GPU computing

Simone Petta

A.A. 2024/2025

Contents

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. Introdurremo anche alcune librerie di python, andando a focalizzarci su elementi a basso livello.

Il parallel computing e' la necessita' di accellerare le computazioni avvalendosi di un numero molteplice di processori. Quando ho un processo di grande 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, rifrazone 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 unità 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 ci 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 parellelismo 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 indipendnti 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,

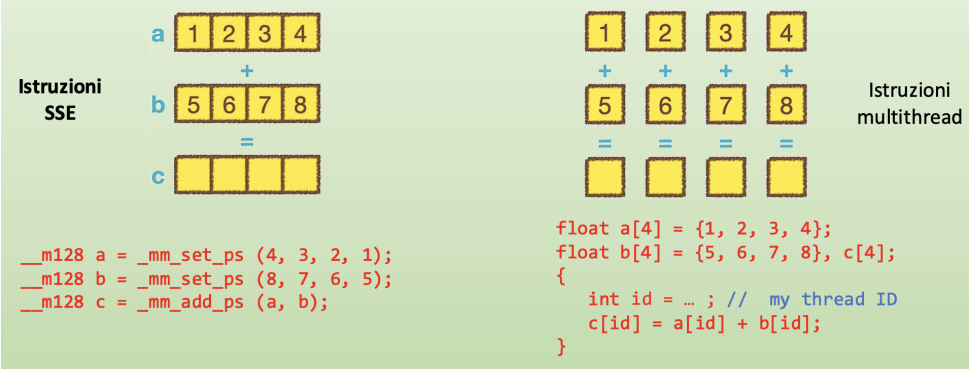


Figure 1.1: Confronto tra SIMT e SIMD

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 accellerate per le GPU useremo delle libreria gia' accellerate 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:

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

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

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

Il risultato finale e':

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

int main(void) {
    hello_world<<<1, 10>>>();
    cudaDeviceReset();
    return 0;
}
```

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 modello 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 prallelismo dell'architettura sottostante per essere eseuito da un linguaggio che ha in se elementi che permettono di usare il parallalismo. 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 conoscela 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 eseguit sequenzialmente secondo un flusso, anche i thread eseguono le cose sequenzialmente ma essendo molteplici possono essere parallali. 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

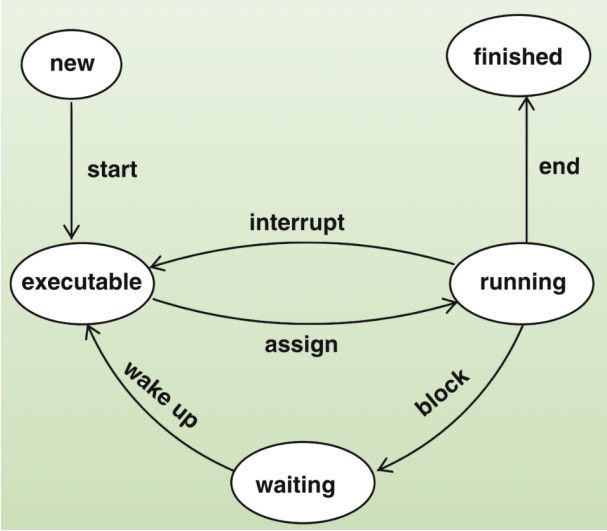


Figure 2.1: Stati di un thread

- 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.

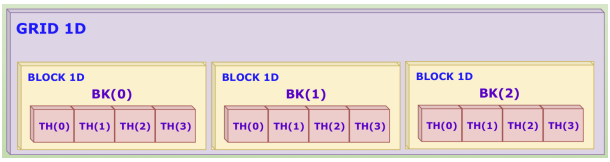


Figure 2.2: Organizzazione dei thread in CUDA

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

Il blocco di thread e' molto importante, dal punto di vista semantico ha un significato, e' 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: e' 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
```

```
*/
int main(int argc, char **argv) {

    // grid and block definition
    dim3 block(4);
    dim3 grid(3);

    // Print from host
    printf("Print from host:\n");
    printf("grid.x = %d\t grid.y = %d\t grid.z = %d\n", grid.x, grid.y,
        grid.z);
    printf("block.x = %d\t block.y = %d\t block.z %d\n\n", block.x,
        block.y, block.z);

    // Print from device
    printf("Print from device:\n");
    checkIndex<<<grid, block>>>();

    // reset device
    cudaDeviceReset();
    return(0);
}
```

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 differentemente rispetto alle dimensioni del blocco:

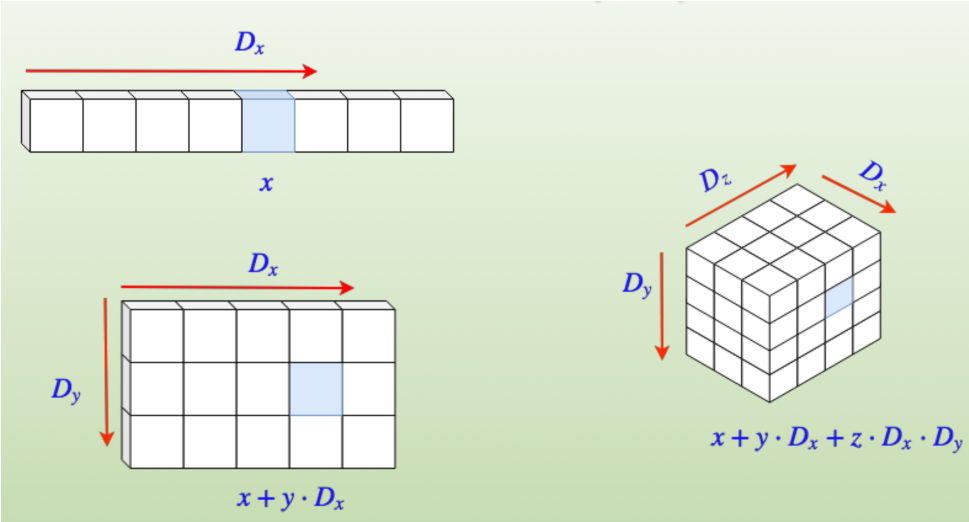


Figure 2.3: Indice univoco dei thread nei blocchi

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:

