

GPU computing

Simone Petta

A.A. 2024/2025

Contents

1	Introduzione	2
1.1	GP-GPU	3
1.2	Architetture eterogenee	3
1.2.1	Parallelismo delle istruzioni	4
1.2.2	Parallelismo dei dati	4
1.3	Parallelismo di task	4
1.4	Tassonomia di Flynn	4
1.5	GPU Nvidia	5
1.5.1	Cuda core	5
1.5.2	CUDA	5
1.5.3	Hello world in cuda	6
2	Modello di programmazione CUDA	7
2.1	Modelli per sistemi paralleli	8
2.1.1	SIMD	8
2.1.2	MIMD	8
2.1.3	PRAM	8
2.1.4	Parallelizzazione di algoritmi	9
2.2	CPU multi-core e programmazione multi-threading	9
2.2.1	Thread	9
2.3	Programmazione multi-core	10
2.4	Programmazione CUDA	10
2.4.1	Organizzazione dei thread	11
2.4.2	Lancio di un kernel CUDA	12
2.4.3	Kernel concorrenti	13
2.4.4	Restrizioni del kernel	13
2.4.5	Memoria	13

Chapter 1

Introduzione

L'obiettivo del corso e' di affrontare il problema del high performance computing. Affronteremo anche il tema delle GPU e del loro utilizzo nel calcolo ad alte prestazioni. Inoltre useremo il framework CUDA per sviluppare applicazioni parallele utilizzando il linguaggio CUDA-c che e' un'estensione del linguaggio c. Introduciamo anche alcune librerie di python, andando a focalizzarci su elementi a basso livello.

Il parallel computing e' la necessita' di accelerare le computazioni avvalendosi di un numero molteplice di processori. Quando ho un processo di grandi dimensioni devo essere in grado di dividere il processo in parti e assegnarlo a vari processi. E' fondamentale la progettazione del problema per spezzettarlo in problemi autonomi.

La CPU ha degli aspetti critici, sono decenni che non vediamo crescere il clock (quante operazioni puo' fare al secondo) si e' andati incontro a misure fisiche che non possono essere superate, come la potenza dissipata. Una naturale estensione e' stata aumentare il numero di core, da multi core a many many core dando vita a device diversi, le GPU, che riscono a mantenere la legge di Moore e a fare si che a costi bassi una workstation diventi una macchina ad alte prestazioni. Bisogna dare credito a Nvidia che ha creato le general purpose GPU, che sono delle GPU che possono essere utilizzate per calcoli generali e non solo per il rendering grafico.

Il parallel computing e' indipendenza, comunicazione e scalabilita' dei processi.

Le motivazioni che portano all'utilizzo della GPU, un esempio e' la riflessione, rifrazione e ombra, la cosa che si fa e' avere un calcolo semplice che si fa per ogni pixel, quindi calcoli semplici ripetuti un numero esorbitante di volte. Vengono anche usate per il deeplearning. Perche' le CNN sono di interesse per la GPU? fanno un'operazione basilare da ripetere un numero esorbitante di volte, fanno la convoluzione, prendono un'immagine ed un kernel e scandiscono l'immagine sottostante estraendone matrici della dimensione del kernel e fare la somma. Un altro esempio e' il calcolo matriciale, vengono fatti due prodotti interni tra i vettori riga e colonna delle matrici, questo puo' essere parallelizzato pensando che gruppi di thread distinti possono occuparsi di ogni entry, fare questa cosa in sincrono costerebbe $O(n^3)$.

1.1 GP-GPU

In sistemi eterogenei, le GPU possono essere utilizzate insieme alle CPU per ottenere prestazioni superiori. La chiave per sfruttare al meglio queste architetture eterogenee e' la suddivisione del lavoro tra le diverse unita' di elaborazione, in modo che ogni tipo di processore possa occuparsi delle operazioni per cui e' piu' adatto. E' importante notare che questa architettura e' di tipo master-slave, dove la CPU funge da master e le GPU da slave. Una funzione gpu e' scritta per sfruttare il parallelismo e CUDA ci permette di pensare il sequenziale nonostante poi i calcoli vengano eseguiti in parallelo, l'obiettivo e' aumentare la potenza di calcolo.

