essere usata per un accesso sia di **lettura** che di **scrittura** dalla CPU
- ha una **larghezza di banda molto grande: throughput** fino a  $\approx 200$  Gb/s
- ha una **latenza molto grande**:  $\approx 400 - 800$  cicli di clock
- la *global* (e anche la *constant* e la *texture*) hanno una *persistent storage duration* ovvero sopravvivono al processo che le ha create (se anche il processo finisce la memoria continua ad esistere)

Puo' essere allocata sia in modo **statico** che **dinamico**:

- **statico**: `__device__ tipo_nome_variabile;` (occhio che va scritto SOPRA il kernel che la chiama, altrimenti il compilatore da **errore**).
- **dinamico**: per esempio

```
int *device_x;
cudaMalloc((void**) &device_x, dim_in_byte);
};
cudaFree(var);
```



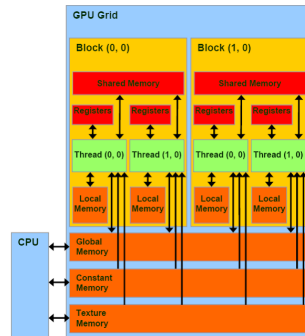
# Memoria Global

- e' la memoria piu' grande che c'e' su un *device*, per esempio 4-8 Gb
- alla Global possono accedere in scrittura e lettura **tutti** i thread (GPU) (pericolo di **race condition**)
- e' l'unica che puo' essere usata per un accesso sia di **lettura** che di **scrittura** dalla CPU
- ha una **larghezza di banda molto grande: throughput** fino a  $\approx 200$  Gb/s
- ha una **latenza molto grande**:  $\approx 400 - 800$  cicli di clock
- la *global* (e anche la *constant* e la *texture*) hanno una *persistent storage duration* ovvero sopravvivono al processo che le ha create (se anche il processo finisce la memoria continua ad esistere)

Puo' essere allocata sia in modo **statico** che **dinamico**:

- **statico**: `__device__ tipo_nome_variabile;` (occhio che va scritto SOPRA il kernel che la chiama, altrimenti il compilatore da **errore**).
- **dinamico**: per esempio

```
int *device_x;
cudaMalloc((void**) &device_x, dim_in_byte
);
cudaFree(device_x);
```



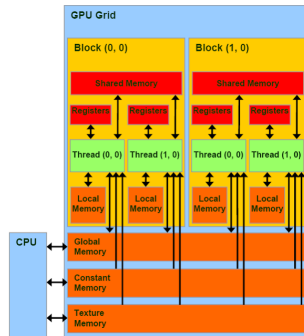
# Memoria Global

- e' la memoria piu' grande che c'e' su un *device*, per esempio 4-8 Gb
- alla Global possono accedere in scrittura e lettura **tutti** i thread (GPU) (pericolo di **race condition**)
- e' l'unica che puo' essere usata per un accesso sia di **lettura** che di **scrittura** dalla CPU
- ha una **larghezza di banda molto grande: throughput** fino a  $\approx 200$  Gb/s
- ha una **latenza molto grande**:  $\approx 400 - 800$  cicli di clock
- la *global* (e anche la *constant* e la *texture*) hanno una *persistent storage duration* ovvero sopravvivono al processo che le ha create (se anche il processo finisce la memoria continua ad esistere)

Puo' essere allocata sia in modo **statico** che **dinamico**:

- **statico**: `__device__ tipo nome_variabile;` (occhio che va scritto SOPRA il kernel che la chiama, altrimenti il compilatore da **errore**).
- **dinamico**: per esempio

```
int *device_x;
cudaMalloc((void**) &device_x, dim_in_byte
);
cudaFree(device_x);
```





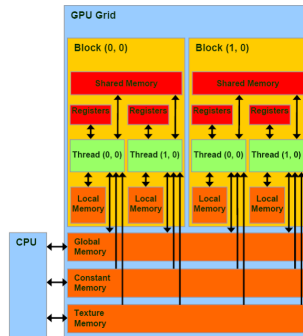
# Memoria Global

- e' la memoria piu' grande che c'e' su un *device*, per esempio 4-8 Gb
- alla Global possono accedere in scrittura e lettura **tutti** i thread (GPU) (pericolo di **race condition**)
- e' l'unica che puo' essere usata per un accesso sia di **lettura** che di **scrittura** dalla CPU
- ha una **larghezza di banda molto grande: throughput** fino a  $\approx 200$  Gb/s
- ha una **latenza molto grande**:  $\approx 400 - 800$  cicli di clock
- la *global* (e anche la *constant* e la *texture*) hanno una *persistent storage duration* ovvero sopravvivono al processo che le ha create (se anche il processo finisce la memoria continua ad esistere)

Puo' essere allocata sia in modo **statico** che **dinamico**:

- **statico**: `__device__ tipo nome_variabile;` (occhio che va scritto SOPRA il kernel che la chiama, altrimenti il compilatore da **errore**).
- **dinamico**: per esempio

```
int *device_x;
cudaMalloc((void**) &devic_x, dim_in_byte);
};
cudaFree(var);
```



# Memoria Global...local

Termini un po' contraddittori...

- **Cos'e' la memoria local?**
- E' una porzione di **Global**, che viene assegnata ad un thread (ha uno scope particolare)
- Ha le stesse caratteristiche di velocita' e throughput della **Global**, alle volte quando si leggono dei testi su CUDA puo' causare dubbi ed essere confusa con altri tipi di memoria dedicate ai singoli thread (i registri che invece sono MOLTO piu' rapidi)
- La dimensione **massima** assegnata ad un thread e 512 kb.
- referenze sulle memorie:  
<https://www.microway.com/hpc-tech-tips/gpu-memory-types-performance-comparison/>

# Memoria Global...local

Termini un po' contraddittori...

- Cos'e' la memoria **local**?
- E' una porzione di **Global**, che viene assegnata ad un thread (ha uno scope particolare)
- Ha le stesse caratteristiche di velocita' e throughput della **Global**, alle volte quando si leggono dei testi su CUDA puo' causare dubbi ed essere confusa con altri tipi di memoria dedicate ai singoli thread (i registri che invece sono MOLTO piu' rapidi)
- La dimensione **massima** assegnata ad un thread e 512 kb.
- referenze sulle memorie:  
<https://www.microway.com/hpc-tech-tips/gpu-memory-types-performance-comparison/>

# Memoria Global...local

Termini un po' contraddittori...

- Cos'è la memoria **local**?
- È una porzione di **Global**, che viene assegnata ad un thread (ha uno scope particolare)
- Ha le stesse caratteristiche di velocità e throughput della **Global**, alle volte quando si leggono dei testi su CUDA può causare dubbi ed essere confusa con altri tipi di memoria dedicate ai singoli thread (i registri che invece sono MOLTO più rapidi)
- La dimensione **massima** assegnata ad un thread è 512 kb.
- referenze sulle memorie:  
<https://www.microway.com/hpc-tech-tips/gpu-memory-types-performance-comparison/>

# Memoria Global...local

Termini un po' contraddittori...

- Cos'è la memoria **local**?
- È una porzione di **Global**, che viene assegnata ad un thread (ha uno scope particolare)
- Ha le stesse caratteristiche di velocità e throughput della **Global**, alle volte quando si leggono dei testi su CUDA può causare dubbi ed essere confusa con altri tipi di memoria dedicate ai singoli thread (i registri che invece sono MOLTO più rapidi)
- La dimensione **massima** assegnata ad un thread è **512 kb**.
- referenze sulle memorie:  
<https://www.microway.com/hpc-tech-tips/gpu-memory-types-performance-comparison/>

# Memoria Global...local

Termini un po' contraddittori...

- Cos'è la memoria **local**?
- È una porzione di **Global**, che viene assegnata ad un thread (ha uno scope particolare)
- Ha le stesse caratteristiche di velocità e throughput della **Global**, alle volte quando si leggono dei testi su CUDA può causare dubbi ed essere confusa con altri tipi di memoria dedicate ai singoli thread (i registri che invece sono MOLTO più rapidi)
- La dimensione **massima** assegnata ad un thread è **512 kb**.
- referenze sulle memorie:

<https://www.microway.com/hpc-tech-tips/gpu-memory-types-performance-comparison/>

# Constant Memory: intro

- Il collo di bottiglia per un calcolo con una GPU, puo' derivare non tanto dalla potenza di calcolo dei processori disponibili, quanto dall'accesso alla memoria (**bandwidth**).
- In pratica alle volte le operazioni di lettura della memoria (dovute al numero di richieste dalle ALU) vengono messe in coda e rallentano quindi l'esecuzione del codice.
- Supponiamo che ci siano dei dati **non vengono modificati** (solo lettura) durante l'esecuzione di un kernel, in questo caso e' possibile utilizzare la cosiddetta **constant memory** (la quantita' dipende dalle c.c., p. es. puo' essere 64 kb).
- chiaramente, visto che questa memoria non puo' essere modificata, il suo utilizzo e' subordinato al tipo di programma che si sta eseguendo!

Ci sono due vantaggi principali all'utilizzo della memoria **constant**:

- 1 una singola lettura ad un'area di memoria **constant** puo' essere trasmessa ad altri 31 thread "vicini" (senza occupare banda)!
- 2 la **constant** e' memorizzata nella cache sul chip dello **streaming multiprocessor**, per questo successive chiamate non aumentano il traffico dalla memoria DRAM.

**Attenzione:** la memoria **constant** e' piuttosto piccola: ci sono 64 kb di memoria constant e 8 kb di cache per ogni **streaming multiprocessor**

**Interessante:** da c.c. 2.0 se il processore e' in grado di determinare che una variabile non sara' modificata all'interno di un blocco (secondo alcune specifiche che non discutiamo qui), quell'oggetto viene messo automaticamente nella **constant**! (senza che il programmatore debba dichiarare qualcosa, altrimenti useremo una sintassi spiegata in seguito).

# Constant Memory: intro

- Il collo di bottiglia per un calcolo con una GPU, puo' derivare non tanto dalla potenza di calcolo dei processori disponibili, quanto dall'accesso alla memoria (**bandwidth**).
- In pratica alle volte le operazioni di lettura della memoria (dovute al numero di richieste dalle ALU) vengono messe in coda e rallentano quindi l'esecuzione del codice.
- Supponiamo che ci siano dei dati **non vengono modificati** (solo lettura) durante l'esecuzione di un kernel, in questo caso e' possibile utilizzare la cosiddetta **constant memory** (la quantita' dipende dalle c.c., p. es. puo' essere 64 kb).
- chiaramente, visto che questa memoria non puo' essere modificata, il suo utilizzo e' subordinato al tipo di programma che si sta eseguendo!

Ci sono due vantaggi principali all'utilizzo della memoria **constant**:

- 1 una singola lettura ad un'area di memoria **\_\_constant\_\_** puo' essere trasmessa ad altri 31 thread "vicini" (senza occupare banda)!
- 2 la **\_\_constant\_\_** e' memorizzata nella cache sul chip dello **streaming multiprocessor**, per questo successive chiamate non aumentano il traffico dalla memoria DRAM.

**Attenzione:** la memoria **constant** e' piuttosto piccola: ci sono 64 kb di memoria constant e 8 kb di cache per ogni **streaming multiprocessor**

**Interessante:** da c.c. 2.0 se il processore e' in grado di determinare che una variabile non sara' modificata all'interno di un blocco (secondo alcune specifiche che non discutiamo qui), quell'oggetto viene messo automaticamente nella **constant**! (senza che il programmatore debba dichiarare qualcosa, altrimenti useremo una sintassi spiegata in seguito).



# Constant Memory: intro

- Il collo di bottiglia per un calcolo con una GPU, puo' derivare non tanto dalla potenza di calcolo dei processori disponibili, quanto dall'accesso alla memoria (**bandwidth**).
- In pratica alle volte le operazioni di lettura della memoria (dovute al numero di richieste dalle ALU) vengono messe in coda e rallentano quindi l'esecuzione del codice.
- Supponiamo che ci siano dei dati **non vengono modificati** (solo lettura) durante l'esecuzione di un kernel, in questo caso e' possibile utilizzare la cosiddetta **constant memory** (la quantita' dipende dalle **c.c.**, p. es. puo' essere 64 kb).
  - chiaramente, visto che questa memoria non puo' essere modificata, il suo utilizzo e' subordinato al tipo di programma che si sta eseguendo!

Ci sono due vantaggi principali all'utilizzo della memoria **constant**:

- 1 una singola lettura ad un'area di memoria **\_\_constant\_\_** puo' essere trasmessa ad altri 31 thread "vicini" (senza occupare banda)!
- 2 la **\_\_constant\_\_** e' memorizzata nella cache sul chip dello **streaming multiprocessor**, per questo successive chiamate non aumentano il traffico dalla memoria DRAM.

**Attenzione:** la memoria **constant** e' piuttosto piccola: ci sono 64 kb di memoria constant e 8 kb di cache per ogni **streaming multiprocessor**

**Interessante:** da c.c. 2.0 se il processore e' in grado di determinare che una variabile non sara' modificata all'interno di un blocco (secondo alcune specifiche che non discutiamo qui), quell'oggetto viene messo automaticamente nella **constant**! (senza che il programmatore debba dichiarare qualcosa, altrimenti useremo una sintassi spiegata in seguito).

# Constant Memory: intro

- Il collo di bottiglia per un calcolo con una GPU, può derivare non tanto dalla potenza di calcolo dei processori disponibili, quanto dall'accesso alla memoria (**bandwidth**).
- In pratica alle volte le operazioni di lettura della memoria (dovute al numero di richieste dalle ALU) vengono messe in coda e rallentano quindi l'esecuzione del codice.
- Supponiamo che ci siano dei dati **non vengono modificati** (solo lettura) durante l'esecuzione di un kernel, in questo caso è possibile utilizzare la cosiddetta **constant memory** (la quantità dipende dalle **c.c.**, p. es. può essere 64 kb).
- chiaramente, visto che questa memoria non può essere modificata, il suo utilizzo è subordinato al tipo di programma che si sta eseguendo!

Ci sono due vantaggi principali all'utilizzo della memoria **constant**:

- 1 una singola lettura ad un'area di memoria **\_\_constant\_\_** può essere trasmessa ad altri 31 thread "vicini" (senza occupare banda)!
- 2 la **\_\_constant\_\_** è memorizzata nella cache sul chip dello **streaming multiprocessor**, per questo successive chiamate non aumentano il traffico dalla memoria DRAM.

**Attenzione:** la memoria **constant** è piuttosto piccola: ci sono 64 kb di memoria constant e 8 kb di cache per ogni **streaming multiprocessor**

**Interessante:** da c.c. 2.0 se il processore è in grado di determinare che una variabile non sarà modificata all'interno di un blocco (secondo alcune specifiche che non discutiamo qui), quell'oggetto viene messo automaticamente nella **constant**! (senza che il programmatore debba dichiarare qualcosa, altrimenti useremo una sintassi spiegata in seguito).

# Constant Memory: intro

- Il collo di bottiglia per un calcolo con una GPU, puo' derivare non tanto dalla potenza di calcolo dei processori disponibili, quanto dall'accesso alla memoria (**bandwidth**).
- In pratica alle volte le operazioni di lettura della memoria (dovute al numero di richieste dalle ALU) vengono messe in coda e rallentano quindi l'esecuzione del codice.
- Supponiamo che ci siano dei dati **non vengono modificati** (solo lettura) durante l'esecuzione di un kernel, in questo caso e' possibile utilizzare la cosiddetta **constant memory** (la quantita' dipende dalle **c.c.**, p. es. puo' essere 64 kb).
- chiaramente, visto che questa memoria non puo' essere modificata, il suo utilizzo e' subordinato al tipo di programma che si sta eseguendo!

Ci sono due vantaggi principali all'utilizzo della memoria **constant**:

- 1 una singola lettura ad un'area di memoria **\_\_constant\_\_** puo' essere trasmessa ad altri 31 thread "vicini" (senza occupare banda)!
- 2 la **\_\_constant\_\_** e' memorizzata nella cache sul chip dello **streaming multiprocessor**, per questo successive chiamate non aumentano il traffico dalla memoria DRAM.

**Attenzione:** la memoria **constant** e' piuttosto piccola: ci sono 64 kb di memoria constant e 8 kb di cache per ogni **streaming multiprocessor**

**Interessante:** da c.c. 2.0 se il processore e' in grado di determinare che una variabile non sara' modificata all'interno di un blocco (secondo alcune specifiche che non discutiamo qui), quell'oggetto viene messo automaticamente nella **constant**! (senza che il programmatore debba dichiarare qualcosa, altrimenti useremo una sintassi spiegata in seguito).

## Constant Memory: intro

- Il collo di bottiglia per un calcolo con una GPU, puo' derivare non tanto dalla potenza di calcolo dei processori disponibili, quanto dall'accesso alla memoria (**bandwidth**).
- In pratica alle volte le operazioni di lettura della memoria (dovute al numero di richieste dalle ALU) vengono messe in coda e rallentano quindi l'esecuzione del codice.
- Supponiamo che ci siano dei dati **non vengono modificati** (solo lettura) durante l'esecuzione di un kernel, in questo caso e' possibile utilizzare la cosiddetta **constant memory** (la quantita' dipende dalle **c.c.**, p. es. puo' essere 64 kb).
- chiaramente, visto che questa memoria non puo' essere modificata, il suo utilizzo e' subordinato al tipo di programma che si sta eseguendo!

Ci sono due vantaggi principali all'utilizzo della memoria **constant**:

- 1 una singola lettura ad un'area di memoria **\_\_constant\_\_** puo' essere trasmessa ad altri 31 thread "vicini" (senza occupare banda)!
- 2 la **\_\_constant\_\_** e' memorizzata nella cache sul chip dello **streaming multiprocessor**, per questo successive chiamate non aumentano il traffico dalla memoria DRAM.

**Attenzione:** la memoria **constant** e' piuttosto piccola: ci sono 64 kb di memoria constant e 8 kb di cache per ogni **streaming multiprocessor**

**Interessante:** da c.c. 2.0 se il processore e' in grado di determinare che una variabile non sara' modificata all'interno di un blocco (secondo alcune specifiche che non discutiamo qui), quell'oggetto viene messo automaticamente nella **constant**! (senza che il programmatore debba dichiarare qualcosa, altrimenti useremo una sintassi spiegata in seguito).

# Constant Memory: intro

- Il collo di bottiglia per un calcolo con una GPU, puo' derivare non tanto dalla potenza di calcolo dei processori disponibili, quanto dall'accesso alla memoria (**bandwidth**).
- In pratica alle volte le operazioni di lettura della memoria (dovute al numero di richieste dalle ALU) vengono messe in coda e rallentano quindi l'esecuzione del codice.
- Supponiamo che ci siano dei dati **non vengono modificati** (solo lettura) durante l'esecuzione di un kernel, in questo caso e' possibile utilizzare la cosiddetta **constant memory** (la quantita' dipende dalle **c.c.**, p. es. puo' essere 64 kb).
- chiaramente, visto che questa memoria non puo' essere modificata, il suo utilizzo e' subordinato al tipo di programma che si sta eseguendo!