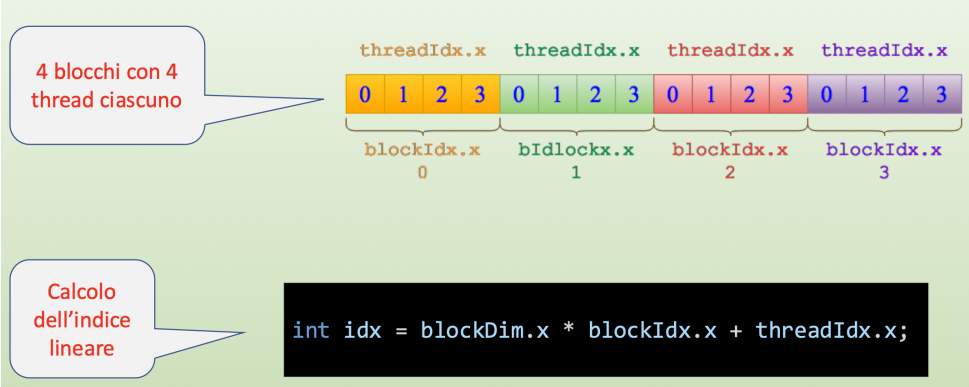


Figure 2.4: Organizzazione della griglia 1D e coordinate 1D

```
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.

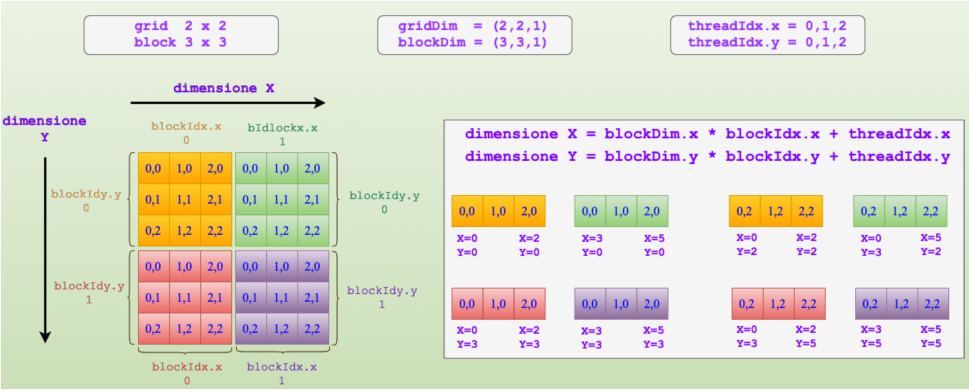
cudaMalloc ritorna un doppio puntatore, che e' quello che viene passato ai kernel per puntare correttamente allo spazio di indirizzamento del device (memoria globale), la size e' in byte, il puntatore e' di tipo void:

```
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.

Non c'e' sempre un mapping 1-1 tra i puntatori host e device, guardiamo ora il mapping tra griglia 2d e coordinate 2d. Se prendiamo una griglia di blocchi 2x2 mostriamo gli indici dei blocchi e dei thread all'interno, mostriamo che poiche' i tre campi vengono definiti quando definiamo le griglie dobbiamo capire come usare x e y, il campo x viene considerato come campo che varia sulle colonne della matrice per converso la y indica le righe della matrice e si muove in verticale pensando che l'origine e' in alto a sinistra. Posso creare un doppio ordine per x e y di indici e le coppie ordinate mi indicizzano tutte le entry della matrice:



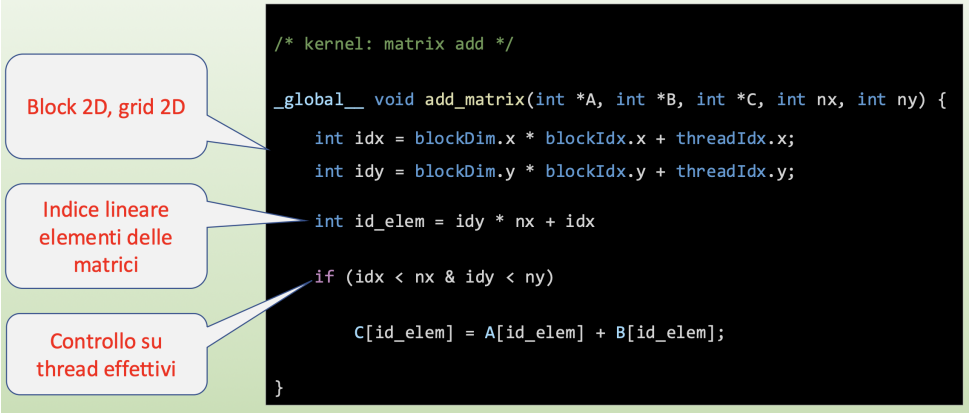
Abbiamo quindi prodotto delle coordinate 2d (x,y) per ogni thread all'interno della griglia 2D e a questo punto e' sufficiente spostarsi nella colonna e nella riga:

```
int ix = blockDim.x * blockIdx.x + threadIdx.x;
int iy = blockDim.y * blockIdx.y + threadIdx.y;
```

A questo punto l'indice linearizzato lo ricaviamo in questo modo:

```
int id = iy * nx + ix;
```

Se si deve trattare una matrice che rappresenta l'immagine, si possono organizzare blocchi di dimensione fissata (16x16) e calcolare la griglia di conseguenza (5x4).



2.5 Modello di esecuzione CUDA

Vogliamo vedere cosa succede a questa miriade di thread che vengono eseguiti nelle GPU, come si arriva ai core. Lo faremo in modo astratto dall'architettura, senza entrare nei dettagli specifici delle implementazioni hardware.

2.5.1 Panoramica

Quando abbiamo un thread che ha le sue risorse private (dei registri assegnati ad uno spazio locale) ed e' a sua volta in grado di chiamare delle funzioni. I thread vengono ordinati in blocchi, il blocco viene attribuito ad uno streaming multi-processor e devono essere eseguiti in parallelo su una risorsa. Vi e' l'uso e lo scambio di memorie veloci, la shared memory per la comunicazione inter-thread. Una griglia (grid) e' un array di thread block che eseguono tutti lo stesso kernel, legge e scrive in global memory e sincronizza le chiamate di kernel tra loro dipendenti. L'architettura della GPU e' disegnata come un array scalabile di streming multiprocessors, il parallelismo hardware della GPU e' ottenuto replicando questo elemento base. Gli elementi di uno streaming multiprocessor tipo sono:

- CUDA core
- Memory: global - shared - texture - constant
- caches
- registri per la memoria locale
- unita' load/store per i/o
- unita' per funzioni specializzate, come funzioni algebrice sin, cos, exp
- scheduler dei wrap

Gli vengono assegnati i blocchi e lui e' in grado di gestire i blocchi attraverso lo scheduler.

Mapping logico - fisico dei thread

In questo panorama dal punto di vista logico si ha un mapping tra thread e core (ogni thread ha un core) un blocco di thread viene eseguito su un streaming multiprocessor e la griglia viene mappata sul device.

Esecuzione

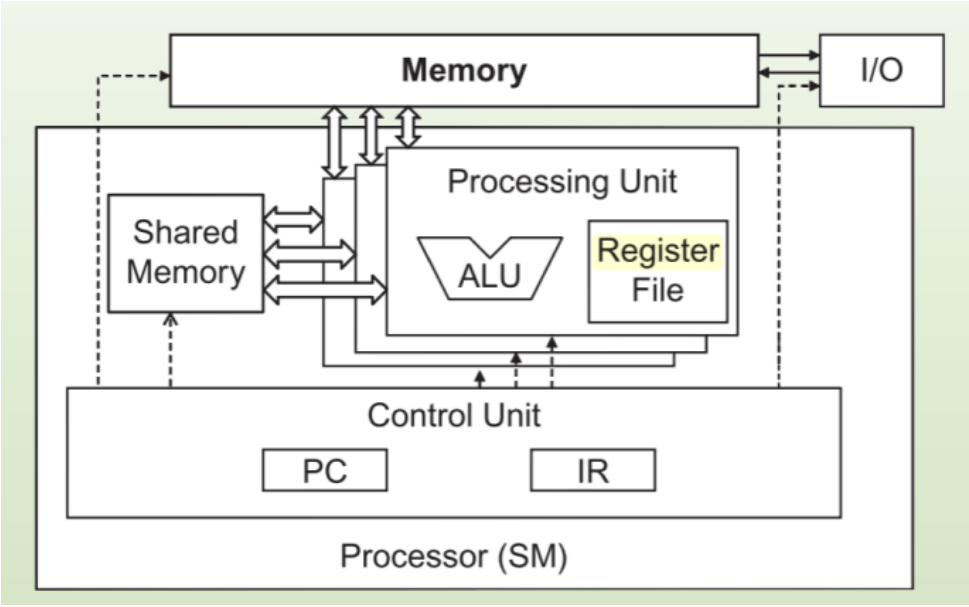
- mapping gerarchia thread in gerarchia processori sulla GPU
- GPU esegue uno o piu' kernel
- streaming multiprocessor gestisce i blocchi di thread
- un thread block e' schedulato solo su un SM
- I core eseguono i thread
- Gli SM eseguono i thread a gruppi di 32 thread chiamati warp

