GPU Computing
Appunti - Bassi Francesca

Accesso a Lagrange

- da terminale (qualsiasi, da putty, dal subsystem linux su windows, sul terminale di git su win)
  ssh emailStudente@cuda.lagrange.di.unimi.it

- Controllare se il compilatore CUDA è presente con il comando di linux
  whereis nvcc

- Controllare che le schede CUDA siano montate sul sistema
  ls -l /dev/nv*

- Chi usa windows con visual studio, includere anche queste direttive per non aver problemi
  #include "cuda_runtime.h"
  #include "device_launch_parameters.h"

Comando	Descrizione
`cudaDeviceReset()`	pulisce tutte le risorse associate al device, inizializza la porzione di device coinvolta nel processamento
`cudaDeviceSynchronize()`	fa aspettare all'host che il device finisca tutte le esecuzioni/task in corso, è bloccante per l'host.
`cudaMalloc()`	corrisponde a `malloc` in c
`cudaMemcpy()`	corrisponde a `mempcy` in c
`cudaMemset()`	corrisponde a `memset` in c
`cudaFree()`	corrisponde a `free` in c
`_synchthreads()`	sincronizza tutti i thread all'interno di un blocco, non si possono sincronizzare i thread di blocchi differenti.
`cudaMemcpyToSymbol()`	come memcpy, copia da un symbol, cioè da una variable, c'è anche la variante `cudaMemcpy`**`From`**`Symbol()`
`cudaMallocHost()`	alloca memoria nella memoria host, ma in pinned memory
`cudaFreeHost()`	libera la memoria nella pinned memory dell'host
`cudaMallocManaged()`	alloca memoria nella UVA (unified virtual addressing), si usano come flag `cudaMemAttachHost` per condividere la memoria solo con l'host, `cudaMemAttachGlobal` per condividere la memoria anche con altre GPU
`cudaHostAlloc()`	con il flag `cudaHostAllocMapped`, si può mappare della memoria device su memoria host, in modo da poter sfruttare la memoria host quando è insufficiente la memoria a disposizione sul device. Ciò permette di evitare il trasferimento esplicito tra host e device (zero-copy memory)

`cudaStreamCreate()`	crea uno stream
`cudaStreamDestroy()`	distrugge uno stream
`cudaMemcpyAsync()`	permette il trasferimento asincrono su pinned memory
`cudaStreamSynchronize()`	blocca l'host finchè lo stream non è terminato, per aspettare la terminazione di tutti gli stream: `cudDeviceSynchronize()`
`cusaStreamQuery()`	viene interrogato e controllato il completamento dello stream
`cudaStreamCreateWithFlags`	Passando come argomento lo stream interessanto e il flag `cudaStreamNonBlocking`, si genera un NON-NULL-stream che non si blocca rispetto ad un NULL-stream
`cudaEventCreate()`	crea un evento
`cudaEventDestroy()`	distrugge un evento
`cudaEventRecord()`	si registra il verificarsi di un evento su uno stream
`cudaEventSynchronize()`	permette di bloccare l'host fino al verificarsi di un evento
`cudaEventQuery()`	permette di controllare se si è verificato un evento
`cudaStreamWaitEvent()`	blocca lo stream finchè non occorre un evento
`cudaElapsedTime()`	serve per misurare il tempo trascorso tra due eventi in ms
`cudaGetDeviceCount()`	ritorna il numero di device utilizzabili in un sistema multi-GPU
`cudaGetDeviceProperties()`	passando il numero di device, ritorna le proprietà del device
`cudaSetDevice()`	serve per impostare quale GPU eseguirà le operazioni
`cudaDeviceCanAccessPeer()`	se è possibile accedere alla global memory peerdevice
`cudaDeviceEnablePeerAccess()`	permette la comunicazione p2p
`cudaDeviceDisablePeerAccess`	disabilita la comunicazione p2p
`__constant__`	variabili destinate alla Constant memory
`__shared__`	variabili destinate alla Shared memory
`__host__`	Eseguito dall'host e chiamato solo dall'host, facoltativo
`__device__`	variabili destinate alla Global memory. Eseguito dal device, chiamato solo dal device, ritorna void
`__global__`	Eseguito dal device, chiamato dall'host o dal device

15/3/19 ---------- lezione 1

1 - Termini dell'Hardware

La **CPU** (Central Processing Unit) è la fonte primaria di calcolo per la macchina, è pensata per essere general purpose e ha la potenza guidata dalla legge di Moore.

Gli aspetti critici che limitano la CPU sono rappresentati da:

- L'aumento del clock, infatti non si può aumentare la frequenza perchè ci sono dei problemi di dissipazione ancora da risolvere.

$$P_{dyn} = \alpha \cdot C_L \cdot V^2 \cdot f$$

con α che rappresenta la probabilità di switch, C_L è la capacità, V è la tensione e f frequenza di clock

- La legge di Moore ha subito una battuta d'arresto, ha raggiunto l'apice.
- Per sopperire al declino del single core si sono adottati processori a multicore

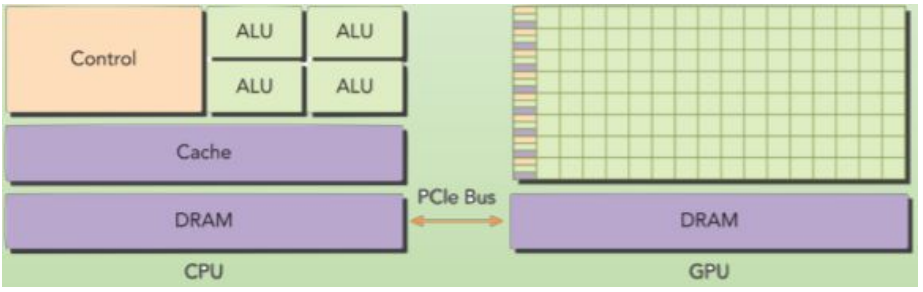
La **GPU** (Graphics Processing Unit) nasce per la grafica e molto più potente per le applicazioni grafiche rispetto alla CPU, quindi si sposta il carico computazionale della grafica da CPU ad un hardware dedicato chiamato GPU. Utile per problemi altamente parallelizzabili perchè dotata di migliaia di core, rispetto alla CPU che ne ha 2, 4, 8.
Viene introdotto il paradigma **GP-GPU** (General purpose - GPU).

1.1 - GP-GPU

- ***General-purpose** GPU*: La GPU è ottimizzata per svolgere calcoli aritmetici, infatti è utilizzata per eseguire computazioni scientifiche, multimediali o ingegneristiche.
- ***Modello eterogeneo***: L'uso di CPU e GPU è essenziale, insieme danno vita al modello di calcolo di co-processing
- ***Separazione***: La parte sequenziale di un programma viene eseguito dalla CPU, la porzione parallela invece eseguita in maniera accelerata dalla GPU.
- ***Trasparente***: L'utente non vede nulla di differente, se non un programma più veloce.
- ***Mapping***: Una function sulla GPU implica riscrivere la function per esporla al parallelismo della GPU aggiungendo le notazioni "C". Il Kernel è ciò che viene eseguito sulla GPU.
- ***Problema***: Bisogna massimizzare la potenza di calcolo, minimizzando l'energia consumata (la GPU non è per niente ottimizzata in termini di consumo energetico)

1.2 - Architetture Eterogenee

La GPU è un **coprocessore**, non può essere utilizzata come piattaforma standalone, infatti è chiamata anche **acceleratore**.
La **CPU** viene chiamata **host**, mentre la **GPU** viene chiamata **device**.
La GPU necessita di trasferimenti diretti di memoria da parte della CPU, che operano congiuntamente attraverso il bus PCI-Express.
La differenza nei due dispositivi è rappresentata principalmente dall'esecuzione dei task



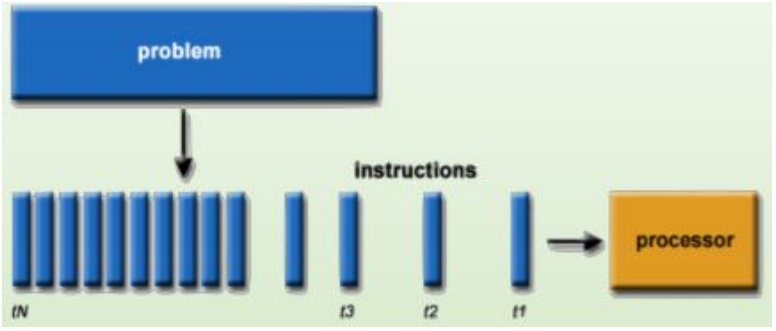
CPU: Pochi core ottimizzati per l'elaborazione sequenziale

GPU: architettura che consiste in migliaia di core progettati per trattare task (semplici) multipli simultaneamente

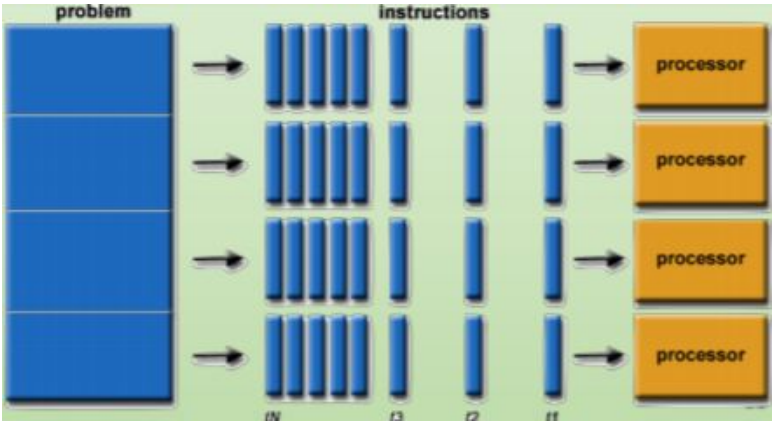
Le applicazioni ibride che sfruttano l'architettura eterogenea, sono caratterizzate da **codice host** (eseguito dalla CPU) e **codice device** (eseguito invece dalla GPU).

- Il **codice host** (della CPU) è responsabile della gestione dell'ambiente, input/output e gestione dei dati per il device, cioè cosa passare come informazioni e cosa impostare prima di caricare i task sul **device**. La **CPU** è ottimizzata per sequenze di operazioni in cui il controllo di flusso è **impredicibile**.
- Il **codice device** invece è usato per accelerare l'esecuzione della sua porzione di codice basandosi sul parallelismo dei dati. I task della **GPU** sono carichi dominati da un flusso di controllo semplice (pochi if e non usare switch!)

2 - Parallelismo - Parallel Computing



Nel caso *sequenziale*, il problema è suddiviso in una serie discreta di istruzioni, **ogni istruzione è eseguita una dopo l'altra**, in maniera sequenziale su un singolo processore.



Nel caso *parallelo*, il problema viene suddiviso in parti che possono essere eseguite concorrentemente, ogni parte è successivamente ridotta ad una serie di istruzioni e tutte **queste parti vengono eseguite simultaneamente** su diversi processori. Un meccanismo di controllo si occupa di raccogliere i risultati parziali.

2.1 - Parallel Computing

Parallel Computing: *Collezione di unità di elaborazione che comunicano e cooperano per risolvere velocemente un problema di grandi dimensioni.*

Lo ***scopo*** è quello di incrementare la velocità delle computazioni.
Il ***vantaggio*** che porta è che molti calcoli vengono svolti in contemporanea. I problemi possono essere suddivisi in problemi di dimensioni più piccole (task) e possono essere risolti simultaneamente e in modo indipendente tra loro (es. prodotto tra matrici).
La ***prospettiva***, il programmatore deve gestire manualmente le risorse di calcolo (core, memoria, I/O).
Il parallelismo è di due tipi nei programmi

- Dei **task**, cioè quando molti task operano in modo parallelo e indipendentemente (funzioni distribuite su diversi core)
- Dei **dati**, cioè quando si può lavorare su più dati allo stesso tempo (dati distribuiti tra thread multipli)

Gli aspetti principali del parallel computing sono legati alle tecnologie esistenti: architetture di calcolo parallelo e programmazione parallela.

2.2 - Architetture parallele

Parallelismo a livello di bit sulla GPU

- infatti le **word size:** da 4, 15, 32, 64 bit per avere più accuratezza nelle operazioni di floating point e spazi di indirizzamento. Si emula il sistema.

Parallelismo via pipeline

- si cerca di sovrapporre le esecuzioni di più istruzioni su pipeline,
- **gli stadi di esecuzione** sono *fetch → decode → execute → write back*
- ***vantaggio***: diversi stadi possono operare in parallelo
- **grado di parallelismo**: numero di stadi di pipeline (parallelismo a livello di istruzione)
- **problema**: la dipendenza tra i dati può limitare il parallelismo
- **necessità**: una ridotta dipendenza tra i dati (data dependency, quando un'istruzione richiede dati prodotti da una istruzione precedente)

Parallelismo dato da molteplici unità funzionali

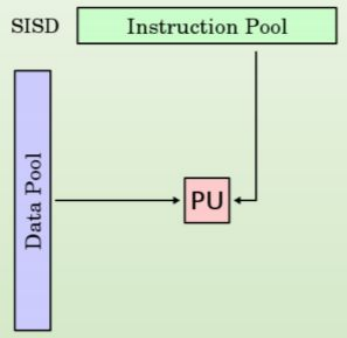
- **Unità funzionali**: tra loro indipendenti come le ALU - arithmetic logical unit, FPU - floating point unit, load/store…
- Le istruzioni possono essere eseguite in parallelo su diverse unità funzionali

Parallelismo a livello di processo o thread

- **Approccio alternativo**: impiegare la logica integrata per porre all'interno del chip diverse unità indipendenti detti core
- **Flusso di controllo**: un singolo flusso di controllo sequenziale può avere più flussi
- **Programmazione parallela:** coordinare flussi, accessi multipli alla memoria, condividere cache e coordinare *sincronizzazione* e *concorrenza*.

2.3 - Tassonomia di Flynn

- **SISD** (Single Instruction Single Data)

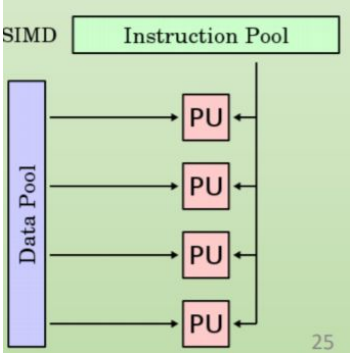


Una unità di computazione che accede a programma e dati

Nessun parallelismo,
le operazioni vengono eseguite sequenzialmente

PU = processing unit

- **SIMD** (Single Instruction Multiple Data)

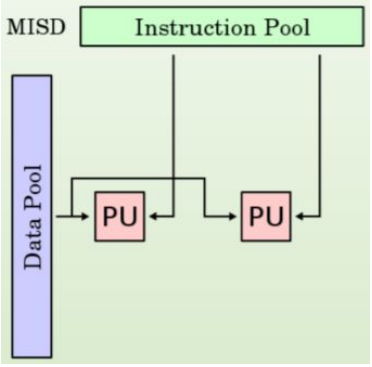


Molte unità di computazione con accesso a memoria privata (distribuita o condivisa), per dati e con una memoria globale unica per le istruzioni.