Ci sono due vantaggi principali all'utilizzo della memoria **constant**:

- 1 una singola lettura ad un'area di memoria **\_\_constant\_\_** puo' essere trasmessa ad altri 31 thread "vicini" (senza occupare banda)!
- 2 la **\_\_constant\_\_** e' memorizzata nella cache sul chip dello **streaming multiprocessor**, per questo successive chiamate non aumentano il traffico dalla memoria DRAM.

**Attenzione:** la memoria **constant** e' piuttosto piccola: ci sono 64 kb di memoria constant e 8 kb di cache per ogni **streaming multiprocessor**

**Interessante:** da c.c. 2.0 se il processore e' in grado di determinare che una variabile non sara' modificata all'interno di un blocco (secondo alcune specifiche che non discutiamo qui), quell'oggetto viene messo automaticamente nella **constant**! (senza che il programmatore debba dichiarare qualcosa, altrimenti useremo una sintassi spiegata in seguito).

# Constant Memory: intro

- Il collo di bottiglia per un calcolo con una GPU, può derivare non tanto dalla potenza di calcolo dei processori disponibili, quanto dall'accesso alla memoria (**bandwidth**).
- In pratica alle volte le operazioni di lettura della memoria (dovute al numero di richieste dalle ALU) vengono messe in coda e rallentano quindi l'esecuzione del codice.
- Supponiamo che ci siano dei dati **non vengono modificati** (solo lettura) durante l'esecuzione di un kernel, in questo caso è possibile utilizzare la cosiddetta **constant memory** (la quantità dipende dalle **c.c.**, p. es. può essere 64 kb).
- chiaramente, visto che questa memoria non può essere modificata, il suo utilizzo è subordinato al tipo di programma che si sta eseguendo!

Ci sono due vantaggi principali all'utilizzo della memoria **constant**:

- 1 una singola lettura ad un'area di memoria **\_\_constant\_\_** può essere trasmessa ad altri 31 thread "vicini" (senza occupare banda)!
- 2 la **\_\_constant\_\_** è memorizzata nella cache sul chip dello **streaming multiprocessor**, per questo successive chiamate non aumentano il traffico dalla memoria DRAM.

**Attenzione:** la memoria **constant** è piuttosto piccola: ci sono 64 kb di memoria constant e 8 kb di cache per ogni **streaming multiprocessor**

**Interessante:** da c.c. 2.0 se il processore è in grado di determinare che una variabile non sarà modificata all'interno di un blocco (secondo alcune specifiche che non discutiamo qui), quell'oggetto viene messo automaticamente nella **constant**! (senza che il programmatore debba dichiarare qualcosa, altrimenti useremo una sintassi spiegata in seguito).

# Constant: *il bello*

## Cosa sono i thread vicini?

- I thread che appartengono ad uno stesso **warp**
- ricordiamoci che il codice viene tramutato in una serie di istruzioni e tutti i thread di **warp** eseguono la **stessa istruzione** con **dati differenti** (a seconda del proprio id): **lockstep!**

Supponiamo quindi che tutti i thread di un **warp** necessitino di accedere ad uno stesso dato:

- il primo thread legge il dato dalla DRAM
- gli altri 31 thread ricevono lo stesso dato, trasmesso dal primo thread e non occupano banda! (il traffico generato dalla **constant** e' 1/32 di quello che sarebbe necessario se ogni thread accedesse.
- le variabili inserite nella constant non sono modificabili, quindi saranno le stesse durante tutta l'esecuzione del kernel. Per questo ha senso immagazzinarle nella cache. Successive chiamate al dato sono quindi estremamente veloci ed evitano completamente l'accesso alla DRAM.

# Constant: *il bello*

Cosa sono i thread vicini?

- I thread che appartengono ad uno stesso **warp**
- ricordiamoci che il codice viene tramutato in una serie di istruzioni e tutti i thread di **warp** eseguono la **stessa istruzione** con **dati differenti** (a seconda del proprio id): **lockstep!**

Supponiamo quindi che tutti i thread di un **warp** necessitino di accedere ad uno stesso dato:

- il primo thread legge il dato dalla DRAM
- gli altri 31 thread ricevono lo stesso dato, trasmesso dal primo thread e non occupano banda! (il traffico generato dalla **constant** e' 1/32 di quello che sarebbe necessario se ogni thread accedesse.
- le variabili inserite nella constant non sono modificabili, quindi saranno le stesse durante tutta l'esecuzione del kernel. Per questo ha senso immagazzinarle nella cache. Successive chiamate al dato sono quindi estremamente veloci ed evitano completamente l'accesso alla DRAM.



# Constant: *il bello*

Cosa sono i thread vicini?

- I thread che appartengono ad uno stesso **warp**
- ricordiamoci che il codice viene tramutato in una serie di istruzioni e tutti i thread di **warp** eseguono la **stessa istruzione** con **dati differenti** (a seconda del proprio id): **lockstep!**

Supponiamo quindi che tutti i thread di un **warp** necessitino di accedere ad uno stesso dato:

- il primo thread legge il dato dalla DRAM
- gli altri 31 thread ricevono lo stesso dato, trasmesso dal primo thread e non occupano banda! (il traffico generato dalla **constant** e' 1/32 di quello che sarebbe necessario se ogni thread accedesse.
- le variabili inserite nella constant non sono modificabili, quindi saranno le stesse durante tutta l'esecuzione del kernel. Per questo ha senso immagazzinarle nella cache. Successive chiamate al dato sono quindi estremamente veloci ed evitano completamente l'accesso alla DRAM.

# Constant: *il bello*

Cosa sono i thread vicini?

- I thread che appartengono ad uno stesso **warp**
- ricordiamoci che il codice viene tramutato in una serie di istruzioni e tutti i thread di **warp** eseguono la **stessa istruzione** con **dati differenti** (a seconda del proprio id): **lockstep!**

Supponiamo quindi che tutti i thread di un **warp** necessitino di accedere ad uno stesso dato:

- il primo thread legge il dato dalla DRAM
- gli altri 31 thread ricevono lo stesso dato, trasmesso dal primo thread e non occupano banda! (il traffico generato dalla **constant** e' 1/32 di quello che sarebbe necessario se ogni thread accedesse.
- le variabili inserite nella constant non sono modificabili, quindi saranno le stesse durante tutta l'esecuzione del kernel. Per questo ha senso immagazzinarle nella cache. Successive chiamate al dato sono quindi estremamente veloci ed evitano completamente l'accesso alla DRAM.

# Constant: *il bello*

Cosa sono i thread vicini?

- I thread che appartengono ad uno stesso **warp**
- ricordiamoci che il codice viene tramutato in una serie di istruzioni e tutti i thread di **warp** eseguono la **stessa istruzione** con **dati differenti** (a seconda del proprio id): **lockstep!**

Supponiamo quindi che tutti i thread di un **warp** necessitino di accedere ad uno stesso dato:

- il primo thread legge il dato dalla DRAM
- gli altri 31 thread ricevono lo stesso dato, trasmesso dal primo thread e non occupano banda! (il traffico generato dalla **constant** e' 1/32 di quello che sarebbe necessario se ogni thread accedesse.
- le variabili inserite nella constant non sono modificabili, quindi saranno le stesse durante tutta l'esecuzione del kernel. Per questo ha senso immagazzinarle nella cache. Successive chiamate al dato sono quindi estremamente veloci ed evitano completamente l'accesso alla DRAM.

# Constant: *il bello*

Cosa sono i thread vicini?

- I thread che appartengono ad uno stesso **warp**
- ricordiamoci che il codice viene tramutato in una serie di istruzioni e tutti i thread di **warp** eseguono la **stessa istruzione** con **dati differenti** (a seconda del proprio id): **lockstep!**

Supponiamo quindi che tutti i thread di un **warp** necessitino di accedere ad uno stesso dato:

- il primo thread legge il dato dalla DRAM
- gli altri 31 thread ricevono lo stesso dato, trasmesso dal primo thread e non occupano banda! (il traffico generato dalla **constant** e' 1/32 di quello che sarebbe necessario se ogni thread accedesse.
- le variabili inserite nella constant non sono modificabili, quindi saranno le stesse durante tutta l'esecuzione del kernel. Per questo ha senso immagazzinarle nella cache. Successive chiamate al dato sono quindi estremamente veloci ed evitano completamente l'accesso alla DRAM.

# Constant: il bello

Cosa sono i thread vicini?

- I thread che appartengono ad uno stesso **warp**
- ricordiamoci che il codice viene tramutato in una serie di istruzioni e tutti i thread di **warp** eseguono la **stessa istruzione** con **dati differenti** (a seconda del proprio id): **lockstep!**

Supponiamo quindi che tutti i thread di un **warp** necessitino di accedere ad uno stesso dato:

- il primo thread legge il dato dalla DRAM
- gli altri 31 thread ricevono lo stesso dato, trasmesso dal primo thread e non occupano banda! (il traffico generato dalla **constant** e' 1/32 di quello che sarebbe necessario se ogni thread accedesse.
- le variabili inserite nella constant non sono modificabili, quindi saranno le stesse durante tutta l'esecuzione del kernel. Per questo ha senso immagazzinarle nella cache. Successive chiamate al dato sono quindi estremamente veloci ed evitano completamente l'accesso alla DRAM.

## Constant: il brutto

Il fatto che un dato constant venga trasmesso a tutti i membri del warp puo' pero' causare dei problemi se non utilizzato correttamente.

- Quando si ha una richiesta di accesso alla memoria constant da parte dei thread di un warp si puo' avere una sola richiesta per ciclo di clock per tutto il warp.
- Questo significa che se i thread del warp necessitano di diverse aree che comunque fanno parte dalla memoria constant dovremo aspettare 32 cicli di clock! ergo il sistema esegue serialmente le richieste e quindi rallenta. (In questo caso la lettura dalla constant diventa piu' lenta che quella dalla global)
- Eh! questo problema assomiglia molto al bank conflict della memoria shared, solo che in questo caso c'e' una sola banca.

## Constant: il brutto

Il fatto che un dato constant venga trasmesso a tutti i membri del warp puo' pero' causare dei problemi se non utilizzato correttamente.

- Quando si ha una **richiesta** di accesso alla memoria constant da parte dei thread di un **warp** si puo' avere **una sola richiesta per ciclo di clock** per tutto il warp.
- Questo significa che se i thread del warp necessitano di diverse aree che comunque fanno parte dalla memoria constant dovremo aspettare 32 cicli di clock! ergo il sistema esegue serialmente le richieste e quindi **rallenta**. (In questo caso la lettura dalla **constant** diventa piu' lenta che quella dalla **global**)
- **Ehi!** questo problema assomiglia molto al **bank conflict** della memoria **shared**, solo che in questo caso c'e' una sola banca.

# Constant: il brutto

Il fatto che un dato constant venga trasmesso a tutti i membri del warp puo' pero' causare dei problemi se non utilizzato correttamente.

- Quando si ha una **richiesta** di accesso alla memoria constant da parte dei thread di un **warp** si puo' avere **una sola richiesta per ciclo di clock** per tutto il warp.
- Questo significa che se i thread del warp necessitano di diverse aree che comunque fanno parte dalla memoria constant dovremo aspettare 32 cicli di clock! ergo il sistema esegue serialmente le richieste e quindi **rallenta**. (In questo caso la lettura dalla **constant** diventa piu' lenta che quella dalla **global**)
- **Ehil** questo problema assomiglia molto al **bank conflict** della memoria **shared**, solo che in questo caso c'e' una sola banca.



# Constant: il brutto

Il fatto che un dato constant venga trasmesso a tutti i membri del warp puo' pero' causare dei problemi se non utilizzato correttamente.

- Quando si ha una **richiesta** di accesso alla memoria constant da parte dei thread di un **warp** si puo' avere **una sola richiesta per ciclo di clock** per tutto il warp.
- Questo significa che se i thread del warp necessitano di diverse aree che comunque fanno parte dalla memoria constant dovremo aspettare 32 cicli di clock! ergo il sistema esegue serialmente le richieste e quindi **rallenta**. (In questo caso la lettura dalla **constant** diventa piu' lenta che quella dalla **global**)
- **Ehi!** questo problema assomiglia molto al **bank conflict** della memoria **shared**, solo che in questo caso c'e' una sola banca.

## Constant: il brutto

Il fatto che un dato constant venga trasmesso a tutti i membri del warp puo' pero' causare dei problemi se non utilizzato correttamente.

- Quando si ha una **richiesta** di accesso alla memoria constant da parte dei thread di un **warp** si puo' avere **una sola richiesta per ciclo di clock** per tutto il warp.
- Questo significa che se i thread del warp necessitano di diverse aree che comunque fanno parte dalla memoria constant dovremo aspettare 32 cicli di clock! ergo il sistema esegue serialmente le richieste e quindi **rallenta**. (In questo caso la lettura dalla **constant** diventa piu' lenta che quella dalla **global**)
- **Ehi!** questo problema assomiglia molto al **bank conflict** della memoria **shared**, solo che in questo caso c'e' una sola banca.

## Copiare la memoria constant

Il seguente comando e' una versione modificata di `cudaMemcpy()` che serve per copiare dati a(e da) memoria *constant*.

```
cudaError_t cudaMemcpyToSymbol (const char * symbol,  
                                const void * src,  
                                size_t count,  
                                size_t offset = 0,  
                                enum cudaMemcpyKind kind = cudaMemcpyHostToDevice  
)
```

Questo comando copia `count` byte dalla memoria che e' in `src` alla memoria che e' puntata in `symbol`, spostata di un `offset`.  
Il `symbol` puo' essere una variabile che risiede sia nella `global` che nella `constant`.  
Il tipo di passaggio puo' essere da `host` a `device` o viceversa

- `symbol` - Symbol destinazione
- `src` - sorgente
- `count` - dimensione in byte da copiare
- `offset` - distanza dal `symbol` di dove vengono copiati i dati
- `kind` - direzione della copia

## Esempio di utilizzo di Constant Memory

```

__constant__ float cangle[360]; // si dichiara la memoria constant dall'HOST
int main(int argc, char** argv)
{
    int size=3200;
    float* darray;           // puntatore array sul device
    float hangle[360];      // array sull'host

    cudaMalloc ((void*)&darray, sizeof(float)*size); // allocc memoria sul device
    cudaMemset (darray, 0, sizeof(float)*size);     // azzerà memoria sul device
    for(int loop=0; loop<360; loop++)
        hangle[loop] = acos( -1.0f ) * loop / 180.0f; // crea array host
    cudaMemcpyToSymbol ( cangle, hangle, sizeof(float)*360 ); // copia su CONSTANT
    test_kernel <<< size/64 , 64 >>> (darray); // lancia il kernel
    cudaFree(darray); // libera memoria
    return 0;
}

__global__ void test_kernel(float* darray) //kernel
{
    int index;
    Index = blockIdx.x * blockDim.x + threadIdx.x; // calcola indice globale

    for(int loop=0; loop<360; loop++) //
        darray[index]= darray [index] + cangle [loop] ;

    return;
}

```

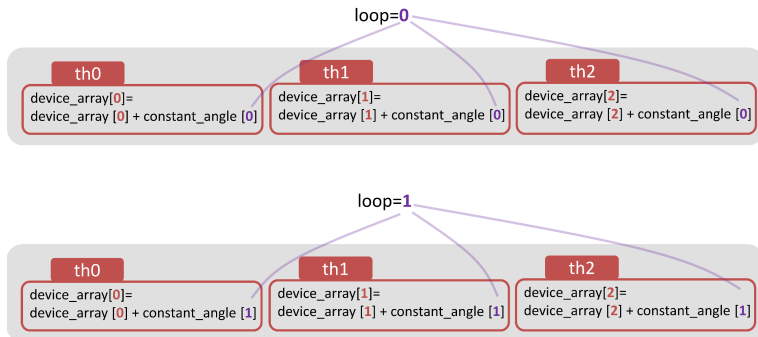
# Conflitto costante?

NO constant  
conflict!

```

__global__ void test_kernel(float* device_array) //kernel
{
    int index;
    index = blockIdx.x * blockDim.x + threadIdx.x;
    for(int loop=0;loop<360;loop++)
        device_array[index]= device_array [index] + constant_angle [loop] ;
    return;
}
  
```

Think per **instruction!**



## Constant: ma non c'è un problema di accessi?

- Nell'esempio precedente si è messo l'array `cangle` sulla memoria constant.
- D'altra parte, si sa che un accesso di thread diversi a diverse aree di cache **serializza** l'accesso alla cache e quindi può essere molto pericoloso.
- In questo caso però non c'è il problema perché?
- la risposta è che bisogna avere una visione del funzionamento del thread a livello di istruzione. Le **singole istruzioni** vengono definite in **lockstep**. Questo significa che in presenza di un loop come quello dell'esempio precedente i singoli passi del loop sono eseguiti in lockstep: **tutti** i thread prima lavorano `loop=0`, poi **tutti** i thread lavorano su `loop=1`, etc.
- Bisogna **liberarsi** dall'immagine mentale secondo cui si ha una associazione **tutto un kernel** ↔ **singolo thread**
- L'idea migliore è **singola istruzione** ↔ **warp**

## Constant: ma non c'è un problema di accessi?

- Nell'esempio precedente si è messo l'array `cangle` sulla memoria constant.
- D'altra parte, si sa che un accesso di thread diversi a diverse aree di cache **serializza** l'accesso alla cache e quindi può essere molto pericoloso.
- In questo caso però non c'è il problema perché?
- la risposta è che bisogna avere una visione del funzionamento dei thread a livello di istruzione. Le **singole istruzioni** vengono definite in **lockstep**. Questo significa che in presenza di un loop come quello dell'esempio precedente i singoli passi del loop sono eseguiti in lockstep: **tutti** i thread prima lavorano `loop=0`, poi **tutti** i thread lavorano su `loop=1`, etc.
- Bisogna **liberarsi** dall'immagine mentale secondo cui si ha una associazione **tutto un kernel** ↔ **singolo thread**
- L'idea migliore è **singola istruzione** ↔ **warp**

## Constant: ma non c'è un problema di accessi?

- Nell'esempio precedente si è messo l'array `cangle` sulla memoria constant.
- D'altra parte, si sa che un accesso di thread diversi a diverse aree di cache **serializza** l'accesso alla cache e quindi può essere molto pericoloso.
- In questo caso però non c'è il problema perché?
- la risposta è che bisogna avere una visione del funzionamento dei thread a livello di istruzione. Le **singole istruzioni** vengono definite in **lockstep**. Questo significa che in presenza di un loop come quello dell'esempio precedente i singoli passi del loop sono eseguiti in lockstep: **tutti** i thread prima lavorano `loop=0`, poi **tutti** i thread lavorano su `loop=1`, etc.
- Bisogna **liberarsi** dall'immagine mentale secondo cui si ha una associazione **tutto un kernel** ↔ **singolo thread**
- L'idea migliore è **singola istruzione** ↔ **warp**



## Constant: ma non c'è un problema di accessi?

- Nell'esempio precedente si è messo l'array `cangle` sulla memoria constant.
- D'altra parte, si sa che un accesso di thread diversi a diverse aree di cache **serializza** l'accesso alla cache e quindi può essere molto pericoloso.
- In questo caso però non c'è il problema perché?
- la risposta è che bisogna avere una visione del funzionamento dei thread a livello di istruzione. Le **singole istruzioni** vengono definite in **lockstep**. Questo significa che in presenza di un loop come quello dell'esempio precedente i singoli passi del loop sono eseguiti in lockstep: **tutti** i thread prima lavorano `loop=0`, poi **tutti** i thread lavorano su `loop=1`, etc.
- Bisogna **liberarsi** dall'immagine mentale secondo cui si ha una associazione **tutto un kernel** ↔ **singolo thread**
- L'idea migliore è **singola istruzione** ↔ **warp**

