UNIVERSITÀ DI PARMA
DIPARTIMENTO DI SCIENZE MATEMATICHE, FISICHE ED INFORMATICHE
Corso di Laurea in Informatica

Programmazione GPU con CUDA

Programmazione parallela e HPC - a.a. 2023/2024

Roberto Alfieri e Alessandro Dal Palù

Programmazione Parallela e HPC: sommario

Programmazione GPU

La GPU è nata come coprocessore per il rendering di immagini su un dispositivo di visualizzazione, ma vista la sua programmabilità dal 2005 si è iniziato ad utilizzarla come coprocessore della CPU.

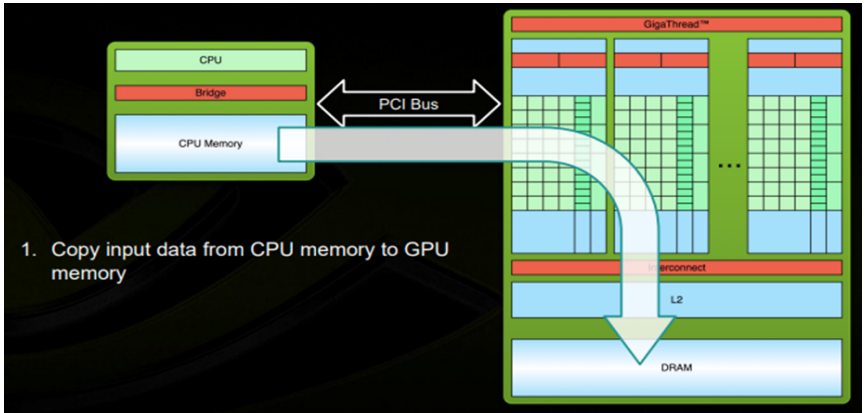
Sono quindi nate le GP-GPU (General Purpose GPU) sprovviste di uscita video.

Ad oggi il costruttore principale di GP-GPU è NVIDIA che fornisce un modello di programmazione **CUDA**, la libreria e il relativo compilatore **nvcc**.

La programmazione consiste nel costruire un kernel di calcolo che verrà tipicamente inviato assieme ai dati dall'host alla GPU, la quale elabora i dati e al termine invia i risultati all'host.

Riferimenti:

- sc11-cuda-c-basics.pdf
- CUDA C++ Programming Guide
- CUDA C++ Best Practices Guide



Architettura GPU vs CPU

La GPU è specializzata per il calcolo «data parallel» e i transistor del chip sono dedicati maggiormente al processamento dei dati piuttosto che al caching o al controllo di flusso.

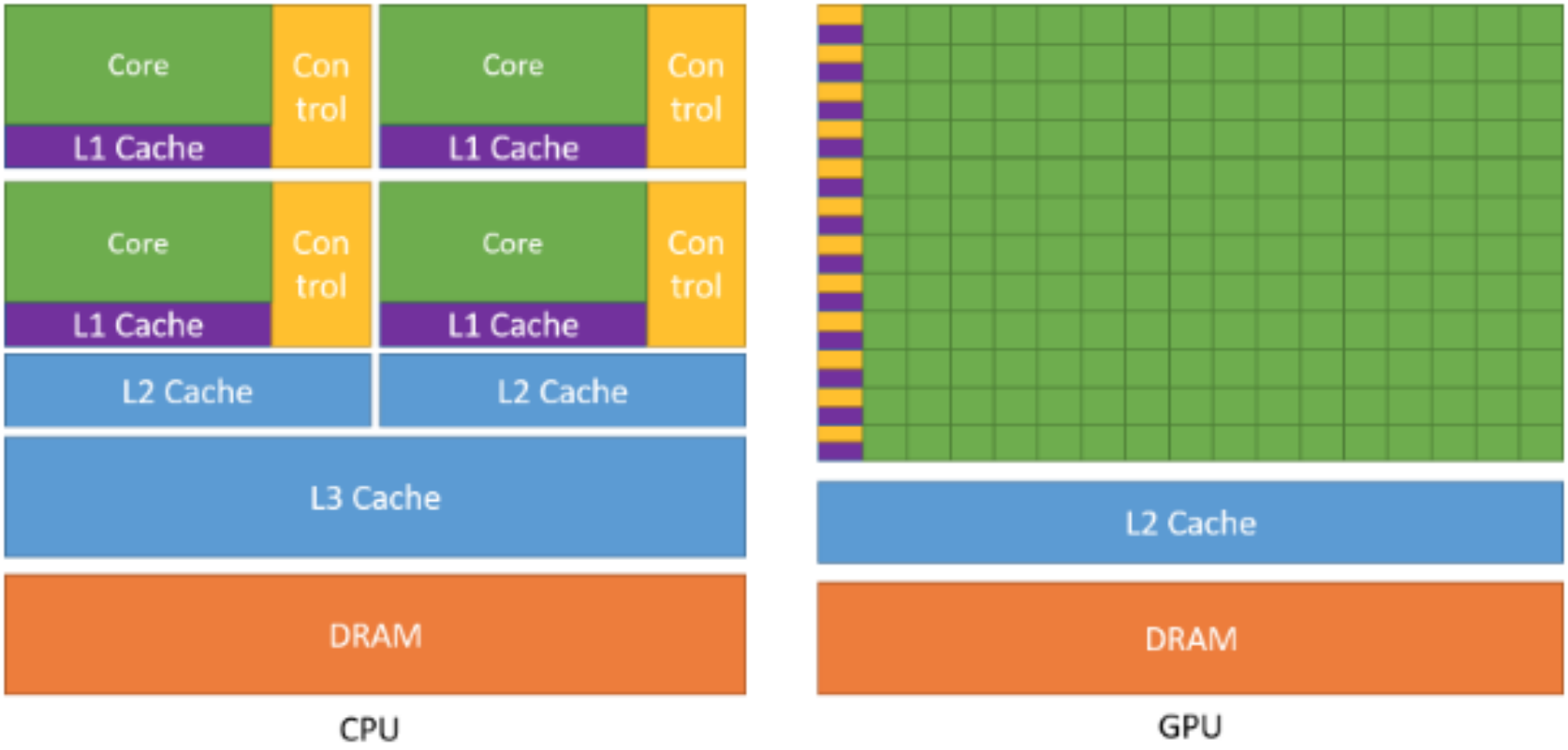
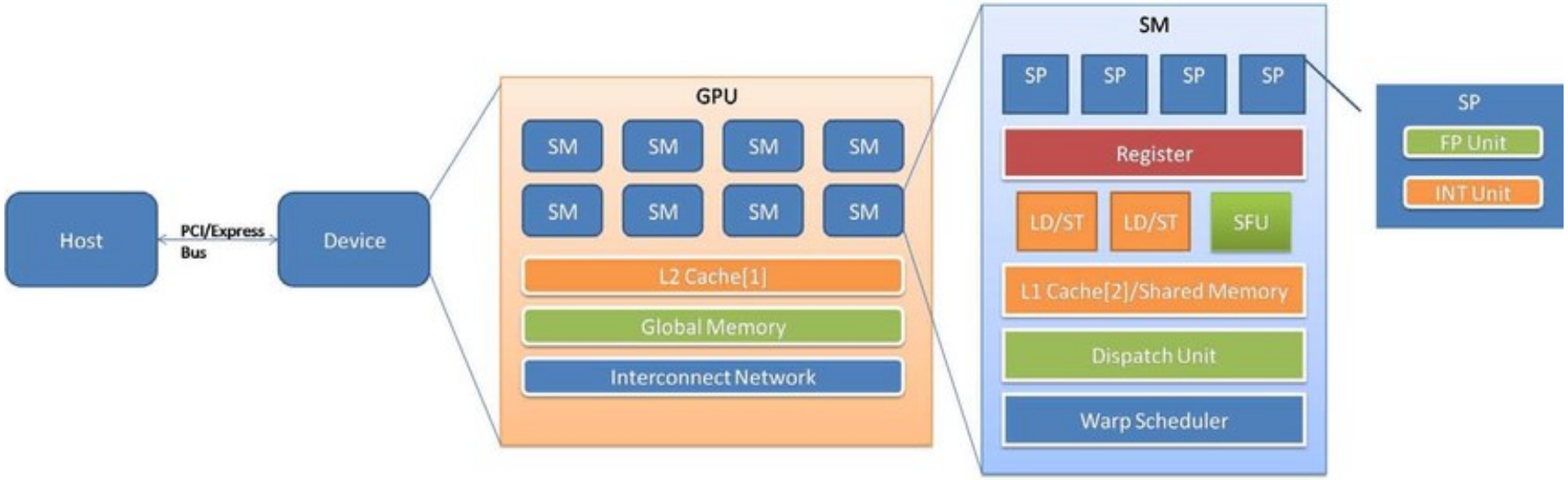