Stessa istruzione: ad ogni passo, un processore centrale invia un'istruzione alle unità che leggono i dati (tutti i processori eseguono la stessa cosa)

Applicazioni con alto grado di parallelismo ne traggono beneficio (computer graphics, simulazioni ecc..)

- **MISD** (Multiple Instruction Single Data)

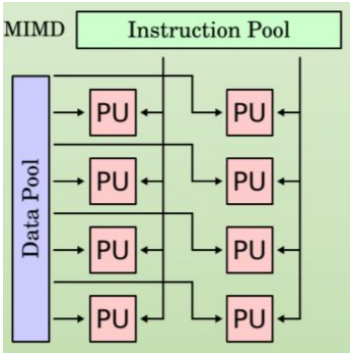


Molte unità di computazione con accesso a memoria privata di un programma, accesso comune a memoria globale unica per i dati.

Stesso dato: ad ogni passo, ogni unità ottiene lo stesso dato e carica un'istruzione da eseguire nella memoria privata

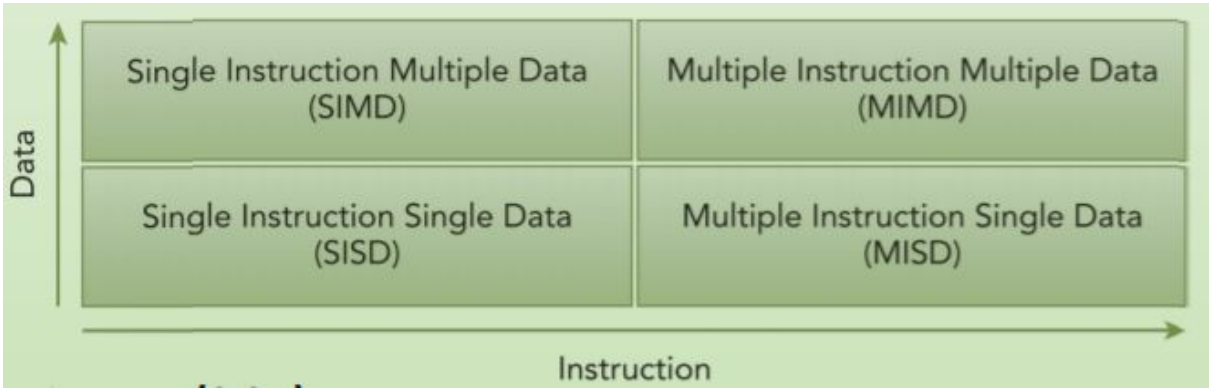
Parallelismo a livello istruzione su medesimo dato (non è diffuso)

- **MIMD** (Multiple Instruction Multiple Data)



Molte unità di computazione con accesso separato a istruzioni e dati su memoria condivisa o distribuita

Passo: in uno step ogni unità carica la propria istruzione e la esegue su un dato separatamente e asincronicamente dagli altri.
Degli esempi possono essere: processori multicore, sistemi distribuiti e data center.



SISD (Single Instruction Single Data)

- Architettura seriale con un solo core (il computer)
- Ogni istruzione viene eseguita una alla volta e su un dato per volta

SIMD (Single instruction Multiple Data)

- Architettura parallela con core multipli
- Tutti i core eseguono nello stesso tempo la stessa istruzione, ma su dati diversi

MIMD (Multiple Instruction Multiple Data)

- Architettura parallela in cui core multipli operano su diversi stream di dati
- Ogni stream esegue istruzioni indipendenti
- Si includono delle SIMD come sub componenti

MISD (Multiple instruction Single Data)

- Architettura non diffusa, dove ogni core lavora sullo stesso stream di dati mediante diversi stream di istruzioni indipendenti.

2.4 - SIMT MODEL (al di fuori del Flynn Model)

SIMT (Single Instruction Multiple Threads)

- **Multithreading**: in sistemi operativi multitasking rappresenta un diffuso modello di programmazione ed esecuzione che consente a thread multipli di coesistere all'interno del contesto di un processo in esecuzione.
- **Modello SIMT** è usato in parallel computing dove si combina il modello **SIMD** con il **multithreading**
- **Condivisione** di risorse, ma esecuzioni indipendenti
- **Estensione** al modello di elaborazione che prevede che un processo abiliti esecuzioni paralleli su sistemi multiprocessore o multicore
- Implementato su diverse GPU, fondamentale sulle GP-GPU (nvidia)

SIMT vs **SIMD**

Il modello **SIMT** ha tre caratteristiche in più del modello **SIMD:**

1. Ogni thread ha il proprio **instruction address counter**
2. Ogni thread ha il proprio **register state** e in generale un **register set**
3. Ogni thread può avere un **execution path** indipendente

3 - Architettura NVIDIA

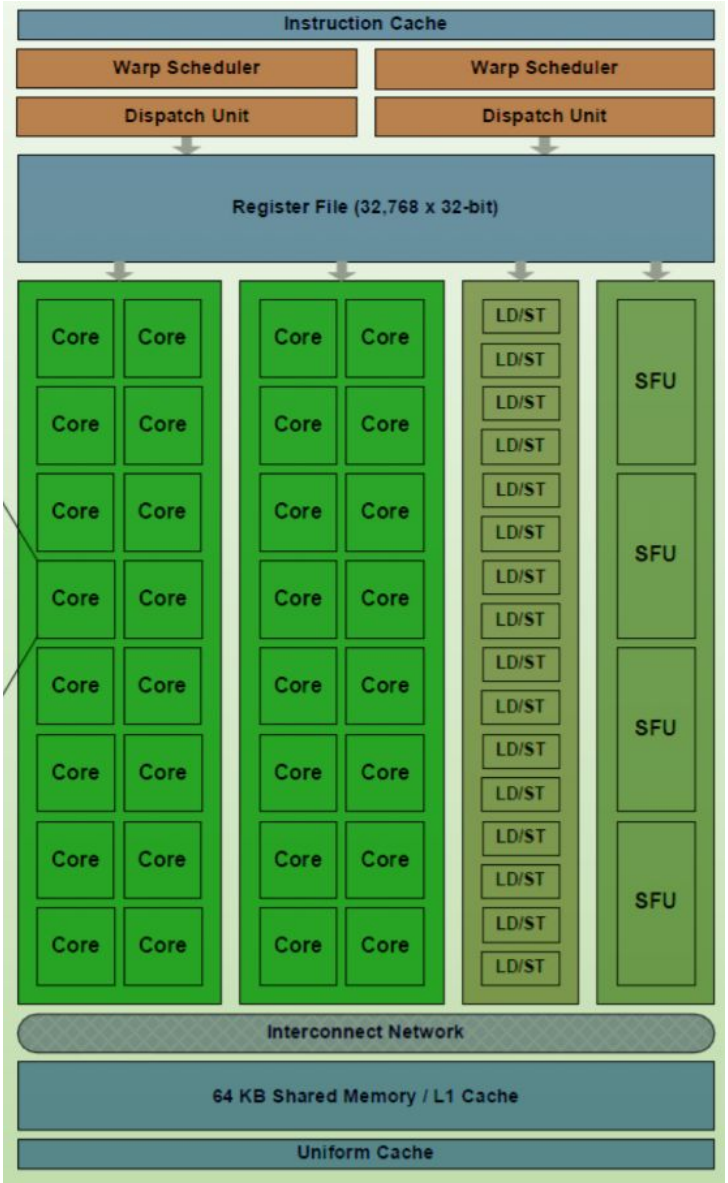
Le architetture si dividono in “generazioni” e ogni generazione ha un nome specifico:

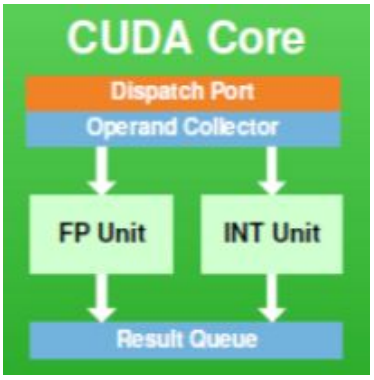
- Fermi (2010)
- Kepler (2012)
- Maxwell (2014)
- Pascal (2016)
- Volta (2018)

Tra di loro si distinguono in base alla **compute capability** (cc).

3.1 - Architettura GPU

L’architettura GPU è costruita attorno ad un array scalabile di **Streaming Multiprocessors (SM)**. Ogni **SM** in una **GPU** è progettato per supportare l’esecuzione concorrente di centinaia di thread.
Molteplici **SM** eseguono thread in gruppi di 32 thread chiamati **warp**. Tutti i thread in un **warp** eseguono la **stessa istruzione nello stesso momento**.

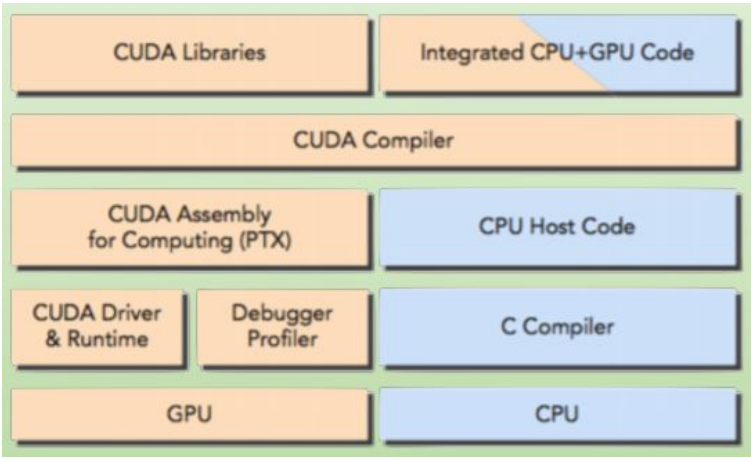




Ogni core è chiamato **CUDA CORE**, è un vettore di unità di processamento, lavora su una singola operazione ed è il **building block** (cioè l'unità minima) di una **SM**.

3.2 - CUDA: Compute Unified Device Architecture

Permette di programmare in general purpose sulle GPU di nvidia, permette l'esplicita gestione della memoria della GPU.
La GPU viene vista come un **compute device** che diventa co-processore della CPU (host), ha una sua DRAM (la global memory) e permette di far partire i thread in parallelo.



Un programma CUDA consiste in due parti
1. il codice **host** eseguito dalla CPU
2. il codice **device** eseguito dalla GPU

Il compilatore di CUDA **nvcc** si occupa di separare il codice **host** da quello **device** durante la compilazione.

Problema: La somma di due vettori `C[*] ← A[*] + B[*]`
su una CPU

```
float *A = malloc (N * sizeof(float));
float *B = malloc (N * sizeof(float));
float *C = malloc (N * sizeof(float));
for (int i = 0 ; i < N; i++)
      C[i] = A[i] + B[i];
```

Su una GPU

```
__global__ void vector_sum (int *A, int *B, int *C) {
int idx = blockDim.x + blockIdx.x + threadIdx.x;
if (idx < N)
      C[idx = A[idx] + B[idx]
}
```

4 - Programmare con CUDA

HelloWorld in C

Esecuzione su CPU
creare un file sorgente con l'estensione **.cu**

```
#include <stdio.h>

int main (void){
      printf("Hello world from CPU!\n");
}
```

compilare con il compilatore cuda **nvcc**
`nvcc hello.cu -o hello`

eseguire
`./hello`

Esecuzione su GPU
Creare un file sorgente con l'estensione **.cu**

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

int main (void){
      printf("Hello World from CPU!\n");
      helloWorldFromGPU<<<1, 10>>>();
      cudaDeviceReset();
}
```

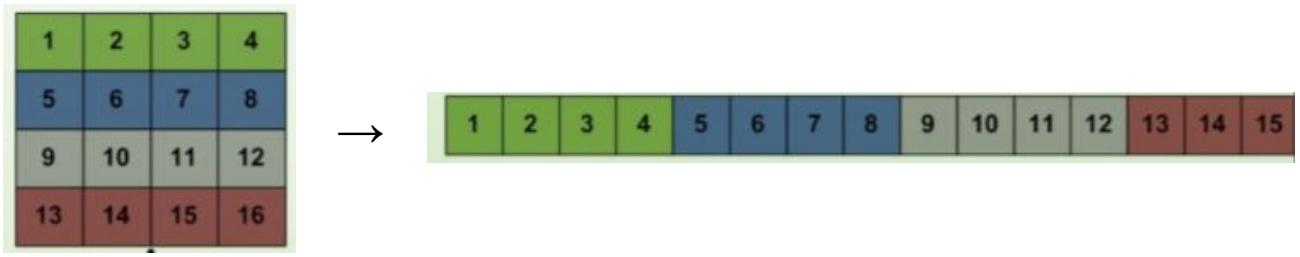
`__global__`: qualificatore che indica al compilatore che è un metodo da eseguire su GPU
`helloWorldFromGPU`: il metodo che deve essere eseguito su GPU
`<<<1,10>>>`: il modo per invocare il kernel, in questa maniera si attiva il metodo specificato con 10 thread

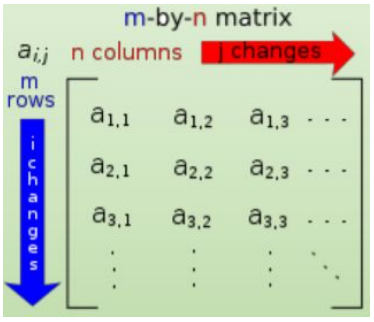
compilare con
`nvcc -arch sm_20 hello.cu -o hello`

eseguire
`./hello`

4.2 - Le matrici

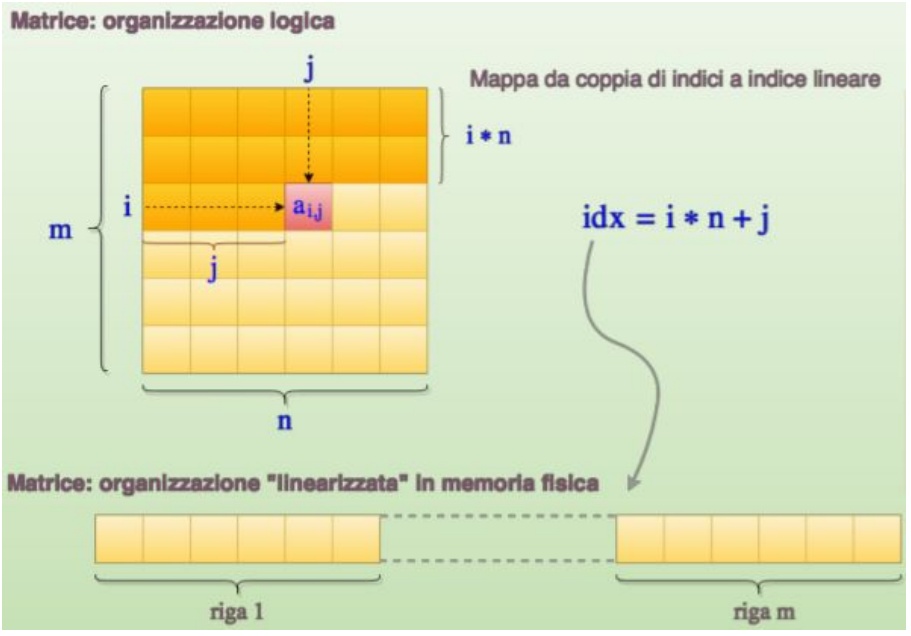
Le matrici per C sono un lungo array continuo





n x m, dove n sono le righe e m le colonne