2.5.2 Warp: architettura SIMT

Gli warp sono gruppi di 32 thread (con ID consecutivi) che messi insieme agli altri warp costituiscono un blocco. Ha una semantica che richiama al modello SIMD perche' idealmente si vorrebbe che i thread nel worp eseguissero simultaneamente la stessa istruzione. Questa cosa e' un problema perche' ogni thread ha un flusso a se, quindi si ricorre al modello SIMT questo implica perdita di efficienza. Ogni thread ha il suo program counter e register state. Ogni thread puo' eseguire cammini distinti di esecuzione delle istruzioni. I thread che compongono un warp iniziano assieme allo stesso indirizzo del progamma. 32 e' un numero magico, ha origine dall'HW ed e' fondamentale nella programmazione CUDA per la scalabilita' e trasparenza. E' l'unita' minima di esecuzione che permette grande efficienza nell'uso della GPU. Ha un forte impatto sulle presatzioni degli algoritmi sviluppati. Concettualmente ha un comportamento modello SIMD ma nella pratica assume un modello SIMT.

HW multi-threading

Questi sono gli elementi HW di un ambiente multi-threading:



Registri

I registri sono usati per le variabili automatiche scalari e le coordinate dei thread:

```
__global__ void matrix_prod(float *a, float *b, float *c) {
    int Row = blockIdx.y * blockDim.y + threadIdx.y;
    int Col = blockIdx.x * blockDim.x + threadIdx.x;
    if (Row < N && Col < N) {
        float sum = 0.0f;
        for (int k = 0; k < N; k++) {
            sum += a[Row * N + k] * b[k * N + Col];
        }
        c[Row * N + Col] = sum;
    }
}
```

Logica warp

Dal punto di vista logico i warp mantiene la consistenza ma il blocco la perde un po', lo vediamo come blocco di thread ma quando viene passato alla macchina e diventa attivo questo viene ripartito in warp (la macchina vede una collezione di 32 thread) e poi lo passano al multiprocessore che si occupa dai warp che gli competono, questo accade indipendentemente dalla dimensione di griglia che abbiamo fissato. I warp condizionano come noi creiamo il kernel, ad esempio un kernel di 40 blocchi non ha molto senso, dato che dobbiamo eseguire gruppi di 32 thread alla volta e' meglio avere multipli di 32 (gli altri rimarrebbero inattivi e si mangerebbero risorse inutili). Uno warp puo' essere attivo o disattivo, e' attivo quando le risorse di computazione gli vengono assegnate, quando e' attivo si puo' trovare in diversi stati:

- selezionato: in esecuzione su un dato path (preso in carico dallo scheduler di warp)
- bloccato: non pronto per l'esecuzione
- candidato: eleggibile se tutti e 32 i core sono liberi e tutti gli argomenti della prossima istruzione sono disponibili

Scheduling dei blocchi

Come funziona lo scheduling dei blocchi, ogni streaming multiprocessor riceve un numero di blocchi (questo varia a seconda di quanti SM ho) questo per non lasciare IDLE gli SM.

Ripartizione risorse

Le risorse vengono ripartite a seconda della griglia che andiamo a definire e nel momento in cui vengono ripartite non c'e' context switch ed e' per questo che cio' che puo' essere attivo e' limitato. Esistono limiti imposti dall'HW, come ad esempio il numero di blocchi che possono essere attivi contemporaneamente su un SM, il numero di thread per blocco, il numero di registri per thread, la dimensione della shared memory per blocco, la dimensione della global memory per blocco.

L'occupancy e' il tasso tra warp attivi e numero massimo di warp per SM:

$$\text{occupancy} = \frac{\text{warp attivi}}{\text{numero massimo di warp per SM}}$$

Esiste un flag di uno strumento di profilazione per calcolare l'occupancy:

```
nvprof --metrics achieved_occupancy ./Application
```

2.5.3 Latency hiding

Il grado di parallelismo a livello di thread utile a massimizzare l'utilizzo delle unita' funzionali di un SM dipende dal numero di warp residenti e attivi nel tempo. La latenza e' definita come il numero di cicli necessari al completamento dell'istruzione. Per massimizzare il throughput occorre che lo scheduler abia sempre warp eleggibili ad ogni ciclo di clock. Si ha cosi' latency hiding quando i warp in attesa di esecuzione possono essere utilizzati per mascherare la latenza delle operazioni in corso. Classificazione dei tipi di istruzione che inducono latenza:

- Istruzioni aritmetiche: tempo necessario per la terminazione dell'operazione
- Istruzioni di memoria: tempo necessario al dato per giungere a destinazione

2.5.4 Warp divergence

La divergenza e' un altro nemico insieme alla latenza, si esplicita in modo chiaro, ci sono dei branch all'interno del codice (if else), semplicemente difronte ad un thread si ha un ritardi sistematico e perdiamo la natura del parallelismo di un warp. Fa cadere il modello SIMD e diventa SIMT.

```
__global__ void pari_dispari_1(int *c) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int a = 0, b = 0;

    if (tid % 2 == 0) {
        a = 2;
    } else {
        b = 1;
    }

    c[tid] = a + b;
}
```

Il compilatore fa dell'ottimizzazione per la divergenza (dove puo' ovviamente), aggiunge dei predicati alle istruzioni che verificano il vero o falso, calcolano quindi il predicato logico e le istruzioni del predicato e a questo punto i thread calcolano i predicati e non si sequenzializzano.

```
if (x < 0) {
    z = x - 2
}
else if (x >= 0) {
```

```
        z = x + 2
    }


    p = (x<0)
    p: z = x - 2
    !p: z = x + 2
```

2.5.5 Sincronizzazione

E' un altro aspetto fondamentale a livello di blocco, i thread possono seguire strade diverse e possono avere tempi diversi. Possiamo arrivare ad uno stadio in cui tutti dei thread hanno finito la loro esecuzione, e' importante quindi impostare delle barriere di sincronizzazione in cui tutti i thread si aspettano, questo per evitare delle race condition e per fare in modo di usare il risultato di altri thread. Si puo' fare a livello di sistema, aspettiamo che venga completato un dato task su entrambi host e divice:

```
    cudaError_t cudaDeviceSynchronize(void);
```

ma anche a livello di blocco, si tratta di utilizzare le primitive di sincronizzazione fornite da CUDA, come `__syncthreads()`, che permette di sincronizzare i thread all'interno di un blocco.

```
    __device__ void __syncthreads(void);
```

Ma anche a livello di warp, attendiamo che tutti i thread in un warp raggiungano lo stesso punto di esecuzione:

```
    __device__ void __syncwarp(mask);
```

Deadlock

Attenzione che sincronizzazione e divergenza possono portare ad un deadlock, quindi è importante progettare attentamente la logica del kernel per evitare situazioni di stallo.

```
    if (threadIdx.x < 16){
        F1();
        __syncthreads();
    }
    else if (threadIdx.x >= 16) {
        F2();
        __syncthreads();
    }
```

La prima meta' dei thread aspetta la seconda meta' per completare la propria esecuzione, creando una situazione di stallo.