1.2 Architetture eterogenee

C'e' la GPU e la CPU, la CPU ha un numero di core. C'e' un bus di cui per la comunicazione tra CPU e GPU.

Quando si crea codice per queste architetture ci si trova con un eseguibile con blocchi di codice che devono essere eseguiti dalla CPU e blocchi dalla GPU.

Le due parti sono destinate a compiti diversi, se i dati sono piccoli e il parallelismo e' basso siamo nel dominio della CPU ed e' quasi inutile fare il parallelismo con la GPU (solo l'overhead del setup iniziale costa di piu') viceversa se i dati sono grandi e il parallelismo e' alto, allora ha senso utilizzare la GPU.

1.2.1 Parallelismo delle istruzioni

Il parallelismo delle istruzioni e' importante (meno di quello dei dati), nel momento in cui il problema puo' essere suddiviso in diverse istruzioni queste vengono eseguite in parallelo. Ci vuole una dotazione hardware adeguata per gestire questo tipo di parallelismo, come ad esempio un numero sufficiente di unità di esecuzione e una gestione della comunicazione tra processi.

1.2.2 Parallelismo dei dati

Noi lavoreremo in una logica sequenziale ma si estende ad una grossa mole di dati. In questo caso, il parallelismo dei dati diventa cruciale, poiché consente di elaborare simultaneamente grandi volumi di informazioni. Utilizzando tecniche di parallelizzazione, possiamo suddividere i dati in blocchi più piccoli e distribuirli su più unità di elaborazione, massimizzando così l'efficienza e riducendo i tempi di calcolo.

1.3 Parallelismo di task

Il fatto di poter avere gruppi di operazioni distinti, ci porta ad un problema non banale di identificazioni di task indipendenti dall'altri, spesso ci si avvale di strumenti come i grafi. Due task sono indipendenti se le operazioni che li compongono non sono dipendenti tra di loro. Questo problema in termini piu' formali e' di colorazione del grado, ogni colore individua un gruppo di nodi indipendenti tra di loro. Non e' un problema banale, perche' e' irrisolvibile NP-hard, quindi ci si accontenta di soluzioni approssimative.

1.4 Tassonomia di Flynn

Si evidenzia i modelli **SISD** single instruction single data dove non c'e' nessun parallelismo e le operazioni vengono eseguite sequenzialmente. In contrapposizione abbiamo il **SIMD** (single instruction multiple data) dove abbiamo un'istruzione che viene eseguita su piu' dati, sono spesso dotate di librerie e hardware consistente che possono svolgere operazioni parallele, in questo caso c'e' l'accesso ad una memoria globale. **MISD** (multiple instruction single data) e' un modello in cui piu' istruzioni vengono eseguite su un singolo dato, mentre il **MIMD** (multiple instruction multiple data) consente l'esecuzione di piu' istruzioni su piu' dati, rappresentando il massimo livello di parallelismo.

Il modello **SIMT** e' interessante e viene introdotto da CUDA, qui ci sono tanti thread che svolgono tante operazioni su thread distinti, deve esserci un sistema book treading. C'e' un'evoluzione del modello SIMD perche' lo cambia con il modello multithreading,