```
//azzeramento della matrice generata dinamicamente
A = (int *) malloc (n * m sizeof(int));

for (int i = 0; i < m; i++){
       for(int j = 0; j < n; j++)
             int idx = i * n + j
             A[idx] = 0;
}
```



22/3/19 ---------- lezione 2

5 - Modelli per sistemi paralleli

Ci sono 4 classificazioni di livelli di astrazione:

- **Modello Macchina**

Il livello più basso che descrive hardware e sistema operativo (registri, memoria, I/O).
Il linguaggio assembly basato su questo livello di astrazione

- **Modello Architetturale**

Rete di interconnessione di piattaforme parallele, organizzazione della memoria e livelli di sincronizzazione tra processi, modalità di esecuzione delle istruzioni tipo **SIMD** o **MIMD**

- **Modello Computazionale**

Modello formale che fornisce metodi analitici per fare predizioni teoriche sulle prestazioni (esempio in base al tempo o sull'uso delle risorse). Ci fornisce dei metodi e tool che ci permettono di usare al meglio gli strumenti sottostanti, tool che ci permettono di costruire e ottenere prestazioni elevate rispetto ai modelli sottostanti. Il modello PRAM estende RAM per architetture parallele.

- **Modello di programmazione parallela**

Si definisce un computer parallelo definendolo come si possa codificare un algoritmo

  - Comprendere la semantica del linguaggio, librerie, compilatore e tool di profiling
  - Di che **tipo** sono le computazioni parallele (instruction/procedural/parallele level)
  - Dare **specifiche** implicite o esplicite per il parallelismo
  - Definire la modalità di **comunicazione** di informazioni tra unità di computazione
  - Definire i meccanismi di **sincronizzazione** per gestire le computazioni e comunicazioni tra diverse unità che operano in parallelo.
  - Fornire il concetto di **Parallel loop** (iterazioni indipendenti), **Parallel task** (moduli assegnati ai processori distinti eseguiti in parallelo)
  - Un programma parallelo è eseguito da processori in un ambiente parallelo tale che IN OGNI PROCESSORE si ha uno o più flussi di esecuzione (i flussi possono essere **processi** o **thread)**
  - Il modello di programmazione parallela ha una organizzazione dello spazio di indirizzamento di tipo **distribuito** (non ci sono variabili shared, si usa il message passing), oppure **condiviso** (uso di variabili shared per scambiare le informazioni)

5.1 - Processi e thread

Processo
Consiste in vari thread che condividono lo stesso spazio di indirizzamento. Processi **distinti** hanno distinte aree di indirizzamento. L'uso differisce per memoria e ambiente di esecuzione. I **processi** sono adatti in ambiente con **memoria distribuita**

Thread
Un thread è un **flusso** di istruzioni di un programma, che assieme a registri particolari e variabili o strutture dati, viene schedulato come **unità indipendente** nelle code di esecuzione dei processi della CPU. I Thread hanno id univoci per permettere al sistema di eseguirli e identificarli. I **thread** sono adatti in ambienti con **memoria condivisa**

5.1 - Passi di Parallelizzazione

Si assume che la parallelizzazione si effettui a partire da un programma o un **algoritmo sequenziale**.

La computazione parallela deve essere suddivisa in **task** dei quali deve essere stabilita la dipendenza. Un task è una **sequenza** di computazioni **eseguite** dallo **stesso processore o core**. Dipendentemente dal modello di memoria, un **task** può accedere alla memoria condivisa o usare tecniche di *message passing*. Dipendente dal programma, i **task** possono essere **staticamente fissati** fin dall'inizio dell'esecuzione o **creati dinamicamente** durante.
Il tempo di computazione del **task** è detto **granularità**, se troppo fine si ha overhead di scheduling, se troppo grezza si può perdere in efficienza.
I **task** sono assegnati ad un processo o thread (scheduling) cercando di bilanciare il carico tra quest'ultimi. Il sistema operativo si occupa di assegnare fisicamente i processi o thread ai vari core disponibili.

5.2 - Caso del Flipping di un'immagine BitMap e Thread

5.2.1 - Formato BitMap (BMP)

Il formato di file windows che permette operazioni di lettura e scrittura molto veloci senza perdita di qualità (richiede una maggio quantità di memoria rispetto ad altri formati). Le immagini bitmap possono avre profondità di 1, 4, 8, 16, 32 bit per pixel.
Le bitmap con 1, 4, 8 bit contenfgono una tavolozza per la conversione dei possibili indici numerici nei rispettivi colori. La versione 3 del formato non supporta il canale alfa o i metadati. I dati raw hanno **54 byte** per l'header (width, height, inizio-fine tavolozza, ecc) poi ogni **pixel occupa 3 byte** (cioè lo spazio (RGB)) della matrice.

5.2.2 - Utilizzo dei Thread

Bisogna incudere la libreria nel sorgente
```
#include <pthread.h>
```

e compilare specificando la libreria
```
gcc <opzioni> -pthread
```

Le funzioni fornite dalle API
`pthread_create()`: crea un thread e lanciare un job
`pthread_join()`: attende la terminazione tra i thread per sincronizzarli, bisogna specificare il thread di cui bisogna attendere la terminazione.
`pthread_attr()`: consente di inizializzare gli attributi del thread
`pthread_attr_setdetachstate()`: assegna/modifica gli attributi appena inizializzati

Usare l'attributo `detachstate`

```
. . .

#define MAXTHREADS   128

int NumThreads;                  // Total number of threads working in parallel
int ThParam[MAXTHREADS];         // Thread parameters ...
pthread_t ThHandle[MAXTHREADS];  // Thread handles
pthread_attr_t ThAttr;           // Pthread attributes
void (*FlipFunc)(pel** img);     // Function pointer to flip the image
void* (*MTFlipFunc)(void *arg);  // Function pointer to flip the image, multi-threaded version

. . .

if (NumThreads > 1) {
pthread_attr_init(&ThAttr);
pthread_attr_setdetachstate(&ThAttr, PTHREAD_CREATE_JOINABLE);
```

Creazione di thread

Ogni thread viene creato usando la funzione `pthread_create()`.
Il terzo argomento della `pthread_create()` indica quale funzione deve eseguire.
Il quarto argomento della `pthread_create()` sono i parametri alla funzione eseguita.

```
for (i = 0; i < NumThreads; i++) {
    ThParam[i] = i;
    ThErr = pthread_create(&ThHandle[i], &ThAttr, MTFlipFunc, (void *)&ThParam[i]);
    if (ThErr != 0) {
        printf("\nThread Creation Error %d. Exiting abruptly... \n",ThErr);
    exit(EXIT_FAILURE);
}
```

Join di thread

Il thread che effettua il join si blocca finchè uno specifico thread non termina.
Il thread che effettua il join può ottenere lo stato del thread che termina.
`pthread_join(x)` indica "*wait unitl thread with the hanfle number x is done*"

```
. . .
pthread_attr_destroy(&ThAttr);
for (i = 0; i < NumThreads; i++) {
    pthread_join(ThHandle[i], NULL);
}
. . .
```

Esecuzione dei task con i thread

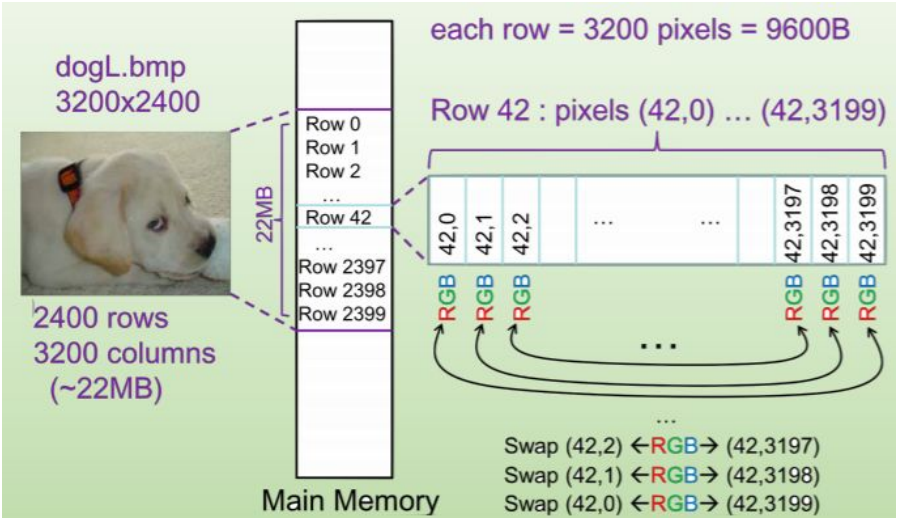
Il main() crea i thread e assegna un unico **tid** ad ogni thread a runtime (con **TheParam[1]**. Infine invoca una funzione per ogni thread. Il main() comunica con il sistema operativo per indicare quanti sono i thread creati (attributo **ThAttr**). La responsabilità di ogni thread è di eseguire la funzione assegnata. Infine in main() attende che ogni thread termini, sincronizzando con la join.

```
. . .
   ThParam[i] = i;
   ThErr = pthread_create(&ThHandle[i], &ThAttr, MTFlipFunc, (void *)&ThParam[i]);
. . .
```

Se deve essere sequenziale o parallelo, vengono definite le vriabili MTFlipFunc e FlipFunc

```
void (*FlipFunc)(pel** img);       // Function pointer to flip the image
void* (*MTFlipFunc)(void *arg);   // Function pointer to flip the image, multi-threaded version

. . .
void FlipImageV(unsigned char** img) {. . .}
void FlipImageH(unsigned char** img)  {. . .}
void *MTFlipV(void* tid)  {. . .}
void *MTFlipH(void* tid)  {. . .}

. . .

if (NumThreads != 1) {
    printf("\nExecuting the multi-threaded version with %d threads …
    MTFlipFunc = (Flip == 'V') ? MTFlipV : MTFlipH;
} else {
    printf("\nExecuting the serial version ...\n");
    FlipFunc = (Flip == 'V') ? FlipImageV : FlipImageH;
    }
}
```

5.2.3 - Flip Verticale



Esempio per un''mmagine di 640x480 (*colonne x righe* o *base x altezza*)

La versione sequenziale sarebbe

```
Row [0]:   [0][0]↔[479][0],   [0][1]↔[479][1] ...   [0][639]↔[479][639]
Row [1]:   [1][0]↔[478][0],   [1][1]↔[478][1] ...   [1][639]↔[478][639]
Row [2]:   [2][0]↔[477][0],   [2][1]↔[477][1] ...   [2][639]↔[477][639]
Row [3]:   [3][0]↔[476][0],   [3][1]↔[476][1] ...   [3][639]↔[476][639]
... ... ... ... ...
Row [239]: [239][0]↔[240][0], [239][1]↔[240][1] ... [239][639]↔[240][639]
```

Cioè fino a metà di 480, cioè la riga 239 si fa lo swap con le righe sottostanti delle varie colonne. infatti la 239 fa swap con la 240 per ogni colonna (639, parte da 0, è un array!!)

La versione parallela usando 4 thread

```
tid = 0 : Pixels [0...159]        Hbytes [0...477]
tid = 1 : Pixels [160...319]    Hbytes [480...959]
tid = 2 : Pixels [320...479]    Hbytes [960...1439]
tid = 3 : Pixels [480...639]    Hbytes [1440...1919]
```

Ogni thread si occupa di un set di colonne, le colonne sono rappresentate da Pixel, ogni pixel ha 3 byte (quindi 159*3 = 477, *Hbytes* rappresentano i byte che ci sono orizzontalmente nell'immagine). Quini ogni thread si prende 160 pixel