2.5.6 Reduction

La reduction e' un operazione molto comune che e' la somma degli elementi di array di grandi dimensioni, se lo vogliamo fare in parallelo non e' cosi' scontata, possiamo fare

questa cosa quando l'operatore soddisfa le proprieta' commutativa e associativa, questo significa che possiamo fare questa operazione riordinando i dati come vogliamo

Il caso sequenziale e' banale:

```
float sum = 0.0f;
for (int i = 0; i < N; i++) {
    sum += array[i];
}
```

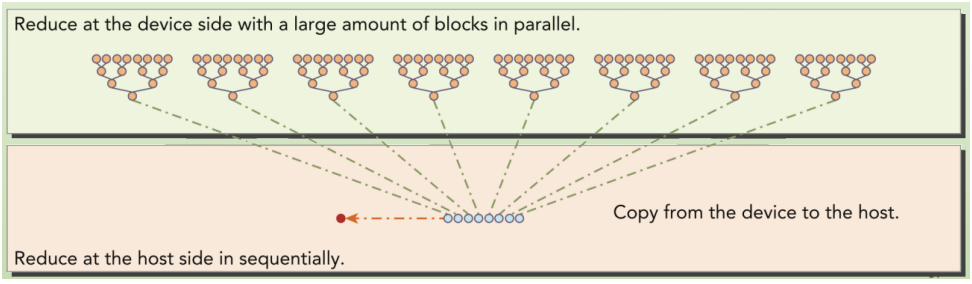
Il caso parallelo e' piu' complesso, dobbiamo dividere i dati in parti uguali, ogni thread calcola la somma della sua parte e poi si fa una somma finale. L'idea quindi e' di fare somma parziali memorizzate in-place nel vettore stesso, sommiami gli elementi di un livello sulla meta' degli elementi del livello sottostante, cosi' facendo teniamo un albero di log n (dove n sono i dati che devono essere sommati). Conviene ricondursi a tecniche ricorsive:

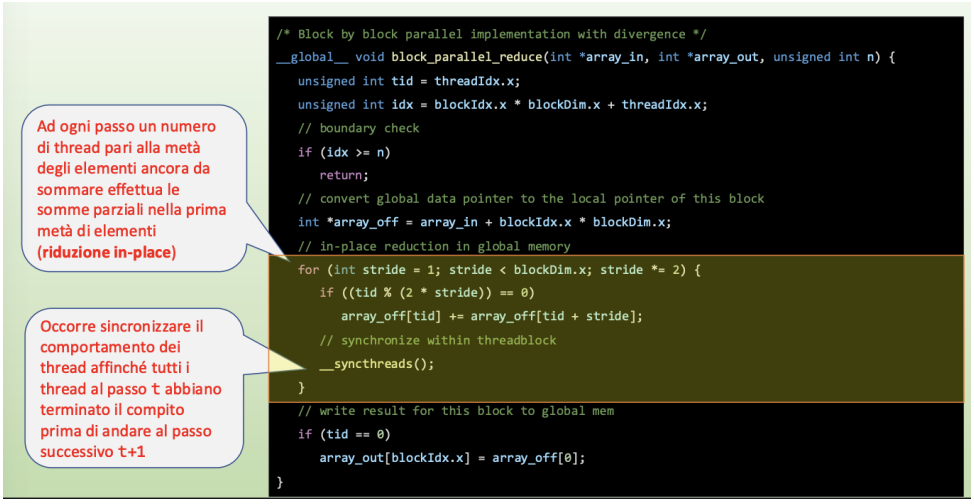
```
int recursiveReduce(int *data, int const size) {
    //terminazione
    if (size == 1) return data[0];

    //rinnova lo stride
    int const stride = size / 2;

    //riduzione in loco
    for (int i = 0; i < stride; i++) {
        data[i] += data[i + stride];
    }

    //chiamata ricorsiva
    return recursiveReduce(data, stride);
}
```

Da qui possiamo pensare ad una strategia ricorsiva:

- Ad ogni passo un numero di thread pari alla meta' degli elementi dell'array effettua le somme parziali nella prima meta' degli elementi, riduzione
- il numero di thread attivi si dimezza ad ogni passo rinnovare lo stride
- occorre sincronizzare il comportamento dei thread affinche' tutti i thread al passo t abbiano terminato il compito prima di andare al passo successivo t + 1 (analogo alla chiamata ricorsiva)

Dobbiamo pensare che i dati li dividiamo in blocchi, e poi combinare i risultati parziali usciti dai blocchi:





Questa soluzione ha il problema della divergenza dei warp, l'if in cui controlliamo che l'indice del thread sia pari, possiamo pero' eliminare l'if:

```
    for (int stride =1; stride < blockDim.x; stride *= 2) {
        //convert tid into local array index
        int index = 2* stride * tid;
        if(index < blockDim.x) {
            idata[index] += idata[index + stride];
        //synchronize within threadblock
        __syncthreads();
    }
```

2.5.7 Operazioni atomiche

Poniamo un problema per porre il pretesto di usare delle operazioni atomiche che di perse' garantiscono l'accesso corretto ed univoco ai dati anche se piu' thread accedono allo stesso dato. L'istogramma di un immagine e' l'intensita' del colore RGB in un'immagine. Per calcolarlo ci serve avere operazioni atomiche perche' dobbiamo incrementare l'array di intensita' RGB accedendo in modo concorrente agli stessi pixel con piu' thread.

Le operazioni atomiche in CUDA eseguono solo operazioni matematiche ma senza interruzione da parte di altri thread questo perche' sono tradotte in una singola istruzione, sono definite da cuda:

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

Le operazioni basilari sono:

- Matematiche: add, subtract, maximum, minimum, increment, decrement
- Bitwise: AND, bitwise OR, bitwise XOR
- Swap: scambiano valore in moemoria con uno nuovo

L'idea per calcolare l'istogramma e' semplice, si alloca una struttura dati (un array lungo 3 * 256) per mantenere i valori dell'istrogramma e , per ogni pixel dell'immagine si effettua una atomicAdd() per incrementare il valore della frequenza dei colori presenti nel pixel.

```
/*
 * Kernel 1D that computes histogram on GPU
 */
__global__ void ppm_histGPU(PPM ppm, int *histogram) {

   uint x = blockIdx.x * blockDim.x + threadIdx.x;

  // pixel out of range
   if (x >= ppm.width * ppm.height)
      return;

   // colors of the pixel
   color R = ppm.image[3 * x];
   color G = ppm.image[3 * x + 1];
   color B = ppm.image[3 * x + 2];

   // use atomic
   atomicAdd(&histogram[R], 1);
   atomicAdd(&histogram[G+256], 1);
   atomicAdd(&histogram[B+512], 1);
}
```

Prodotto matriciale

Come facciamo ad ottenere il prodotto tra matrici sfruttando la blocchettizzazione. Quello che dobbiamo fare e' definire una griglia definendo il mapping tra indici di thread e delle matrici. Da notare che le dimensioni rilevanti sono 2, numero di righe n della amtrice A e il numero m di colonne della matrice B.