## Constant: ma non c'è un problema di accessi?

- Nell'esempio precedente si è messo l'array `cangle` sulla memoria constant.
- D'altra parte, si sa che un accesso di thread diversi a diverse aree di cache **serializza** l'accesso alla cache e quindi può essere molto pericoloso.
- In questo caso però non c'è il problema perché?
- la risposta è che bisogna avere una visione del funzionamento dei thread a livello di istruzione. Le **singole istruzioni** vengono definite in **lockstep**. Questo significa che in presenza di un loop come quello dell'esempio precedente i singoli passi del loop sono eseguiti in lockstep: **tutti** i thread prima lavorano `loop=0`, poi **tutti** i thread lavorano su `loop=1`, etc.
- Bisogna **liberarsi** dall'immagine mentale secondo cui si ha una associazione **tutto un kernel** ↔ **singolo thread**
- L'idea migliore è **singola istruzione** ↔ **warp**

## Constant: ma non c'è un problema di accessi?

- Nell'esempio precedente si è messo l'array `cangle` sulla memoria constant.
- D'altra parte, si sa che un accesso di thread diversi a diverse aree di cache **serializza** l'accesso alla cache e quindi può essere molto pericoloso.
- In questo caso però non c'è il problema perché?
- la risposta è che bisogna avere una visione del funzionamento dei thread a livello di istruzione. Le **singole istruzioni** vengono definite in **lockstep**. Questo significa che in presenza di un loop come quello dell'esempio precedente i singoli passi del loop sono eseguiti in lockstep: **tutti** i thread prima lavorano `loop=0`, poi **tutti** i thread lavorano su `loop=1`, etc.
- Bisogna **liberarsi** dall'immagine mentale secondo cui si ha una associazione **tutto un kernel** ↔ **singolo thread**
- L'idea migliore è **singola istruzione** ↔ **warp**

## Altro esempio: Convoluzione

- Consideriamo una convoluzione tra due funzioni
- ogni punto di una funzione  $f$  viene pesato con una funzione  $g$
- $(f * g)(x) = \int f(x - t)g(t)dt$ , la funzione  $g$  viene chiamata filtro (mask)

Dal punto di vista numerico:

f=	3	4	3	10	9	11	10	3	2	4	3	3
g=	-1	0	1									
f*g=	4	0	6	6	1	1	-8	-8	1	1	-1	-3

```

__global__ void convolution_1D_basic(float *N, float *M, float *P,
int Mask_Width, int Width) {
int i = blockIdx.x*blockDim.x + threadIdx.x;
float Pvalue = 0;
int N_start_point = i - (Mask_Width/2); // posizione della mask
for (int j = 0; j < Mask_Width; j++) {
    if (N_start_point + j >= 0 && N_start_point + j < Width) {
        Pvalue += N[N_start_point + j]*M[j];
    }
}
P[i] = Pvalue;
}

```

**Consiglio** per fare una convoluzione velocemente, meglio passare attraverso la FFT... sempre con CUDA!

## Altro esempio: Convoluzione

- Consideriamo una convoluzione tra due funzioni
- ogni punto di una funzione  $f$  viene pesato con una funzione  $g$
- $(f * g)(x) = \int f(x - t)g(t)dt$ , la funzione  $g$  viene chiamata filtro (mask)

Dal punto di vista numerico:

```
f=  3  4  3  10  9  11  10  3  2  4  3  3
g= -1  0  1
f*g= 4  0  6  6  1  1  -8  -8  1  1  -1  -3
```

```
__global__ void convolution_1D_basic(float *N, float *M, float *P,
int Mask_Width, int Width) {
int i = blockIdx.x*blockDim.x + threadIdx.x;
float Pvalue = 0;
int N_start_point = i - (Mask_Width/2); // posizione della mask
for (int j = 0; j < Mask_Width; j++) {
    if (N_start_point + j >= 0 && N_start_point + j < Width) {
        Pvalue += N[N_start_point + j]*M[j];
    }
}
P[i] = Pvalue;
}
```

Consiglio per fare una convoluzione velocemente, meglio passare attraverso la FFT... sempre con CUDA!

## Altro esempio: Convoluzione

- Consideriamo una convoluzione tra due funzioni
- ogni punto di una funzione  $f$  viene pesato con una funzione  $g$
- $(f * g)(x) = \int f(x - t)g(t)dt$ , la funzione  $g$  viene chiamata filtro (mask)

Dal punto di vista numerico:

f=	3	4	3	10	9	11	10	3	2	4	3	3
g=	-1	0	1									
f*g=	4	0	6	6	1	1	-8	-8	1	1	-1	-3

```

__global__ void convolution_1D_basic(float *N, float *M, float *P,
int Mask_Width, int Width) {
int i = blockIdx.x*blockDim.x + threadIdx.x;
float Pvalue = 0;
int N_start_point = i - (Mask_Width/2); // posizione della mask
for (int j = 0; j < Mask_Width; j++) {
    if (N_start_point + j >= 0 && N_start_point + j < Width) {
        Pvalue += N[N_start_point + j]*M[j];
    }
}
P[i] = Pvalue;
}

```

**Consiglio** per fare una convoluzione velocemente, meglio passare attraverso la FFT... sempre con CUDA!

# Registri

I Registri sono usati per mettere in memoria **scalari** o **piccoli array** con un accesso frequente per ogni thread. Per esempio: architettura Kepler, 255 registri (ogni registro=4 byte) per thread / 64 kb

- tanti meno registri sono necessari ad un **kernel**, tanti piu' blocchi possono essere assegnati ad una SM.
- Il numero di registri per kernel puo' essere limitato al tempo di compilazione con `-maxregcount max_registers`
- Singole variabili e array non dinamici vengono **automaticamente** messi nei **registri** sul chip (e questo garantisce che i costi di lettura e scrittura siano minimi), se le dimensioni diventano **grandi** vengono messi nella **local memory** e quindi la lettura e scrittura possono diventare centinaia di volte piu' lenti.
- **Attenzione:** ricordiamo che la cosiddetta **local memory** ed i **registri** sono due cose **differenti**. La cosiddetta local e' una area della global memory che e' riservata al singolo thread: per questo motivo e' lenta come la global!
- se si vuole scrivere un codice ottimizzato e' **necessario** controllare quanti registri vengono usati da ogni thread, per evitare colli di bottiglia dovuti a variabili finite nella **local memory**.

# Registri

I Registri sono usati per mettere in memoria **scalari** o **piccoli array** con un accesso frequente per ogni thread. Per esempio: architettura Kepler, 255 registri (ogni registro=4 byte) per thread / 64 kb

- tanti meno registri sono necessari ad un **kernel**, tanti piu' blocchi possono essere assegnati ad una SM.
- Il numero di registri per kernel puo' essere limitato al tempo di compilazione con `-maxregcount max_registers`
- Singole variabili e array non dinamici vengono **automaticamente** messi nei **registri** sul chip (e questo garantisce che i costi di lettura e scrittura siano minimi), se le dimensioni diventano **grandi** vengono messi nella **local memory** e quindi la lettura e scrittura possono diventare centinaia di volte piu' lenti.
- **Attenzione:** ricordiamo che la cosiddetta **local memory** ed i **registri** sono due cose **differenti**. La cosiddetta local e' una area della global memory che e' riservata al singolo thread: per questo motivo e' lenta come la global!
- se si vuole scrivere un codice ottimizzato e' **necessario** controllare quanti registri vengono usati da ogni thread, per evitare colli di bottiglia dovuti a variabili finite nella **local memory**.



# Registri

I Registri sono usati per mettere in memoria **scalari** o **piccoli array** con un accesso frequente per ogni thread. Per esempio: architettura Kepler, 255 registri (ogni registro=4 byte) per thread / 64 kb

- tanti meno registri sono necessari ad un **kernel**, tanti piu' blocchi possono essere assegnati ad una SM.
- Il numero di registri per kernel puo' essere limitato al tempo di compilazione con `-maxregcount max_registers`
- Singole variabili e array non dinamici vengono **automaticamente** messi nei **registri** sul chip (e questo garantisce che i costi di lettura e scrittura siano minimi), se le dimensioni diventano **grandi** vengono messi nella **local memory** e quindi la lettura e scrittura possono diventare centinaia di volte piu' lenti.
- **Attenzione:** ricordiamo che la cosiddetta **local memory** ed i **registri** sono due cose **differenti**. La cosiddetta local e' una area della global memory che e' riservata al singolo thread: per questo motivo e' lenta come la global!
- se si vuole scrivere un codice ottimizzato e' **necessario** controllare quanti registri vengono usati da ogni thread, per evitare colli di bottiglia dovuti a variabili finite nella **local memory**.

# Registri

I Registri sono usati per mettere in memoria **scalari** o **piccoli array** con un accesso frequente per ogni thread. Per esempio: architettura Kepler, 255 registri (ogni registro=4 byte) per thread / 64 kb

- tanti meno registri sono necessari ad un **kernel**, tanti piu' blocchi possono essere assegnati ad una SM.
- Il numero di registri per kernel puo' essere limitato al tempo di compilazione con `-maxregcount max_registers`
- Singole variabili e array non dinamici vengono **automaticamente** messi nei **registri** sul chip (e questo garantisce che i costi di lettura e scrittura siano minimi), se le dimensioni diventano **grandi** vengono messi nella **local memory** e quindi la lettura e scrittura possono diventare centinaia di volte piu' lenti.
- **Attenzione:** ricordiamo che la cosiddetta **local memory** ed i **registri** sono due cose **differenti**. La cosiddetta local e' una area della global memory che e' riservata al singolo thread: per questo motivo e' lenta come la global!
- se si vuole scrivere un codice ottimizzato e' **necessario** controllare quanti registri vengono usati da ogni thread, per evitare colli di bottiglia dovuti a variabili finite nella local memory.

# Registri

I Registri sono usati per mettere in memoria **scalari** o **piccoli array** con un accesso frequente per ogni thread. Per esempio: architettura Kepler, 255 registri (ogni registro=4 byte) per thread / 64 kb

- tanti meno registri sono necessari ad un **kernel**, tanti piu' blocchi possono essere assegnati ad una SM.
- Il numero di registri per kernel puo' essere limitato al tempo di compilazione con `-maxregcount max_registers`
- Singole variabili e array non dinamici vengono **automaticamente** messi nei **registri** sul chip (e questo garantisce che i costi di lettura e scrittura siano minimi), se le dimensioni diventano **grandi** vengono messi nella **local memory** e quindi la lettura e scrittura possono diventare centinaia di volte piu' lenti.
- **Attenzione:** ricordiamo che la cosiddetta **local memory** ed i **registri** sono due cose **differenti**. La cosiddetta local e' una area della global memory che e' riservata al singolo thread: per questo motivo e' lenta come la global!
- se si vuole scrivere un codice ottimizzato e' **necessario** controllare quanti registri vengono usati da ogni thread, per evitare colli di bottiglia dovuti a variabili finite nella **local memory**.

# Registri

I Registri sono usati per mettere in memoria **scalari** o **piccoli array** con un accesso frequente per ogni thread. Per esempio: architettura Kepler, 255 registri (ogni registro=4 byte) per thread / 64 kb

- tanti meno registri sono necessari ad un **kernel**, tanti piu' blocchi possono essere assegnati ad una SM.
- Il numero di registri per kernel puo' essere limitato al tempo di compilazione con `-maxregcount max_registers`
- Singole variabili e array non dinamici vengono **automaticamente** messi nei **registri** sul chip (e questo garantisce che i costi di lettura e scrittura siano minimi), se le dimensioni diventano **grandi** vengono messi nella **local memory** e quindi la lettura e scrittura possono diventare centinaia di volte piu' lenti.
- **Attenzione:** ricordiamo che la cosiddetta **local memory** ed i **registri** sono due cose **differenti**. La cosiddetta local e' una area della global memory che e' riservata al singolo thread: per questo motivo e' lenta come la global!
- se si vuole scrivere un codice ottimizzato e' **necessario** controllare quanti registri vengono usati da ogni thread, per evitare colli di bottiglia dovuti a variabili finite nella **local memory**.

# Registri

I Registri sono usati per mettere in memoria **scalari** o **piccoli array** con un accesso frequente per ogni thread. Per esempio: architettura Kepler, 255 registri (ogni registro=4 byte) per thread / 64 kb

- tanti meno registri sono necessari ad un **kernel**, tanti piu' blocchi possono essere assegnati ad una SM.
- Il numero di registri per kernel puo' essere limitato al tempo di compilazione con `-maxregcount max_registers`
- Singole variabili e array non dinamici vengono **automaticamente** messi nei **registri** sul chip (e questo garantisce che i costi di lettura e scrittura siano minimi), se le dimensioni diventano **grandi** vengono messi nella **local memory** e quindi la lettura e scrittura possono diventare centinaia di volte piu' lenti.
- **Attenzione:** ricordiamo che la cosiddetta **local memory** ed i **registri** sono due cose **differenti**. La cosiddetta local e' una area della global memory che e' riservata al singolo thread: per questo motivo e' lenta come la global!
- se si vuole scrivere un codice ottimizzato e' **necessario** controllare quanti registri vengono usati da ogni thread, per evitare colli di bottiglia dovuti a variabili finite nella **local** memory.

# Texture: perché un altro tipo di memoria?

## Problemi di accesso alle memorie veloci

- **constant memory**: se tutti i thread accedono alla stessa area di memoria questa è **ri-distribuita** (broadcastata! w la Crusca!) a costo zero a tutti i thread, ma se ci sono accessi a aree differenti l'accesso è serializzato (c'è un'unica banca a cui possono accedere i thread del warp).
- **shared memory**: similmente attenzione ai **bank conflict**. La memoria shared è suddivisa in 32 "banche" differenti. Se più thread provano ad accedere ad (aree diverse di) una singola banca, l'accesso è serializzato. Se accedono alla stessa banca e nella stessa area, il dato viene **ri-distribuito** (broadcastato) a costo zero. Notiamo che non esiste una coerenza spaziale tra i vari elementi della memoria shared (a meno che il programmatore non abbia costruito la memoria shared in modo apposito).

Domanda, se mi servisse:

- una memoria che non devo aggiornare ( la **constant** andrebbe bene per questo...)
- ma, nella singola istruzione, thread differenti vogliono accedere ad aree di memoria **differenti** ma contigue dal punto di vista geometrico (questa caratteristica mi fa invece scartare la **constant**)
- (e non posso usare la **shared** che invece è già usata per altri scopi)

Soluzione: Usiamo la memoria **Texture**

<http://cuda-programming.blogspot.com/2013/04/texture-references-object-in-cuda.html>

# Texture: perche' un altro tipo di memoria?

Problemi di accesso alle memorie veloci

- **constant memory**: se tutti i thread accedono alla stessa area di memoria questa e' **ri-distribuita** (broadcastata! w la Crusca!) a costo zero a tutti i thread, ma se ci sono accessi a aree differenti l'accesso e' serializzato (c'e' un'unica banca a cui possono accedere i thread del warp).
- **shared memory**: similmente attenzione ai **bank conflict**. La memoria shared e' suddivisa in 32 "banche" differenti. Se piu' thread provano ad accedere ad (aree diverse di) una singola banca, l'accesso e' serializzato. Se accedono alla stessa banca e nella stessa area, il dato viene **ri-distribuito** (broadcastato) a costo zero. Notiamo che non esiste una coerenza spaziale tra i vari elementi della memoria shared (a meno che il programmatore non abbia costruito la memoria shared in modo apposito).

Domanda, se mi servisse:

- una memoria che non devo aggiornare ( la **constant** andrebbe bene per questo...)
- ma, nella singola istruzione, thread differenti vogliono accedere ad aree di memoria **differenti** ma contigue dal punto di vista geometrico (questa caratteristica mi fa invece scartare la **constant**)
- (e non posso usare la **shared** che invece e' gia' usata per altri scopi)

Soluzione: Usiamo la memoria **Texture**

<http://cuda-programming.blogspot.com/2013/04/texture-references-object-in-cuda.html>

# Texture: perche' un altro tipo di memoria?

Problemi di accesso alle memorie veloci

- **constant memory**: se tutti i thread accedono alla stessa area di memoria questa e' **ri-distribuita** (broadcastata! w la Crusca!) a costo zero a tutti i thread, ma se ci sono accessi a aree differenti l'accesso e' serializzato (c'e' un'unica banca a cui possono accedere i thread del warp).
- **shared memory**: similmente attenzione ai **bank conflict**. La memoria shared e' suddivisa in **32 "banche"** differenti. Se piu' thread provano ad accedere ad (aree diverse di) una singola banca, l'accesso e' serializzato. Se accedono alla stessa banca e nella stessa area, il dato viene **ri-distribuito** (broadcastato) a costo zero. Notiamo che non esiste una coerenza spaziale tra i vari elementi della memoria shared (a meno che il programmatore non abbia costruito la memoria shared in modo apposito).

**Domanda**, se mi servisse:

- una memoria che non devo aggiornare ( la **constant** andrebbe bene per questo...)
- ma, nella singola istruzione, thread differenti vogliono accedere ad aree di memoria **differenti** ma contigue dal punto di vista geometrico (questa caratteristica mi fa invece scartare la **constant**)
- (e non posso usare la **shared** che invece e' gia' usata per altri scopi)

**Soluzione**: Usiamo la memoria **Texture**

<http://cuda-programming.blogspot.com/2013/04/texture-references-object-in-cuda.html>



# Texture: perche' un altro tipo di memoria?

Problemi di accesso alle memorie veloci

- **constant memory**: se tutti i thread accedono alla stessa area di memoria questa e' **ri-distribuita** (broadcastata! w la Crusca!) a costo zero a tutti i thread, ma se ci sono accessi a aree differenti l'accesso e' serializzato (c'e' un'unica banca a cui possono accedere i thread del warp).
- **shared memory**: similmente attenzione ai **bank conflict**. La memoria shared e' suddivisa in 32 "banche" differenti. Se piu' thread provano ad accedere ad (aree diverse di) una singola banca, l'accesso e' serializzato. Se accedono alla stessa banca e nella stessa area, il dato viene **ri-distribuito** (broadcastato) a costo zero. Notiamo che non esiste una coerenza spaziale tra i vari elementi della memoria shared (a meno che il programmatore non abbia costruito la memoria shared in modo apposito).

**Domanda**, se mi servisse:

- una memoria che non devo aggiornare ( la **constant** andrebbe bene per questo...)
- ma, nella singola istruzione, thread differenti vogliono accedere ad aree di memoria **differenti** ma contigue dal punto di vista geometrico (questa caratteristica mi fa invece scartare la **constant**)
- (e non posso usare la **shared** che invece e' gia' usata per altri scopi)

**Soluzione**: Usiamo la memoria **Texture**

<http://cuda-programming.blogspot.com/2013/04/texture-references-object-in-cuda.html>

# Texture: perche' un altro tipo di memoria?

Problemi di accesso alle memorie veloci

- **constant memory**: se tutti i thread accedono alla stessa area di memoria questa e' **ri-distribuita** (broadcastata! w la Crusca!) a costo zero a tutti i thread, ma se ci sono accessi a aree differenti l'accesso e' serializzato (c'e' un'unica banca a cui possono accedere i thread del warp).
- **shared memory**: similmente attenzione ai **bank conflict**. La memoria shared e' suddivisa in 32 "banche" differenti. Se piu' thread provano ad accedere ad (aree diverse di) una singola banca, l'accesso e' serializzato. Se accedono alla stessa banca e nella stessa area, il dato viene **ri-distribuito** (broadcastato) a costo zero. Notiamo che non esiste una coerenza spaziale tra i vari elementi della memoria shared (a meno che il programmatore non abbia costruito la memoria shared in modo apposito).