```
void *MTFlipV(void* tid) {
    struct Pixel pix; //temp swap pixel
    int row, col;
    long ts = *((int *) tid);       // My thread ID is stored here
    ts *= ip.Hbytes / NumThreads; // start index
    long te = ts + ip.Hbytes / NumThreads - 1; // end index
    for (col = ts; col <= te; col += 3) {
. . .
```

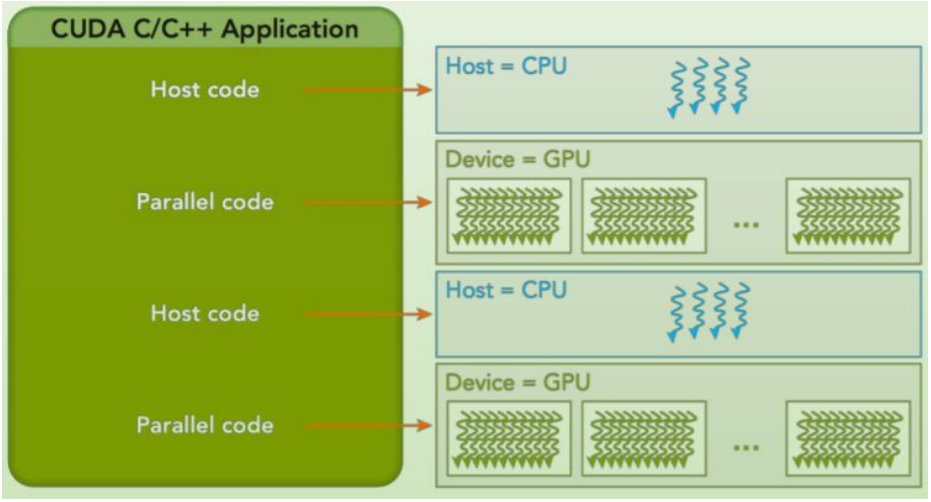
pix è la struttura dati in cui risiedono i 3 byte per l'RGB
row e *col* sono indici per navigare nell'immagine
ts indica il tid del thread
ts viene moltiplicato per ip.Hbytes /NumThreads, cioè in base al byte che si sta elaborando, si sa che thread è coinvolto.
te indica il limite delle colonne da esaminare. Per ogni colonna fa il flip sull'asse x

6 - Programmare in CUDA C pt2

6.1 - Elementi chiave

La programmazione con CUDA prevede alcuni elementi

- **Controllo della memoria, dati e thread** da parte del programmatore
- **Kernel**, è il programma sequenziale eseguito dalla GPU
- **Host**, opera indipendentemente dal **device** (cioè la GPU)
- **Host code** programma in ANSI C
- **Device code** programma in CUDA C
- Le computazioni di (kernel) GPU e CPU sono **asincrone**
- I trasferimenti tra memorie CPU e GPU invece sono **sincrone**
- Il **compilatore è il nvcc**, e genera il codice eseguibile per **host** e **device** (fat-binary)



6.1.1 - Passi per la programmazione in CUDA. Formula base

1. Setup dei dati su host (CPU-accessible memory)
2. Alloca memoria per i dati sulla GPU
3. Copia i dati da host a GPU
4. Alloca memoria per output su host (CPU)
5. Alloca memoria per output su device (GPU)
6. Lancia il kernel su GPU
7. Copia l'output da GPU a host
8. Reset delle memorie

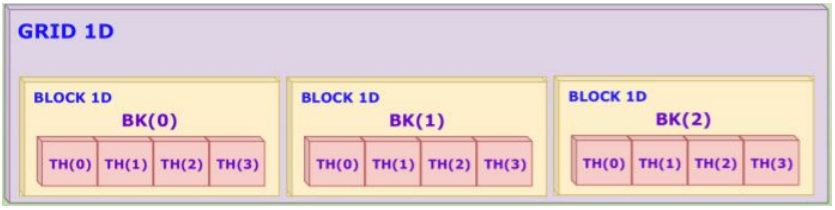
6.2 - Organizzazione e gestione dei Thread

CUDA utilizza una gerarchia astratta per i thread ed è su due livelli

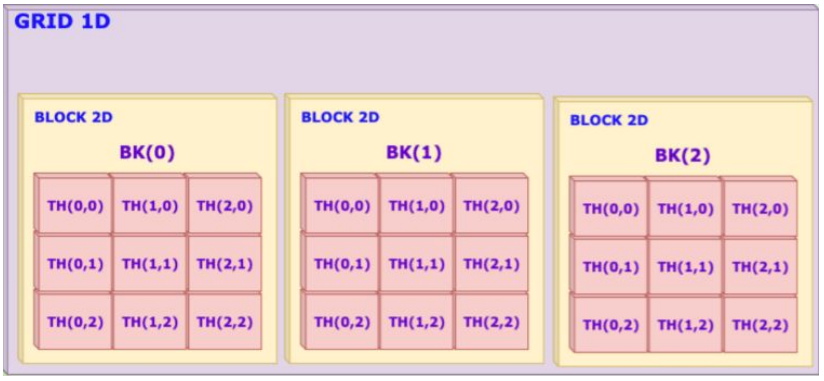
- **Grid**: una griglia ordinata di blocchi
- **Block**: una collezione ordinata di thread

Le strutture grid e block possono avere diverse dimensioni: 1D, 2D, 3D e si possono usare le 9 combinazioni tra queste strutture. La numerazione è *base x altezza*, *colonna x riga*

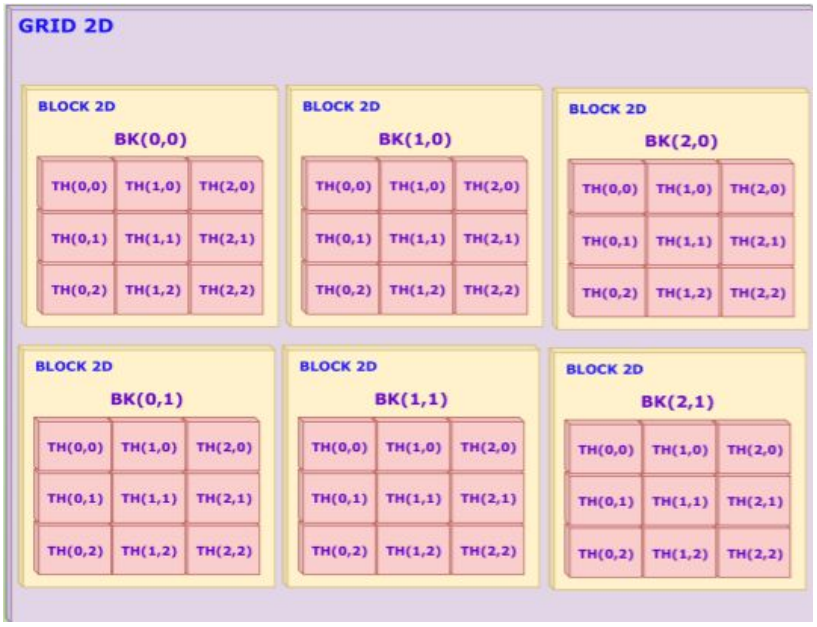
Grid 1D e Block 1D



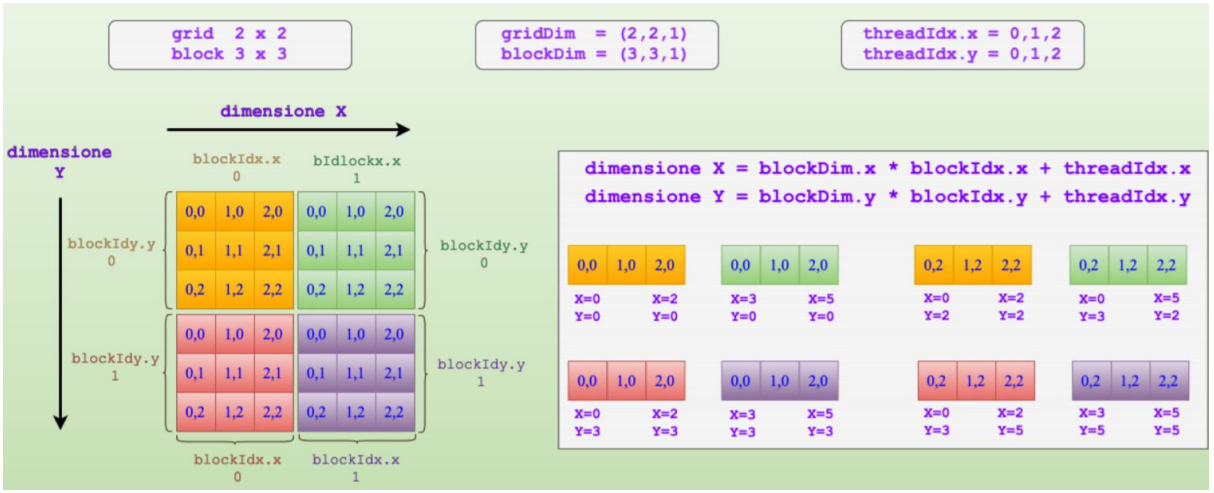
Grid 1D e Block 2D

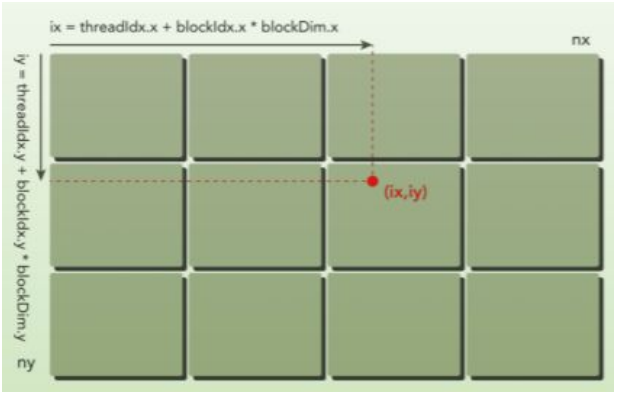


Grid 2D e Block 2D



Lavorare con le matrici





Nel caso 2D ci sono 3 tipi di indici

1. indici di thread e blocchi
2. indice dell'elemento della matrice
3. indice (offset) della memoria lineare globale

Trovo le coordinate 2D del thread

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

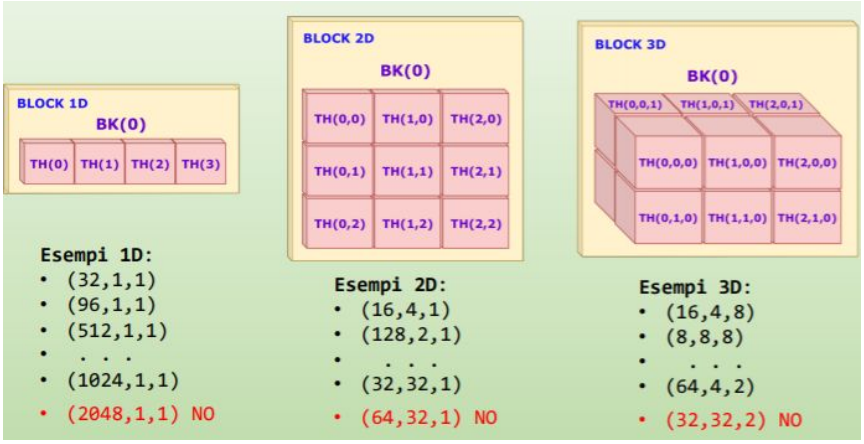
Mappo le coordinate della matrice sulla memoria lineare globale: `idx = iy * nx +ix`

Infine si controlla se la dimensione della griglia non collima con quella della matrice

```
if (ix < nx) ...
if (iy < ny) ...
```

6.2.1 - Informazioni sul blocco

Importante: UN BLOCCO può contenere al massimo 1024 thread.



Un blocco di thread è un gruppo di thread che possono cooperare tra di loro mediante:

- **Block-local synchronization**
  I thread di un unico blocco possono sincronizzarsi tra loro e non possono assolutamente sincronizzarsi con thread di altri blocchi.
- **Block-local shared memory**
  I thread di un unico blocco possono condividere la memoria tra loro e non possono accedere a quella di altri blocchi.

Tutti i thread in una **grid** condividono lo stesso spazio di **global memory**.

I thread vengono identificati mediante delle coordinate

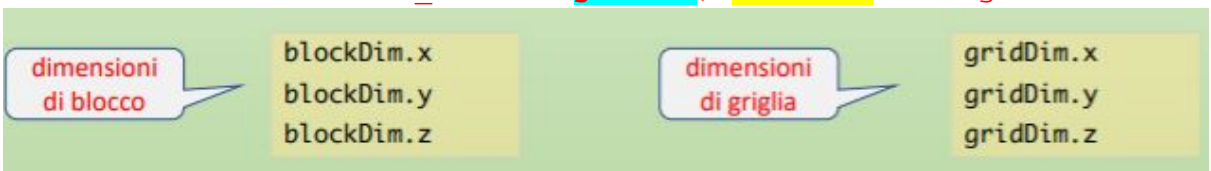
- **blockIdx** (indice di blocco nella grid)
- **threadIdx** (indice di thread nel blocco)

Ci si accede agli indici tramite le variabili **built-in**
Le coordinate sono di tipo **uint3** (**x**, **y**, **z**), **builtin**, pre-inizializzate e possono essere accedute all'interno del kernel. Quando il kernel è eseguito, **blockIdx** e **threadIdx** vengono assegnate a runtime da CUDA.

Le dimensioni di grid e block sono definite alla chiamata del kernel e i valori sono rispettivamente

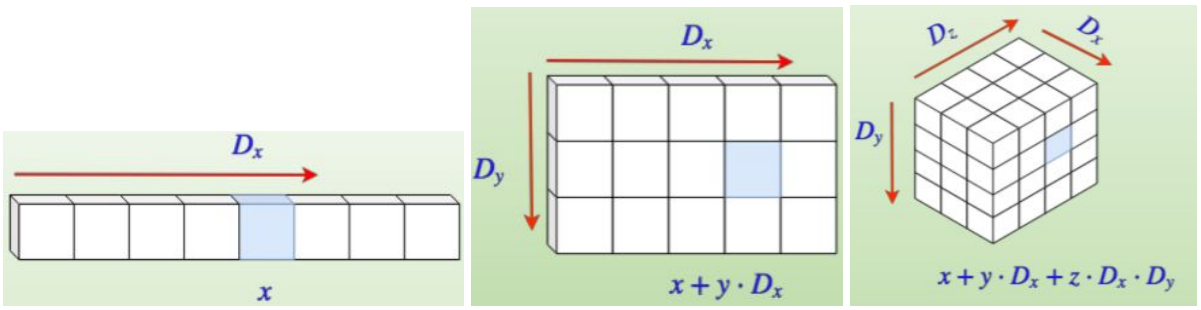
- **blockDim** (dimensione del blocco, il numero di thread per blocco)
- **gridDim** (dimensione della griglia, il numero di blocchi per griglia, la griglia è una)

infatti si chiama così → `kernel_name <<<gridDim, blockDim>>>(argumentList)`



6.2.2 - Trovare l'indice

1D **2D** **3D**



Grid 1D Block 1D:
IDthread = blockId.x * blockDim + threadIdx.x → *0 * Dx + x*

Grid 1D Block 2D
IDthread = blockIdx * blockDim.x * blockDim.y + threadIdx.y * blockDim.x + threadIdx.x
(<0, GridDim-1> * BlockDim (x*y) + Dy * Dx (scorro tra le righe) + x (il thread))
(scorro i blocchi - scorro all'interno della matrice)

Grid 1D Block 3D
IDthread = blockIdx.x * BlockDim.x * BlockDim.y * BlockDim.z +
threadIdx.z * blockDim.y * blockDim.x + threadIdx.y * blockDim.x + threadIdx.x
(scorro la dimensione 3D (trovo il piano)
+ scorro la dimensione 2D (trovo il blocco)
+ scorro la dimensione 1D (trovo l'unità))

Grid 2D Block 1D
IDblocco = blockIdx.y * gridDim.x + blockIdx.x
IDthread = blockIdx.x * blockDim.x + threadIdx.x

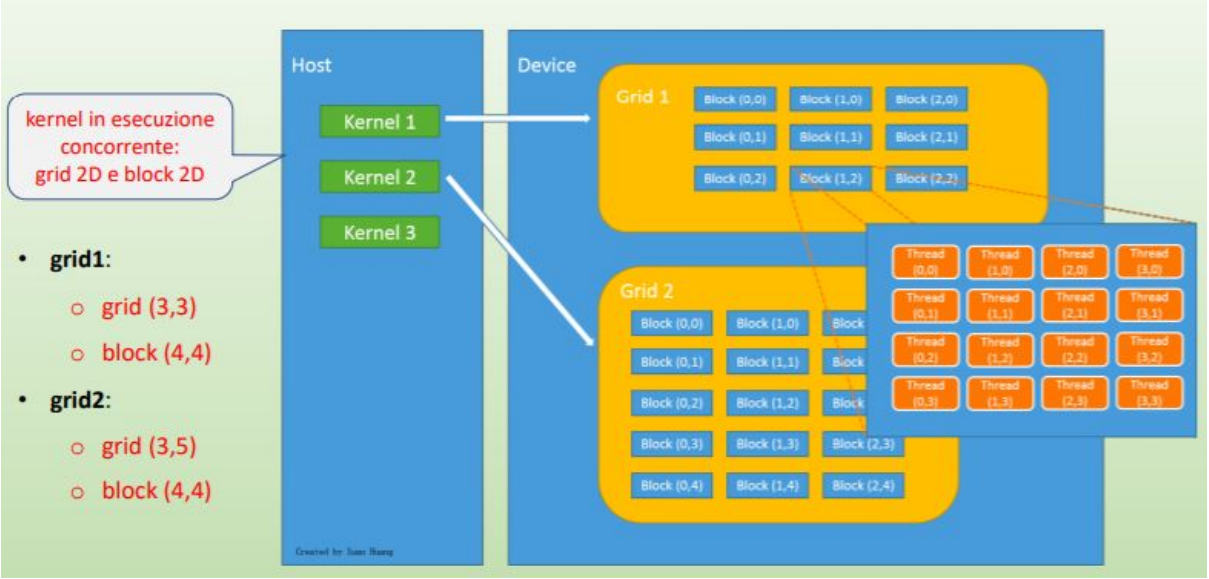
Grid 2D Block 2D
IDthread = IDblocco * (blockDim.x * blockDim.y) + threadIdx.y * blockDim.x + threadIdx.x

Grid 2D Block 3D
IDthread = IDblocco * (blockDim.x * blockDim.y * blockDim*z) +
threadIdx.x * (blockDim.x * blockDim.y) +
threadIdx.y * blockDim.x + threadIdx.x

Grid 3D Block 1D
IDblocco = blockIdx.x + blockIdx.y * gridDim.x + gridDim.x * gridDim.y * blockIdx.z
IDthread = IDblocco * blockDim.x + htreadIdx.x

Grid 3D Block 2D
IDthread = IDblocco * (bloccoDim.x * blockDim.y) + threadIdx.y * blockDim.x + threadIdx.x

Grid 3D Block 3D
IDthread = IDblocco * (blockDim.x * blockDim.y * blockDim.z) +
threadIdx.z * (blockDim.x * blockDim.y) +
threadIdx.y * blockDim.x + threadIdx.x