Figure 1: *The GPU Devotes More Transistors to Data Processing*

https://docs.nvidia.com/cuda/cuda-c-programming-guide/

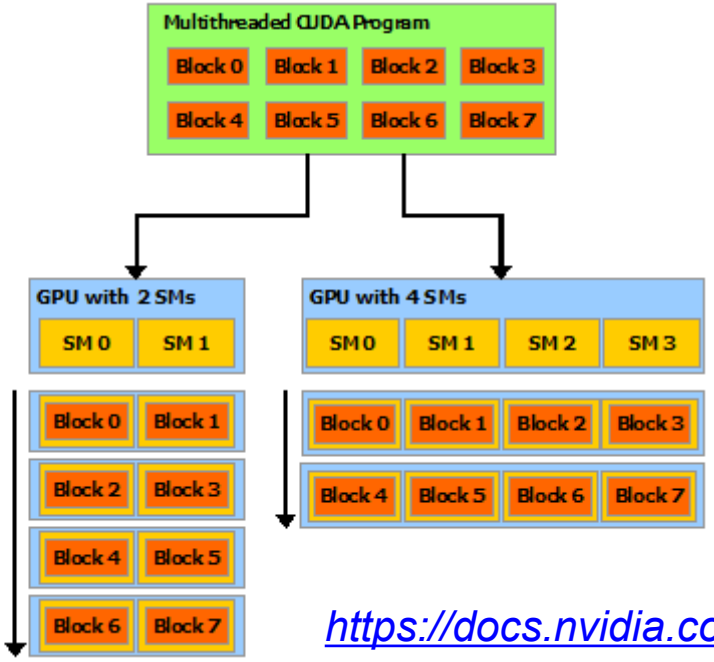
Architettura GPU: SP, Warp e SM

L'unità elementare di calcolo è il **Thread**. Un thread viene eseguito da un Cuda core (**Streaming Processor** – SP).
I thread vengono eseguiti in parallelo in gruppi di 32 (Warp) su SP gestiti da uno **Steaming Multiprocessor** (SM).
Ogni SM dispone di N SP (N=64 su P100, V100 e A100), un Warp scheduler, una memoria veloce (**Register**) privata per ogni thread, e una memoria veloce (L1 cache / **Shared memory**) condivisa tra i thread del SM.
Una GPU contiene un certo numero di SM (es. 56 su P100, 108 su A100) e un'ampia **memoria Globale** condivisa tra tutti i thread della GPU.



Architettura GPU: thread, block e grid

Il programma Cuda non vede SP, Warp e SM, ma vede **thread, block e grid**.
Un **thread block** è un gruppo di thread eseguiti all'interno di un SM che quindi condivide oltre la memoria globale anche la memoria shared. Un SM esegue contemporaneamente i thread su uno (o più) warp, mentre gli altri warp del blocco sono gestiti dalla scheduler dell'SM.
Una applicazione CUDA può creare un numero di thread block, di uguale dimensione, formando una **Grid**. L'interazione tra i blocchi di una grid avviene tramite la memoria globale. Non ci sono garanzie sull'ordine di esecuzione dei blocchi di una griglia.