**Domanda**, se mi servisse:

- una memoria che non devo aggiornare ( la **constant** andrebbe bene per questo...)
- ma, nella singola istruzione, thread differenti vogliono accedere ad aree di memoria **differenti** ma contigue dal punto di vista geometrico (questa caratteristica mi fa invece scartare la **constant**)
- (e non posso usare la **shared** che invece e' gia' usata per altri scopi)

**Soluzione**: Usiamo la memoria **Texture**

<http://cuda-programming.blogspot.com/2013/04/texture-references-object-in-cuda.html>

# Texture: perche' un altro tipo di memoria?

Problemi di accesso alle memorie veloci

- **constant memory**: se tutti i thread accedono alla stessa area di memoria questa e' **ri-distribuita** (broadcastata! w la Crusca!) a costo zero a tutti i thread, ma se ci sono accessi a aree differenti l'accesso e' serializzato (c'e' un'unica banca a cui possono accedere i thread del warp).
- **shared memory**: similmente attenzione ai **bank conflict**. La memoria shared e' suddivisa in 32 "banche" differenti. Se piu' thread provano ad accedere ad (aree diverse di) una singola banca, l'accesso e' serializzato. Se accedono alla stessa banca e nella stessa area, il dato viene **ri-distribuito** (broadcastato) a costo zero. Notiamo che non esiste una coerenza spaziale tra i vari elementi della memoria shared (a meno che il programmatore non abbia costruito la memoria shared in modo apposito).

**Domanda**, se mi servisse:

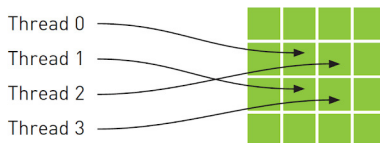
- una memoria che non devo aggiornare ( la **constant** andrebbe bene per questo...)
- ma, nella singola istruzione, thread differenti vogliono accedere ad aree di memoria **differenti** ma contigue dal punto di vista geometrico (questa caratteristica mi fa invece scartare la **constant**)
- (e non posso usare la **shared** che invece e' gia' usata per altri scopi)

**Soluzione**: Usiamo la memoria **Texture**

<http://cuda-programming.blogspot.com/2013/04/texture-references-object-in-cuda.html>

## La memoria texture

- **NON** e' sul chip, ma e' **cached** sul chip! (questo significa che successivi utilizzi della medesima memoria *texture* diventano estremamente veloci).
- puo' essere utilizzata per ottenere una banda efficace maggiore, riducendo le chiamate alla global sulla DRAM.
- la cache associata e' piccola, ordine di pochi kb (**64 kb**) per SM
- e' una memoria in sola **lettura** (da parte del kernel)
- pensata per le *texture* per immagini vettoriali, che hanno una notevole localita' spaziale e in questo caso molto efficiente
- ogni SM ha varie *texture fetch units*. Quindi ci sono delle unita' dedicate per accedere alla fetch.
- Ha una **coerenza spaziale** particolare, in un array come quello in figura i valori, normalmente, non sono nella stessa cache, con la texture si'!



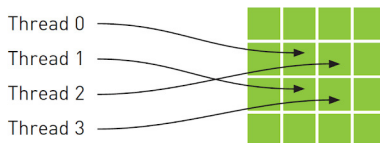
*Figure:* from: CUDA by example. Le aree di memoria **NON** sono consecutive (dal punto di vista della aritmetica dei puntatori)

Un po' di gergo:

- leggere la texture viene chiamato *texture fetch*
- *texel* (texture element) e' il nome degli elementi degli array 1D, 2D o 3D della texture

## La memoria texture

- **NON** e' sul chip, ma e' **cached** sul chip! (questo significa che successivi utilizzi della medesima memoria *texture* diventano estremamente veloci).
- puo' essere utilizzata per ottenere una banda efficace maggiore, riducendo le chiamate alla global sulla DRAM.
- la cache associata e' piccola, ordine di pochi kb (64 kb) per SM
- e' una memoria in sola **lettura** (da parte dei kernel)
- pensata per le *texture* per immagini vettoriali, che hanno una notevole localita' spaziale e in questo caso molto efficiente
- ogni SM ha varie *texture fetch units*. Quindi ci sono delle unita' dedicate per accedere alla fetch.
- Ha una **coerenza spaziale** particolare, in un array come quello in figura i valori, normalmente, non sono nella stessa cache, con la texture si'!



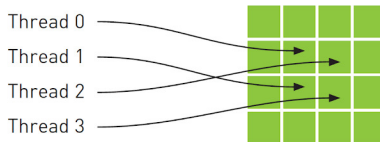
*Figure:* from: CUDA by example. Le aree di memoria **NON** sono consecutive (dal punto di vista della aritmetica dei puntatori)

Un po' di gergo:

- leggere la texture viene chiamato *texture fetch*
- *texel* (texture element) e' il nome degli elementi degli array 1D, 2D o 3D della *texture*

# La memoria texture

- **NON** e' sul chip, ma e' **cached** sul chip! (questo significa che successivi utilizzi della medesima memoria *texture* diventano estremamente veloci).
- puo' essere utilizzata per ottenere una banda efficace maggiore, riducendo le chiamate alla global sulla DRAM.
- la cache associata e' piccola, ordine di pochi kb (64 kb) per **SM**
- e' una memoria in sola **lettura** (da parte dei kernel)
- pensata per le *texture* per immagini vettoriali, che hanno una notevole localita' spaziale e in questo caso molto efficiente
- ogni SM ha varie *texture fetch units*. Quindi ci sono delle unita' dedicate per accedere alla fetch.
- Ha una **coerenza spaziale** particolare, in un array come quello in figura i valori, normalmente, non sono nella stessa cache, con la texture si!



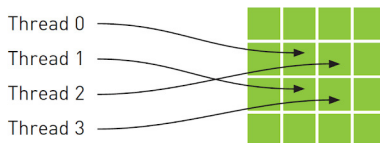
*Figure:* from: CUDA by example. Le aree di memoria **NON** sono consecutive (dal punto di vista della aritmetica dei puntatori)

Un po' di gergo:

- leggere la texture viene chiamato *texture fetch*
- *texel* (texture element) e' il nome degli elementi degli array 1D, 2D o 3D della *texture*

## La memoria texture

- **NON** e' sul chip, ma e' **cached** sul chip! (questo significa che successivi utilizzi della medesima memoria *texture* diventano estremamente veloci).
- puo' essere utilizzata per ottenere una banda efficace maggiore, riducendo le chiamate alla global sulla DRAM.
- la cache associata e' piccola, ordine di pochi kb (64 kb) per **SM**
- e' una memoria in sola **lettura** (da parte dei kernel)
- pensata per le *texture* per immagini vettoriali, che hanno una notevole localita' spaziale e in questo caso molto efficiente
- ogni SM ha varie *texture fetch units*. Quindi ci sono delle unita' dedicate per accedere alla fetch.
- Ha una **coerenza spaziale** particolare, in un array come quello in figura i valori, normalmente, non sono nella stessa cache, con la texture si!



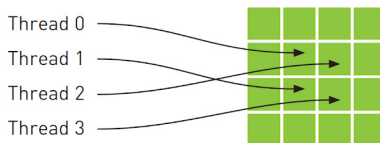
*Figure:* from: CUDA by example. Le aree di memoria **NON** sono consecutive (dal punto di vista della aritmetica dei puntatori)

Un po' di gergo:

- leggere la texture viene chiamato *texture fetch*
- *texel* (texture element) e' il nome degli elementi degli array 1D, 2D o 3D della texture

# La memoria texture

- **NON** e' sul chip, ma e' **cached** sul chip! (questo significa che successivi utilizzi della medesima memoria *texture* diventano estremamente veloci).
- puo' essere utilizzata per ottenere una banda efficace maggiore, riducendo le chiamate alla global sulla DRAM.
- la cache associata e' piccola, ordine di pochi kb (64 kb) per **SM**
- e' una memoria in sola **lettura** (da parte dei kernel)
- pensata per le *texture* per immagini vettoriali, che hanno una notevole localita' spaziale e in questo caso molto efficiente
- ogni SM ha varie *texture fetch units*. Quindi ci sono delle unita' dedicate per accedere alla fetch.
- Ha una **coerenza spaziale** particolare, in un array come quello in figura i valori, normalmente, non sono nella stessa cache, con la texture si'!



*Figure:* from: CUDA by example. Le aree di memoria **NON** sono consecutive (dal punto di vista della aritmetica dei puntatori)

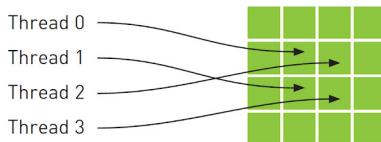
Un po' di gergo:

- leggere la texture viene chiamato *texture fetch*
- *texel* (texture element) e' il nome degli elementi degli array 1D, 2D o 3D della texture



## La memoria texture

- **NON** e' sul chip, ma e' **cached** sul chip! (questo significa che successivi utilizzi della medesima memoria *texture* diventano estremamente veloci).
- puo' essere utilizzata per ottenere una banda efficace maggiore, riducendo le chiamate alla global sulla DRAM.
- la cache associata e' piccola, ordine di pochi kb (64 kb) per **SM**
- e' una memoria in sola **lettura** (da parte dei kernel)
- pensata per le *texture* per immagini vettoriali, che hanno una notevole localita' spaziale e in questo caso molto efficiente
- ogni SM ha varie *texture fetch units*. Quindi ci sono delle unita' dedicate per accedere alla fetch.
- Ha una **coerenza spaziale** particolare, in un array come quello in figura i valori, normalmente, non sono nella stessa cache, con la texture si'!



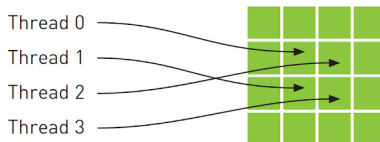
*Figure:* from: CUDA by example. Le aree di memoria **NON** sono consecutive (dal punto di vista della aritmetica dei puntatori)

Un po' di gergo:

- leggere la texture viene chiamato *texture fetch*
- *texel* (texture element) e' il nome degli elementi degli array 1D, 2D o 3D della texture

## La memoria texture

- **NON** e' sul chip, ma e' **cached** sul chip! (questo significa che successivi utilizzi della medesima memoria *texture* diventano estremamente veloci).
- puo' essere utilizzata per ottenere una banda efficace maggiore, riducendo le chiamate alla global sulla DRAM.
- la cache associata e' piccola, ordine di pochi kb (64 kb) per **SM**
- e' una memoria in sola **lettura** (da parte dei kernel)
- pensata per le *texture* per immagini vettoriali, che hanno una notevole localita' spaziale e in questo caso molto efficiente
- ogni SM ha varie *texture fetch units*. Quindi ci sono delle unita' dedicate per accedere alla fetch.
- Ha una **coerenza spaziale** particolare, in un array come quello in figura i valori, normalmente, non sono nella stessa cache, con la texture si'!



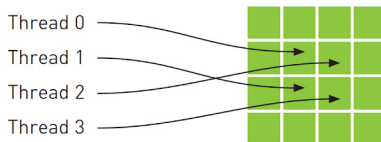
*Figure:* from: CUDA by example. Le aree di memoria **NON** sono consecutive (dal punto di vista della aritmetica dei puntatori)

Un po' di gergo:

- leggere la texture viene chiamato *texture fetch*
- *texel* (texture element) e' il nome degli elementi degli array 1D, 2D o 3D della texture

## La memoria texture

- **NON** e' sul chip, ma e' **cached** sul chip! (questo significa che successivi utilizzi della medesima memoria *texture* diventano estremamente veloci).
- puo' essere utilizzata per ottenere una banda efficace maggiore, riducendo le chiamate alla global sulla DRAM.
- la cache associata e' piccola, ordine di pochi kb (64 kb) per **SM**
- e' una memoria in sola **lettura** (da parte dei kernel)
- pensata per le *texture* per immagini vettoriali, che hanno una notevole localita' spaziale e in questo caso molto efficiente
- ogni SM ha varie *texture fetch units*. Quindi ci sono delle unita' dedicate per accedere alla fetch.
- Ha una **coerenza spaziale** particolare, in un array come quello in figura i valori, normalmente, non sono nella stessa cache, con la texture si'!



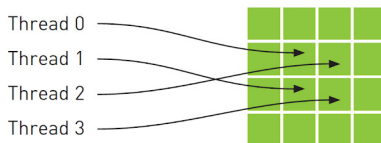
*Figure:* from: CUDA by example. Le aree di memoria **NON** sono consecutive (dal punto di vista della aritmetica dei puntatori)

Un po' di gergo:

- leggere la texture viene chiamato *texture fetch*
- *texel* (texture element) e' il nome degli elementi degli array 1D, 2D o 3D della texture

## La memoria texture

- **NON** e' sul chip, ma e' **cached** sul chip! (questo significa che successivi utilizzi della medesima memoria *texture* diventano estremamente veloci).
- puo' essere utilizzata per ottenere una banda efficace maggiore, riducendo le chiamate alla global sulla DRAM.
- la cache associata e' piccola, ordine di pochi kb (64 kb) per **SM**
- e' una memoria in sola **lettura** (da parte dei kernel)
- pensata per le *texture* per immagini vettoriali, che hanno una notevole localita' spaziale e in questo caso molto efficiente
- ogni SM ha varie *texture fetch units*. Quindi ci sono delle unita' dedicate per accedere alla fetch.
- Ha una **coerenza spaziale** particolare, in un array come quello in figura i valori, normalmente, non sono nella stessa cache, con la texture si'!



*Figure:* from: CUDA by example. Le aree di memoria **NON** sono consecutive (dal punto di vista della aritmetica dei puntatori)

Un po' di gergo:

- leggere la texture viene chiamato *texture fetch*
- *texel* (texture element) e' il nome degli elementi degli array 1D, 2D o 3D della texture

# Texture: uso

- 1 La memoria va **dichiarata** nell'**host**, per esempio:

```
texture<float> MY_TEXTURE ; // contiene float
```

**Attenzione** la dichiarazione, **non allocca** la memoria, **non associa** la memoria al nome!

- 2 Non si possono costruire direttamente dei puntatori alla **texture**. **Prima** si deve avere preparato un oggetto nella memoria del **device**. Per esempio: `cudaMalloc((void*)& in_global, sizeof(float));`
- 3 A questo punto si fa il **binding** alla memoria Texture della gpu

```
cudaBindTexture (size *t offset, // puntatore per allinare texture
                 MY_TEXTURE, // riferimento alla texture
                 const void * in_global, // area di memoria sul device da legare alla texture
                 size_t size); // dimensione dell'area di memoria puntata da MY_TEXTURE
```

Legare una quantità di dati di dimensione `size` dell'area di memoria puntata da `in_global` alla memoria texture dichiarata con il nome `MY_TEXTURE`

- 4 Bisogna **leggere** la memoria texture (non si accede semplicemente...), con degli "strumenti appositi", per esempio:

```
x=tex1Dfetch(MY_TEXTURE, i) // associa a x il valore i-esimo dell'area in MY_TEXTURE
```

- 5 Dopo l'utilizzo e' bene **liberare** la memoria: `cudaUnbindTexture (MY_TEXTURE)`

# Texture: uso

- 1 La memoria va **dichiarata** nell'**host**, per esempio:

```
texture<float> MY_TEXTURE ; // contiene float
```

**Attenzione** la dichiarazione, **non allocca** la memoria, **non** associa la memoria al nome!

- 2 **Non** si possono costruire direttamente dei puntatori alla **texture**. **Prima** si deve avere preparato un oggetto nella memoria del **device**. Per esempio: `cudaMalloc((void**) & in_global, sizeof(float))`;

- 3 A questo punto si fa il **binding** alla memoria Texture della gpu

```
cudaBindTexture (size *t offset, // puntatore per allinare texture
                 MY_TEXTURE, // riferimento alla texture
                 const void * in_global, // area di memoria sul device da legare alla texture
                 size_t size); // dimensione dell'area di memoria puntata da MY_TEXTURE
```

Legna una quantita' di dati di dimensione `size` dell'area di memoria puntata da `in_global` alla memoria texture dichiarata con il nome `MY_TEXTURE`

- 4 Bisogna **leggere** la memoria texture (non si accede semplicemente...), con degli "strumenti appositi", per esempio:

```
x=tex1Dfetch(MY_TEXTURE, i) // associa a x il valore i-esimo dell'area in MY_TEXTURE
```

- 5 Dopo l'utilizzo e' bene **liberare** la memoria: `cudaUnbindTexture(MY_TEXTURE)`

# Texture: uso

- 1 La memoria va **dichiarata** nell'**host**, per esempio:

```
texture<float> MY_TEXTURE ; // contiene float
```

**Attenzione** la dichiarazione, **non allocca** la memoria, **non** associa la memoria al nome!

- 2 **Non** si possono costruire direttamente dei puntatori alla **texture**. **Prima** si deve avere preparato un oggetto nella memoria del **device**. Per esempio: `cudaMalloc((void**) & in_global, sizeof(float));`
- 3 A questo punto si fa il **binding** alla memoria Texture della gpu

```
cudaBindTexture (size *t offset, // puntatore per allinare texture
                 MY_TEXTURE, // riferimento alla texture
                 const void * in_global, // area di memoria sul device da legare alla texture
                 size_t size); // dimensione dell'area di memoria puntata da MY_TEXTURE
```

Legna una quantita' di dati di dimensione `size` dell'area di memoria puntata da `in_global` alla memoria texture dichiarata con il nome `MY_TEXTURE`

- 4 Bisogna **leggere** la memoria texture (non si accede semplicemente...), con degli "strumenti appositi", per esempio:

```
x=tex1Dfetch(MY_TEXTURE, i) // associa a x il valore i-esimo dell'area in MY_TEXTURE
```

- 5 Dopo l'utilizzo e' bene **liberare** la memoria: `cudaUnbindTexture (MY_TEXTURE)`

# Texture: uso

- 1 La memoria va **dichiarata** nell'**host**, per esempio:

```
texture<float> MY_TEXTURE ; // contiene float
```

**Attenzione** la dichiarazione, **non allocca** la memoria, **non** associa la memoria al nome!

- 2 **Non** si possono costruire direttamente dei puntatori alla **texture**. **Prima** si deve avere preparato un oggetto nella memoria del **device**. Per esempio: `cudaMalloc((void*)& in_global, sizeof(float));`
- 3 A questo punto si fa il **binding** alla memoria Texture della gpu

```
cudaBindTexture (size *t offset, // puntatore per allinare texture
                 MY_TEXTURE, // riferimento alla texture
                 const void * in_global, // area di memoria sul device da legare alla texture
                 size_t size); // dimensione dell'area di memoria puntata da MY_TEXTURE
```