```c
#include <stdio.h>

/*
 * Mostra DIMs e IDs di grid, block e thread
 */
__global__ void checkIndex(void) {
    printf("threadIdx:(%d, %d, %d) blockIdx:(%d, %d, %d) "
        "blockDim:(%d, %d, %d) gridDim:(%d, %d, %d)\n",
        threadIdx.x, threadIdx.y, threadIdx.z,
        blockIdx.x, blockIdx.y, blockIdx.z,
        blockDim.x, blockDim.y, blockDim.z,
        gridDim.x,gridDim.y,gridDim.z);
}

int main(int argc, char **argv) {
    // definisce grid e struttura dei blocchi
    dim3 block(4);
    dim3 grid(3);
    // controlla dim. dal lato host
    printf("grid.x %d grid.y %d grid.z %d\n", grid.x, grid.y, grid.z);
    printf("block.x %d block.y %d block.z %d\n", block.x, block.y, block.z);
    // controlla dim. dal lato device
    checkIndex<<<grid, block>>>();
    // reset device
    cudaDeviceReset();
    return(0);
}
```

1. **__global__** è la funzione del kernel che verrà eseguita dalla GPU
2. Come dimensionare griglia e blocchi
   `dim3 block (4)`
   `dim3 grid (3)`
   Si crea una struttura 1D 1D, composta da:
   una fila di 3 blocchi e per ogni blocco una riga di 4 thread
3. La CPU una volta lanciata l'istruzione, cioè `checkIndex <<<grid, block>>>()`, per la GPU, prosegue con l'esecuzione
4. Con `cudaDeviceReset()` resetta la GPU
   `cudaDeviceReset()` pulisce tutte le risorse associate al device, cioè inizializza il device (solo l'area coinvolta nel processamento)

6.2.3 - Debugging del codice con le printf

Invece di utilizzare i tool di debugging avanzati, è consigliato utilizzare mezzi molto più semplici per verificarne la correttezza.

- **printf**: disponibile nei device da Fermi in poi
- **kernel <<<1,1>>>**: si forza il kernel ad eseguire un blocco con un solo thread, emula il comportamento dell'esecuzione sequenziale su un singolo dato.

Per avere una print corretta su standard output è utile usare **cudaDeviceSynchronize** o **cudaDeviceReset**

`cudaDeviceSynchronize`, fa aspettare all'host che il **device** finisca le elaborazioni. Quindi è bloccante e fa attendere all'**host** il completamento dei task sul device.

Esercizio: filtrare gli indici di un kernel CUDA di cui coordinate sulla componente x e y se sommate risultino un multiplo di 5.
Il filtro viene applicato alla funzione eseguita dalla GPU, non si può far partire specifici thread perchè la GPU esegue l'intero warp (cioè il minimo set, composto da 32 thread)

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>

__global__ void checkIndex(void) {

    if ((threadIdx.x + threadIdx.y) % 5 == 0)
        printf("threadIdx:(%d, %d, %d) blockIdx:(%d, %d, %d) "
            "blockDim:(%d, %d, %d) gridDim:(%d, %d, %d)\n",
            threadIdx.x, threadIdx.y, threadIdx.z,
            blockIdx.x, blockIdx.y, blockIdx.z,
            blockDim.x, blockDim.y, blockDim.z,
            gridDim.x, gridDim.y, gridDim.z);
}

int main(int argc, char **argv) {
    dim3 grid(2,2,1);
    dim3 block(8,7,1);

    printf("grid.x %d grid.y %d grid.z %d\n", grid.x, grid.y, grid.z);
    printf("block.x %d block.y %d block.z %d\n", block.x, block.y, block.z);

    checkIndex<<<grid, block>>>();

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

Proprietà del Kernel

QUALIFICATORI	ESECUZIONE	CHIAMATA	NOTE
`__global__`	Eseguito dal device	Dall'host e dalla compute cap. 3 anche dal device	Restituisce un tipo void
`__device__`	Eseguito dal device	Solo dal device	
`__host__`	Eseguito dall'host	Solo dall'host	Può essere omesso

`__device__` e `__host__` possono essere usati insieme, le funzioni vengono compilate sia per host che per device

Ci sono delle **restrizioni sul kernel**

1. Accede alla sola memoria del device
2. Deve restituire un tipo void (infatti i valori vengono salvati sulla memoria ram)
3. Non supporta le variabili statiche
4. Non supporta puntatori a funzioni
5. Esibisce un comportamento asincrono rispetto all'host

6.3 - CUDA Runtime API vs CUDA Driver API

Complessità vs Controllo

- Le **runtime API** alleggeriscono la gestione del codice device provvedendo:
  - inizializzazioni implicite, context management, gestione dei moduli
  - semplifica il codice, facendo scarseggiare il livello di controllo delle API
- Le **driver API** offrono una grana più fine del controllo, soprattutto sui context e il caricamento dei moduli
  - **kernel launches** sono molto più complessi da implementare, come anche l'esecuzione e la configurazione dei parametri del kernel, tutto è da specificare esplicitamente con chiamate a funzione (explicit function calls)
  - i Driver API sono sono language-independent perchè lavorano sul binario di cuda (cubin objects).

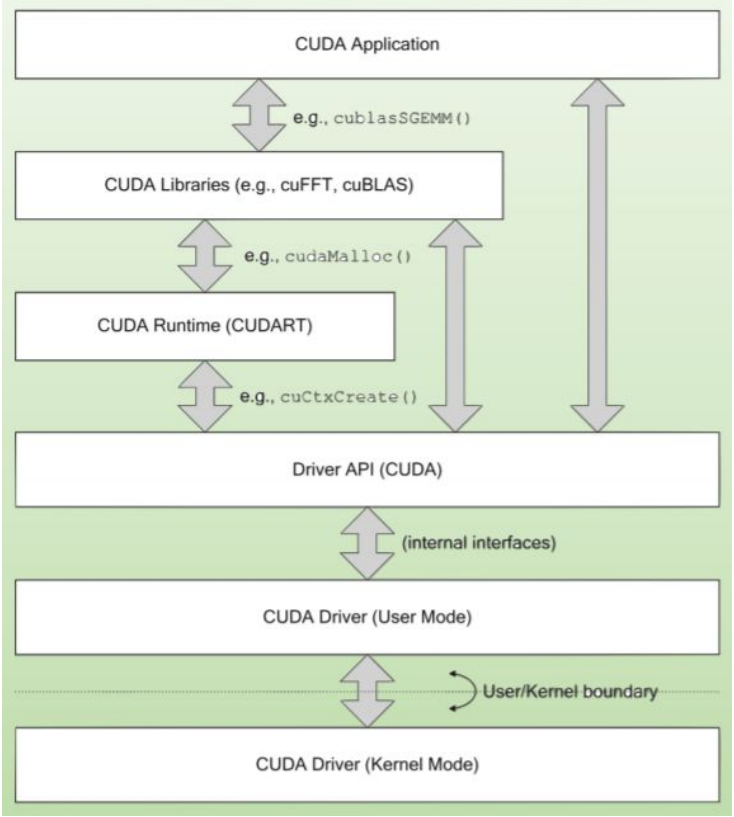
Context management

Il context management si può fare attraverso i **driver API** , ma non è esposto nelle **runtime API**. Infatti le **runtime API** scelgono da sole quale **context** usare per un **thread**. Se il contesto viene fatto mediante la chiamata del thread attraverso i **driver API**, si utilizza quello, ma se non esiste, si utilizza il ***primary context***.

Il **primary context** è creato quando necessario, uno per device e per processo, infine distrutto quando non ci sono più reference per il context.
Con `cudaDeviceSynchronize()` si sincronizza il contesto, mentre con `cudaDeviceReset()` si elimina.

6.4 - Livelli di SW CUDA

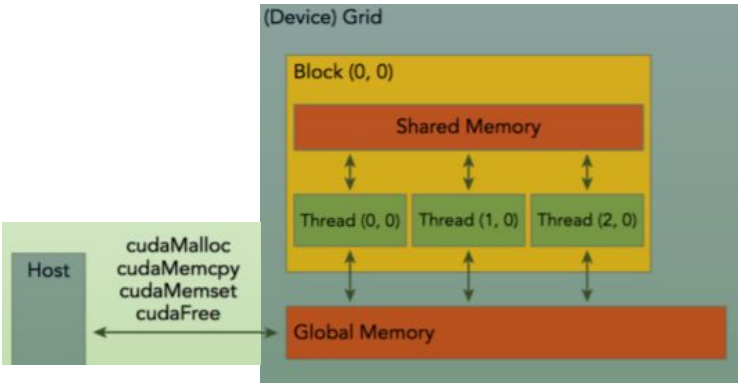
Tutti i livelli operano con i privilegi utente, eccetto il livello del driver che opera in kernel mode. In CUDA host e device memory allocati a un programma CUDA, non possono essere acceduti da un ‘altro programma CUDA (isolation).



6.5 - Gestione della memoria

Funzioni C standard	Funzioni CUDA C
malloc	cudaMalloc
memcpy	cudaMemcpy
memset	cudaMemset
free	cudaFree

Quando si usa **cudaMemcpy** per copiare i dati da host e device, **implicitamente viene fatta della sincronizzazione** dalla parte dell’host, quindi questo deve attendere che la copia dei dati sia completa



6.5.1 - cudaMalloc e cudaMemcpy

Segnatura:
`cudaError_t cudaMalloc (void** devPtr, size_t size)`
devPtr è un puntatore a un puntatore in global memory device

Segnatura:
`cudaError_t cudaMemcpy (void* dst, const void* src, size_t count, cudaMemcpyKind kind)`

KIND: i cudaMemcpyKind si intende che *kind* deve essere uno di questi:
`cudaMemcpyHostToHost` `cudaMemcpyHostToDevice`
`cudaMemcpyDeviceToHost` `cudaMemcpyDeviceToDevice`
Questa funzione esibisce un comportamento sincrono che blocca il programma host fino a che il trasferimento non viene completato. Per capire se **destination** e **src** sono puntatori a memoria CPU o GPU si guarda la variabile **kind**

Errore:
`cudaError_t` return → `success` oppure `cudaErrorMemoryAllocation`
Ogni chiamata CUDA (eccetto il kernel) restituisce un tipo enumerativo `cudaError_t`

Errore:
`char* cudaGetErrorString(cudaError_t error)`
Conversione del messaggio di errore in forma leggibile

6.5.2 - Gestione errori

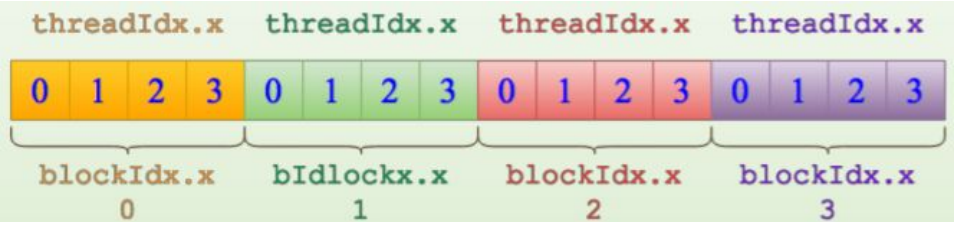
Dato che le **chiamate CUDA** sono **asincrone**, è difficile sapere quale routine o function genera un errore, quindi si definiscono delle macro per la gestione degli errori, per fare un wrap delle chiamate CUDA.