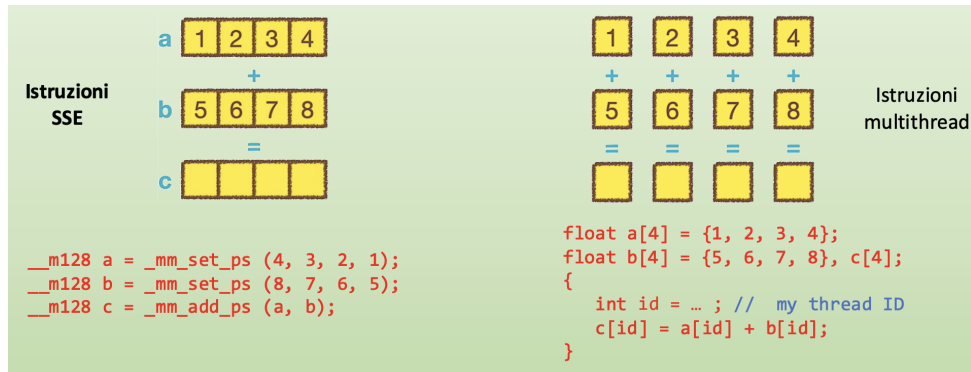


Figure 1.1: Confronto tra SIMD e SIMT

decade il vincolo della singola istruzione, questo e' dato dalla presenza di branch nel codice. Questo modello e' molto utile perche' ci permette di pensare in modo sequenziale anche se complica un po' l'architettura perche' bisogna fare comunicare i thread. Questo e' il modello implementato dai processori GPU.

Nel simt si fissano gli operandi, mentre nel simd no.

1.5 GPU Nvidia

CUDA e' una libreria che permette di fare applicazioni sulle GPU

Ogni scheda e' composta da streaming multi processor, ogni streaming contiene dei core che costituiscono l'architettura della scheda, ci sono gli elementi di memoria shared e global.

1.5.1 Cuda core

Un core ha una logica di processazione vettoriale, piu' dati insieme che possono essere caratterizzati da diverse unita' di calcolo (floating point).

1.5.2 CUDA

E' la piattaforma che ci permette di dare luogo ad un modello di programmazione per le schede Nvidia. Guardando a CUDA nello specifico per progettare delle applicazioni accelerate per le GPU useremo delle librerie gia' accelerate per GPU (come il compilatore).

Ci sono due famiglie di api:

- CUDA Runtime
- CUDA Driver: sono a basso livello e sono piu' difficili da utilizzare perche' non astraggono determinate cose

Programma CUDA

Un programma cuda e' composto da due parti:

- codice host eseguito su CPU
- codice device eseguito su GPU

Il compilatore separa il codice host dal codice device in fase di compilazione, generando un file eseguibile che contiene entrambe le parti.

1.5.3 Hello world in cuda

Per scrivere programmi cuda va modificata la sintassi del c, si scrivono delle funzioni che sono destinate al kernel cuda:

```
1  __global__ void hello_world() {
2      printf("Hello, World from GPU!\n");
3  }
```

Per invocarla va utilizzata questa sintassi in cui specifichiamo il numero di blocchi e il numero di thread per ogni blocco:

```
1  hello_world<<<num_blocks, threads_per_block>>>();
```

Il risultato finale e':

```
1  __global__ void hello_world() {
2      printf("Hello, World from GPU!\n");
3  }
4
5  int main(void) {
6      hello_world<<<1, 10>>>();
7      cudaDeviceReset();
8      return 0;
9  }
```

In questa funzione stamperemo 10 volte "Hello, World from GPU!".

La funzione `cudaDeviceReset` serve a ripristinare lo stato del dispositivo GPU. Viene utilizzata per liberare le risorse allocate sulla GPU e riportare il dispositivo a uno stato pulito. È buona norma chiamare questa funzione alla fine di un programma CUDA per garantire che tutte le risorse siano correttamente rilasciate.