Chapter 3

Architetture delle GPU

3.1 Loop Unrolling

L'idea e' di ottimizzare i loop ripetendo le istruzioni nel corpo del loop. Invece di eseguire un'iterazione alla volta, il compilatore o il programmatore espande il loop per eseguire più iterazioni in un singolo passaggio. Questo riduce l'overhead del controllo del loop e può migliorare l'uso della pipeline della CPU o della GPU. Come esempio vediamo il for della parallel reduction vista in precedenza:

```
    for (int stride = blockDim.x / 2; stride > 0; stride /= 2) {
        if (threadIdx.x < stride) {
            thisBlock[threadIdx.x] += thisBlock[threadIdx.x + stride];
        }
        __syncthreads();
    }
```

Per farlo e' possibile aggiungere un descrittore volatile che implica che ogni volta che l'istruzione viene eseguita va direttamente alla shared memory, evitando ottimizzazioni indesiderate da parte del compilatore passando per la cache:

```
    if (tid < 32) {
        volatile int *vmem = thisBlock;
        vmem[tid] += vmem[tid + 32];
        vmem[tid] += vmem[tid + 16];
        vmem[tid] += vmem[tid + 8];
        vmem[tid] += vmem[tid + 4];
        vmem[tid] += vmem[tid + 2];
        vmem[tid] += vmem[tid + 1];
    }
```

32 non e' un numero casuale ma stiamo facendo l'unrolling a livello di warp:

3.1.1 Block unrolling

Possiamo estendere l'idea di loop unrolling anche a livello di block. In questo caso, invece di unrollare le operazioni all'interno di un singolo warp, possiamo unrollare le operazioni su più thread all'interno di un block. Questo approccio può portare a un utilizzo più efficiente delle risorse della GPU e a una riduzione del numero di sincronizzazioni necessarie.

3.2 Architetture delle GPU

Il PCI Express sono i bus che permettono di unire CPU e GPU ed e' normalmente il collo di bottiglia del sistema. Ci sono dei controller da entrambe le parti che permettono di parlare con il BUS, all'interno di CPU e GPU ci sono bus per comunicazione interna molto piu' veloci.

Il PCI Express BUS parla con dei controller che a loro volta si interfacciano con i livello di cache piu' basso dell'architettura.

3.2.1 Compute Capability

E' il termine usato per descrivere la versione hw degli acceleratori GPU che appartengono alle famiglie di dispositivi e che hanno una data architettura interna. E' un parametro che

va passato al comiplatore, dobbiamo tenere in considerazione questi cambiamenti generazionali delle GPU per scrivere degli algoritmi efficienti per l'architettura che abbiamo a disposizione. Non c'e' grande compatibilita' tra versioni.

3.2.2 Architettura interna GPU

Ci sono i vari straming multi processors che comunicano usando una shared memory che e' una memoria cache, poi ci sono i vari bus e controllori che parlano con la CPU.

Host interface

E' un controller interno alla GPU che si interfaccia verso bus PCie. Consente alla GPU di trasferire dati da e per la CPU. Ha due tipi di memorie cache: una L1 interna al sm e una L2 condivisa tra SM. Ogni cache l1 e' parallela e combina attivita' con le unita' load/store. Un memory controller dedicato trasporta dati dentro e fuori dalla GDDR5 global memory.

Giga-thread scheduler

Si occupa di assegnare ai vari SM disponibili ti blocchi del kernel con una politica di round robin. L'assegnamento di blocchi a SM e' un attivita' veloce, l'esecuzione molto piu' lenta. La variabili gridDim, blockIdx e blockDim sono pasate dal giga thread scheduler ad ogni core della GPU che eseguira' il corrispondente blocco.

3.2.3 Scheduler di warp

Ogni blocco e' assegnato a un SM. Ogni SM contiene 2 warp scheduler e 2 instruction dispatch unit. I due scheduler selezionano 2 warp da eseguire in parallelo e distribuiscono un'istruzione a ognuno dei 16 core o alle 16 unita' o alle 4 fpu. L'archittura fermi puo' trattare simultaneamente 8 blocchi = 48 warp per SM per un totale di 1536 thread alla volta residenti su un singolo SM.

3.2.4 Fermi: kernel e memoria

La memoria e' fissa e' puo' essere divisa in...

3.2.5 Transparent scalability

E' la capacita' di eseguire lo stesso codice applicativo su diverse architetture, numero variabile di compute core e di SM.

3.2.6 Parallelizzazione dinamica

Potremmo voler lanciare un kern da un kernel, il kernel padre puo' consumare l'output prodotto dal figlio tutto senza la CPU:

```
__global__ void kernel_figlio() {
    // Codice del kernel figlio
}

__global__ void kernel_padre() {
    // Codice del kernel padre
    kernel_figlio<<<1, 1>>>();
}
```

Bilanciare il lavoro

Se mi accorgo che un blocco di dati e' molto grande e richiede tanto tempo potrei dividere questo blocco in altri blocchi e dividere il lavoro con altri thread. Questo e' utile per evitare che un singolo blocco di dati rallenti l'intero processo di calcolo.

Chapter 4

Memorie

Esiste una gerarchia di memoria, partono da memorie molto veloci come i registri, le cache arriviamo poi alla main memory e alla disk memory, sempre piu' lente e sempre piu' grandi.

Ci sono i principi di localita' e gerarchia di memorie. Il principio di localita' spaziale afferma che i programmi tendono ad accedere a una porzione ristretta di memoria in un dato momento, se accedo ad i e' molto probabile che acceda ad un indirizzo li' vicino (delta i), il che rende vantaggioso mantenere i dati frequentemente utilizzati nelle memorie piu' veloci. La gerarchia di memorie sfrutta questo principio organizzando le memorie in livelli, con memorie piu' veloci e costose in alto e memorie piu' lente e economiche in basso. Mentre la localita' temporale si riferisce alla tendenza ad eseguire la stessa istruzione piu' volte in un breve intervallo di tempo.

Il modello di memoria CUDA, abbiamo visto che esistono delle memorie programmabili che hanno pattern di accesso utilizzabili con le API di sistema e devono essere gestite direttamente, devono essere gestite in maniera diversa per la lettura e la scrittura. Poi ci sono delle memorie non programmabili, le cache L1 e L2.

4.1 Registri

Sono le memorie piu' veloci e piu' piccole. Sono ripartiti tra i warp attivi (assegnati ad ogni thread). Le variabili locali sono quelle che vengono assegante ai registri senza che sia specificato nessuno dei qualificatori generalmente risiede in un registro. Meno registri usa il kernel, piu' blocchi di thread e' probabile che risiedano sul multiprocessore, quindi meno ne usiamo meglio e'.

4.2 Local Memory

E' una memoria lenta come la global memory. E' una memoria locale ai thread ed e' usata per contenere le variabili automatiche (grandi) non contenute nei registri:

- Array locali i cui indici non possono essere determinati a compile-trasferimento
- Strutture locali grandi che consumano troppo spazio registro
- Ogni variabile che non puo' essere allocata nei registri a causa numero limitato (register spilling)

Risiede nella device memory, pertanto gli accessi hanno stessa latenza e ampiezza di banda della global memory e sono soggetti anche agli stessi vincoli di di coalescenza. Il compilatore nvcc si preoccupa della sua allocazione e non e' controllata dal programmatore.

4.3 Constant Memory

Risiede in device memory ed ha una cache dedicata in ogni SM. E' ideale per ospitare dati a cui si accede in modo uniforme e in sola lettura. Una variabile dichiarata come `__constant__` viene dichiarata con scope globale, ad di fuori di qualsiasi kernel. L'utilizzo migliore come dice il nome e' allocarci dati che devono essere distribuiti tra tanti kernel.

```
cudaError_t cudaMemcpyToSymbol(const void *symbol, const void *src,
    size_t count);
```

E' pari a 64KB per tutte le compute capability.