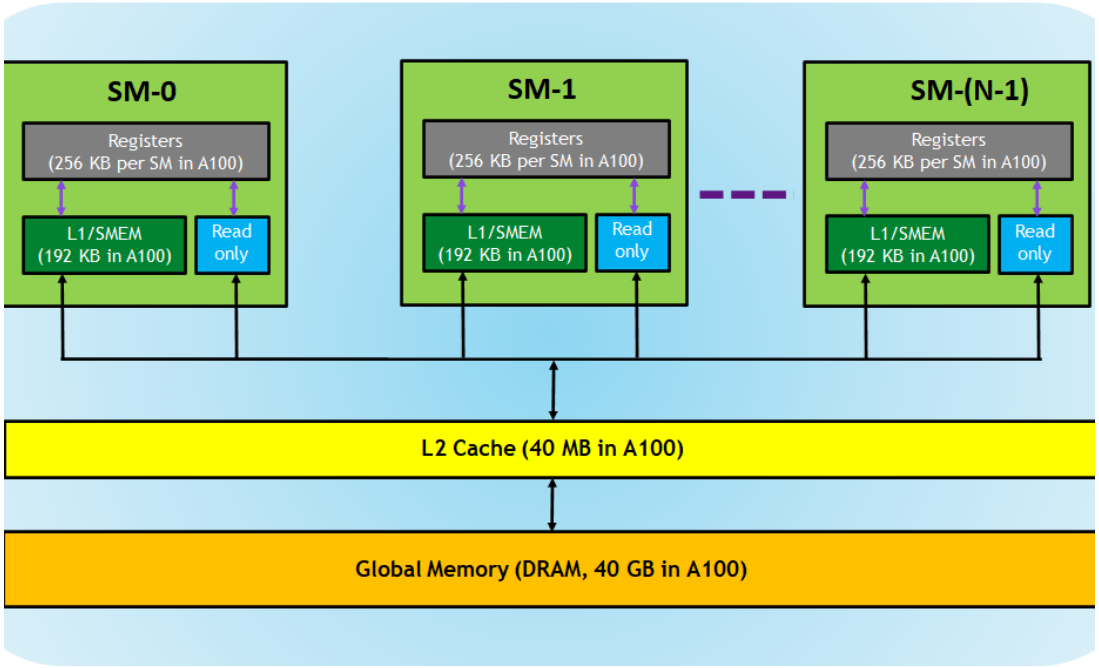
https://docs.nvidia.com/cuda/cuda-c-programming-guide/

Architettura GPU: P100 V100 A100

	SM	Cuda Core per SM	Cuda Core total	Freq. core MHz	TFlops fp64 Peak(*)	TFlops fp64 nbody	L1/Shared per SM	Register per SM	Global memory
P100	56	64	3584	1190	4.2	2.8	64KB	256KB	12GB
V100	80	64	5120	1380	7.0	5.2	96KB	256KB	32GB
A100	108	64	6912	1410	9.7	7.4	164KB	256KB	40/80 GB

(*) PeakPerf= TotalCore x Freq.

Max thread per SM 2038
Max thread per block 1024



CUDA

CUDA (acronimo di **Compute Unified Device Architecture**) è un framework per l'elaborazione parallela creato da NVIDIA nel 2006 per le proprie schede GPU. **CUDA Toolkit** fornisce gli strumenti di sviluppo, librerie e la documentazione necessari per creare applicazioni per l'architettura CUDA.

I sistemi operativi supportati sono **Linux e Windows**.

I linguaggi di programmazione disponibili nell'ambiente di sviluppo CUDA sono estensioni dei linguaggi più diffusi per scrivere programmi. Il principale è '**CUDA-C**' (C con estensioni NVIDIA); altri sono estensioni di Python, Fortran, Java e MATLAB.

CUDA: blocchi e griglie

I thread sono organizzati 1D, 2D o 3D nel blocco (max 1024 thread per block)

I blocchi sono organizzati su **una griglia** 1D, 2D o 3D (max 65535 block per dimensione)

La chiamata di un kernel è simile ad una chiamata a funzione,
ma include le dimensioni di griglia e blocchi.

Esempio 1D:

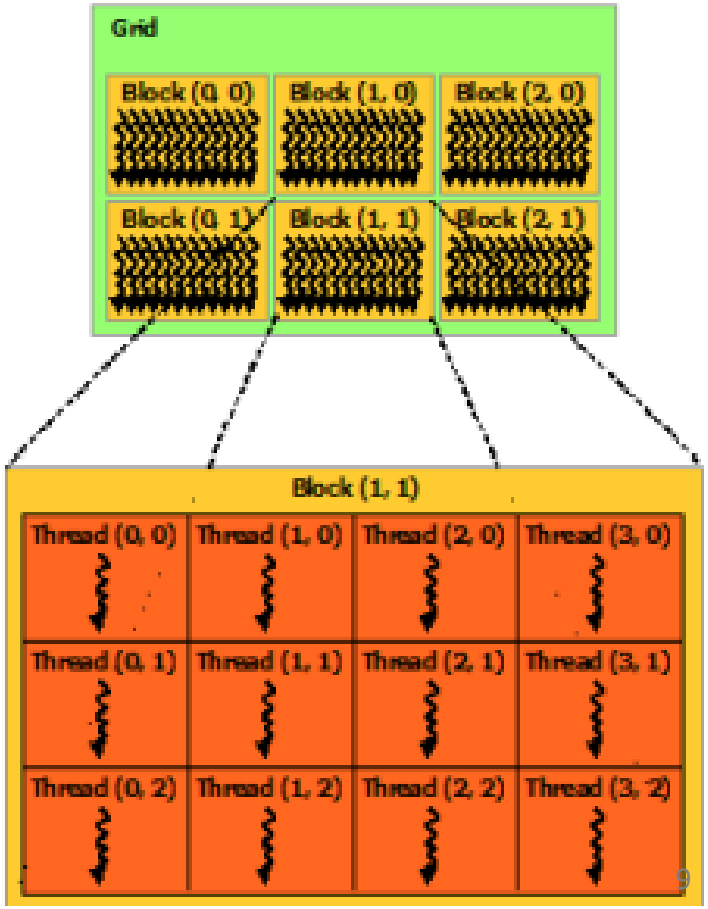
mykernel<<<1024,1024>>>()

Per geometrie 2D o 3D conviene usare dim3

dim3 è un vettore di interi usato in Cuda per definire le geometrie di griglie e blocchi di un kernel.

Esempio:

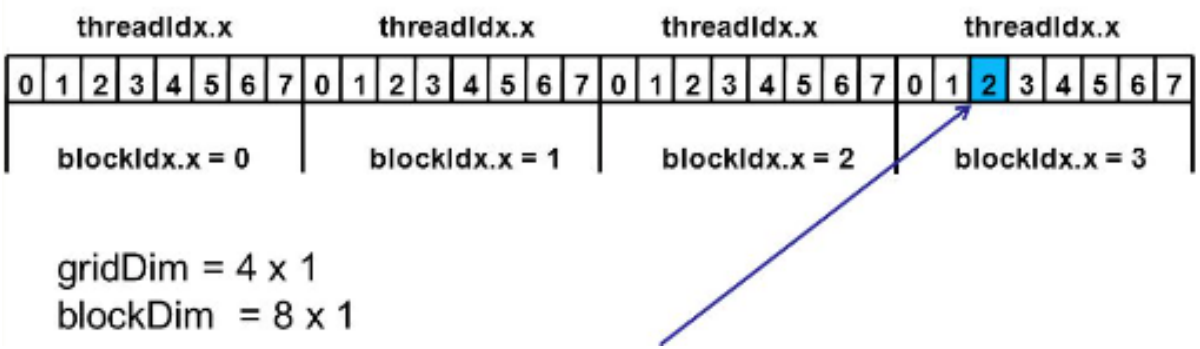
```
dim3 grid( 512 );              // 512 x 1 x 1
dim3 block( 32, 32);           // 32 x 32 x 1
myKernel<<< grid, block >>>();
```