Legna una quantita' di dati di dimensione `size` dell'area di memoria puntata da `in_global` alla memoria texture dichiarata con il nome `MY_TEXTURE`

- 4 Bisogna **leggere** la memoria texture (non si accede semplicemente...), con degli "strumenti appositi", per esempio:

```
x=tex1Dfetch(MY_TEXTURE, i) // associa a x il valore i-esimo dell'area in MY_TEXTURE
```

- 5 Dopo l'utilizzo e' bene **liberare** la memoria: `cudaUnbindTexture (MY_TEXTURE)`



# Texture: uso

- 1 La memoria va **dichiarata** nell'**host**, per esempio:

```
texture<float> MY_TEXTURE ; // contiene float
```

**Attenzione** la dichiarazione, **non allocca** la memoria, **non** associa la memoria al nome!

- 2 **Non** si possono costruire direttamente dei puntatori alla **texture**. **Prima** si deve avere preparato un oggetto nella memoria del **device**. Per esempio: `cudaMalloc((void**) & in_global, sizeof(float));`
- 3 A questo punto si fa il **binding** alla memoria Texture della gpu

```
cudaBindTexture (size *t offset, // puntatore per allinare texture
                 MY_TEXTURE, // riferimento alla texture
                 const void * in_global, // area di memoria sul device da legare alla texture
                 size_t size); // dimensione dell'area di memoria puntata da MY_TEXTURE
```

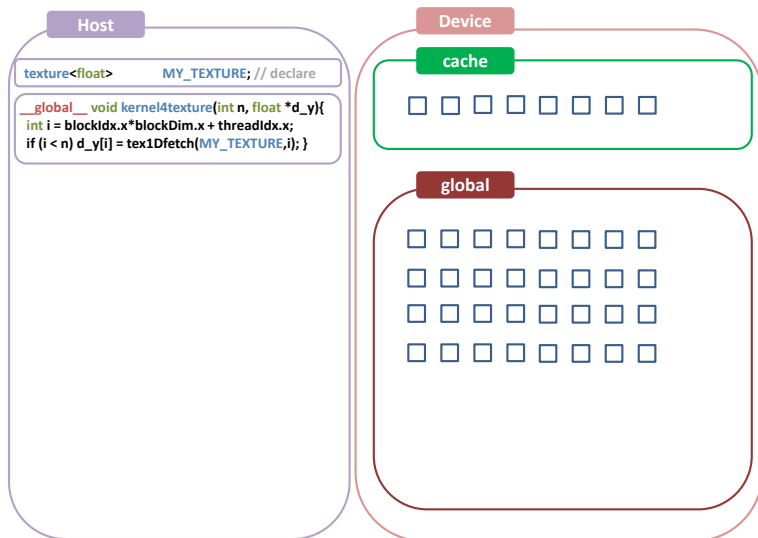
Legna una quantita' di dati di dimensione `size` dell'area di memoria puntata da `in_global` alla memoria texture dichiarata con il nome `MY_TEXTURE`

- 4 Bisogna **leggere** la memoria texture (non si accede semplicemente...), con degli "strumenti appositi", per esempio:

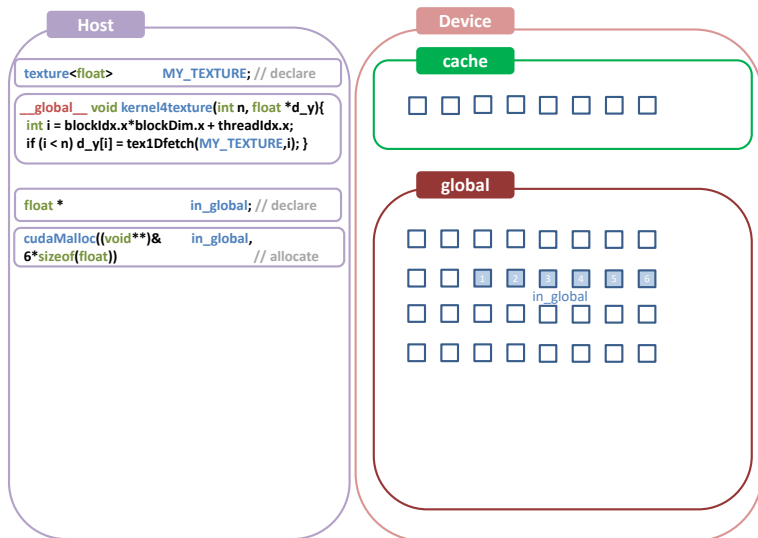
```
x=tex1Dfetch(MY_TEXTURE, i) // associa a x il valore i-esimo dell'area in MY_TEXTURE
```

- 5 Dopo l'utilizzo e' bene **liberare** la memoria: `cudaUnbindTexture (MY_TEXTURE)`

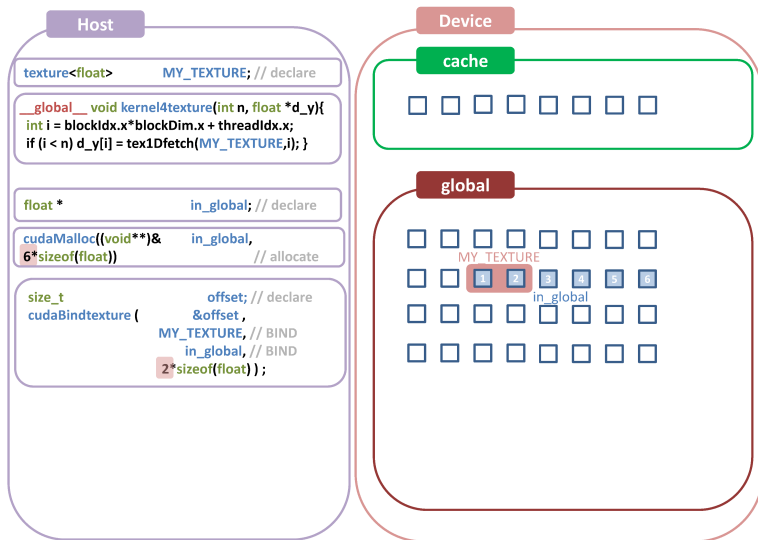
# Texture: visualizzata



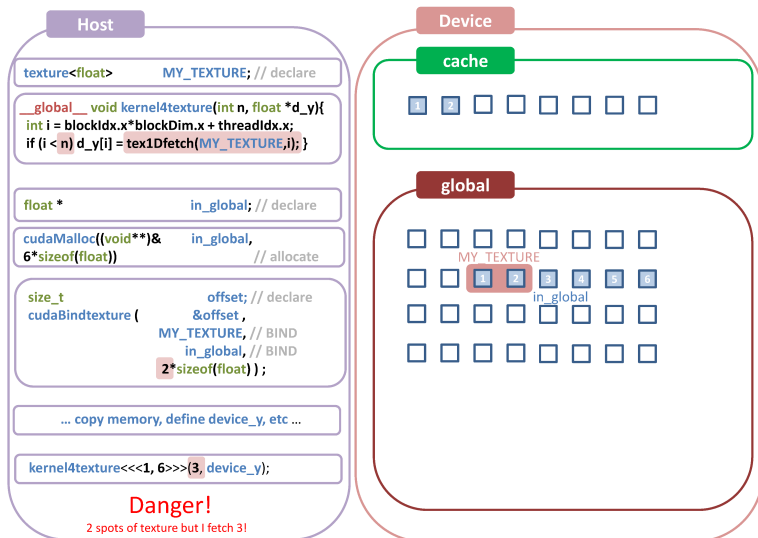
# Texture: visualizzata



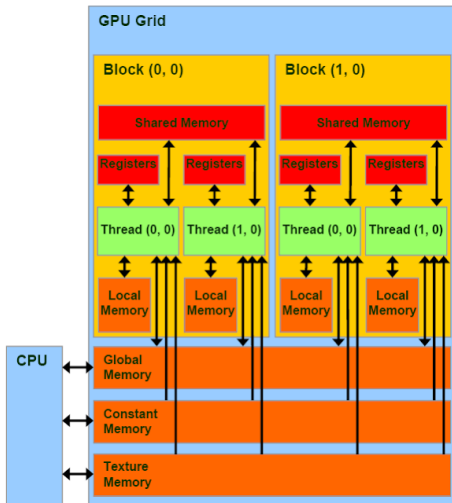
# Texture: visualizzata



# Texture: visualizzata



# Struttura delle memorie: review



- i **registri** (Kepler) sono 255 per thread (64kb). Velocita' di accesso  $\approx 1$  ciclo.
- la **shared** memory ( $\approx 64$  kb per **SM**) e' condivisa da tutti i thread nel blocco: e' **veloce**. I thread di un blocco non accedono alla shared di un altro blocco. Velocita' di accesso  $\approx 5$  cicli.
- la **constant** memory ( $\approx 64$  kb), velocita' di accesso  $\approx 5$  cicli (sulla cache).
- la **local** memory ( $\approx 512$  kb) e' assegnata ad ogni singolo thread, e' solo un sottoinsieme della global: e' **lenta** come la global  $\approx 500$  cicli.
- la **global** memory (DRAM  $\approx 4$ Gb) e' **lenta** ( $\approx 500$  cicli). Bandwidth circa 200 Gb/s. Vista da tutti i thread.
- la **texture** memory ( $\approx 12/48$  kb), cache e coerenza spaziale.

I valori qui riportati sono solo **indicativi**, e servono solo per avere un'idea di quanto sono gli **ordini di grandezza** delle varie memorie associate (ovviamente dipendono dalle **compute capabilities** di ogni scheda).

*Figure:* da <http://cuda-programming.blogspot.it/2013/01/what-is-constant-memory-in-cuda.html>.

# Outline

1 *Struttura Fisica*

2 *Kernel*

3 *Struttura Logica*

4 *Memorie*

5 ***Prodotto Matrici***

6 *Stream*

7 *Esempi e strumenti*

# Prodotto matrice x matrice usando la Global Memory

Supponiamo che  $A$  e  $B$  siano matrici float  $N \times N$ .

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

Un esempio di algoritmo per il prodotto:

- ogni **thread** calcola un elemento della matrice prodotto  $C$ . Dato che deve fare  $N$  prodotti e  $N - 1$  somme:  $O(N)$ .
- Devono essere calcolati  $N^2$  elementi, il totale richiede  $O(N^3)$  operazioni di caricamento da memoria alla ALU
- ogni **thread** puo ottenere solo un elemento in parallelo e memorizzarlo nella `shared memory`
- quando tutti i **thread** hanno caricato di dati di cui hanno bisogno, possono anche accedere a tutti gli elementi che sono stati caricati dagli altri **thread** che appartengono allo stesso blocco, per esempio condividendo una colonna o una riga.
- **PROBLEMA** la memoria `shared` e' piccola (poche decine di kb  $\approx$  48), quindi questo algoritmo puo' funzionare solo per matrici piccole: sarebbe utile pensare in termini di *blocking*...



# Prodotto matrice x matrice usando la Global Memory

Supponiamo che  $A$  e  $B$  siano matrici float  $N \times N$ .

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

Un esempio di algoritmo per il prodotto:

- ogni **thread** calcola un elemento della matrice prodotto  $C$ . Dato che deve fare  $N$  prodotti e  $N - 1$  somme:  $O(N)$ .
- Devono essere calcolati  $N^2$  elementi, il totale richiede  $O(N^3)$  operazioni di caricamento da memoria alla ALU
- ogni **thread** puo ottenere solo un elemento in parallelo e memorizzarlo nella `shared memory`
- quando tutti i **thread** hanno caricato di dati di cui hanno bisogno, possono anche accedere a tutti gli elementi che sono stati caricati dagli altri **thread** che appartengono allo stesso blocco, per esempio condividendo una colonna o una riga.
- **PROBLEMA** la memoria `shared` e' piccola (poche decine di kb  $\approx$  48), quindi questo algoritmo puo' funzionare solo per matrici piccole: sarebbe utile pensare in termini di *blocking*...

# Prodotto matrice x matrice usando la Global Memory

Supponiamo che  $A$  e  $B$  siano matrici float  $N \times N$ .

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

Un esempio di algoritmo per il prodotto:

- ogni **thread** calcola un elemento della matrice prodotto  $C$ . Dato che deve fare  $N$  prodotti e  $N - 1$  somme:  $O(N)$ .
- Devono essere calcolati  $N^2$  elementi, il totale richiede  $O(N^3)$  operazioni di caricamento da memoria alla ALU
- ogni **thread** puo ottenere solo un elemento in parallelo e memorizzarlo nella `shared memory`
- quando tutti i **thread** hanno caricato di dati di cui hanno bisogno, possono anche accedere a tutti gli elementi che sono stati caricati dagli altri **thread** che appartengono allo stesso blocco, per esempio condividendo una colonna o una riga.
- **PROBLEMA** la memoria `shared` e' piccola (poche decine di kb  $\approx$  48), quindi questo algoritmo puo' funzionare solo per matrici piccole: sarebbe utile pensare in termini di *blocking*...

# Prodotto matrice x matrice usando la Global Memory

Supponiamo che  $A$  e  $B$  siano matrici float  $N \times N$ .

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

Un esempio di algoritmo per il prodotto:

- ogni **thread** calcola un elemento della matrice prodotto  $C$ . Dato che deve fare  $N$  prodotti e  $N - 1$  somme:  $O(N)$ .
- Devono essere calcolati  $N^2$  elementi, il totale richiede  $O(N^3)$  operazioni di caricamento da memoria alla ALU
- ogni **thread** puo ottenere solo un elemento in parallelo e memorizzarlo nella `shared memory`
- quando tutti i **thread** hanno caricato di dati di cui hanno bisogno, possono anche accedere a tutti gli elementi che sono stati caricati dagli altri **thread** che appartengono allo stesso blocco, per esempio condividendo una colonna o una riga.
- **PROBLEMA** la memoria `shared` e' piccola (poche decine di kb  $\approx 48$ ), quindi questo algoritmo puo' funzionare solo per matrici piccole: sarebbe utile pensare in termini di *blocking*...

# Prodotto matrice x matrice usando la Global Memory

Supponiamo che  $A$  e  $B$  siano matrici float  $N \times N$ .

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

Un esempio di algoritmo per il prodotto:

- ogni **thread** calcola un elemento della matrice prodotto  $C$ . Dato che deve fare  $N$  prodotti e  $N - 1$  somme:  $O(N)$ .
- Devono essere calcolati  $N^2$  elementi, il totale richiede  $O(N^3)$  operazioni di caricamento da memoria alla ALU
- ogni **thread** puo ottenere solo un elemento in parallelo e memorizzarlo nella `shared memory`
- quando tutti i **thread** hanno caricato di dati di cui hanno bisogno, possono anche accedere a tutti gli elementi che sono stati caricati dagli altri **thread** che appartengono allo stesso blocco, per esempio condividendo una colonna o una riga.
- **PROBLEMA** la memoria `shared` e' piccola (poche decine di kb  $\approx 48$ ), quindi questo algoritmo puo' funzionare solo per matrici piccole: sarebbe utile pensare in termini di *blocking*...

# Prodotto tra matrici usando la Shared

Per prima cosa dividiamo le matrici in **tile** (=tegole) di dimensione  $NB \times NB$ . Dovremo quindi fare prodotti matrice matrice per le **tile** e poi sommare i risultati.

$$C_{ij} = \sum_{s=1}^{N/NB} \left( \sum_{k=1}^{NB} A_{ik}^s B_{kj}^s \right)$$

- 1 l'idea e' che un **block** si prenda in carico una **tile** della matrice prodotto.
- 2 nel **blocco** ogni thread viene usato per calcolare un singolo elemento di matrice
- 3 ogni **blocco** carica dalla **global** alla **shared** una **tile** di  $A_{ik}^s$  e uno di  $B_{kj}^s$  (all'inizio  $s=0$ ).
- 4 **Attenzione** prima di fare i prodotti devo eseguire un `__syncthreads` altrimenti gli e.d.m. delle **tile** potrebbero non essere stati tutti caricati.
- 5 ad ogni **i thread** nel blocco associo due indici  $i$  e  $j$  (che saranno gli indici dell'elemento di matrice da calcolare).
- 6 il singolo **thread** calcola il **prodotto scalare** del pezzo di riga di  $A$  con la sua stessa  $i$  e del pezzo di colonna di  $B$  nella **tile** con il suo indice di colonna  $j$  (che sono nella **shared**):  $C_{ij}^s = \sum_{k=1}^{NB} A_{ik}^s B_{kj}^s$
- 7 una volta fatto il prodotto scalare, ogni thread aspetta che tutti gli altri thread nel blocco abbiano eseguito finito il loro prodotto.
- 8 A questo punto il **blocco** passa alla prossima coppia di **tile**:  $s = s + 1$  e si torna al punto 4.

Con questa procedura, ogni singolo elemento di matrice di  $A$  e  $B$  viene letto dalla **global** tante volte quante sono le **tile** invece che tante volte quante sono le colonne della matrice  $A$

Per una spiegazione grafica: <http://www.es.ele.tue.nl/~mwijtvliet/5KK73/?page=mmcuda>

# Prodotto tra matrici usando la Shared

Per prima cosa dividiamo le matrici in **tile** (=tegole) di dimensione  $NB \times NB$ . Dovremo quindi fare prodotti matrice matrice per le **tile** e poi sommare i risultati.

$$C_{ij} = \sum_{s=1}^{N/NB} \left( \sum_{k=1}^{NB} A_{ik}^s B_{kj}^s \right)$$

- 1 l'idea e' che un **block** si prenda in carico una **tile** della matrice prodotto.
- 2 nel **blocco** ogni thread viene usato per calcolare un singolo elemento di matrice
- 3 ogni **blocco** carica dalla `global` alla `shared` una **tile** di  $A_{ik}^s$  e uno di  $B_{kj}^s$  (all'inizio  $s=0$ ).
- 4 **Attenzione** prima di fare i prodotti devo eseguire un `__syncthreads` altrimenti gli e.d.m. delle **tile** potrebbero non essere stati tutti caricati.
- 5 ad ogni **i thread** nel blocco associo due indici  $i$  e  $j$  (che saranno gli indici dell'elemento di matrice da calcolare).
- 6 il singolo **thread** calcola il **prodotto scalare** del pezzo di riga di  $A$  con la sua stessa  $i$  e del pezzo di colonna di  $B$  nella **tile** con il suo indice di colonna  $j$  (che sono nella `shared`):  $C_{ij}^s = \sum_{k=1}^{NB} A_{ik}^s B_{kj}^s$
- 7 una volta fatto il prodotto scalare, ogni thread aspetta che tutti gli altri thread nel blocco abbiano eseguito finito il loro prodotto.
- 8 A questo punto il **blocco** passa alla prossima coppia di **tile**:  $s = s + 1$  e si torna al punto 4.