```
#define CHECK(call) {
    const cudaError_t error = call;
    if (error != cudaSuccess) {
        printf("Error: %s:%d, ", __FILE__, __LINE__);
        printf("code:%d, reason: %s\n", error, cudaGetErrorString(error));
        exit(1);
    }
}
```

Poi usare
`CHECK(cudaMemcpy(d_C, gpuRef, nBytes, cudaMemcpyHostToDevice)) ;`

per motivi di debugging di kernel, si può usare il codice
```
kernel_function<<<grid, block>>> (argumentList);
CHECK (cudaDeviceSynchronize);
```

6.5.2 - Esempio con Grid 1D e Block 1D (coordinate 1D)



il kernel:

```
__global__ void add_vect(int *a, int *b, int *c) {
      int idx = blockDim.x * blockIdx.x + thread.x
      if (idx < N)
            c[idx] = a[idx] + b[idx]
}
```

Allocazione su host e device:
Allocazione su CPU

```
float *h_A, *h_B, *h_C;
h_A = (float *) malloc(nBytes);
h_B = (float *) malloc(nBytes);
h_C = (float *) malloc(nBytes);
```

Allocazione su GPU

```
float *d_A, *d_B, *d_C;
cudaMalloc((void**) &d_A, nBytes);
cudaMalloc((void**) &d_B, nBytes);
cudaMalloc((void**) &d_C, nBytes);
```

si usano identificatori specifici per i dispositivi.
Un errore comune è quello di confondere i puntatori di due diversi spazi di memoria CPU e GPU. Un puntatore alla memoria device NON PUò essere deferenzato nel codice host e viceversa.

Un assegnamento sbagliato è: `gpuRef = d_C`
Quello giusto è: `cudaMemcpy(gpuRef, d_C, nBytes, cudaMemcpyDeviceToHost)`

Nella **Unified Memory** questi errori non sono importanti perchè consente da CUDA 6 di accedere alle memorie CPU e GPU da un singolo puntatore

Schema generale

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

_global__ void add_vect(…)
 . . .


1. allocare la memoria device per
   h_a, h_b e h_c

2. Lanciare il kernel con argomenti
    h_a, h_b e h_c

3. Copiare dalla memoria device alla
   memoria host c il risultato in h_c

3. Liberare la memoria hos e device con
   cudaFree(*ptr);
```

Codice facendo la somma dei vettori

```c
#include <stdio.h>
#include <cuda_runtime.h>
#define N 1024*1024  // vector size
#define TxB 32       // threads x block

/* kernel: vector add */
_global__ void add_vect(int *a, int *b, int *c) {
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    if (idx < N)
        c[idx] = a[idx] + b[idx];
}

int main(void) {
    int *a, *b, *c;
    int *dev_a, *dev_b, *dev_c;
    int nBytes = N * sizeof(int);

    // malloc host memory
    a = (int *) malloc(nBytes);
    b = (int *) malloc(nBytes);
    c = (int *) malloc(nBytes);

    // malloc device memory
    cudaMalloc((void**) &dev_a, nBytes);
    cudaMalloc((void**) &dev_b, nBytes);
    cudaMalloc((void**) &dev_c, nBytes);

    // fill the arrays 'a' and 'b' on the CPU
    for (int i = 0; i < N; i++) {
        a[i] = rand() % 10;
        b[i] = rand() % 10;
    }
```

```c
    // copy the arrays 'a' and 'b' to the GPU
    cudaMemcpy(dev_a, a, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(dev_b, b, nBytes, cudaMemcpyHostToDevice);

    add_vect<<<N / TxB, TxB>>>(dev_a, dev_b, dev_c);

    // copy the array 'c' back from the GPU to the CPU
    cudaMemcpy(c, dev_c, nBytes, cudaMemcpyDeviceToHost);

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

    // Free host memory
    free(a);
    free(b);
    free(c);

    // free the memory allocated on the GPU
    cudaFree(dev_a);
    cudaFree(dev_b);
    cudaFree(dev_c);

    return 0;
}
```

Misurare il tempo con la CPU

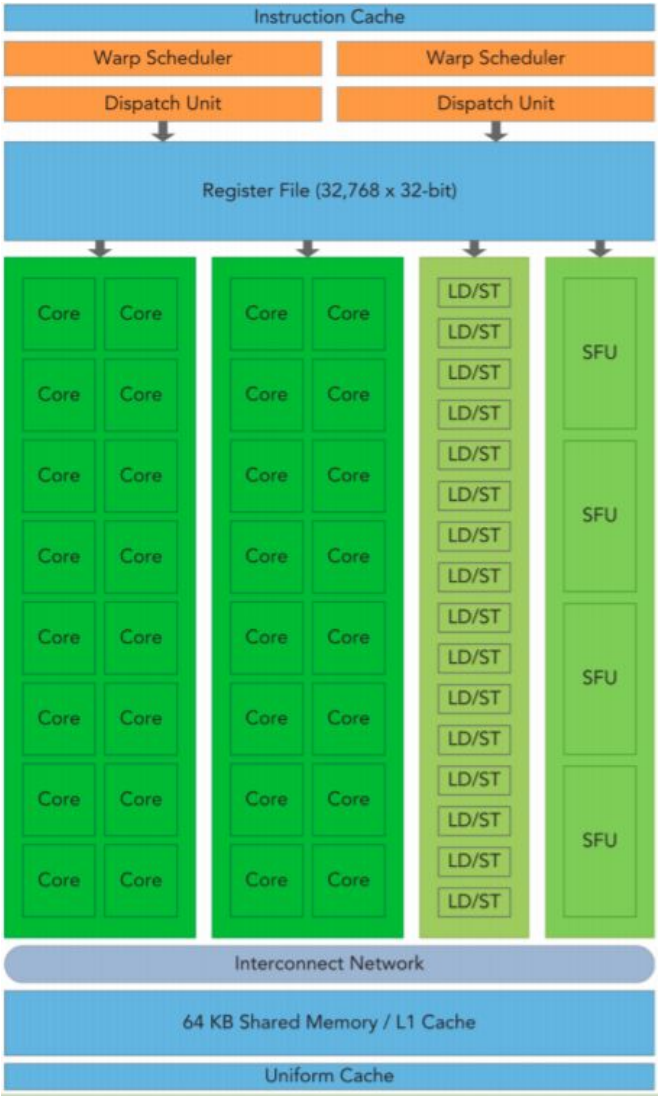
```c
double cpuSecond() {
    struct timeval tp;
    gettimeofday(&tp,NULL);
    return ((double)tp.tv_sec + (double)tp.tv_usec*1.e-6);
}

// misura del tempo di esecuzione del kernel
double iStart = cpuSecond();
kernel_name<<<grid, block>>>(argument list);
cudaDeviceSynchronize();
double iElaps = cpuSecond() - iStart;
```

Utilizzando **nvprof**, un tool di profiling, permette di vedere il tempo delle attività su cpu e gpu

```
$ nvprof [nvprof_args] <application> [application_args]
```

```
$ nvprof add_vect
 ==8132== Profiling application: a.out
==8132== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 37.38%  7.3920us         1  7.3920us  7.3920us  7.3920us  add_vect(int*, int*, int*)
 32.36%  6.4000us         2  3.2000us  2.8480us  3.5520us  [CUDA memcpy HtoD]
 30.26%  5.9840us         1  5.9840us  5.9840us  5.9840us  [CUDA memcpy DtoH]

==8132== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 99.73%  174.54ms         3  58.179ms  3.7540us  174.53ms  cudaMalloc
  0.10%  172.88us         3  57.625us  6.9580us  149.96us  cudaFree
  0.07%  115.60us        83  1.3920us     100ns  53.943us  cuDeviceGetAttribute
  0.06%  103.47us         3  34.490us  18.391us  49.043us  cudaMemcpy
```

7 - Architettura di una GPU



Streaming Multiprocessors (SM)
L'architettura della GPU è disegnata come un array scalabile di **SM** (streaming multiprocessors). Il **parallelismo** hardware della GPU è ottenuto replicando l'elemento base. Ci sono 32 core, 32 come il numero dei thread gestiti da un blocco.

I componenti chiave della SM Fermi:
- Cuda cores
- Shared Memory/L1 cache
- Register file
- Load/Store Units
- Special Function Units
- Warp Scheduler

Hardware multi-threading
Architettura **SIMT** - warp level (si utilizza una sola istruzione su molti thread, quindi sono uniformati)
Ogni warp ha un **contesto in esecuzione** a runtime, mantenuto **on-chip** (ne velocizza lo switch tra contesti), che consiste in:
- *Program counters*
- *Registri a 32 bit ripartiti tra i thread*
Sono usati per le variabili locali automatiche scalari e coordinate dei thread. I dati nei registri sono privati ai thread (scope).
- *Shared memory ripartita tra blocchi*
La memoria condivisa permette di condividere la memoria tra i thread dello stesso blocco.

- Ogni thread viene eseguito su CUDA core e ha un suo spazio privato di **memoria** per registri, **chiamate** di funzioni e **variabili** C automatiche.
- Un **thread block** è un gruppo di thread eseguiti **concorrentemente** e possono essere cooperare sulla stessa memoria e venendo sincronizzati insieme.
- Una **GRID** è 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

Esecuzione
Viene mappata la **gerarchia di thread** nella **gerarchia di processori** sulla GPU
La GPU esegue uno o più kernel
Uno streaming multiprocessor (SM) esegue uno o più thread block
Un thread block è schedulato solo su un SM
I core del SM eseguono i thread
Gli SM eseguono i thread a gruppi di **32 thread** chiamati **warp**

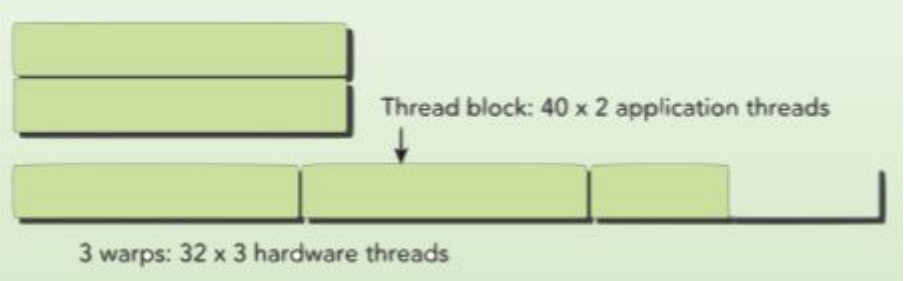
7.1 - WARP architettura SIMT

In un **warp** ci sono 32 thread (per architettura). Idealmente tutti i thread di un warp vengono eseguiti in parallelo allo stesso tempo (modello SIMD: Single Instruction, Multiple Data)
Ogni thread può seguire **cammini distinti** di esecuzione delle istruzioni (parallelismo a livello thread). Il **warp** è l'unità minima di esecuzione, ha un forte impatto sulle prestazioni degli algoritmi sviluppato. Concettualmente ha un comportamento **SIMD**, ma assume il **SIMT**

Architettura SIMT è il modello Single Instruction, Multiple Thread e la differenza è che il **SIMD** espone l'intero set di dati all'istruzione che deve essere eseguita, mentre il **SIMT** specifica il comportamento di esecuzione e branching di un singolo thread.

Esempio: un blocco da 128 thread viene suddiviso in 4 warp.

Esempio: se usassi 2 blocchi da 40 thread, in totale sarebbero 80 thread (40*2) e i warp coinvolti sono 3, cioè 96 thread, di cui 80 usati e 16 no.



$$warpPerBlock = \frac{ThreadPerBlock}{warpSize} = \frac{40}{32} = 2$$

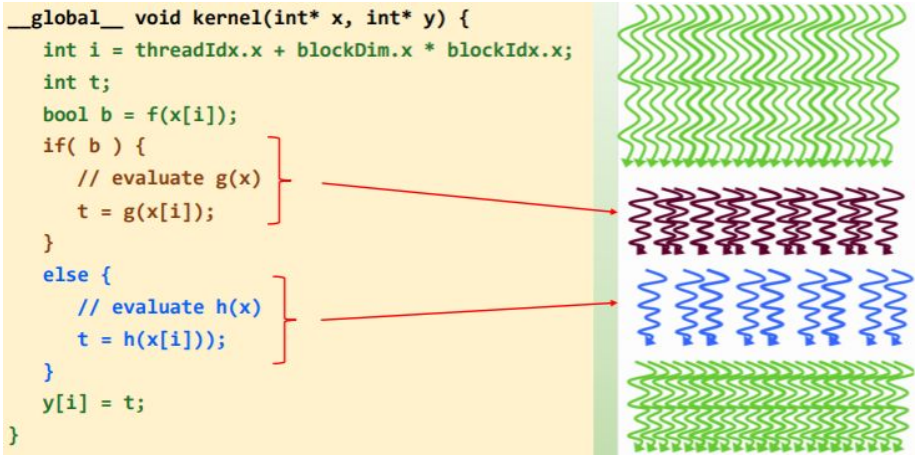
Un blocco è attivo dopo che le risorse gli sono state assegnate, i warp sono anch'essi attivi.
Un **warp attivo** può essere di tre tipi:

- *Selezionato*: in esecuzione su un dato path
- *Bloccato*: non pronto per l'esecuzione
- *Candidato*: eleggibile se 32 core sono liberi e tutti gli argomenti della prossima istruzione disponibili

Un **thread** può essere nello stato di:

- *Attivo*: quando appartiene al path in esecuzione del warp corrente
- *Inattivo*: quando non nel path in esecuzione o uscito prima degli altri dal warp o ultimo di blocchi non multipli di 32.

7.2 - Divergenza ed esecuzione



Le GPU non possiedono sistemi complessi come quelli della CPU per fare il branch prediction, quindi avere dei thread con dei kernel che cambiano il flow di esecuzione, li desincronizza dato che il warp **esegue serialmente** ogni branch path, disabilitando i thread che non appartengono a quel dato path. La divergenza introduce la degradazione delle prestazioni. Il problema è isolato all'interno del warp, quindi i warp possono avere path di esecuzione differenti tra loro. Il caso peggiore è quando ogni thread ha un path differente.