4.4 Texture Memory

E' una memoria in sola lettura, dotata di cache e risiede in device memory. E' particolarmente utile perche' include un supporto hw efficiente per filtraggio o interpolazione floating-point nel processo di lettura dei dati. E' ottimizzata per la localita' spaziale 2D, quindi dati espressi sotto forma di matrici. I thread di un warp che usano la texture memory per accedere a dati 2D hanno migliori prestazioni rispetto a quelle standard.

4.5 Shared Memory

E' una memoria programmabile on-chip con un bandwidth molto piu' alto e minor latenza della local o global memory. E' suddivisa in moduli della stessa ampizza, chiamati bank che possono essere acceduti da un thread alla volta. Abbiamo 32 banchi di memoria che devono essere utilizzati in maniera opportuna per essere acceduti simultaneamente. Ogni SM ha una quantita' limitata di shared memory che viene ripartita tra i blocchi di thread. Bisogna farne un uso parco ancora una volta per non limitare il numero di warp attivi. Con i blocchi di thread condivide anche lifetime e scope. E' fondamentale per la comunicazione inter-thread di un blocco, poiche' permette di scambiare dati in modo rapido e con bassa latenza. L'accesso deve essere sincronizzato per mezzo delle seguente CUDA runtime call:

```
__syncthreads();
```

4.5.1 Organizzazione

La SMEM dell'arch Fermi sono suddivisi in blocchi da 4 byte (chiamate word) da 32 bit (0 64). Ogni word puo' contenere tipi base (1 int, 1 float, 2 short, 4 char). la bandwitdth e' di 32 bit ogni 2 cicli di clock. Nei 32 banchi andiamo ad accedere simultaneamente a 32 word, ed e' l'accesso piu' veloce che possiamo avere:

4.5.2 SMEM runtime

Viene ripartita tra tutti i blocchi residenti in un SM. Maggiore e' la shared memory richiesta da un kernel, minore e' il numero di blocchi attivi concorrenti. Il contenuto della shared memory ha lo stesso lifetime del blocco cui e' stata assegnata. L'accesso e' per warp: caso migliore 1 transazione x 32 thread, caso peggiore 32 transazioni.

4.5.3 Pattern di accesso

Se un'operazione di load o store eseguita da un warp richiede al piu' un accesso per bank, si puo' effettuare in una sola transizione il traferimento dati dalla shared memory al warp.

- Caso broadcast: un singolo valore letto da tutti i thread in un warp da un singolo bank
- Caso parallelo: un singolo valore letto da un singolo bank. In questo caso le cose vanno bene, per ogni thread accediamo al corrispondente banco di memoria.
- Conflitto doppio: tutti i thread di un warp richiedono una word di indice doppio il threadId.x
- Nessun conflitto: strutture di 3 word come, ad esempio, le terne di punti nello spazio.

4.5.4 Conflitto di accesso

Il bank conflit e' quando diversi indirizzi di shared memory insistono sullo stesso bank. L'hw effettua tante transizioni quante ne sono necessarie per risolvere il conflitto, riducendo le prestazioni.

4.5.5 Configurazione SMEM / cache L1

Ogni SM ha 64 Kb di memoria on-chip, la sherd memory e L1 cache condividono questa risorsa HW. Cuda fornisce due metodi per configurare la L1 cache e la shared memory:

```
1   cudaError_t cudaDeviceSetCacheConfig(cudaFuncCache cacheConfig);
```

Dove l'argomento cacheConfig specifica come deve essere ripartita la memoria:

- cudaFuncCachePreferNone: no preference
- cudaFuncCachePreferShared: prefer shared memory 48KB shared e 16 Kb L1
- cudaFuncCachePreferL1: prefer L1 cache
- cudaFuncCachePreferEqual: equal preference

4.5.6 Osservazioni

Il latency hiding e' sempre una cosa indesiderata, il ritardo che puo' incorrere alla SMEM e all'ottenimento dei dati non e' in generale un problema anche in caso di conflitti di banchi, perche' se si riesce a tenere alto il numero di warp attivi, thread in esecuzione e blocchi letti in esecuzione si ha sempre un sostituto nel momento in cui un warp va in attesa. Inter-block conflict, non esiste un conflitto tra thread appartenenti a blocchi differenti, il problema sussite solo a livello di warp dello stesso blocco. Il modo piu' semplice per avere prestazioni elevate e' quello di fare in modo che un warp acceda a word consecutive in memoria shared. Caching: con lo scheduling efficace le prestazioni anche in presenza di conflitti a livello di SMEM sono di gran lunga migliori se paragonate a quelle in cui il cammino che i dati percorrono passa attraverso la cache L2 o peggio, arriva in global memory.

4.5.7 Allocazione statica delle SMEM

Una variabile in shared memory puo' anche essere dichiarata sia locale a un kernel sia globale in un file sorgente. E' dichiarata con il qualificatore `__shared__`. Puo' essere dichiarata sia staticamente sia dinamicamente.

4.5.8 Allocazione dinamica della SMEM

Se la dimensione non e' nota a tempo di compilazione e' possibile dichiarare una variabile adimensionale con la keyword extern:

- Dinamicamente si possono allocare solo array 1D
- puo' essere sia internal al kernel sia esterna

4.5.9 Uso tipico

- Caricare i dati dalla device memory alla shared memory
- Sincronizzare i thread del blocco al termine della copia che ognuno effettua sulla shared memory (cosi' che ogni thread possa elaborare dati certi nel prosieguo)
- Elabora i dati in shared memory
- Sincronizza per essere certi che la shared memory contenga i risultati aggiornati
- Scrivi i risultati dalla device memory alla host memory

4.5.10 Prodotto matriciale con SMEM

Nella versione senza memoria shared ogni thread e' responsabile del calcolo di un elemento della matrice prodotto. Dobbiamo tenere conto che abbiamo una suddivisione logica dei thread che stanno in un blocco e degli elementi in memoria corrispondenti. Passi con shared memory:

- Occorre caricare in memoria shared i dati relativi ad ogni blocco prima di svolgere le somme e i prodotti
- Ogni thread accumula i risultati di ognuno di questi prodotti (scandendo tutti i blocchi) in un registro
- Alla fine scrive su global memory

Qui l'idea e' di trasferire gli elementi di riga e colonna delle due matrici in shared memory perche' dobbiamo accedere ad ogni elemento tante volte e quindi l'accesso a quella memoria sarebbe piu' veloce.

L'idea e':

- Caricare il primo tile
- Effettuare le somme parziali per ogni cella

- Riptere per il numero di tile contenuti nelle zone delle matrici da caricare
- Scrivere i risultati dalla shared memory alla global memory

4.5.11 Prodotto convolutivo con SMEM

La convoluzione e' un prodotto tra due vettori, tendenzialmente un vettore piu' grande ed uno piu' piccolo chiamato kernel o maschera.

Una volta che abbiamo la scrittura matematica l'istanziazione e' con vettori, nell'esempio vediamo il segnale e la maschera, viene evidenziato il segnale centrale perche' e' quello in output della convoluzione: Occorre fare shiftare la maschera con uno stride = 1 e calcolare prodotti e somme.