Con questa procedura, ogni singolo elemento di matrice di  $A$  e  $B$  viene letto dalla `global` tante volte quante sono le **tile** invece che tante volte quante sono le colonne della matrice  $A$

Per una spiegazione grafica: <http://www.es.ele.tue.nl/~mwijtvliet/5KK73/?page=mmcuda>

# Prodotto tra matrici usando la Shared

Per prima cosa dividiamo le matrici in **tile** (=tegole) di dimensione  $NB \times NB$ . Dovremo quindi fare prodotti matrice matrice per le **tile** e poi sommare i risultati.

$$C_{ij} = \sum_{s=1}^{N/NB} \left( \sum_{k=1}^{NB} A_{ik}^s B_{kj}^s \right)$$

- 1 l'idea e' che un **block** si prenda in carico una **tile** della matrice prodotto.
- 2 nel **blocco** ogni thread viene usato per calcolare un singolo elemento di matrice
- 3 ogni **blocco** carica dalla `global` alla `shared` una **tile** di  $A_{ik}^s$  e uno di  $B_{kj}^s$  (all'inizio  $s=0$ ).
- 4 **Attenzione** prima di fare i prodotti devo eseguire un `__syncthreads` altrimenti gli e.d.m. delle **tile** potrebbero non essere stati tutti caricati.
- 5 ad ogni **i thread** nel blocco associo due indici  $i$  e  $j$  (che saranno gli indici dell'elemento di matrice da calcolare).
- 6 il singolo **thread** calcola il **prodotto scalare** del pezzo di riga di  $A$  con la sua stessa  $i$  e del pezzo di colonna di  $B$  nella **tile** con il suo indice di colonna  $j$  (che sono nella `shared`):  $C_{ij}^s = \sum_{k=1}^{NB} A_{ik}^s B_{kj}^s$
- 7 una volta fatto il prodotto scalare, ogni thread aspetta che tutti gli altri thread nel blocco abbiano eseguito finito il loro prodotto.
- 8 A questo punto il **blocco** passa alla prossima coppia di **tile**:  $s = s + 1$  e si torna al punto 4.

Con questa procedura, ogni singolo elemento di matrice di  $A$  e  $B$  viene letto dalla `global` tante volte quante sono le **tile** invece che tante volte quante sono le colonne della matrice  $A$

Per una spiegazione grafica: <http://www.es.ele.tue.nl/~mwijtvliet/5KK73/?page=mmcuda>

# Prodotto tra matrici usando la Shared

Per prima cosa dividiamo le matrici in **tile** (=tegole) di dimensione  $NB \times NB$ . Dovremo quindi fare prodotti matrice matrice per le **tile** e poi sommare i risultati.

$$C_{ij} = \sum_{s=1}^{N/NB} \left( \sum_{k=1}^{NB} A_{ik}^s B_{kj}^s \right)$$

- 1 l'idea e' che un **block** si prenda in carico una **tile** della matrice prodotto.
- 2 nel **blocco** ogni thread viene usato per calcolare un singolo elemento di matrice
- 3 ogni **blocco** carica dalla `global` alla `shared` una **tile** di  $A_{ik}^s$  e uno di  $B_{kj}^s$  (all'inizio  $s=0$ ).
- 4 **Attenzione** prima di fare i prodotti devo eseguire un `__syncthreads` altrimenti gli e.d.m. delle **tile** potrebbero non essere stati tutti caricati.
- 5 ad ogni **i thread** nel blocco associo due indici  $i$  e  $j$  (che saranno gli indici dell'elemento di matrice da calcolare).
- 6 il singolo **thread** calcola il **prodotto scalare** del pezzo di riga di  $A$  con la sua stessa  $i$  e del pezzo di colonna di  $B$  nella **tile** con il suo indice di colonna  $j$  (che sono nella `shared`):  $C_{ij}^s = \sum_{k=1}^{NB} A_{ik}^s B_{kj}^s$
- 7 una volta fatto il prodotto scalare, ogni thread aspetta che tutti gli altri thread nel blocco abbiano eseguito finito il loro prodotto.
- 8 A questo punto il **blocco** passa alla prossima coppia di **tile**:  $s = s + 1$  e si torna al punto 4.

Con questa procedura, ogni singolo elemento di matrice di  $A$  e  $B$  viene letto dalla `global` tante volte quante sono le **tile** invece che tante volte quante sono le colonne della matrice  $A$

Per una spiegazione grafica: <http://www.es.ele.tue.nl/~mwijtvliet/5KK73/?page=mmcuda>



# Prodotto tra matrici usando la Shared

Per prima cosa dividiamo le matrici in **tile** (=tegole) di dimensione  $NB \times NB$ . Dovremo quindi fare prodotti matrice matrice per le **tile** e poi sommare i risultati.

$$C_{ij} = \sum_{s=1}^{N/NB} \left( \sum_{k=1}^{NB} A_{ik}^s B_{kj}^s \right)$$

- 1 l'idea e' che un **block** si prenda in carico una **tile** della matrice prodotto.
- 2 nel **blocco** ogni thread viene usato per calcolare un singolo elemento di matrice
- 3 ogni **blocco** carica dalla `global` alla `shared` una **tile** di  $A_{ik}^s$  e uno di  $B_{kj}^s$  (all'inizio  $s=0$ ).
- 4 **Attenzione** prima di fare i prodotti devo eseguire un `__syncthreads` altrimenti gli e.d.m. delle **tile** potrebbero non essere stati tutti caricati.
- 5 ad ogni **i thread** nel blocco associo due indici  $i$  e  $j$  (che saranno gli indici dell'elemento di matrice da calcolare).
- 6 il singolo **thread** calcola il **prodotto scalare** del pezzo di riga di  $A$  con la sua stessa  $i$  e del pezzo di colonna di  $B$  nella **tile** con il suo indice di colonna  $j$  (che sono nella `shared`):  $C_{ij}^s = \sum_{k=1}^{NB} A_{ik}^s B_{kj}^s$
- 7 una volta fatto il prodotto scalare, ogni thread aspetta che tutti gli altri thread nel blocco abbiano eseguito finito il loro prodotto.
- 8 A questo punto il **blocco** passa alla prossima coppia di **tile**:  $s = s + 1$  e si torna al punto 4.

Con questa procedura, ogni singolo elemento di matrice di  $A$  e  $B$  viene letto dalla `global` tante volte quante sono le **tile** invece che tante volte quante sono le colonne della matrice  $A$

Per una spiegazione grafica: <http://www.es.ele.tue.nl/~mwijtvliet/5KK73/?page=mmcuda>

## Prodotto tra matrici usando la Shared

Per prima cosa dividiamo le matrici in **tile** (=tegole) di dimensione  $NB \times NB$ . Dovremo quindi fare prodotti matrice matrice per le **tile** e poi sommare i risultati.

$$C_{ij} = \sum_{s=1}^{N/NB} \left( \sum_{k=1}^{NB} A_{ik}^s B_{kj}^s \right)$$

- 1 l'idea e' che un **block** si prenda in carico una **tile** della matrice prodotto.
- 2 nel **blocco** ogni thread viene usato per calcolare un singolo elemento di matrice
- 3 ogni **blocco** carica dalla `global` alla `shared` una **tile** di  $A_{ik}^s$  e uno di  $B_{kj}^s$  (all'inizio  $s=0$ ).
- 4 **Attenzione** prima di fare i prodotti devo eseguire un `__syncthreads` altrimenti gli e.d.m. delle **tile** potrebbero non essere stati tutti caricati.
- 5 ad ogni **i thread** nel blocco associo due indici  $i$  e  $j$  (che saranno gli indici dell'elemento di matrice da calcolare).
- 6 il singolo **thread** calcola il **prodotto scalare** del pezzo di riga di  $A$  con la sua stessa  $i$  e del pezzo di colonna di  $B$  nella **tile** con il suo indice di colonna  $j$  (che sono nella `shared`):  $C_{ij}^s = \sum_{k=1}^{NB} A_{ik}^s B_{kj}^s$
- 7 una volta fatto il prodotto scalare, ogni thread aspetta che tutti gli altri thread nel blocco abbiano eseguito finito il loro prodotto.
- 8 A questo punto il **blocco** passa alla prossima coppia di **tile**:  $s = s + 1$  e si torna al punto 4.

Con questa procedura, ogni singolo elemento di matrice di  $A$  e  $B$  viene letto dalla `global` tante volte quante sono le **tile** invece che tante volte quante sono le colonne della matrice  $A$

Per una spiegazione grafica: <http://www.es.ele.tue.nl/~mwijtvliet/5KK73/?page=mmcuda>

## Prodotto tra matrici usando la Shared

Per prima cosa dividiamo le matrici in **tile** (=tegole) di dimensione  $NB \times NB$ . Dovremo quindi fare prodotti matrice matrice per le **tile** e poi sommare i risultati.

$$C_{ij} = \sum_{s=1}^{N/NB} \left( \sum_{k=1}^{NB} A_{ik}^s B_{kj}^s \right)$$

- 1 l'idea e' che un **block** si prenda in carico una **tile** della matrice prodotto.
- 2 nel **blocco** ogni thread viene usato per calcolare un singolo elemento di matrice
- 3 ogni **blocco** carica dalla `global` alla `shared` una **tile** di  $A_{ik}^s$  e uno di  $B_{kj}^s$  (all'inizio  $s=0$ ).
- 4 **Attenzione** prima di fare i prodotti devo eseguire un `__syncthreads` altrimenti gli e.d.m. delle **tile** potrebbero non essere stati tutti caricati.
- 5 ad ogni **i thread** nel blocco associo due indici  $i$  e  $j$  (che saranno gli indici dell'elemento di matrice da calcolare).
- 6 il singolo **thread** calcola il **prodotto scalare** del pezzo di riga di  $A$  con la sua stessa  $i$  e del pezzo di colonna di  $B$  nella **tile** con il suo indice di colonna  $j$  (che sono nella `shared`):  $C_{ij}^s = \sum_{k=1}^{NB} A_{ik}^s B_{kj}^s$
- 7 una volta fatto il prodotto scalare, ogni thread aspetta che tutti gli altri thread nel blocco abbiano eseguito finito il loro prodotto.
- 8 A questo punto il **blocco** passa alla prossima coppia di **tile**:  $s = s + 1$  e si torna al punto 4.

Con questa procedura, ogni singolo elemento di matrice di  $A$  e  $B$  viene letto dalla `global` tante volte quante sono le **tile** invece che tante volte quante sono le colonne della matrice  $A$

Per una spiegazione grafica: <http://www.es.ele.tue.nl/~mwijtvliet/5KK73/?page=mmcuda>

## Prodotto tra matrici usando la Shared

Per prima cosa dividiamo le matrici in **tile** (=tegole) di dimensione  $NB \times NB$ . Dovremo quindi fare prodotti matrice matrice per le **tile** e poi sommare i risultati.

$$C_{ij} = \sum_{s=1}^{N/NB} \left( \sum_{k=1}^{NB} A_{ik}^s B_{kj}^s \right)$$

- 1 l'idea e' che un **block** si prenda in carico una **tile** della matrice prodotto.
- 2 nel **blocco** ogni thread viene usato per calcolare un singolo elemento di matrice
- 3 ogni **blocco** carica dalla `global` alla `shared` una **tile** di  $A_{ik}^s$  e uno di  $B_{kj}^s$  (all'inizio  $s=0$ ).
- 4 **Attenzione** prima di fare i prodotti devo eseguire un `__syncthreads` altrimenti gli e.d.m. delle **tile** potrebbero non essere stati tutti caricati.
- 5 ad ogni **i thread** nel blocco associo due indici  $i$  e  $j$  (che saranno gli indici dell'elemento di matrice da calcolare).
- 6 il singolo **thread** calcola il **prodotto scalare** del pezzo di riga di  $A$  con la sua stessa  $i$  e del pezzo di colonna di  $B$  nella **tile** con il suo indice di colonna  $j$  (che sono nella `shared`):  $C_{ij}^s = \sum_{k=1}^{NB} A_{ik}^s B_{kj}^s$
- 7 una volta fatto il prodotto scalare, ogni thread aspetta che tutti gli altri thread nel blocco abbiano eseguito finito il loro prodotto.
- 8 A questo punto il **blocco** passa alla prossima coppia di **tile**:  $s = s + 1$  e si torna al punto 4.

Con questa procedura, ogni singolo elemento di matrice di  $A$  e  $B$  viene letto dalla `global` tante volte quante sono le **tile** invece che tante volte quante sono le colonne della matrice  $A$

Per una spiegazione grafica: <http://www.es.ele.tue.nl/~mwijtvliet/5KK73/?page=mmcuda>

## Prodotto tra matrici usando la Shared

Per prima cosa dividiamo le matrici in **tile** (=tegole) di dimensione  $NB \times NB$ . Dovremo quindi fare prodotti matrice matrice per le **tile** e poi sommare i risultati.

$$C_{ij} = \sum_{s=1}^{N/NB} \left( \sum_{k=1}^{NB} A_{ik}^s B_{kj}^s \right)$$

- 1 l'idea e' che un **block** si prenda in carico una **tile** della matrice prodotto.
- 2 nel **blocco** ogni thread viene usato per calcolare un singolo elemento di matrice
- 3 ogni **blocco** carica dalla `global` alla `shared` una **tile** di  $A_{ik}^s$  e uno di  $B_{kj}^s$  (all'inizio  $s=0$ ).
- 4 **Attenzione** prima di fare i prodotti devo eseguire un `__syncthreads` altrimenti gli e.d.m. delle **tile** potrebbero non essere stati tutti caricati.
- 5 ad ogni **i thread** nel blocco associo due indici  $i$  e  $j$  (che saranno gli indici dell'elemento di matrice da calcolare).
- 6 il singolo **thread** calcola il **prodotto scalare** del pezzo di riga di  $A$  con la sua stessa  $i$  e del pezzo di colonna di  $B$  nella **tile** con il suo indice di colonna  $j$  (che sono nella `shared`):  $C_{ij}^s = \sum_{k=1}^{NB} A_{ik}^s B_{kj}^s$
- 7 una volta fatto il prodotto scalare, ogni thread aspetta che tutti gli altri thread nel blocco abbiano eseguito finito il loro prodotto.
- 8 A questo punto il **blocco** passa alla prossima coppia di **tile**:  $s = s + 1$  e si torna al punto 4.

Con questa procedura, ogni singolo elemento di matrice di  $A$  e  $B$  viene letto dalla `global` tante volte quante sono le `tile` invece che tante volte quante sono le colonne della matrice  $A$

Per una spiegazione grafica: <http://www.es.ele.tue.nl/~mwijtvliet/5KK73/?page=mmcuda>

## Prodotto tra matrici usando la Shared

Per prima cosa dividiamo le matrici in **tile** (=tegole) di dimensione  $NB \times NB$ . Dovremo quindi fare prodotti matrice matrice per le **tile** e poi sommare i risultati.

$$C_{ij} = \sum_{s=1}^{N/NB} \left( \sum_{k=1}^{NB} A_{ik}^s B_{kj}^s \right)$$

- 1 l'idea e' che un **block** si prenda in carico una **tile** della matrice prodotto.
- 2 nel **blocco** ogni thread viene usato per calcolare un singolo elemento di matrice
- 3 ogni **blocco** carica dalla `global` alla `shared` una **tile** di  $A_{ik}^s$  e uno di  $B_{kj}^s$  (all'inizio  $s=0$ ).
- 4 **Attenzione** prima di fare i prodotti devo eseguire un `__syncthreads` altrimenti gli e.d.m. delle **tile** potrebbero non essere stati tutti caricati.
- 5 ad ogni **i thread** nel blocco associo due indici  $i$  e  $j$  (che saranno gli indici dell'elemento di matrice da calcolare).
- 6 il singolo **thread** calcola il **prodotto scalare** del pezzo di riga di  $A$  con la sua stessa  $i$  e del pezzo di colonna di  $B$  nella **tile** con il suo indice di colonna  $j$  (che sono nella `shared`):  $C_{ij}^s = \sum_{k=1}^{NB} A_{ik}^s B_{kj}^s$
- 7 una volta fatto il prodotto scalare, ogni thread aspetta che tutti gli altri thread nel blocco abbiano eseguito finito il loro prodotto.
- 8 A questo punto il **blocco** passa alla prossima coppia di **tile**:  $s = s + 1$  e si torna al punto 4.

Con questa procedura, ogni singolo elemento di matrice di  $A$  e  $B$  viene letto dalla `global` tante volte quante sono le **tile** invece che tante volte quante sono le colonne della matrice  $A$ !

Per una spiegazione grafica: <http://www.es.ele.tue.nl/~mwijtvliet/5KK73/?page=mmcuda>

# Outline

- 1 *Struttura Fisica*
- 2 *Kernel*
- 3 *Struttura Logica*
- 4 *Memorie*
- 5 *Prodotto Matrici*
- 6 *Stream***
- 7 *Esempi e strumenti*

# Lo stream 0

Consideriamo la sincronizzazione a livello di sistema:

- **molte** chiamate a CUDA API e i **tutti** i kernel sono asincroni rispetto all'**HOST**, ergo restituiscono immediatamente il controllo all'**host** che puo' lavorare contemporaneamente al **device**.
- D'altra parte se abbiamo un codice con 2 kernel lanciati uno subito dopo l'altro:
  - `kernel1<<<X1, Y1>>> (...)`; il **kernel1** comincia l'esecuzione, la CPU continua al nuovo statement
  - `kernel2<<<X2, Y2>>> (...)`; il **kernel2** e' messo nella coda e comincerà a funzionare DOPO che il **kernel1** avrà finito. La CPU continua al prossimo statement
- Se non viene specificato altrimenti, **tutti** i comandi di CUDA finiscono nel cosiddetto **stream 0**, e vengono messi uno dietro l'altro rispetto al **device**, come visto sopra.
- In generale uno **stream** in CUDA, e' un insieme di comandi CUDA che viene eseguito nell'ordine di scrittura. Lo stream in pratica definisce una **sincronizzazione** implicita tra comandi che girano sul **device**.



# Lo stream 0

Consideriamo la sincronizzazione a livello di sistema:

- **molte** chiamate a CUDA API e i **tutti** i kernel sono asincroni rispetto all'**HOST**, ergo restituiscono immediatamente il controllo all'**host** che puo' lavorare contemporaneamente al **device**.
- D'altra parte se abbiamo un codice con 2 kernel lanciati uno subito dopo l'altro:
  - `kernel1<<<X1,Y1>>> (...)` ; il **kernel1** comincia l'esecuzione, la CPU continua al nuovo statement
  - `kernel2<<<X2,Y2>>> (...)` ; il **kernel2** e' messo nella coda e comincerà a funzionare DOPO che il **kernel1** avrà finito. La CPU continua al prossimo statement
- Se non viene specificato altrimenti, **tutti** i comandi di CUDA finiscono nel cosiddetto **stream 0**, e vengono messi uno dietro l'altro rispetto al **device**, come visto sopra.
- In generale uno **stream** in CUDA, e' un insieme di comandi CUDA che viene eseguito nell'ordine di scrittura. Lo stream in pratica definisce una **sincronizzazione** implicita tra comandi che girano sul **device**.