Si può **misurare l'efficienza** con il comando

```
nvprof --metrics branch_efficency ./kernel
```

Invocations	Metric Name	Metric Description	Min	Max	Avg
Device "Tesla M2090 (0)"					
Kernel: evenODD2(int*, int)					
1	branch_efficiency	Branch Efficiency	99.83%	99.83%	99.83%
Kernel: evenODD1(int*)					
1	branch_efficiency	Branch Efficiency	100.00%	100.00%	100.00%

In questo caso non si nota particolarmente la differenza perchè il compilatore ci mette lo zampino e ottimizza il sorgente.

```
nvcc -g -G kernel.cu -o kernel
nvprof --metrics branch_efficency ./kernel
```

Invocations	Metric Name	Metric Description	Min	Max	Avg
Device "Tesla M2090 (0)"					
Kernel: evenODD2(int*, int)					
1	branch_efficiency	Branch Efficiency	99.83%	99.83%	99.83%
Kernel: evenODD1(int*)					
1	branch_efficiency	Branch Efficiency	83.33%	83.33%	83.33%

Si può impedire al compilatore (**nvcc**) di non ottimizzare, infatti riprovando la metrica, si nota la divergenza del tempo di esecuzione

Branch Efficiency = $100 \cdot \frac{\#Branches - \#DivergentBranches}{\#Branches}$

Il compilatore interviene sostituendo i path condizionali.

```
nvprof --events branch,divergent_branch ./kernel
```

Invocations	Event Name	Min	Max	Avg	Total
Device "GeForce GT 650M (0)"					
Kernel: evenODD1(int*, int)					
1	branch	229376	229376	229376	229376
1	divergent_branch	32768	32768	32768	32768
Kernel: evenODD2(int*, int)					
1	branch	327680	327680	327680	327680
1	divergent_branch	0	0	0	0

Risoluzione della divergenza:
Si utilizza la granularità dei warp, non quella dei thread, perchè si svolge a livello di blocco.

```
/*
 * Kernel con divergenza dei warp risolta (solo se N > 64)
 */
__global__ void pari_dispari_2(int *c, int N) {
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
  int a = 0, b = 0, i;

  int wid = tid / warpSize; // indice dei warp wid = 0,1,2,3,...
  if (!(wid % 2))
     a = 2;           // Branch1: thread tid = 0-31, 64-95,...
  else
     b = 1;           // Branch2: thread tid = 32-63, 96-127,..

     // right index
     if (!(wid%2))  // even
        i = 2*(tid%32)  + (tid/64)*64;
     else          // odd
         i = 2*(tid%32)+1 + (tid/64)*64;
    if (i<N)
       c[i] = a + b;
}
```

In questo codice faccio la somma tra pari e dispari e la metto in un array c.

Calcolo il **tid**, cioè l'id del thread

Calcolo il **wid**, corrisponde all'id identificativo del warp, dato che abbiamo 64 thread, saranno 2 i warp in questo caso. Il risultato è tra **0** e **1** (es tid = 34, warpsize = 32 → 34/32 = 1, sono nel secondo warp)

Suddivido l'assegnamento del valore (a o b) tr i due warp, il warp pari (0) farà assegnare a tutti i suoi 32 thread "a = 2", l'altro warp "b = 0"

Ora si calcola l'indice per posizionare correttamente la somma di a + b nell'array c
l'indice lo calcolo in due pezzi: **2 * (tid%32)**, mi restituisce un valore tra 0 e 31, lo moltiplico *2 per scorrere tutte le posizioni pari dell'array c (nel l'else fa lo stesso calcolo +1 per le posizioni dispari), infine si somma **(tid/64)*64** per gestire i thread da 64 a 95, infatti in caso di tid = 65, 2*(65%32) = 2 * 1 = 2 → si andrebbe a sovrascrivere erroneamente sull'indice 2, invece bisogna spaziare l'indice di 64 e ottenere 2 + 64 per aver l'indice corretto

7.2.1 - Predicati

CUDA introduce i predicati, istruzioni che vengono eseguite se e solo se il flag è a true:
`p: a = b + c; //viene eseguito solo se p è vero, una istruzione.`

```
if (z < 0.0)                →    p = (x < 0.0);
      z = x - 2.0;          →    p: z = x - 2.0;
else
      z = sqrt(x);          →    !p: sqrt(x);
```

Warp Voting
Il compilatore adotta questa tecnica per verificare se tutti i warp seguono la stessa strada, in base all'esito della tecnica il compilatore agirà di conseguenza, usando anche i predicati.

7.2.3 - Occupancy

Misurare l'occupancy Warp attivi
L'occupancy è il tasso tra warp attivi e numero massimo di warp per SM

$$occupancy = \frac{warp\ attivi}{max\ numero\ warp}$$

`nvprof --metrics achieved_occupancy ./kernel`

```
Invocations                               Metric Name  Metric Description         Min         Max         Avg
Device "GeForce GT 650M (0)"
    Kernel: matrix_prod(float*, float*, float*)
          1                    achieved_occupancy  Achieved Occupancy    0.879916    0.879916    0.879916
```

7.2.3 - Latenza nell'esecuzione

Latency hiding
Rappresenta il grado di parallelismo a livello di thread, serve per massimizzare l'utilizzo delle unit funzionali di un SM, dipende dal numero di **warp** residenti e attivi nel tempo.

La Latenza è definita come **numero di cicli necessari** al completamento di un'istruzione. Per massimizzare il **throughput** occorre che lo scheduler abbia sempre war eleggibili ad ogni ciclo di clock. Si ha **latency hiding** intercambiando la computazione tra warp.
Le **istruzioni aritmetiche** non introducono molta latenza (10-20 cicli di clock), mentre le **istruzioni di memoria** cioè di lettura e scrittura sulla global memory richiedono fino a 400 o 800 cicli di clock, quindi pesano sulla latenza complessiva.

7.2.4 - Sincronizzazione

La sincronizzazione può essere eseguita a due livelli

- **System-level**: si attende che il task venga completato sia su **host** che su **device**
- **Block-level**: si attende che tutti i thread di un blocco raggiungano lo stesso punto di esecuzione sul **device**.

Segnatura: `cudaError_t cudaDeviceSynchronize(void)`
Blocca l'applicazione finchè tutte le operazioni CUDA siano completate

Segnatura: `__device__ void __syncthreads(void)`
Sincronizza tutti i thread all'interno di **un blocco**, non si possono sincronizzare i thread di blocchi differenti. Può provocare deadlock.

```
//some device functions F1 & F2

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

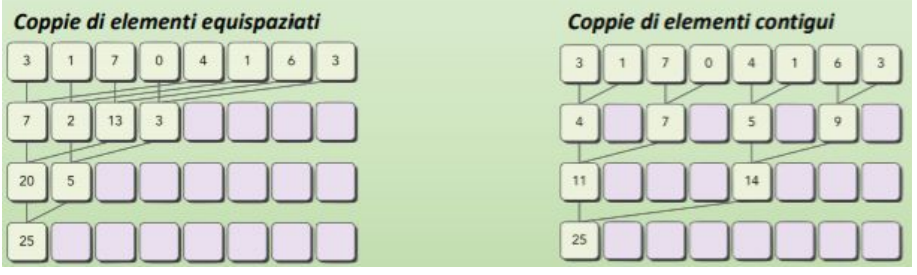
Questa situazione genera un deadlock, perchè la prima metà dei thread nel warp esegue il ramo F1, mentre la seconda metà F2, essendo in flow differenti, non riusciranno mai a raggiungere lo stesso punto.

7.2.5 - Parallel Reduction

Approccio Sequenziale per la reduce →

```
int sum = 0;
for (int i = 0; i < N; i++)
        sum += array[i];
```

Approccio parallelo
- Si suddivide il vettore in parti più piccole
- SI attivano i thread per la somma parziale sui pezzi
- Si somma le somme parziali

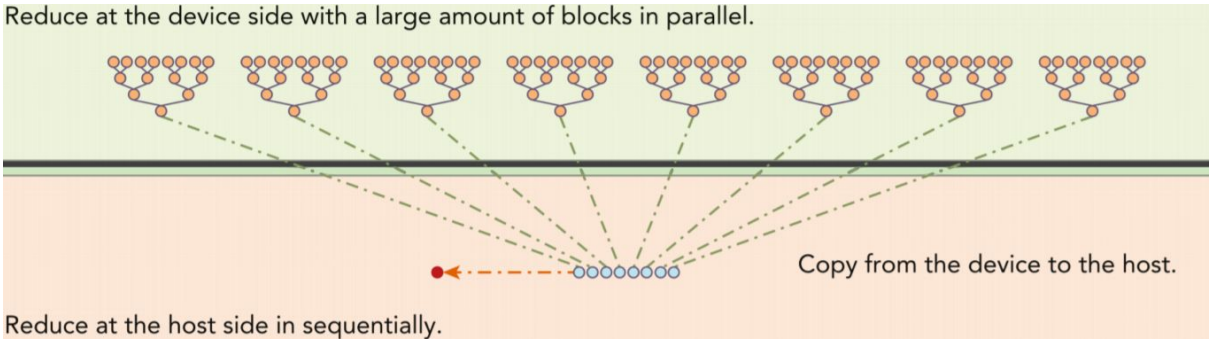


```
int recursiveReduce(int *data, int const size) {

        // 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);
}
```

Strategia parallela e ricorsiva:
Ad ogni passo, un numero di thread pari alla metà degli elementi dell'array effettua le somme. Il numero di thread attivi deve dimezzarsi ad ogni passo (rinnovare lo stride). Occorre sincronizzare i thread in modo che al passo t abbiano tutti completato il loro compito.

Il dimezzamento viene effettuato all'assegnamento della variabile stride.

Somma parziale dicotomica:
- Soluzione locale: somma parziale sincrona sui blocchi della grid
- Vincolo: la dimensione del blocco deve essere una potenza di 2
- Svantaggio: è una soluzione che introduce divergenza



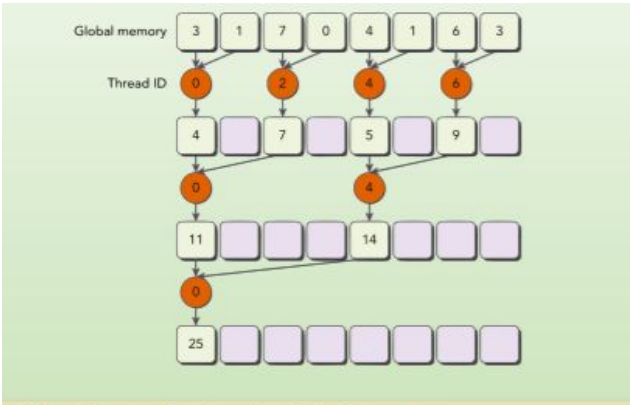
Ad ogni passo un numero di thread pari alla metà degli elementi ancora da sommare effettua le somme parziali nella prima metà di elementi (**riduzione in-place**)

Occorre sincronizzare il comportamento dei thread affinché tutti i thread al passo `t` abbiano terminato il compito prima di andare al passo successivo `t+1`

```
/* Block by block parallel implementation with divergence */
__global__ void block_parallel_reduce(int *array_in, int *array_out, unsigned int n) {
          unsigned int tid = threadIdx.x;
          unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;
          // boundary check
          if (idx >= n)
          return;
          // convert global data pointer to the local pointer of this block
          int *array_off = array_in + blockIdx.x * blockDim.x;
          // in-place reduction in global memory
          for (int stride = 1; stride < blockDim.x; stride *= 2) {
                    if ((tid % (2 * stride)) == 0)
                              array_off[tid] += array_off[tid + stride];
                    // synchronize within threadblock
                    __syncthreads();
          }
          // write result for this block to global mem
          if (tid == 0)
          array_out[blockIdx.x] = array_off[0];
}
```

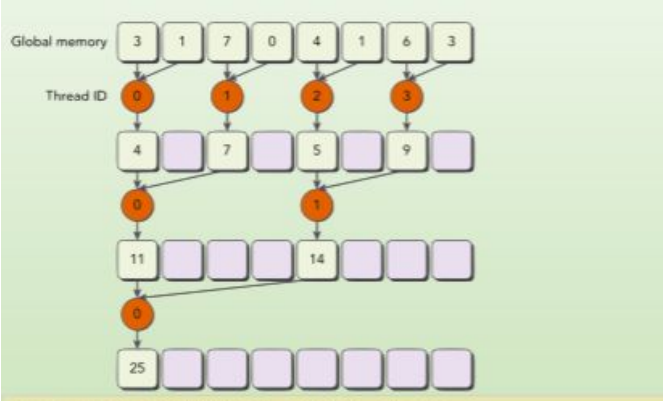
Divergenza in parallel reduction

- Inefficiente per la troppa divergenza




```
// in-place reduction in global memory
for (int stride = 1; stride < blockDim.x; stride *= 2) {
   if ((tid % (2 * stride)) == 0)
      idata[tid] += idata[tid + stride];
}
```

- Schema efficiente perchè riduce quasi a zero la divergenza




```
// in-place reduction in global memory
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];
}
```

37

7.3 - Loop Unrolling

è più costoso per la GPU fare cicli che eseguire più calcoli matematici quindi un loop come

Non Ottimizzato

```
for (int i = 0; i < n; i++)
      a[i] = b[i] + c[i]
```

→

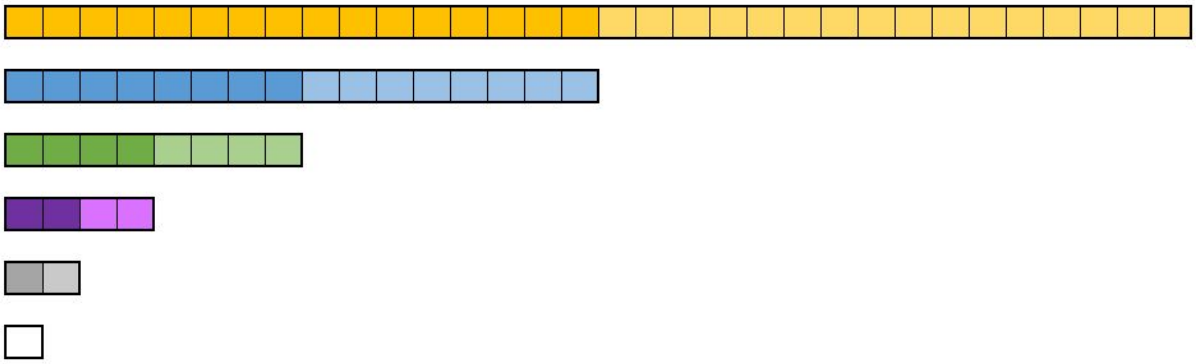
Ottimizzato