Chapter 2

Modello di programmazione CUDA

2.1 Modelli per sistemi paralleli

E' qualcosa di complesso che richiede una parte HW e software, bisogna necessariamente fare delle astrazioni perche' non ci potremo occupare di tutti i dettagli. Abbiamo questa astrazione:

- Livello macchina: assembly
- Modello architetturale: SIMD o MIMD, si prevedono anche processi in cui ci sono tante unita' di calcolo che interagiscono tra di loro, gestione multi processi e multi thread
- Modello computazionale: e' un modello formale che consente di descrivere la macchina astraendo dai modelli sottostanti ed e' fondamentale per la creazione degli algoritmi. Processore, memoria, scambio dati con un BUS. E' inerentemente sequenziale ma e' utile per creare l'algoritmo data la size dei dati. Nell'ambito del parallelismo passiamo dal modello RAM al PRAM (modello RAM per architetture parallele)
- Modello di programmazione parallela: e' il modo con cui gestiamo il sistema di calcolo parallelo esprimendo algoritmi con un linguaggio che espone il parallelismo dell'architettura sottostante per essere eseguito da un linguaggio che ha in se elementi che permettono di usare il parallelismo. Vi sono aspetti di comunicazione per gestire il multi threading, si affronta quindi la suddivisione del compito in processi e thread. Fino ad arrivare a problemi legati all'uso o della gestione di spazi di indirizzamento condivisi.

2.1.1 SIMD

Si parla di unita' molteplici che lavorano sui dati, laddove i dati permettono di sviluppare meccanismi paralleli. E' un mondo vasto che si puo' vedere anche nelle architetture comuni dove si ha la divisione di computazione in diverse ALU specializzate, una che lavora su un singolo registro ed una che lavora vettorialmente ed e' quindi in grado di lavorare su piu' dati contemporaneamente. Nelle GPU parleremo di thread che lavorano simultaneamente in modalita' SIMD.

2.1.2 MIMD

E' un'altro dei modelli molto diffusi paralleli in cui si hanno le istruzioni di tanti dati simultaneamente che vengono eseguite da un numero di processori veramente grandi.

2.1.3 PRAM

Il modello PRAM e' il modello di calcolo parallelo piu' semplice, in questo caso abbiamo una memoria condivisa, i processori lavorano in maniera SIMD ovvero fanno la stessa istruzione, esistono anche processori inattivi. Ci da quanti passi ci servono per risolvere l'algoritmo parallelo, anche in questo caso potremmo fare studio di complessita' per risolvere il problema parallelo.

2.1.4 Parallelizzazione di algoritmi

Quando vorremo vedere la complessita' computazionale degli algoritmi dovremo fare delle astrazioni ed usare qualcosa simile al PRAM. Nella pratica si partira' da un algoritmo sequenziale, dovremo vedere come dividere la computazione in parallelo per capire come le varie parti vengano attribuite ai vari thread, tendenzialmente la partizione logica dei task la faremo noi, la parte dello scheduling se ne occupa il sistema ma e' bene conoscerla perche' ad esempio in CUDA i processi di scheduling sono fortemente condizionanti dalla potenza del sistema e dovremo adeguare gli algoritmi sulla base dei meccanismi di scheduling. Quello che vorremo ottenere e' una massima occupancy delle unita' di calcolo. Dipendentemente dal modello di memoria, un task può accedere a memoria condivisa o usare tecniche di message passing.

2.2 CPU multi-core e programmazione multi-threading

Un programma in esecuzione e' un processo che viene allocato nel OS e ha una serie di risorse che ha, convive con altri processi nel sistema ed e' una cosa abbastanza pesante, mentre i thread che compongono un processo sono dei sotto-processi ma piu' leggeri. Un processo in genere consiste di tanti thread. In genere vengono creati da delle operazioni di sistema come le fork in UNIX e vengono terminati dall'OS. I processi hanno il loro contesto e vengono eseguiti sequenzialmente secondo un flusso, anche i thread eseguono le cose sequenzialmente ma essendo molteplici possono essere paralleli. Il thread e' un singolo flusso di istruzione che si trova in un processo che lo scheduler si occupa di eseguire concorrentemente ad altri threads.