# Lo stream 0

Consideriamo la sincronizzazione a livello di sistema:

- **molte** chiamate a CUDA API e i **tutti** i kernel sono asincroni rispetto all'**HOST**, ergo restituiscono immediatamente il controllo all'**host** che puo' lavorare contemporaneamente al **device**.
- D'altra parte se abbiamo un codice con 2 kernel lanciati uno subito dopo l'altro:
  - `kernel1<<<X1,Y1>>> (...)` ; il **kernel1** comincia l'esecuzione, la CPU continua al nuovo statement
  - `kernel2<<<X2,Y2>>> (...)` ; il **kernel2** e' messo nella coda e comincerà a funzionare DOPO che il **kernel1** avrà finito. La CPU continua al prossimo statement
- Se non viene specificato altrimenti, **tutti** i comandi di CUDA finiscono nel cosiddetto **stream 0**, e vengono messi uno dietro l'altro rispetto al **device**, come visto sopra.
- In generale uno **stream** in CUDA, e' un insieme di comandi CUDA che viene eseguito nell'ordine di scrittura. Lo stream in pratica definisce una **sincronizzazione** implicita tra comandi che girano sul **device**.

# Lo stream 0

Consideriamo la sincronizzazione a livello di sistema:

- **molte** chiamate a CUDA API e i **tutti** i kernel sono asincroni rispetto all'**HOST**, ergo restituiscono immediatamente il controllo all'**host** che puo' lavorare contemporaneamente al **device**.
- D'altra parte se abbiamo un codice con 2 kernel lanciati uno subito dopo l'altro:
  - `kernel1<<<X1, Y1>>> (...)` ; il **kernel1** comincia l'esecuzione, la CPU continua al nuovo statement
  - `kernel2<<<X2, Y2>>> (...)` ; il **kernel2** e' messo nella coda e comincerà a funzionare DOPO che il **kernel1** avrà finito. La CPU continua al prossimo statement
- Se non viene specificato altrimenti, **tutti** i comandi di CUDA finiscono nel cosiddetto **stream 0**, e vengono messi uno dietro l'altro rispetto al **device**, come visto sopra.
- In generale uno **stream** in CUDA, e' un insieme di comandi CUDA che viene eseguito nell'ordine di scrittura. Lo stream in pratica definisce una **sincronizzazione** implicita tra comandi che girano sul **device**.

# Molti stream

Nelle versioni piu' recenti delle compute capabilities ( $> 2.0$ ) e' possibile lanciare piu' **kernel** concorrenti. Questo viene ottenuto costruendo **piu' di uno stream**

- Il vantaggio e' che comandi di stream differenti possono lavorare in modo **concorrente** sulla GPU. Ergo **puo' esserci** sovrapposizione tra due comandi che appartengono a due diversi cuda stream.
- Chiaramente i **limiti di utilizzo delle GPU** continuano ad esistere, per questo motivo se una persona ha scritto un codice in cui il kernel utilizza il 100% delle risorse della GPU, utilizzare piu' kernel contemporaneamente sara' de facto inutile, in quanto questi kernel verranno serializzati.
- In pratica quindi ogni **stream** e' un insieme di comandi ordinati (uno di seguito all'altro). Comandi di uno stesso stream seguono una logica FIFO, mentre sono possono agire in modo concorrente (con le dovute precauzioni) rispetto a comandi di stream differenti.

Quali vantaggi ci sono nell'utilizzare degli stream?

- Utilizzo migliore delle risorse
- **Concorrenza** di esecuzione dei kernel

# Molti stream

Nelle versioni piu' recenti delle compute capabilities ( $> 2.0$ ) e' possibile lanciare piu' **kernel** concorrenti. Questo viene ottenuto costruendo **piu' di uno stream**

- Il vantaggio e' che comandi di stream differenti possono lavorare in modo **concorrente** sulla GPU. Ergo **puo' esserci** sovrapposizione tra due comandi che appartengono a due diversi cuda stream.
- Chiaramente i **limiti di utilizzo delle GPU** continuano ad esistere, per questo motivo se una persona ha scritto un codice in cui il kernel utilizza il 100% delle risorse della GPU, utilizzare piu' kernel contemporaneamente sara' de facto inutile, in quanto questi kernel verranno serializzati.
- In pratica quindi ogni **stream** e' un insieme di comandi ordinati (uno di seguito all'altro). Comandi di uno stesso stream seguono una logica FIFO, mentre sono possono agire in modo concorrente (con le dovute precauzioni) rispetto a comandi di stream differenti.

Quali vantaggi ci sono nell'utilizzare degli stream?

- Utilizzo migliore delle risorse
- **Concorrenza** di esecuzione dei kernel

# Molti stream

Nelle versioni piu' recenti delle compute capabilities ( $> 2.0$ ) e' possibile lanciare piu' **kernel** concorrenti. Questo viene ottenuto costruendo **piu' di uno stream**

- Il vantaggio e' che comandi di stream differenti possono lavorare in modo **concorrente** sulla GPU. Ergo **puo' esserci** sovrapposizione tra due comandi che appartengono a due diversi cuda stream.
- Chiaramente i **limiti di utilizzo delle GPU** continuano ad esistere, per questo motivo se una persona ha scritto un codice in cui il kernel utilizza il 100% delle risorse della GPU, utilizzare piu' kernel contemporaneamente sara' de facto inutile, in quanto questi kernel verranno serializzati.
- In pratica quindi ogni **stream** e' un insieme di comandi ordinati (uno di seguito all'altro). Comandi di uno stesso stream seguono una logica FIFO, mentre sono possono agire in modo concorrente (con le dovute precauzioni) rispetto a comandi di stream differenti.

Quali vantaggi ci sono nell'utilizzare degli stream?

- Utilizzo migliore delle risorse
- **Concorrenza** di esecuzione dei kernel

# Molti stream

Nelle versioni piu' recenti delle compute capabilities ( $> 2.0$ ) e' possibile lanciare piu' **kernel** concorrenti. Questo viene ottenuto costruendo **piu' di uno stream**

- Il vantaggio e' che comandi di stream differenti possono lavorare in modo **concorrente** sulla GPU. Ergo **puo' esserci** sovrapposizione tra due comandi che appartengono a due diversi cuda stream.
- Chiaramente i **limiti di utilizzo delle GPU** continuano ad esistere, per questo motivo se una persona ha scritto un codice in cui il kernel utilizza il 100% delle risorse della GPU, utilizzare piu' kernel contemporaneamente sara' de facto inutile, in quanto questi kernel verranno serializzati.
- In pratica quindi ogni **stream** e' un insieme di comandi ordinati (uno di seguito all'altro). Comandi di uno stesso stream seguono una logica FIFO, mentre sono possono agire in modo concorrente (con le dovute precauzioni) rispetto a comandi di stream differenti.

Quali vantaggi ci sono nell'utilizzare degli stream?

- Utilizzo migliore delle risorse
- **Concorrenza** di esecuzione dei kernel

# Molti stream

Nelle versioni piu' recenti delle compute capabilities ( $> 2.0$ ) e' possibile lanciare piu' **kernel** concorrenti. Questo viene ottenuto costruendo **piu' di uno stream**

- Il vantaggio e' che comandi di stream differenti possono lavorare in modo **concorrente** sulla GPU. Ergo **puo' esserci** sovrapposizione tra due comandi che appartengono a due diversi cuda stream.
- Chiaramente i **limiti di utilizzo delle GPU** continuano ad esistere, per questo motivo se una persona ha scritto un codice in cui il kernel utilizza il 100% delle risorse della GPU, utilizzare piu' kernel contemporaneamente sara' de facto inutile, in quanto questi kernel verranno serializzati.
- In pratica quindi ogni **stream** e' un insieme di comandi ordinati (uno di seguito all'altro). Comandi di uno stesso stream seguono una logica FIFO, mentre sono possono agire in modo concorrente (con le dovute precauzioni) rispetto a comandi di stream differenti.

Quali vantaggi ci sono nell'utilizzare degli stream?

- Utilizzo migliore delle risorse
- **Concorrenza** di esecuzione dei kernel



# Comandi per gli stream

Uno stream e' un oggetto di CUDA, per questo va dichiarato, allocato, etc..

- `cudaStream_t mioStream;` dichiara una *handle* ad uno stream (il suo nome...)
- `cudaStreamCreate (mioStream);` e' una funzione che definisce lo stream
- `cudaStreamDestroy (mioStream);` e' una funzione fa due cose:
  - 1 Sincronizza l'host finche' lo stream non ha finito (ergo l'host non continua fino a che lo stream ha terminato l'esecuzione)
  - 2 deallocca lo stream

Lo stream e' il quarto parametro che si inserisce nelle *triple angle brackets*

```
mioKernel <<< mioGrid, mioBlocco, memoria_shared, mioStream >>(arg1, arg2);
```

L'oggetto "stream" puo' essere passato ad alcune CUDA API. Per esempio, esiste un altro modo di copiare la memoria oltre al `cudaMemcpy`:

```
cudaMemcpyAsync(dst, src, size, direzione, mioStream); // vedremo poi che e' utile
```

# Comandi per gli stream

Uno stream e' un oggetto di CUDA, per questo va dichiarato, allocato, etc..

- `cudaStream_t mioStream;` dichiara una *handle* ad uno stream (il suo nome...)
- `cudaStreamCreate (mioStream);` e' una funzione che definisce lo stream
- `cudaStreamDestroy (mioStream);` e' una funzione fa due cose:
  - 1 Sincronizza l'host finche' lo stream non ha finito (ergo l'host non continua fino a che lo stream ha terminato l'esecuzione)
  - 2 deallocca lo stream

Lo stream e' il **quarto** parametro che si inserisce nelle **triple angle brackets**

```
mioKernel <<< miaGrid, mioBlocco, memoria_shared, mioStream >>(arg1, arg2);
```

L'oggetto "stream" puo' essere passato ad alcune CUDA API. Per esempio, esiste un altro modo di copiare la memoria oltre al `cudaMemcpy`:

```
cudaMemcpyAsync(dst, src, size, direzione, mioStream); // vedremo poi che e' utile
```

# Comandi per gli stream

Uno stream e' un oggetto di CUDA, per questo va dichiarato, allocato, etc..

- `cudaStream_t mioStream;` dichiara una *handle* ad uno stream (il suo nome...)
- `cudaStreamCreate (mioStream);` e' una funzione che definisce lo stream
- `cudaStreamDestroy (mioStream);` e' una funzione fa due cose:
  - 1 **Sincronizza** l'host finche' lo stream non ha finito (ergo l'host non continua fino a che lo stream ha terminato l'esecuzione)
  - 2 **dealloca** lo stream

Lo stream e' il **quarto** parametro che si inserisce nelle **triple angle brackets**

```
mioKernel <<< mioGrid, mioBlocco, memoria_shared, mioStream >>(arg1, arg2);
```

L'oggetto "stream" puo' essere passato ad alcune CUDA API. Per esempio, esiste un altro modo di copiare la memoria oltre al `cudaMemcpy`:

```
cudaMemcpyAsync(dst, src, size, direzione, mioStream); // vedremo poi che e' utile
```

# Comandi per gli stream

Uno stream e' un oggetto di CUDA, per questo va dichiarato, allocato, etc..

- `cudaStream_t mioStream;` dichiara una *handle* ad uno stream (il suo nome...)
- `cudaStreamCreate (mioStream);` e' una funzione che definisce lo stream
- `cudaStreamDestroy (mioStream);` e' una funzione fa due cose:
  - 1 **Sincronizza** l'host finche' lo stream non ha finito (ergo l'host non continua fino a che lo stream ha terminato l'esecuzione)
  - 2 **dealloca** lo stream

Lo stream e' il **quarto** parametro che si inserisce nelle **triple angle brackets**

```
mioKernel <<< miaGrid, mioBlocco, memoria_shared, mioStream >>(arg1, arg2);
```

L'oggetto "stream" puo' essere passato ad alcune CUDA API. Per esempio, esiste un altro modo di copiare la memoria oltre al `cudaMemcpy`:

```
cudaMemcpyAsync(dst, src, size, direzione, mioStream); // vedremo poi che e' utile
```

# Comandi per gli stream

Uno stream e' un oggetto di CUDA, per questo va dichiarato, allocato, etc..

- `cudaStream_t mioStream;` dichiara una *handle* ad uno stream (il suo nome...)
- `cudaStreamCreate (mioStream);` e' una funzione che definisce lo stream
- `cudaStreamDestroy (mioStream);` e' una funzione fa due cose:
  - 1 **Sincronizza** l'host finche' lo stream non ha finito (ergo l'host non continua fino a che lo stream ha terminato l'esecuzione)
  - 2 **dealloca** lo stream

Lo stream e' il **quarto** parametro che si inserisce nelle **triple angle brackets**

```
mioKernel <<< miaGrid, mioBlocco, memoria_shared, mioStream >>(arg1, arg2);
```

L'oggetto "stream" puo' essere passato ad alcune CUDA API. Per esempio, esiste un altro modo di copiare la memoria oltre al `cudaMemcpy`:

```
cudaMemcpyAsync(dst, src, size, direzione, mioStream); // vedremo poi che e' utile
```

# cudaMemcpyAsync

Tutti gli informatici sanno che la memoria puo' essere:

- **pageable**: puo' essere spostata nella memoria virtuale
- **page locked** (pinned): **NON** puo' essere spostata nella memoria virtuale

Se la memoria e' allocata (invece che con `cudaMalloc`) tramite `cudaHostAlloc()`, allora diventa **page locked**.

```

cudaHostAlloc( (void**)&host_a,
               FULL_DATA_SIZE * sizeof(int),
               cudaHostAllocDefault ); // memoria page locked

cudaMemcpyAsync( dev_a, host_a+i, // copia asincrona
                 N * sizeof(int),
                 cudaMemcpyHostToDevice, // direzione della copia
                 qualeStream ); // in quale stream va questo comando?

```

- 1 `cudaMemcpyAsync()` e' **asincrono** rispetto all'**HOST** (ergo non appena chiamato, l'host riprende e a lavorare)
- 2 `cudaMemcpyAsync()` e' (parzialmente) **asincrono** anche rispetto a **CUDA**, nel senso che puo' operare se nel frattempo un kernel di un **altro** stream sta gia' girando!

Giusto per ricordare:

`cudaMemcpy` invece e' **sincrono** sia rispetto all'HOST che rispetto a CUDA.

## *Esempi sugli stream*

Vediamo come usare gli stream:

`S4158-cuda-streams-best-practices-common-pitfalls.pdf`

## Attenzione allo stream 0!

- lo **stream 0** (quello di default) ha una **sincronizzazione differente** rispetto agli altri stream
- ovvero: **tutti gli altri** stream devono aspettare che finisca (a meno di avere costruito gli altri stream in modo particolare)
- se lancio vari kernel nel mio codice e, nel primo lancio di kernel, non metto il 4to ingresso nelle triple angle brackets, questo vuol dire che quel kernel e' **automaticamente** nello stream 0.
- lo stesso vale per le CUDA api che hanno come parametro **OPZIONALE** lo stream. Se non viene inserito questo parametro, CUDA **accende** lo **stream 0**, creando problemi a tutti gli altri stream!

Per esempio:

```
miokernel0 <<< mioGrid0, mioBlocco0, 0 >>> (argomento0, 3)  
miokernel1 <<< mioGrid1, mioBlocco1, 0, mioStream1 >>> (argomento, altroArgomento);  
miokernel2 <<< mioGrid2, mioBlocco2, 0, mioStream2 >>> (argom, argomento);
```

Il primo kernel, **sembra innocuo**, ma in pratica, non avendo il quarto ingresso nelle <<< >>> appartiene allo stream 0 e quindi "blocca" i kernel seguenti!



## Attenzione allo stream 0!

- lo **stream 0** (quello di default) ha una **sincronizzazione differente** rispetto agli altri stream
- ovvero: **tutti gli altri** stream devono aspettare che finisca (a meno di avere costruito gli altri stream in modo particolare)
- se lancio vari kernel nel mio codice e, nel primo lancio di kernel, non metto il 4to ingresso nelle triple angle brackets, questo vuol dire che quel kernel e' **automaticamente** nello stream 0.
- lo stesso vale per le CUDA api che hanno come parametro **OPZIONALE** lo stream. Se non viene inserito questo parametro, CUDA **accende** lo **stream 0**, creando problemi a tutti gli altri stream!

Per esempio:

```
miokernel0 <<<miaGrid0, mioBlocco0, 0>>> (argomento0, 3)  
miokernel1 <<<miaGrid1, mioBlocco1, 0, mioStream1>>>(argomento, altroArgomento);  
miokernel2 <<<miaGrid2, mioBlocco2, 0, mioStream2>>>(argom, argomento);
```

Il primo kernel, **sembra innocuo**, ma in pratica, non avendo il quarto ingresso nelle <<< >>> appartiene allo stream 0 e quindi "blocca" i kernel seguenti!

# Attenzione allo stream 0!

- lo **stream 0** (quello di default) ha una **sincronizzazione differente** rispetto agli altri stream
- ovvero: **tutti gli altri** stream devono aspettare che finisca (a meno di avere costruito gli altri stream in modo particolare)
- se lancio vari kernel nel mio codice e, nel primo lancio di kernel, non metto il 4to ingresso nelle triple angle brackets, questo vuol dire che quel kernel e' **automaticamente** nello stream 0.
- lo stesso vale per le CUDA api che hanno come parametro **OPZIONALE** lo stream. Se non viene inserito questo parametro, CUDA **accende** lo **stream 0**, creando problemi a tutti gli altri stream!

Per esempio:

```
miokernel0 <<<miaGrid0, mioBlocco0, 0>>> (argomento0, 3)  
miokernel1 <<<miaGrid1, mioBlocco1, 0, mioStream1>>>(argomento, altroArgomento);  
miokernel2 <<<miaGrid2, mioBlocco2, 0, mioStream2>>>(argom, argomento);
```

Il primo kernel, **sembra innocuo**, ma in pratica, non avendo il quarto ingresso nelle <<< >>> appartiene allo **stream 0** e quindi "blocca" i kernel seguenti!

## Attenzione allo stream 0!

- lo **stream 0** (quello di default) ha una **sincronizzazione differente** rispetto agli altri stream
- ovvero: **tutti gli altri** stream devono aspettare che finisca (a meno di avere costruito gli altri stream in modo particolare)
- se lancio vari kernel nel mio codice e, nel primo lancio di kernel, non metto il 4to ingresso nelle triple angle brackets, questo vuol dire che quel kernel e' **automaticamente** nello stream 0.
- lo stesso vale per le CUDA api che hanno come parametro **OPZIONALE** lo stream. Se non viene inserito questo parametro, CUDA **accende** lo **stream 0**, creando problemi a tutti gli altri stream!

Per esempio:

```
miokernel0 <<<miaGrid0, mioBlocco0, 0>>> (argomento0, 3)  
miokernel1 <<<miaGrid1, mioBlocco1, 0, mioStream1>>>(argomento, altroArgomento);  
miokernel2 <<<miaGrid2, mioBlocco2, 0, mioStream2>>>(argom, argomento);
```

Il primo kernel, **sembra innocuo**, ma in pratica, non avendo il quarto ingresso nelle <<< >>> appartiene allo **stream 0** e quindi "blocca" i kernel seguenti!

## Attenzione allo stream 0!

- lo **stream 0** (quello di default) ha una **sincronizzazione differente** rispetto agli altri stream
- ovvero: **tutti gli altri** stream devono aspettare che finisca (a meno di avere costruito gli altri stream in modo particolare)
- se lancio vari kernel nel mio codice e, nel primo lancio di kernel, non metto il 4to ingresso nelle triple angle brackets, questo vuol dire che quel kernel e' **automaticamente** nello stream 0.
- lo stesso vale per le CUDA api che hanno come parametro **OPZIONALE** lo stream. Se non viene inserito questo parametro, CUDA **accende** lo **stream 0**, creando problemi a tutti gli altri stream!