CUDA: identificativi dei thread e dei blocchi

- **threadIdx (.x, .y, .z)** identifica il thread all'interno del blocco
- **blockIdx (.x, .y, .z)** identifica il blocco all'interno della griglia
- **blockDim: (.x, .y, .z)** descrive quanti thread ci sono in un blocco
- **gridDim (.x, .y, .z)** descrive quanti blocchi ci sono nella griglia

Esempio: mapping di un vettore su blocchi 1D



I thread dei blocchi vengono associati agli elementi del vettore e possono mapparne l'indice:

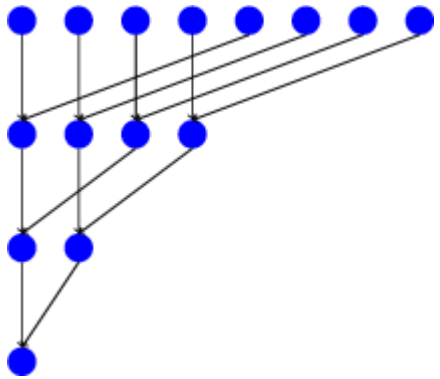
```
idx = (blockIdx.x * blockDim.x) + threadIdx.x;
vettore[idx]=idx;
```

Sincronismo dei thread a livello di blocco

Poichè i warp di un blocco sono eseguiti in un ordine indefinito lo Stream Multiprocessor (SM) consente di creare una barriera di sincronizzazione che forza i thread del blocco ad attendere che tutti raggiungano la barriera.
All'interno del kernel occorre inserire la chiamata **__syncthreads();**

Esempio: Riduzione somma del vettore V[N].
Al termine la somma si trova in V[0] in log N iterazioni.
Dopo ogni iterazione occorre una barriera di sincronismo poichè I warp del blocco hanno un ordine di esecuzione indefinito.

```
int i = blockDim.x/2;
  while (i != 0) {
       if (threadIdx.x < i)
       V[threadIdx.x] += V[threadIdx.x + i];
       __syncthreads();
       i /= 2;
  }
```



Prefissi per la caratterizzazione delle funzioni CUDA

__global__ questa keyword definisce una funzione GPU kernel, può essere invocata solo dalla CPU (host), non dalla GPU. Deve essere di tipo void

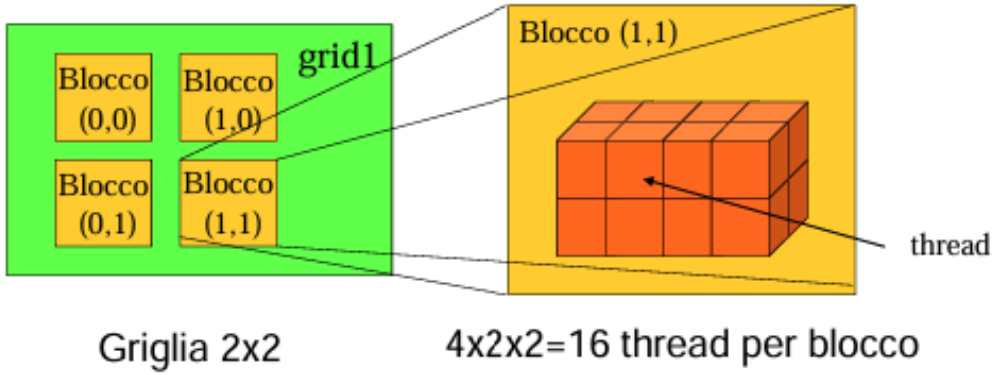
__device__ definisce una funzione GPU kernel, può essere invocata solo dalla GPU stessa (device). Deve essere di tipo void.

__host__ definisce una funzione eseguibile ed invocabile dall'HOST (default)

Esempio:

```
__global__ void KernelFunc(...)
{  …. }

main() {
dim3 DimGrid(2, 2);       // 4 blocks
dim3 DimBlock(4, 2, 2); // 16 threads per block
KernelFunc<<< DimGrid, DimBlock, >>>(...);
}
```



CUDA: modello di calcolo

Il programma inizia l'esecuzione sull'host (CPU)

Il programma chiede al Device (GPU) di eseguire un kernel

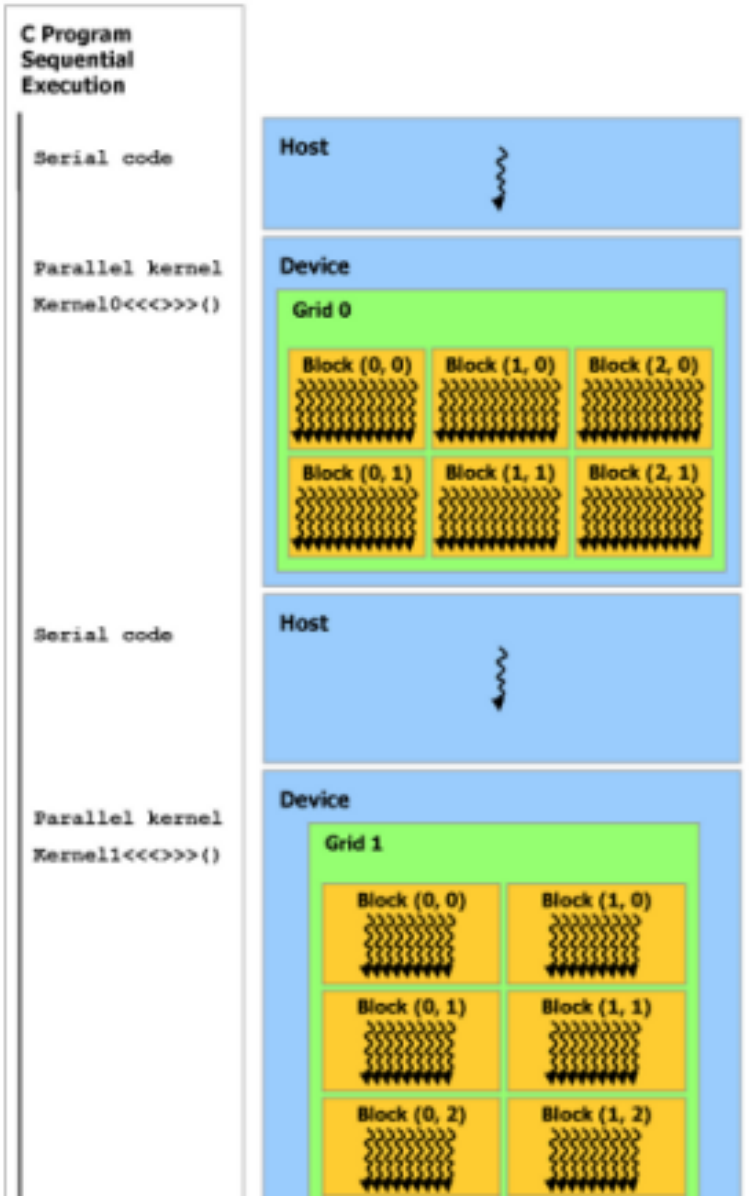
La chiamata è **asincrona:** è possibile far eseguire alla CPU altre parti di codice in parallelo alla GPU.