Da notare che ci sono casi particolari, nell'esempio partiamo dal terzo elemento perche' c'e' corrispondenza reale con la maschera, si potrebbe anche adottare approcci diversi relativamente a dati mancanti.

Questa cosa si estende naturalmente a piu' dimensioni, questo e' l'esempio del 2D:

Ha senso pensare all'uso della shared memory perche' alcune celle della matrice di input vengono riutilizzate più volte durante il calcolo della convoluzione (tutte quelle sotto la maschera sicuramente).

Questo e' un problema ideale per il parallel computing:

- Determinare la mappa tra thread ed elementi di output
- Semplice approccio nei casi 1D e 2D e' organizzare i thread in grid corrispondenti in modo tale che ogni thread calcoli un elemento della matrice di convoluzione
- Problema: non soddisfa per inefficienza negli accessi alla global memory

Tiling

Divido i dati in blocchi (ad esempio, 16 elementi in 4 blocchi da 4 thread), ogni thread nel blocco fa un prodotto della convoluzione. Per ridurre l'accesso alla global memory, in cache/memoria condivisa si tengono i dati a cui l'accesso è fatto più frequentemente, ovvero i valori del "blocco di dati" assegnato al block (tutti i thread devono calcolare sullo stesso insieme di dati, o quasi), tenendo conto della dimensione della maschera (serve avere i dati "adiacenti" al blocco, quelli che sbordano (sarebbe molto utile un'immagine, si capirebbe subito)).

I dati da caricare in smem sono più dei thread nel blocco ("alone" che va al di fuori del blocco di dati stesso); in smem carico tutti i possibili dati a cui il blocco deve fare l'accesso.

Chi carica che dati in memoria? I dati esterni al blocco potrebbero essere anche più del blocco stesso (maschera "grossa"). La soluzione è dare un ordine ai thread e dividere il più equamente possibile i caricamenti in memoria tra i thread del blocco. Si può fare facendo un tiling dei dati da caricare: il blocco viene "ripetuto" sui dati da caricare, secondo un ordine dei thread.

4.6 Global Memory

Nei computer moderni esiste una gerarchia di memorie per minimizzare latenze e massimizzare throughput. In genere, si ha l'illusione virtuale di una grande memoria, tutta a bassa latenza, anche se la memoria con effettivamente bassa latenza è poca e si ha una memoria ad alta capacità e alta latenza.

All'interno delle GPU abbiamo, dalla latenza più alta alla più bassa:

- Device Memory
- L2 Cache
- L1/shared
- Registers

Le gerarchie di memorie, comprese quelle CUDA, fanno fede ai principi di:

- **Località spaziale**: se l'istruzione all'indirizzo i viene eseguita, probabilmente dopo verrà eseguita quella all'indirizzo $i + \Delta i$
- **Località temporale**: se un'istruzione viene eseguita al tempo t, probabilmente verrà eseguita anche al tempo $t + \Delta t$ (dove Δt è piccolo)

Registers: Le memorie più veloci in assoluto, con lifetime del kernel. Vengono ripartiti tra i warp attivi, le variabili dichiarate nel codice device senza qualificatori generalmente risiedono in un registro.

Meno registri usa il kernel, più blocchi di thread è probabile che risiedano sull'SM (il compilatore usa un'euristica per ottimizzare questo parametro). **Register spilling**: se si usano più registri di quelli consentiti le variabili si riversano nella local memory.

Local Memory: Si tratta di una memoria *lenta* (collocata off-chip, alta latenza, bassa bandwidth). Si tratta di una memoria locale ai thread.

Usata per contenere le variabili automatiche (grandi) non contenute nei registri, o per le variabili al di fuori causa spilling. La local memory risiede nella device memory, pertanto gli accessi hanno stessa latenza e ampiezza di banda della global memory e sono soggetti anche agli stessi vincoli di coalescenza.

Da CC 2.0 ci sono parti poste in cache L1 a livello di SM e in cache L2 a livello di device. Il compilatore `nvcc` si preoccupa della sua allocazione e non è controllata dal programmatore.

Constant Memory: Risiede nella device memory (64K per tutte le CC) ed ha una cache dedicata in ogni SM (8K). Definibile tramite l'attributo `__constant__`.

Ospita dati in sola lettura, ideale per accessi uniformi. Ha scope globale, va dichiarata al di fuori di qualsiasi kernel e viene dichiarata staticamente, quindi è visibile a tutti i kernel nella stessa unità di compilazione.

La constant memory può essere inizializzata dall'host usando:

```
1 cudaError_t cudaMemcpyToSymbol(const void* symbol,
2     const void* src, size_t count)
```

Lavora bene quando tutti i thread di un warp leggono dallo stesso indirizzo di memoria (raggiunge l'efficienza dei registri); se i thread di un warp leggono da indirizzi diversi allora le letture vengono serializzate, riducendo l'efficienza.

Texture Memory: Risiede nella device memory e (può avere) una read-only cache per-SM ed è acceduta solo attraverso di essa. La cache include un supporto hardware efficiente per filtraggio o interpolazione floating-point nel processo di lettura dei dati.

Ottimizzata per la località spaziale 2D, quindi dati espressi sotto forma di matrici. I thread in un warp che usano la texture memory per accedere a dati 2D hanno migliori prestazioni rispetto a quelle standard, quindi è adatta per applicazioni in cui servono classiche elaborazioni di immagini/video. Per altre applicazioni l'uso della texture memory potrebbe essere più lento della global memory.

Global Memory: La più grande, con più alta latenza e più comunemente usata memoria su GPU. Ha scope e lifetime globale (da qui il nome). Dichiarazione (codice host):

Statica	`__device__ int a[N];`
Dinamica	`cudaMalloc((void **)&d_a, N);  cudaFree(d_a);`

Corrisponde alla memoria fisica, con "global" si intende una divisione logica. L'accesso da parte di thread appartenenti a blocchi distinti può potenzialmente portare a modifiche incoerenti. La global memory è accessibile attraverso transazioni da 32, 64, o 128 byte; le transazioni avvengono solo per gruppi di valori, non si può accedere a un valore singolo.

I valori contenuti nella memoria allocata non sono inizializzati, ma si possono inizializzare con dati provenienti dall'host (`cudaMemcpy()`) oppure con un valore specifico

```
1 cudaError_t cudaMemset(void* devPtr, int value, size_t count)
```

Assegna il valore `value` a tutti gli indirizzi contenuti nel blocco di memoria.

La memoria allocata e' opportunamente allineata per ogni tipo di variabile. La `cudaMalloc()` restituisce `cudaErrorMemoryAllocation` in caso di fallimento.

Lo **specificatore** `__device__` indica una variabile che risiede unicamente sul device. Risiede nella memoria globale (e quindi oggetti distinti per device diversi), ha il lifetime del contesto CUDA in cui è stata creata. Può essere acceduta da tutti i thread e dall'host tramite la libreria runtime:

- `cudaGetSymbolAddress()`, `cudaGetSymbolSize()`: per ottenere indirizzo e dimensione di una variabile, rispettivamente
- `cudaMemcpyToSymbol()`, `cudaMemcpyFromSymbol()`: per copiare verso e da una variabile, rispettivamente

Riassunto dichiarazione di variabili:

QUALIFIER	VARIABLE	MEMORY	SCOPE	LIFESPAN
	`float var`	Register	Local	Thread
	`float var[100]`	Local	Local	Thread
`__shared__`	`float var`	Shared	Block	Block
`__device__`	`float var`	Global	Global	Application
`__constant__`	`float var`	Constant	Global	Application

4.6.1 Pinned memory

La pinned memory (o page-locked memory) in CUDA è una tecnica che serve per ottimizzare il trasferimento dei dati tra la memoria del sistema (RAM) e la memoria della GPU (VRAM).