Per esempio:

```
miokernel0 <<<miaGrid0, mioBlocco0, 0>>> (argomento0, 3)  
miokernel1 <<<miaGrid1, mioBlocco1, 0, mioStream1>>>(argomento, altroArgomento);  
miokernel2 <<<miaGrid2, mioBlocco2, 0, mioStream2>>>(argom, argomento);
```

Il primo kernel, **sembra innocuo**, ma in pratica, non avendo il quarto ingresso nelle <<< >>> appartiene allo **stream 0** e quindi "blocca" i kernel seguenti!

# Outline

- 1 *Struttura Fisica*
- 2 *Kernel*
- 3 *Struttura Logica*
- 4 *Memorie*
- 5 *Prodotto Matrici*
- 6 *Stream*
- 7 *Esempi e strumenti*

# Esercizio

Proviamo a costruire un codice che calcoli l'integrale di una funzione in un intervallo  $[c, d]$ .

- 1 ci sia una funzione che calcoli una cubica:  $-3x^3 + 2x^2 + 5x + 3$
- 2 ci sia un kernel che:
  - sia funzione del numero totale di punti usati  $N$
  - sia funzione del punto di partenza dell'intervallo
  - sommi la funzione da un punto (che dipende dall'identità del thread) al successivo e la metta in un ingresso di un array
- 3 lanciare il kernel con molti `thread` e molti `blocchi`, dove ad ogni thread si fa integrare un sottoinsieme dei punti
- 4 decidere la struttura dei blocchi e dei thread in modo che ci siano "pochi" thread inattivi.
- 5 modificare la struttura dei blocchi e thread e vedere l'impatto sulla velocità
- 6 modificare il kernel in modo che ogni thread calcoli un sottointervallo e non un solo punto

## *inizio trapezi: kernel e funzione da integrare*

```
#include <iostream>
#include <ctime>
using namespace std;
#include <cuda.h>
#include <math_constants.h>
#include <cuda_runtime.h>

__device__ float myfunction(float x) // funzione lanciata dal device sul device
{
return -3*x*x*x+2.0f*x*x + 5*x + 3.0f;
}

__global__ void integratorKernel(float *a, float c, float deltaX, int N) // kernel
{
int idx = blockIdx.x * blockDim.x + threadIdx.x;
float x = c + (float)idx * deltaX; // c=punto partenza intervallo integrazione

if (idx<N)
{
a[idx] = myfunction(x)+myfunction(x+deltaX);
}
}
```

## funzione che lancia il kernel

```

// cudaIntegrate() e' la funzione che fa partire il calcolo di f(x) su [c,d].
__host__ float cudaIntegrate(float c, float d, int N) // funzione che gira sull'host
float deltaX = (d-c)/N;                               // deltaX
cudaError_t errorcode = cudaSuccess;                  // codice di errore
int size = N*sizeof(float);                           // dimensione dell'array delle somme

float* a_h = (float *)malloc(size);                   // allocca memoria sull'host
float* a_d;                                           // alloccala sul device
if (( errorcode = cudaMalloc((void **)&a_d,size))!= cudaSuccess)
{
cout << "cudaMalloc(): " << cudaGetErrorString(errorcode) << endl;
exit(1);
}
int block_size = 256;                                // dimensione blocchi
int n_blocks = N/block_size + ( N % block_size == 0 ? 0 : 1); // numero blocchi

integratorKernel <<< n_blocks, block_size >>> (a_d, c, deltaX, N); // lancia kernel

if((errorcode = cudaMemcpy(a_h, a_d, sizeof(float)*N, cudaMemcpyDeviceToHost))!=cudaSuccess)
// copia da device a host
{
cout << "cudaMemcpy(): " << cudaGetErrorString(errorcode) << endl;
exit(1);
}

```



## operazione di riduzione e calcolo tempi

```

float sum = 0.0;
for(int i=0; i<N; i++) sum += a_h[i]; // somma tutto l'array: poco efficiente
sum *= deltaX/2.0;                  // moltiplica per deltaX/2
free(a_h);
cudaFree(a_d);
return sum;
}
// fine della funzione cudaIntegrate
// funzione che converte il tempo in microsecondi (gira sull'host)
__host__ double diffclock(clock_t clock1, clock_t clock2)
{
double diffticks = clock1-clock2;
double diffms = diffticks/(CLOCKS_PER_SEC/1000);
return diffms;
}
// host main program
int main()
{
clock_t start = clock();
float answer = cudaIntegrate(0.0,1.0,1000); // fai integrazione
clock_t end = clock();

cout << "The answer is " << answer << endl;
cout << "Computation time: " << diffclock(end,start);
cout << "  micro seconds" << endl;

return 0;
}

```

# CUDA Events: misurare la velocità di esecuzione

Per misurare la velocità di esecuzione è essenziale essere in grado di distinguere tra il tempo passato durante il calcolo parallelo e quello seriale del codice, per questo sono state implementate delle funzioni che chiamano i cosiddetti *CUDA Events*. Questi oggetti sono in pratica dei *timestamp* associati a delle azioni CUDA. Un evento è un oggetto che va prima dichiarato

```

cudaEvent_t inizio, fine;           // dichiaro 2 eventi, l'inizio e la fine
cudaEventCreate( &inizio);         // creo l'evento, e' una sorta di inizializzazione
cudaEventRecord( inizio, 0);       // faccio partire l'evento

...

cudaEventRecord(fine, 0);          // parte l'evento fine?
cudaEventSynchronize(&fine);      // serve per poter scrivere, altrimenti

cudaEventElapsedTime( &tempopassato, inizio, fine); // calcola quanto tempo e' passato

...
cudaEventDestroy( &start);
cudaEventDestroy( &fine);

```

- `cudaEventSynchronize()` serve per essere sicuri che tutti i thread abbiano finito di lavorare prima di leggere il timestamp. Il `cudaEventRecord(&fine)` è asincrono, se non faccio la sincronizzazione, potrebbe essere che il codice da `host` che deve scrivere a video o su un file il valore del timestamp, possa scriverlo **prima** che l'evento sia finito realmente
- `cudaEventDestroy()` cancella l'evento in modo che non occupi memoria o crei errori
- gli eventi non vanno bene per misurare la CPU!

## CUDA Events: misurare la velocità di esecuzione

Per misurare la velocità di esecuzione è essenziale essere in grado di distinguere tra il tempo passato durante il calcolo parallelo e quello seriale del codice, per questo sono state implementate delle funzioni che chiamano i cosiddetti *CUDA Events*. Questi oggetti sono in pratica dei *timestamp* associati a delle azioni CUDA. Un evento è un oggetto che va prima dichiarato

```

cudaEvent_t inizio, fine;           // dichiaro 2 eventi, l'inizio e la fine
cudaEventCreate( &inizio);        // creo l'evento, e' una sorta di inizializzazione
cudaEventRecord( inizio, 0);      // faccio partire l'evento

...

cudaEventRecord(fine, 0);         // parte l'evento fine?
cudaEventSynchronize(&fine);     // serve per poter scrivere, altrimenti

cudaEventElapsedTime( &tempopassato, inizio, fine); // calcola quanto tempo e' passato

...
cudaEventDestroy(&start);
cudaEventDestroy(&fine);

```

- `cudaEventSynchronize()` serve per essere sicuri che tutti i thread abbiano finito di lavorare prima di leggere il timestamp. Il `cudaEventRecord(&fine)` è asincrono, se non faccio la sincronizzazione, potrebbe essere che il codice da `host` che deve scrivere a video o su un file il valore del timestamp, possa scriverlo **prima** che l'evento sia finito realmente
- `cudaEventDestroy()` cancella l'evento in modo che non occupi memoria o crei errori
- gli eventi non vanno bene per misurare la CPU!

## CUDA Events: misurare la velocità di esecuzione

Per misurare la velocità di esecuzione è essenziale essere in grado di distinguere tra il tempo passato durante il calcolo parallelo e quello seriale del codice, per questo sono state implementate delle funzioni che chiamano i cosiddetti *CUDA Events*. Questi oggetti sono in pratica dei *timestamp* associati a delle azioni CUDA. Un evento è un oggetto che va prima dichiarato

```

cudaEvent_t inizio, fine;           // dichiaro 2 eventi, l'inizio e la fine
cudaEventCreate( &inizio);        // creo l'evento, e' una sorta di inizializzazione
cudaEventRecord( inizio, 0);      // faccio partire l'evento

...

cudaEventRecord(fine, 0);         // parte l'evento fine?
cudaEventSynchronize(&fine);     // serve per poter scrivere, altrimenti

cudaEventElapsedTime( &tempopassato, inizio, fine); // calcola quanto tempo e' passato

...
cudaEventDestroy(&start);
cudaEventDestroy(&fine);

```

- `cudaEventSynchronize()` serve per essere sicuri che tutti i thread abbiano finito di lavorare prima di leggere il `timestamp`. Il `cudaEventRecord(&fine)` è asincrono, se non faccio la sincronizzazione, potrebbe essere che il codice da `host` che deve scrivere a video o su un file il valore del `timestamp`, possa scriverlo **prima** che l'evento sia finito realmente
- `cudaEventDestroy()` cancella l'evento in modo che non occupi memoria o crei errori
- gli eventi non vanno bene per misurare la CPU!

## CUDA Events: misurare la velocità di esecuzione

Per misurare la velocità di esecuzione è essenziale essere in grado di distinguere tra il tempo passato durante il calcolo parallelo e quello seriale del codice, per questo sono state implementate delle funzioni che chiamano i cosiddetti *CUDA Events*. Questi oggetti sono in pratica dei *timestamp* associati a delle azioni CUDA. Un evento è un oggetto che va prima dichiarato

```

cudaEvent_t inizio, fine;           // dichiaro 2 eventi, l'inizio e la fine
cudaEventCreate( &inizio);        // creo l'evento, e' una sorta di inizializzazione
cudaEventRecord( inizio, 0);      // faccio partire l'evento

...

cudaEventRecord(fine, 0);         // parte l'evento fine?
cudaEventSynchronize(&fine);     // serve per poter scrivere, altrimenti

cudaEventElapsedTime( &tempopassato, inizio, fine); // calcola quanto tempo e' passato

...
cudaEventDestroy(&start);
cudaEventDestroy(&fine);

```

- `cudaEventSynchronize()` serve per essere sicuri che tutti i thread abbiano finito di lavorare prima di leggere il `timestamp`. Il `cudaEventRecord(&fine)` è asincrono, se non faccio la sincronizzazione, potrebbe essere che il codice da `host` che deve scrivere a video o su un file il valore del `timestamp`, possa scriverlo **prima** che l'evento sia finito realmente
- `cudaEventDestroy()` cancella l'evento in modo che non occupi memoria o crei errori
- gli eventi non vanno bene per misurare la CPU!

## CUDA Events: misurare la velocità di esecuzione

Per misurare la velocità di esecuzione è essenziale essere in grado di distinguere tra il tempo passato durante il calcolo parallelo e quello seriale del codice, per questo sono state implementate delle funzioni che chiamano i cosiddetti *CUDA Events*. Questi oggetti sono in pratica dei *timestamp* associati a delle azioni CUDA. Un evento è un oggetto che va prima dichiarato

```

cudaEvent_t inizio, fine;           // dichiaro 2 eventi, l'inizio e la fine
cudaEventCreate( &inizio);        // creo l'evento, e' una sorta di inizializzazione
cudaEventRecord( inizio, 0);      // faccio partire l'evento

...

cudaEventRecord(fine, 0);         // parte l'evento fine?
cudaEventSynchronize(&fine);     // serve per poter scrivere, altrimenti

cudaEventElapsedTime( &tempopassato, inizio, fine); // calcola quanto tempo e' passato

...
cudaEventDestroy(&start);
cudaEventDestroy(&fine);

```

- `cudaEventSynchronize()` serve per essere sicuri che tutti i thread abbiano finito di lavorare prima di leggere il `timestamp`. Il `cudaEventRecord(&fine)` è asincrono, se non faccio la sincronizzazione, potrebbe essere che il codice da `host` che deve scrivere a video o su un file il valore del `timestamp`, possa scriverlo **prima** che l'evento sia finito realmente
- `cudaEventDestroy()` cancella l'evento in modo che non occupi memoria o crei errori
- gli eventi non vanno bene per misurare la CPU!

# Occupancy

Cos'è l'occupancy?

- un **warp** è considerato **attivo** dal momento della "partenza" del primo dei suoi thread alla fine del calcolo dell'ultimo dei thread.
- Si definisce **occupancy**:

$$\text{occupancy} = \frac{\text{warp attivi}}{\text{max teorica warp attivi}}$$

l'occupancy è una misura della **efficienza** di un kernel, dipende da molti fattori:

- dalla struttura della grid e dei **blocchi**
- dall'uso delle memorie (shared, registri)
- dallo specifico kernel (p.es ci sono divergenze?)
- tra gli strumenti forniti da NVIDIA esiste l'occupancy calculator:  
<https://docs.nvidia.com/cuda/cuda-occupancy-calculator/index.html> che ci aiuta nei lanci di kernel, dicendo la max occupancy ottenibile.
- Per fare debugging, si può usare *nvprof*:  
<https://devblogs.nvidia.com/cuda-pro-tip-nvprof-your-handy-universal-gpu-profiler/>

# Occupancy

Cos'è l'occupancy?

- un **warp** è considerato **attivo** dal momento della "partenza" del primo dei suoi thread alla fine del calcolo dell'ultimo dei thread.
- Si definisce **occupancy**:

$$\text{occupancy} = \frac{\text{warp attivi}}{\text{max teorica warp attivi}}$$

l'occupancy è una misura della **efficienza** di un kernel, dipende da molti fattori:

- dalla struttura della grid e dei blocchi
- dall'uso delle memorie (shared, registri)
- dallo specifico kernel (p.es ci sono divergenze?)
- tra gli strumenti forniti da NVIDIA esiste l'occupancy calculator:  
<https://docs.nvidia.com/cuda/cuda-occupancy-calculator/index.html> che ci aiuta nei lanci di kernel, dicendo la max occupancy ottenibile.
- Per fare debugging, si può usare *nvprof*:  
<https://devblogs.nvidia.com/cuda-pro-tip-nvprof-your-handy-universal-gpu-profiler/>



# Occupancy

Cos'è l'occupancy?

- un **warp** è considerato **attivo** dal momento della "partenza" del primo dei suoi thread alla fine del calcolo dell'ultimo dei thread.
- Si definisce **occupancy**:

$$\text{occupancy} = \frac{\text{warp attivi}}{\text{max teorica warp attivi}}$$

l'occupancy è una misura della **efficienza** di un kernel, dipende da molti fattori:

- dalla struttura della **grid** e dei **blocchi**
- dall'uso delle memorie (shared, registri)
- dallo specifico kernel (p.es ci sono divergenze?)
- tra gli strumenti forniti da NVIDIA esiste l'occupancy calculator:  
<https://docs.nvidia.com/cuda/cuda-occupancy-calculator/index.html> che ci aiuta nei lanci di kernel, dicendo la max occupancy ottenibile.
- Per fare debugging, si può usare *nvprof*:  
<https://devblogs.nvidia.com/cuda-pro-tip-nvprof-your-handy-universal-gpu-profiler/>

# Occupancy

Cos'è l'occupancy?

- un **warp** è considerato **attivo** dal momento della "partenza" del primo dei suoi thread alla fine del calcolo dell'ultimo dei thread.
- Si definisce **occupancy**:

$$\text{occupancy} = \frac{\text{warp attivi}}{\text{max teorica warp attivi}}$$

l'occupancy è una misura della **efficienza** di un kernel, dipende da molti fattori:

- dalla struttura della **grid** e dei **blocchi**
- dall'uso delle memorie (shared, registri)
- dallo specifico kernel (p.es ci sono divergenze?)
- tra gli strumenti forniti da NVIDIA esiste l'occupancy calculator:  
<https://docs.nvidia.com/cuda/cuda-occupancy-calculator/index.html> che ci aiuta nei lanci di kernel, dicendo la max occupancy ottenibile.
- Per fare debugging, si può usare *nvprof*:  
<https://devblogs.nvidia.com/cuda-pro-tip-nvprof-your-handy-universal-gpu-profiler/>

# Occupancy

Cos'è l'occupancy?

- un **warp** è considerato **attivo** dal momento della "partenza" del primo dei suoi thread alla fine del calcolo dell'ultimo dei thread.
- Si definisce **occupancy**:

$$\text{occupancy} = \frac{\text{warp attivi}}{\text{max teorica warp attivi}}$$

l'occupancy è una misura della **efficienza** di un kernel, dipende da molti fattori:

- dalla struttura della **grid** e dei **blocchi**
- dall'uso delle memorie (shared, registri)
- dallo specifico kernel (p.es ci sono divergenze?)
- tra gli strumenti forniti da NVIDIA esiste l'occupancy calculator:  
<https://docs.nvidia.com/cuda/cuda-occupancy-calculator/index.html> che ci aiuta nei lanci di kernel, dicendo la max occupancy ottenibile.
- Per fare debugging, si può usare *nvprof*:  
<https://devblogs.nvidia.com/cuda-pro-tip-nvprof-your-handy-universal-gpu-profiler/>

# Occupancy

Cos'è l'occupancy?

- un **warp** è considerato **attivo** dal momento della "partenza" del primo dei suoi thread alla fine del calcolo dell'ultimo dei thread.
- Si definisce **occupancy**:

$$\text{occupancy} = \frac{\text{warp attivi}}{\text{max teorica warp attivi}}$$

l'occupancy è una misura della **efficienza** di un kernel, dipende da molti fattori:

- dalla struttura della **grid** e dei **blocchi**
- dall'uso delle memorie (shared, registri)
- dallo specifico kernel (p.es ci sono divergenze?)
- tra gli strumenti forniti da NVIDIA esiste l'occupancy calculator:  
<https://docs.nvidia.com/cuda/cuda-occupancy-calculator/index.html> che ci aiuta nei lanci di kernel, dicendo la max occupancy ottenibile.
- Per fare debugging, si può usare *nvprof*:  
<https://devblogs.nvidia.com/cuda-pro-tip-nvprof-your-handly-universal-gpu-profiler/>

# Occupancy

Cos'è l'occupancy?

- un **warp** è considerato **attivo** dal momento della "partenza" del primo dei suoi thread alla fine del calcolo dell'ultimo dei thread.
- Si definisce **occupancy**:

$$\text{occupancy} = \frac{\text{warp attivi}}{\text{max teorica warp attivi}}$$

l'occupancy è una misura della **efficienza** di un kernel, dipende da molti fattori:

- dalla struttura della **grid** e dei **blocchi**
- dall'uso delle memorie (shared, registri)
- dallo specifico kernel (p.es ci sono divergenze?)
- tra gli strumenti forniti da NVIDIA esiste l'occupancy calculator:  
<https://docs.nvidia.com/cuda/cuda-occupancy-calculator/index.html> che ci aiuta nei lanci di kernel, dicendo la max occupancy ottenibile.
- Per fare debugging, si può usare *nvprof*:  
<https://devblogs.nvidia.com/cuda-pro-tip-nvprof-your-handy-universal-gpu-profiler/>