E' possibile lanciare più kernel in parallelo (se non ci sono dipendenze)

Se si vuole mettere la CPU in attesa della terminazione di tutte le operazioni in esecuzione sul device, si può usare:

cudaDeviceSynchronize();

La CPU colleziona risultati dalla GPU e prosegue



Esempio: somma scalare + vettore

Supponendo di avere un vettore A di 64 elementi e 2 blocchi da 32 thread:

Programma CPU

```
void add_cpu(float *a, float b, int N)
{
for (int idx=0; idx<N; idx++)
   A[idx] = A[idx]+b;
}


void main()
{
….
add_cpu(A,b,N);
}
```

Programma GPU

```
__global__ void add_gpu(float *A, float b, int N)
{
 int idx = (blockIdx.x * blockDim.x) + threadIdx.x;
  A[idx] = A[idx]+b;
}


void main()
{
….
add_gpu<<<2,32>>>(A,b,N);
}
```

Variabili CUDA e tipi di memoria

Le memorie presenti su una GPU sono di diverso tipo e il loro utilizzo ha un impatto notevole sui tempi di trasferimenti dati e quindi sulle prestazione dell'applicazione.
Le variabili possono essere locali, shared, globali o costanti:

Variable declaration	Memory	Scope	Lifetime
`int localVar;`	register	thread	thread
`int localArray[10];`	local	thread	thread
`__shared__ int sharedVar;`	shared	block	block
`__device__ int globalVar;`	global	grid	application
`__constant__ int constantVar;`	constant	grid	application

__device__ __constant__ **a**llocazione dall'host (al di fuori di main())
__shared__ local e register: allocazione dal kernel

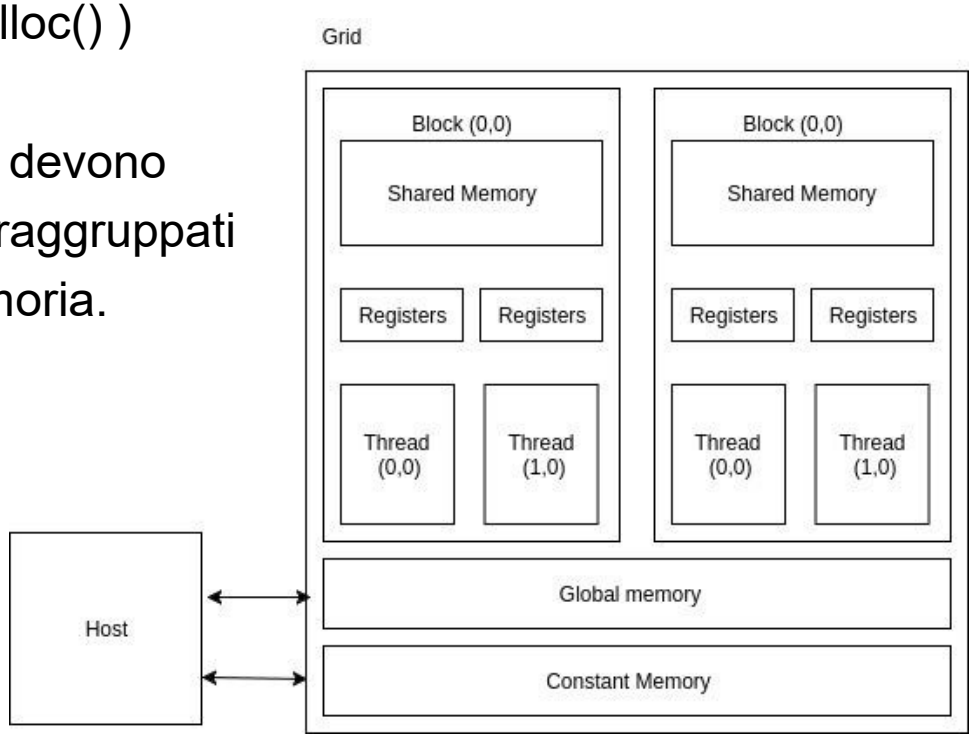
Variabili senza caratterizzazione

- Scalari e piccoli array: memorizzati nei register
- Array di più di 4 elementi: memorizzati in local memory

Memoria globale

E’ la memoria più grande disponibile sul device
Accessibile in lettura e scrittura da tutti i thread della griglia (scope: grid)
La sola possibilità di comunicazione tra host e device
Mantiene il suo stato tra un kernel il successivo (application lifetime)
Elevata latenza di accesso (centinaia di cicli di clock).
Le variabili vengono creare in memoria globale in modo statico (__global__) o dinamico (cudaMalloc())

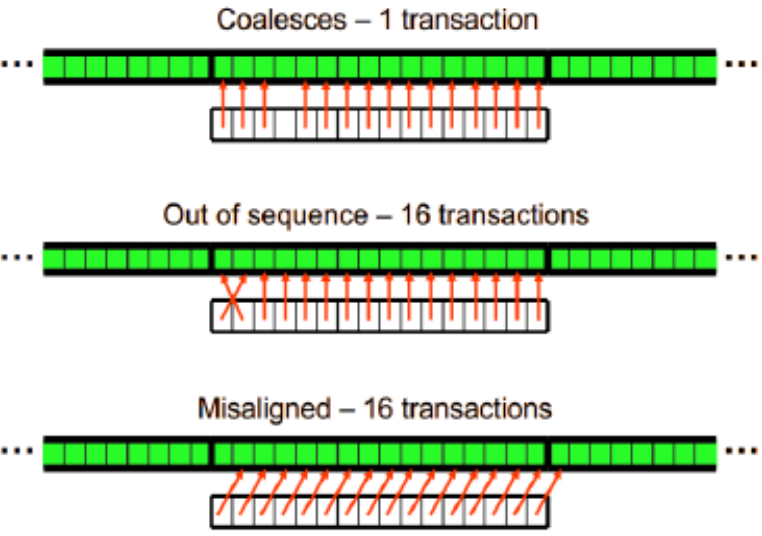
Per sfruttare al meglio la banda gli accessi devono essere coalesced, ovvero devono essere raggruppati e eseguiti con un’unica transazione di memoria.