2.2.1 Thread

Ha un ciclo di vita, viene generato e viene ucciso, il processo chiude quando tutti i thread hanno terminato la loro esecuzione:

- New: viene generato, chi lo ha generato sa chi e'
- Executable: il thread e' pronto per essere eseguito quando ha tutte le risorse necessarie
- Running: il thread sta attualmente eseguendo le sue istruzioni
- Waiting: il thread sta aspettando che una risorsa diventi disponibile
- Finished: il thread ha completato la sua esecuzione e viene rimosso

I vantaggi sono:

- Si possono avere tanti flussi di esecuzioni che possono avvantaggiarsi di sistemi multi-core
- Visibilita' dei dati globale
- Semplicita' di gestione eventi asincroni come l'io

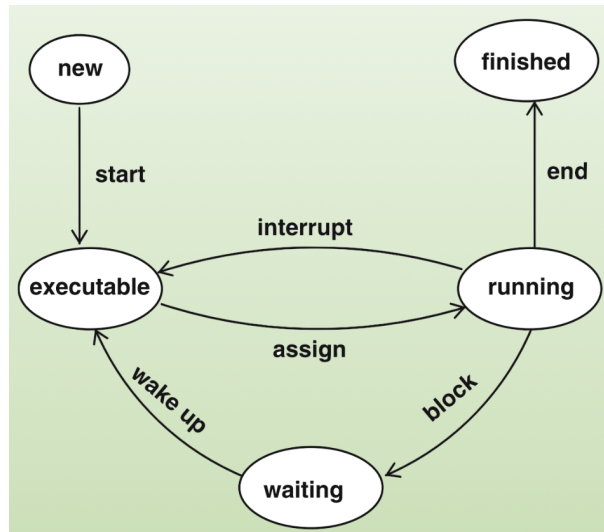


Figure 2.1: Stati di un thread

- Velocita' di context switching

Le difficolta' sono la concorrenza e il problema di avere uno spazio privato.

2.3 Programmazione multi-core

Noi vedremo la programmazione multi-thread su architetture multi-core. L'applicazione deve essere progettata considerando diversi fattori del sistema di calcolo parallelo multi-core (e many core come le GPU), bisogna trovare di task, bilanciarli, suddividere i dati sui vari task (il problema e i dati lo devono consentire) e farlo in modo che tutti i task possano lavorare su queste porzioni di dati indipendentemente. Ci sono quindi tanti aspetti da tenere in considerazione sulla dipendenza e indipendenza dei dati. Test e debugging sono anchessi importanti quando si sviluppano gli algoritmi in ambito parallelo perche' ci sono tanti flussi di esecuzione.

2.4 Programmazione CUDA

Pensare in parallelo significa avere chiaro quali feature la GPU espone al programmatore. Si lavora come su pthread o OpenMP. Si scrive una porzione di CUDA C (semplice estensione di C) per l'esecuzione sequenziale e lo si estende a migliaia di thread (permette di pensare 'ancora' in sequenziale)

- I dati stanno su host
- allocare spazio su GPU
- copiare i dati da host a GPU
- allocare memoria per l'output

- lanciare il kernel che elabora dati in ingresso e li mette in output, mette una copia su memoria condivisa
- cancellare le memorie allocate

2.4.1 Organizzazione dei thread

Cuda presenta una gerarchia astratta di thread che si distribuisce su due livelli:

- grid: una griglia ordinata di blocchi
- block: un insieme di thread che possono cooperare tra loro

Questi possono avere dimensioni 1D, 2D o 3D. Tutto questo fa 9 combinazioni tra grid e block.

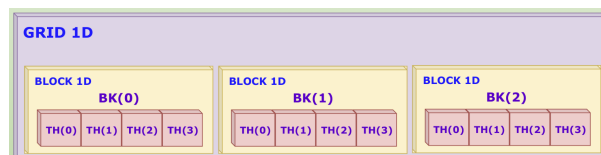


Figure 2.2: Organizzazione dei thread in CUDA

Ci sono delle limitazioni, un blocco può contenere al massimo 1024 thread.