Si vuole evitare il page fault della virtual memory (CPU, di default la memoria host allocata è paginabile). Esistono delle primitive per definire una memoria pinned, ovvero viene tolta la pagina dal meccanismo di virtualizzazione in modo che l'host non possa "toglierla" mentre il device la deve usare. Blocca la memoria in modo da poter fare trasferimenti asincroni al device.

Una volta "pinnata", la memoria non sparirà dal sistema di virtualizzazione automatico della memoria host, quindi si può lavorare su quella memoria in maniera asincrona. La pinned memory può essere acceduta direttamente dal device, in modalità asincrona. Può essere letta e scritta con una bandwidth più alta rispetto alla memoria paginabile.

Da notare che eccessi di allocazione di pinned memory potrebbero far degradare le prestazioni dell'host (ridurre la memoria paginabile inficia l'uso della virtual memory), Per allocare esplicitamente memoria pinned:

```
cudaError_t cudaMallocHost(void **devPtr, size_t count);
```

E per deallocarla:

```
cudaError_t cudaFreeHost(void *devPtr);
```

Questa allocazione sostituisce la malloc "normale", su host. Rende i trasferimenti host-device significativamente più veloci, al costo di un tempo più alto di allocazione.

4.6.2 Unified Virtual Addressing UVA

Si vuole avere un unico spazio di indirizzamento tra CPU e tutte le GPU. Tutti i puntatori (CPU e GPU) appartengono allo stesso spazio di indirizzi virtuali, di conseguenza è possibile passare un puntatore da host a device e viceversa senza ambiguità, entrambi possono "capire" a cosa punta quell'indirizzo.

La unified memory è una memoria "*comoda*", fornisce un puntatore unico per tutte le CPU e GPU presenti nel sistema. Spazio di indirizzamento unico per CPU e GPU.

Con "**Managed Memory**" si fa riferimento ad allocazioni della unified memory. All'interno di un kernel si possono usare entrambi i tipi di memoria:

- managed memory, controllata dal sistema
- un-managed memory, controllata esplicitamente dall'applicazione

Tutte le operazioni CUDA valide sulla memoria del dispositivo sono valide anche sulla memoria managed.

Per fare allocazione dinamica:

```

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

"rimpiazza" `cudaMalloc`, la `flag` indica chi condivide il puntatore con il device:

- `cudaMemAttachHost`: solo la CPU
- `cudaMemAttachGlobal`: anche tutte le altre GPU

Nuova **keyword**: `__managed__`, si tratta di un qualifier che denota scope globale, accessibile da CPU e GPU.

Con l'uso "misto" di memoria bisogna porre attenzione alla sincronizzazione tra CPU e GPU, onde evitare problemi.

4.6.3 Pattern di Accesso alla Global Memory

Gli accessi alla memoria del dispositivo possono avvenire in transazioni da 32, 64 o 128 byte. Le applicazioni GPU tendono (a volte) ad essere limitate dalla memory bandwidth, quindi massimizzare il throughput effettivo è importante.

In generale, per rendere efficienti le transazioni in memoria:

- minimizzare il numero di transazioni per servire il massimo numero di accessi
- considerare che il numero di transazioni e throughput ottenuto variano con la CC

Per migliorare le prestazioni in lettura e scrittura occorre ricordare che:

- le istruzioni vengono eseguite a livello di warp e gli accessi in memoria dipendono dalle operazioni svolte nel warp
- per un dato indirizzo si esegue un'operazione di loading o storing (gestione diversa)
- i 32 thread presentano una singola richiesta di accesso, che viene servita da una o più transazioni in memoria

In base a come sono distribuiti gli indirizzi di memoria, gli accessi alla stessa possono essere classificati in pattern distinti. Tutti gli accessi a memoria globale passano dalla cache L2, molti passano anche dalla L1. Se entrambe le memorie vengono usate gli accessi sono da 128 byte, altrimenti, se viene usata solo la L2, gli accessi sono a 32 byte.

Per le architetture che usano cache L1, queste possono essere esplicitamente abilitate o disabilitate a compile time.

Bisogna rispettare allineamento e coalescenza per sfruttare al meglio le transazioni di memoria; per avere accessi in memoria efficienti è necessario combinare in un unica transazione accessi multipli a memoria allineati e coalescenti.

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

Accesso **coalescente:** quando tutti i 32 thread in un warp accedono a un blocco contiguo di memoria.

In un SM i dati seguono pipeline attraverso i seguenti tre cache/buffer paths dipendentemente da quale tipo di device memory si accede:

- L1/L2 cache
- Constant cache

- Read-only cache

L1/L2 cache rappresenta il default path. Il fatto che un'operazione di `load` in global memory passi attraverso la cache L1 dipende da CC e compiler options.

Scritture: Le write vengono servite in modo diverso, non viene usata la cache L1, ma le `store` sono cachate solo in L2, prima di essere inviate alla device memory in segmenti da 32 byte; vengono trasferiti 1,2 o 4 segmenti alla volta.

Quando forzati a fare accessi (letture/scritture) non coalescenti si può usare la shared memory come "passaggio" per rendere le operazioni effettive in memoria coalescenti.

AoS vs SoA: I dati possono essere divisi in:

- **Array of Structures AoS**:

```
struct Particle { float x, y, z; };
Particle* P;
float x = P[idx].x;  // stride = sizeof(Particle)
```

  Questo porta a distanza tra accessi (stride) alta, rompendo la coalescenza

- **Structure of Arrays SoA**:

```
float *Px, *Py, *Pz;
float x = Px[idx];   // stride = 1
```

  In questo modo lo stride è ridotto, riducendo così il numero di transazioni necessarie

TL;DR: Un warp può effettuare accessi

- coalescenti: i 32 thread leggono dati contigui, massima efficienza
- non coalescenti/strided: dati a distanza > 1, possono servire più transazioni per la stessa quantità di dati, fino a 32 diverse

In generale (per CC superiori a 2) le transazioni coprono 128 byte. All'interno di un singolo segmento da 128 byte, la memoria è organizzata in "banks" (4 da 32 byte solitamente), anche se un thread legge solo 4 byte, dovrà trasferire l'intero bank.

La cache L1 serve load/store anche con granularità a 32 byte.

Matrice trasposta con SMEM

Il problema del parallelizzare l'operazione di trasposizione di una matrice è che in lettura gli accessi sono per riga, quindi coalescenti, mentre le scritture sulla matrice trasposta sono per colonna, quindi non coalescenti.

Un'idea per risolvere il problema può essere:

- Caricare dalla global memory alla smem le celle che il blocco corrente deve trasporre, riga per riga
- Leggere una colonna in smem e scrivere una riga in global memory

Esempio:

```
__shared__ float tile[BDIMY][BDIMX];
// Coordinate originali
int y = blockIdx.y * blockDim.y + threadIdx.y;
int x = blockIdx.x * blockDim.x + threadIdx.x;

// ...
// Trasferimento in smem e sincronizzazione

// Nuovi indici
int y = blockIdx.x * blockDim.x + threadIdx.y;
int x = blockIdx.y * blockDim.y + threadIdx.x;

// ...
// Scrivere sulla matrice output,
//      controlli invertiti tra riga e colonna
```