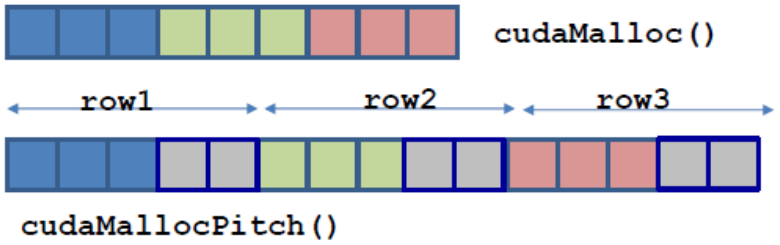


Memoria globale: coalescing

Per sfruttare al meglio la banda gli accessi devono essere coalesced, ovvero tutti i thread del warp accedono a memorie consecutive della memoria globale.
In questo caso l'hardware combina (coalesces) gli accessi in un'unica transazione.

L'allineamento dei dati in memoria à fondamentale per avere accessi coalesced
cudaMalloc() garantisce l'allineamento del primo elemento nella memoria globale, utile quindi per array 1D
cudaMallocPitch() è ideale per array 2D:gli elementi sono paddati e allineati in memoria
cudaMemcpy2D() trasferisce dati 2D allineati con Pitch.





Memoria locale e registri

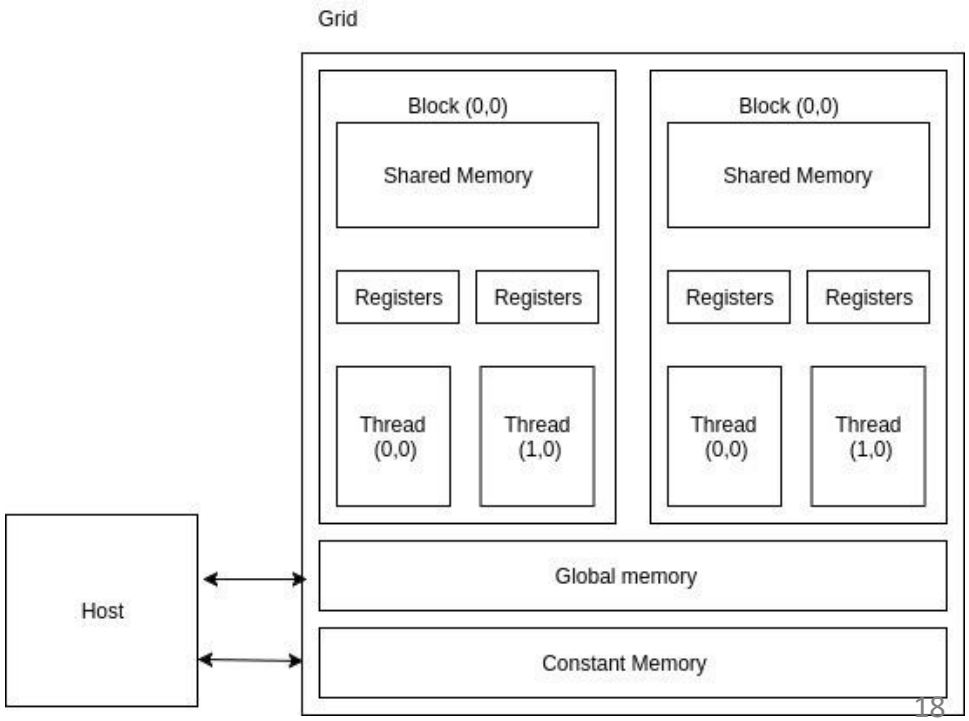
I **dati privati di un thread** devono essere allocati nel kernel e possono risiedere su register file o local memory

Register File: spazio privato di ogni singolo processore on-chip. Alta velocità (circa 100 volte > global)

Local Memory: Memoria privata di un thread sul device (con cache) per variabili che non stanno nei register (tipicamente array con piu' di 4 elementi).

Esempio:

```
__global__ void KernelFunc(...)
{
int a, b[10];
….
}
```



Memoria shared

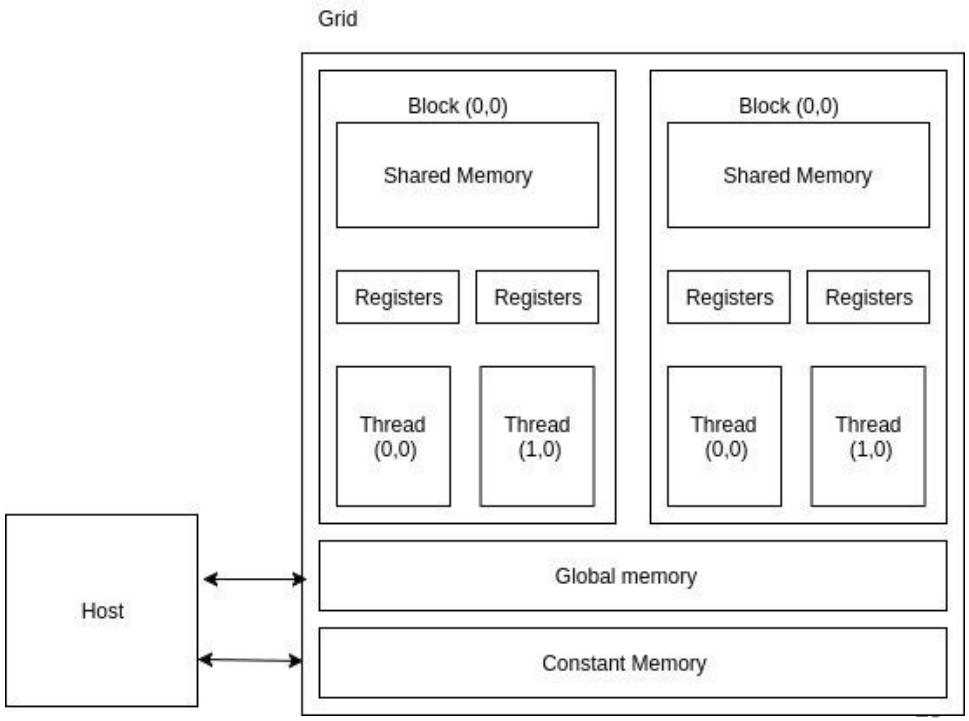
Memoria on-chip condivisa tra i thread del blocco
Bassa latenza (2 cicli di clock)