Il blocco di thread è molto importante, dal punto di vista semantico ha un significato, è un gruppo di thread che possono cooperare tra loro tramite:

- block-local synchronization, sincronizzazione: cooperare per uno stesso compito avvantaggiandosi da operazioni fatte da altri thread
- block-local communication, comunicazione tramite la shared memory: è una memoria cache che quindi ha tempi di accesso di molto ridotti

I thread vengono identificati univocamente tramite l'id di blocco e l'id di thread. `blockIdx` e `threadIdx` sono specificati in variabili globali, un kernel a runtime ha accesso a queste informazioni che vengono assegnate dinamicamente, hanno tre valori x, y, z di tipo `uint32`.

```

1
2 #include <stdio.h>
3
4 __global__ void checkIndex(void) {
5     printf("threadIdx: (%d, %d, %d) blockIdx: (%d, %d, %d) "
6           "blockDim: (%d, %d, %d) gridDim: (%d, %d, %d)\n",
7           threadIdx.x, threadIdx.y, threadIdx.z,
8           blockIdx.x, blockIdx.y, blockIdx.z,
9           blockDim.x, blockDim.y, blockDim.z,
10          gridDim.x, gridDim.y, gridDim.z);
11 }
12
13 /*
14 * MAIN

```

```

15 */
16 int main(int argc, char **argv) {
17
18     // grid and block definition
19     dim3 block(4);
20     dim3 grid(3);
21
22     // Print from host
23     printf("Print from host:\n");
24     printf("grid.x = %d\t grid.y = %d\t grid.z = %d\n", grid.x, grid.y,
25           grid.z);
26     printf("block.x = %d\t block.y = %d\t block.z %d\n\n", block.x,
27           block.y, block.z);
28
29     // Print from device
30     printf("Print from device:\n");
31     checkIndex<<<grid, block>>>();
32
33     // reset device
34     cudaDeviceReset();
35     return(0);
36 }

```

Abbiamo accesso alle dimensioni della griglia e del blocco tramite le variabili `blockDim` e `gridDim`, che sono anchessi di tipo `uint32`. Anche in questo caso abbiamo tre componenti `x`, `y`, `z`.

L'indice univoco del thread nei blocchi si calcola diversamente rispetto alle dimensioni del blocco:

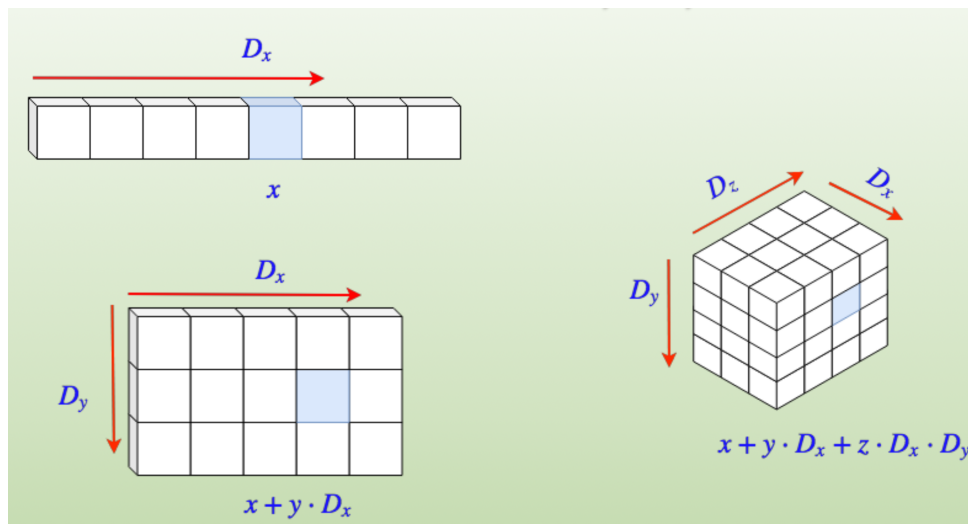


Figure 2.3: Indice univoco dei thread nei blocchi

2.4.2 Lancio di un kernel CUDA

Quando prendo una funzione e la lancio con un kernel significa che prendo quella funzione e la lancio sulla GPU, i parametri di configurazione di esecuzione sono fatti in questo modo:

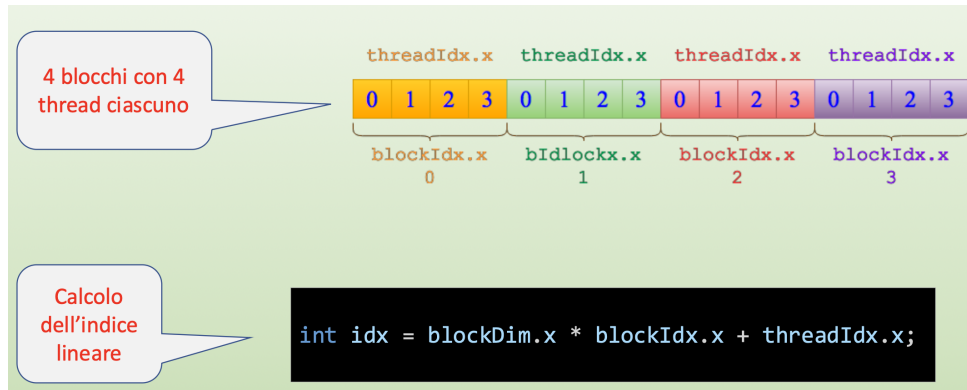


Figure 2.4: Organizzazione della griglia 1D e coordinate 1D

```
1 kernel<<<gridDim, blockDim>>>(args...);
```

Il primo valore `gridDim` specifica la dimensione della griglia di blocchi, mentre il secondo valore `blockDim` specifica la dimensione di ciascun blocco di thread. Gli argomenti `args...` rappresentano i parametri da passare al kernel.

2.4.3 Kernel concorrenti

Potrebbe verificarsi che numerosi kernel vengano lanciati sullo stesso host (dallo stesso processo o da processi diversi), potrei trovarmi nella situazione in cui ho diversi blocchi concorrenti sullo stesso device.

Esiste l'api `cudaDeviceSynchronize` che permette di sincronizzare l'esecuzione dei kernel, significa che la cpu si sincronizza con tutti i kernel, praticamente il lancio del kernel diventa bloccante.

2.4.4 Restrizioni del kernel

- Accede alla memoria globale
- Restituisce void
- non supporta numero variabile di argomenti, devono essere specificati
- non supporta le variabili statiche
- ha un comportamento asincrono rispetto al chiamante

2.4.5 Memoria

E' praticamente una trasposizione 1 a 1 della memoria in C, le funzioni servono per allocare memoria sul device in global memory (ricordasi che i thread accedono tutti alla global memory):

- `cudaMalloc`: alloca memoria sul device

- `cudaFree`: dealloca memoria sul device
- `cudaMemcpy`: copia dati tra host e device
- `cudaMemset`: imposta un valore nella memoria del device

Il trasferimento e l'allocazione di memoria sono operazioni sincrone, l'host si ferma in attesa del completamento di queste operazioni.

`cudaMalloc` ritorna un doppio puntatore, che e' quello che viene passato ai kernel per puntare correttamente allo spazio di indirizzamento del device (memoria globale), la size e' in byte, il puntatore e' di tipo void:

```
1  cudaError_t cudaMalloc(void** devPtr, size_t size);
```

restituisce un codice di errore. La `cudaFree` e' quella che libera la memoria.

I puntatori su host e device sono diversi, non e' possibile fare assegnamenti tra uno e l'altro invece bisogna usare la `cudaMemcpy`.