Le variabili shared vengono allocate dal kernel con la dichiarazione __shared__.
Esempio:

```
__global__ void KernelFunc(...)
{
__shared__ float c[10];
….
}
```

Le variabili shared vengono usate come cache gestita dal programmatore:

- crea area in shared
- copia dati da global a shared
- opera sui dati in shared
- copia dati da shared a global



Memoria shared: bank conflict

La shared memory è organizzata in linee da 32 banchi, ciascuno con ampiezza di 4-byte.
I dati vengono distribuiti ciclicamente su banchi successivi
Gli accessi alla memoria avvengono per warp
Si ha un **bank conflict** quando thread differenti dello stesso warp tentano di accedere a dati differenti sullo stesso banco
Ogni conflitto viene servito e risolto serialmente

Memoria costante

Il fatto che alcuni dati di un programma sono costanti può essere sfruttato per migliorare le prestazioni attraverso l'utilizzo di cache.

Constant memory: Le GPU NVIDIA forniscono 64KB di memoria costante che viene gestita in modo diverso dalla memoria globale standard. In alcune situazioni, l'utilizzo della memoria costante anziché della memoria globale può ridurre la larghezza di banda della memoria . L'allocazione è solo statica e può essere istanziata con cudaMemcpyToSymbol()

Esempio:

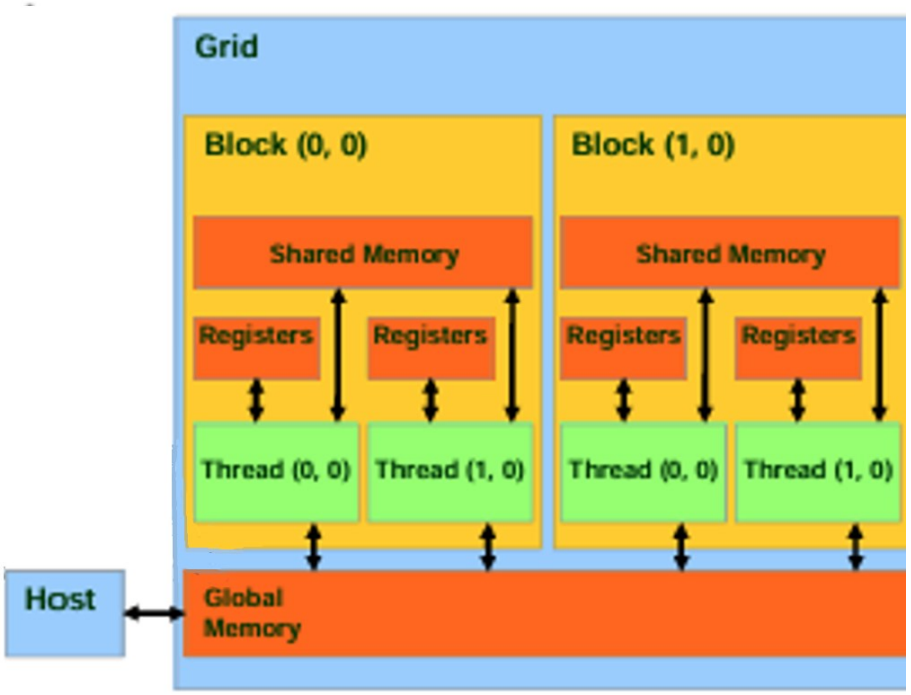
```
__constant__ float dfactor;
int main(void)  {
    float factor=9.0f;
    cudaMemcpyToSymbol(dfactor, &factor, sizeof(float), 0, cudaMemcpyHostToDevice);
…
```

Texture memory: è un altro tipo di memoria di sola lettura. Con l'avvento di CUDA, la sofisticata memoria di texture della GPU può essere utilizzata anche per l'elaborazione generica.. Come la memoria costante, la memoria texture viene memorizzata nella cache del chip, quindi può fornire una larghezza di banda effettiva maggiore rispetto a quella ottenuta quando si accede alla DRAM off-chip.

Allocazione della memoria CUDA

cudaMalloc()
- alloca oggetti nella memoria globale
- 2 parametri:
  - Indirizzo del puntatore (void)
  - Numero di byte

cudaFree()
- libera la memoria globale

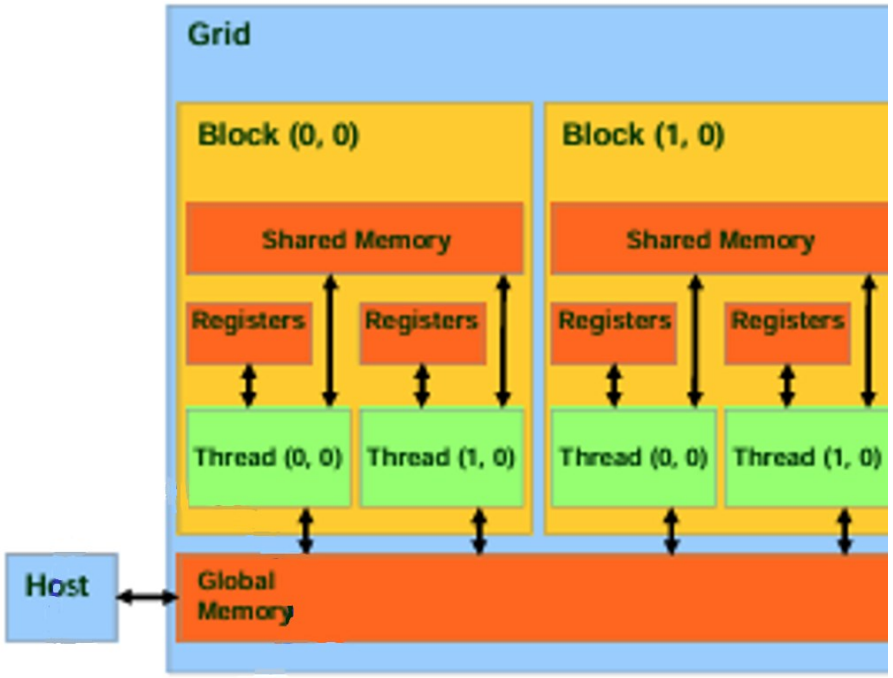


Esempio: Allocare un array di 5 elementi

```
float* D;
int size = sizeof(float) * 5 ;
cudaMalloc((void**)&D, size);
cudaFree(D);
```

Trasferimento dati host-device

cudaMemcpy()
- trasferisce dati da e per la memeoria globale
- richiede 4 parametri
  -- Puntatore destinazione
  -- Puntatore sorgente
  -- Numero di byte
  -- Tipo di trasferimento:
      HostToDevice
      DeviceToHost
      DeviceToDevice



Esempio:

```
float H[5];
float* D;
int size = sizeof(float) * 5;
cudaMalloc((void**)&D, size);
cudaMemcpy(D, H, size, cudaMemcpyHostToDevice);
myKernel<<<grid,block>>>();
cudaMemcpy(H, D, size, cudaMemcpyDeviceToHost);
```

Sincronismo host-device

Il kernel viene eseguito in modo asincrono: il controllo ritorna alla CPU immediatamente

La CPU deve sincronizzarsi prima di accedere ai risultati

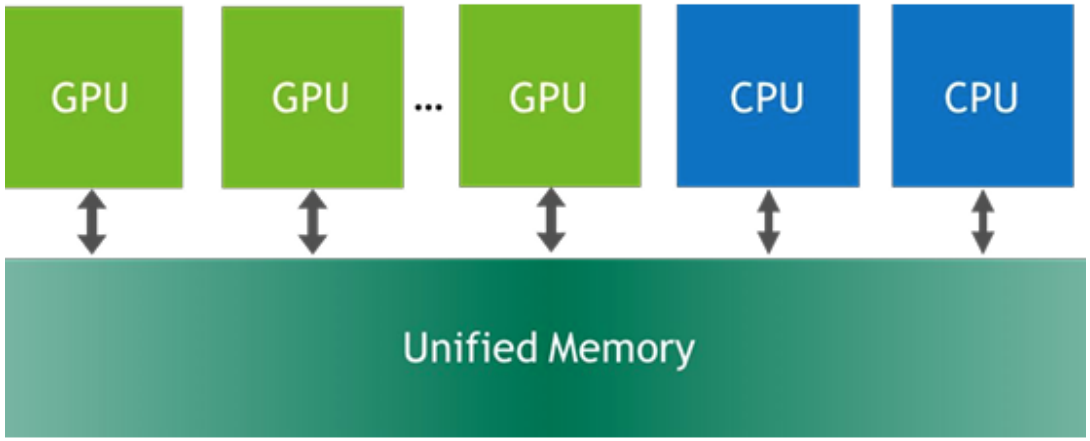
cudaMemcpy() Copia inzia quando le chiamate CUDA sono state completate

cudaMemcopyAsync() Asincrono, non blocca la CPU

cudaDeviceSynchronize() Blocca la CPU fino a quando le chiamate CUDA sono state completate

Unified memory in CUDA

Dalla versione 6 di CUDA e dalla tecnologia hardware Pascal (P100) è stata introdotto il supporto all' «Unified memory» che consente di avere una immagine di memoria coerente con uno spazio di indirizzamento comune tra host e device.
L'allocazione di memoria avviene con il prefisso di caratterizzazione **__managed__** o con **cudaMallocManaged(),** eliminando quindi la necessità di cudaMemcpy().



Esempio:

```
__managed__  int  A[N];
int main(void)
{
  float *B;
 cudaMallocManaged(&B, N*sizeof(float));
...
```

https://developer.nvidia.com/blog/unified-memory-cuda-beginners/

Race condition

L’accesso contemporaneo a varibile condivise è soggetto a “race condition”.
Esempio:

```
__global__ void race(void)
{
  __shared__ int my_shared_variable;
    my_shared_variable = threadIdx.x;
 }
```

- Risultato indefinito
- L’ordine di accesso alla variabile non è noto senza esplicito coordinamento

Funzioni atomiche

CUDA fornisce operazioni atomiche per la gestione delle race condition.

Una operazione atomica garantisce che solo un singolo thread ha accesso a una porzione di memoria finchè l'operzione non è completata.

Le operazioni atomiche lavorano solo con signed e unsigned int.

Addizione/sottrazione: atomicAdd(), atomicSub()
Minimo/Massimo: atomicMin(), atomicMax()
Incremento/decremento condizionale: atomicInc() atomicDec()

Esempio:

```
 __managed__ int  result=0;

__global__ void sum(int *input)
 {
 atomicAdd(result, input[threadIdx.x]);
 }
```

Sezione critica: gestione con atomicCAS e atomicExch

Le istruzioni atomiche possono essere utilizzate anche per proteggere una sezione critica. Si considerino le seguenti istruzioni atomiche:

- **int atomicExch(int* address, int val)** legge il valore attuale in address e lo restituisce, quindi in modo atomico scrive in address il nuovo valore "val"
- **int atomicCAS(int* address, int compare, int val)** legge il valore attuale in address e lo restituisce; "val" viene scritto in address solo se il valore attuale coincide con "compare" .

Esempio di gestione di una sezione critica:

```
__managed__ int lock = 0;
__global__ void kernel() {
 // ...
if (threadIdx.x==0) {
    do {} while(atomicCAS(&lock, 0, 1));    // set lock (lock settato a 1 solo se è 0)
     // critical code section
     atomicExch(&lock, 0);                  // release lock
   }
 }
}
```

Il thread 0 di ogni blocco è usato per la mutua esclusione

CUDA Timing

CUDA ha alcune funzioni per il timing e ha un typedef specifico: cudaEvent_t

cudaEvent_t start, stop;
float time;

cudaEventCreate(&start); // crea un evento
cudaEventCreate(&stop);

cudaEventRecord(start, 0); // registra un evento nello stream di default (0).
…
cudaEventRecord(stop, 0); // registra un evento nello stream di default (0).
Questa operazione è asincrona e potrebbe ritornare prima della registrazione.

cudaEventSynchronize(stop); // attende il completamento della registrazione

CudaEventElapsedTime(&time, start, stop); // tempo in millisecondi tra i due eventi.

cudaEventDestroy(start); // distrugge l'evento
cudaEventDestroy(stop);