```
for (int i = 0, i<n; i+=2)
      a[i] = b[i]+c[i]
      a[i+1] = b[i+1] + c[i+1]
```

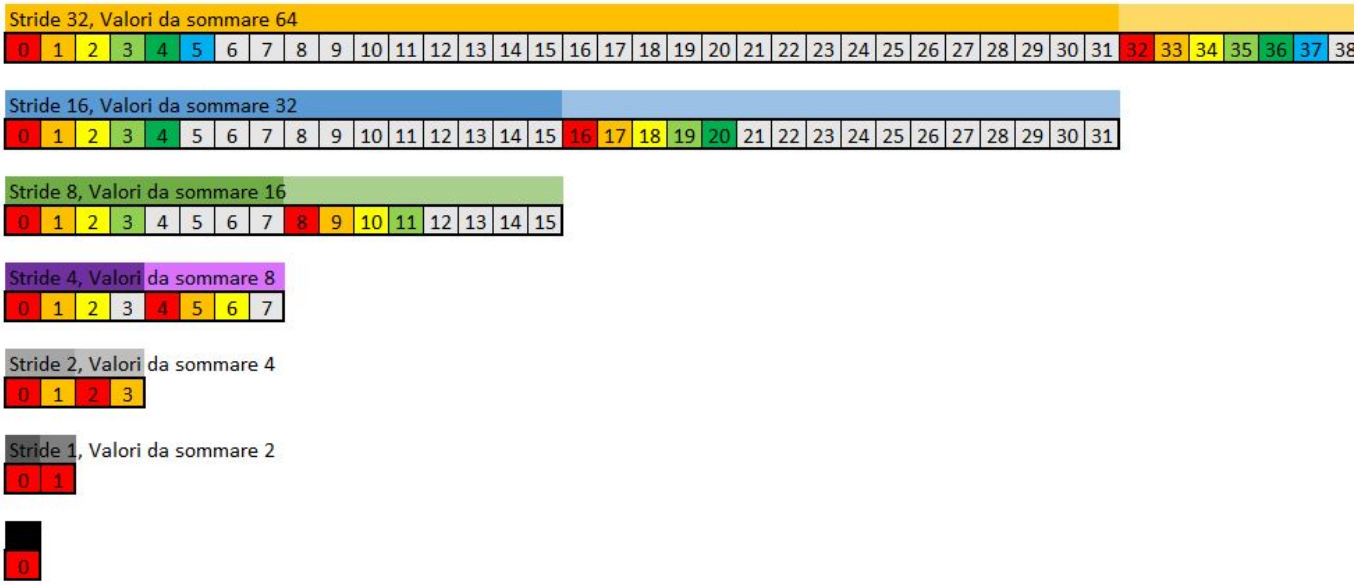
- **Si può fare anche il warpUnrolling, cioè fare l'unrolling a livello di blocco:**

```
__global__ void blockParReduceUroll (int *in, int *out, ulong n){
      uint tid = threadIdx.x;
//BlockDim è di 1024, informazione data dal main
      ulong idx = blockIdx.x * blockDim.x + threadIdx.x;

      id (idx >= n)
            return;          //controllo che l'idx non superi la dimensione del blocco

// inzializzo thisBlock al valore di inizio della global memory, ma nel punto in cui inizia
// la matrice con i dati da analizzare
      int *thisBlock = in + blockIdx.x * blockDim.x;

// inizia la reduction
// con lo shift di stride, questo valore shifta di un bit, cioè da 1024 subirà questa modifica di
// indici: 1024/2 = 512 → 128 → 64 → 32. Quando raggiunge 32 esce dal ciclo.
      for (int strinde = blockDim.x/2; stride > 32; stride >>= 1) {
            if (id < stride)

// tutti i thread dall'ID minore della metà della dimensione del blocco
// Ogni thread fino a 512, poi 128, poi 64, calcolano la somma tra loro e l'altro blocco
                  thisBlock[tid] = thisBlock [tid + Stride];
      }

//Quando esce dal ciclo for, i pimi 32 warp eseguiranno il seguente frammento di codice
      if (tid < 32) {

//Sono rimasti in campo solo 32 thread, questi 32 man mano eseguono
// CONTEMPORANEAMENTE il codice, quindi sono sincronizzati a livello di istruzione (tutti
// eseguiranno insieme "vmem[tid] += vmem[tid + 32]; ", quindi alla prossima istruzione, ci
// saranno tutte le somme corrette e i thread sono sincronizzati tra loro.
            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];
      }
}
if (tid == 0)
      out[blockIdx.x] = thisBlock;
```

Il concetto è che ad ogni ciclo for, si riduce lo stride, cioè si riducono i thread che effettivamente fanno dei calcoli utili perchè sommano se stessi all'altra metà.



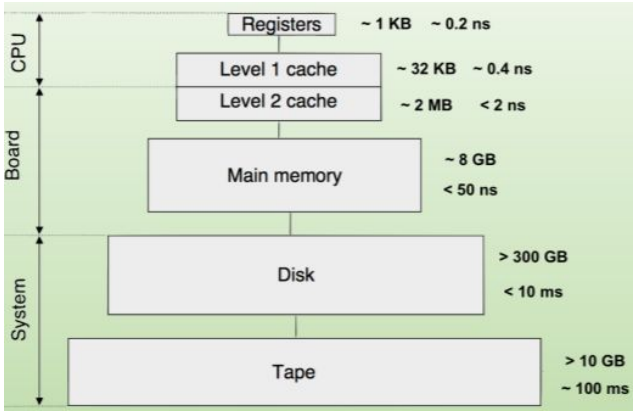
I vari colori corrispondono alle somme che vengono fatte ad ogni istruzione



8 - I modelli di memoria e shared memory

8.1 - modello di memoria cuda

La velocità delle computazioni è influenzata anche dalla velocità di spostamento dei dati, più sono veloci le memorie (lettura e scrittura) più sono costose.



Principio di località

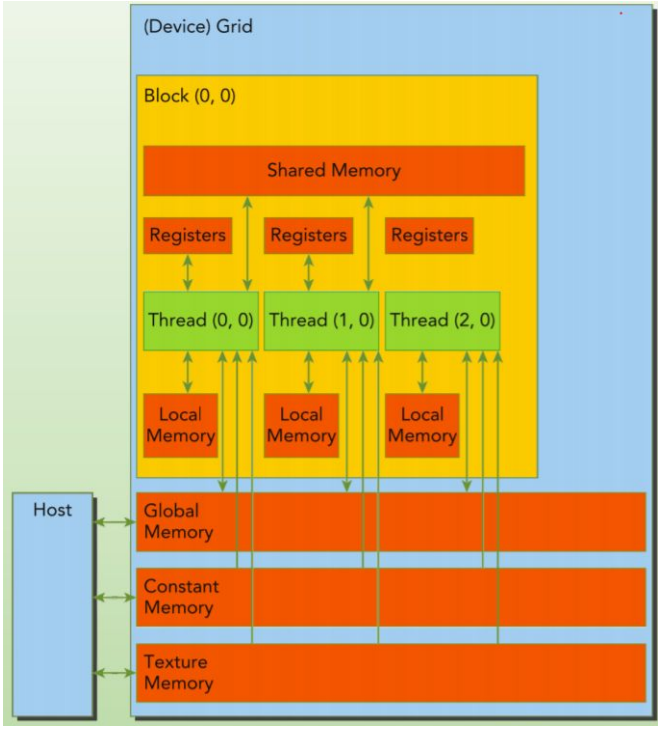
Località spaziale:
Con l'esecuzione di una funzione a indirizzo i, è molto probabile che l'istruzione con indirizzo $i + \Delta i$ venga eseguita (con Δi pari ad un piccolo intero).

Località temporale
Con l'esecuzione di un'istruzione al tempo t, è molto probabile che la stessa istruzione verrà eseguita al tempo $t + \Delta t$ (dove Δt è un piccolo intero).

8.1.2 - Tipi di memorie

Ci sono due tipi di memorie:
Non-Programmabile: non c'è nessun controllo sull'allocazione dei dati, questi vengono gestiti automaticamente.

Programmabile: in cui il programmatore ha il controllo esplicito di lettura e scrittura dei dati che transitano nella memoria.



Le memorie programmabili sono:

- **Register**

Sono le memorie più veloci in assoluto, sono ripartiti tra i warp attivi all'interno del kernel.
Una variabile dichiarata senza nessun qualificatore (`__device__`/`__shared__`/`__constant__`), viene gestita risiede automaticamente in un registro.

Meno registri usa il kernel, più blocchi di thread è probabile che risiedano sul multiprocessore.

register spilling: se si usano più registri dei consentiti, le variabili vengono spostate nella local memory.
Il numero massimo di registri per thread si può impostare manualmente a tempo di compilazione usando l'opzione

```
nvcc -maxrregcount 18 kernel.cu
```

Register bounds
Per minimizzare lo *spilling* si può ricorrere al qualificatore `__launch_bounds_` nella definizione del kernel.

```
__global__ void
__launch_bounds__(maxThreadsPerBlock, minBlocksPerMultiprocessor) MyKernel(...) {

...

}
```

Oppure usando il flag di compilazione `-maxregcount=32`

- **Local Memory**

La memoria è lenta come la global memory. è una memoria **locale** ai **thread**.
Viene usata per contenere: le *variabili automatiche* non contenute nei registri, gli *array locali* (i cui indici non sono determinati a compile-time), *strutture locali* di grosse dimensioni, *variabili spostate dai registri* a causa del register spilling.

La local memory **risiede nella device memory**, quindi gli cessi hanno la stessa latenza e ampiezza di banda della global memory. L'allocazione non è controllata dal programmatore ma dal compilatore nvcc. Le GPU con compute capability 2.0 e superiori, i dati sono posti anche in cache L1 a livello di SM e L2 a livello di device

- **Constant memory**

Risiede in **device memory** e ha una cache dedicata per ogni SM. è una memoria utile per ospitare dati a cui si accede in modo uniforme e in sola lettura, per dichiarare una variabile per questa area di memoria bisogna usare il qualificatore `__constant__`.
La variabile costante deve avere scope globale, quindi al di fuori di qualsiasi kernel. La dimensione massima di una variabile constant è di 64kb.
La constant memory può essere inizializzata dall'host usando

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

In cui si copia **count** byte della memoria puntata da **src** alla memoria puntata da **symbol**, che può appunto risiedere in global memory o in constant memory.
La constant memory lavora in maniere ottimizzata **quando tutti i thread di un warp leggono dallo stesso indirizzo di memoria.**

- **Shared Memory**

Memoria programmabile on-chip, quindi con banda molto più alta e minore latenza rispetto alla local memory e alla global memory. è suddivisa in moduli della stessa dimensione chiamati **bank** che possono essere acceduti simultaneamente. Ogni richiesta di **n indirizzi** che riguardano **n distinti bank** sono serviti simultaneamente. Con i blocchi di thread condivide anche scope e lifetime.
La shared memory serve come base per la comunicazione **inter-thread**, cioè i thread in un block possono cooperare scambiandosi i dati che risiedono nella shared memory.
L'accesso alla shared memory deve essere sincronizzato: `void _syncthreads()`

- **Texture memory**

Risiede nella **device memory**, ha una cache per ogni SM. La read-only cache è acceduta solo attraverso essa. 2La cache read-only supporta in termini hardware il filtraggio o interpolazione floating point nel processo di lettura dei dati. La texture memory è ottimizzata anche per la località spaziale 2D (per dati espressi con matrici, i thread in un warp che

usano la texture memory per leggere dati 2D hanno prestazioni migliori). è adatta ad applicazioni come elaborazioni di immagini e video.

- **Global memory**

è la memoria più grande e con la latenza maggiore. Il "global" deriva dallo scope di uesta memoria e dalla lifetime globale (può essere acceduta da ogni thread in ogni SM)

dichiarazione statica: `__device__ int a[N]`

dichiarazione dinamica:
```
cudaMalloc((void**)&d_a, N);
cudaFree(d_a)
```

L'accesso da parte dei thread appartamenti a blocchi distinti e non sincronizzabili, da potenzialmente modifiche incoerenti. La global memory corrisponde alla **device memory** ed è accessibile attraverso transazioni da 32, 64 o 128 byte.

L'ottimizzazione delle transazioni in memoria sono importanti per le prestazioni. Quando un warp esegue operazioni di load e store, il numero di transazioni richieste dipendono da due fattori:

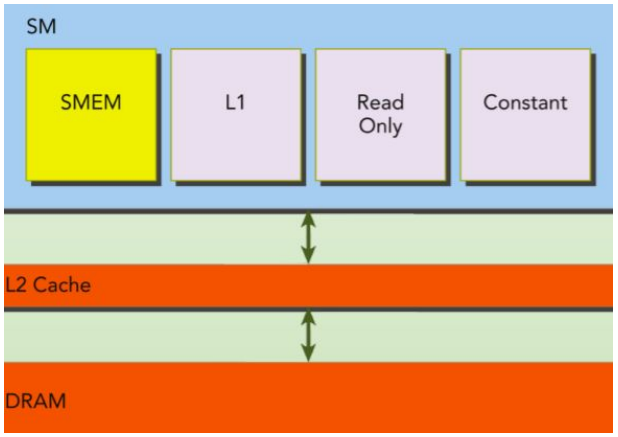
- **Distribuzione** degli indirizzi attraverso i thread di un warp.
- **Allineamento** degli indirizzi di memoria nell transazioni.

- **Cache su GPU:** Non sono programmabili, sono di 4 tipi
  - ❖ L1 (una per ogni SM)
  - ❖ L2 (condivisa tra tutte le SM)
  - ❖ Read-only constant
  - ❖ Read-only texture

L1 e L2 sono usate per memorizzare i dati in memoria locale e globale. Le operazioni di load possono essere cached, mentre quelle di memory store no. Ogni SM ha una read-only constant e una read-only-texture per migliorare le prestazioni dei rispettivi spazi di memoria.

Tabella riassuntiva:

MEMORY	ON/OFF CHIP	CACHED	ACCESS	SCOPE	LIFETIME
Register	On	n/a	R/W	1 thread	Thread
Local	Off	†	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	†	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation



8.2 - Uso della shared memory (SMEM)

Viene utilizzata come canale di comunicazione per i thread dello stesso blocc..

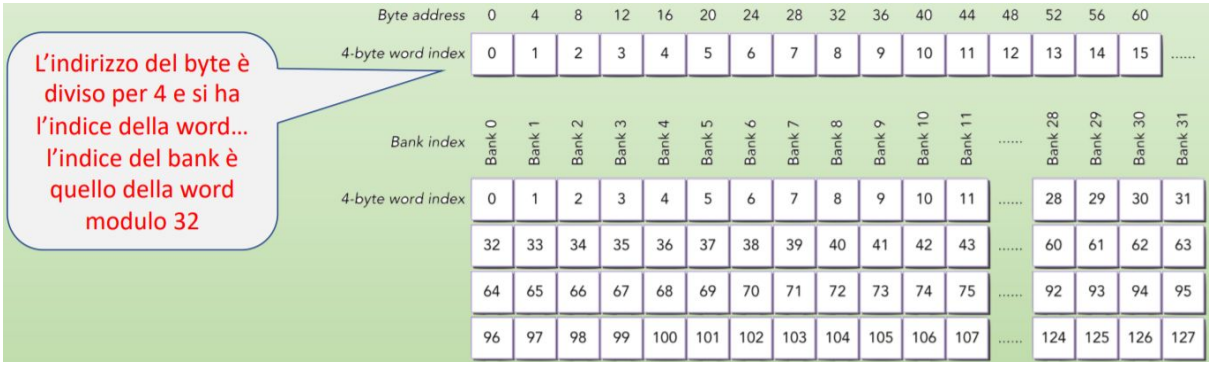
La cache per la global memory viene gestita direttamente dal programmatore.

Memoria scratch pad per elaborare dati on-chip e migliorare i pattern di accesso alla global memory.

Latenza e banda sono migliori rispetto alla global memory.

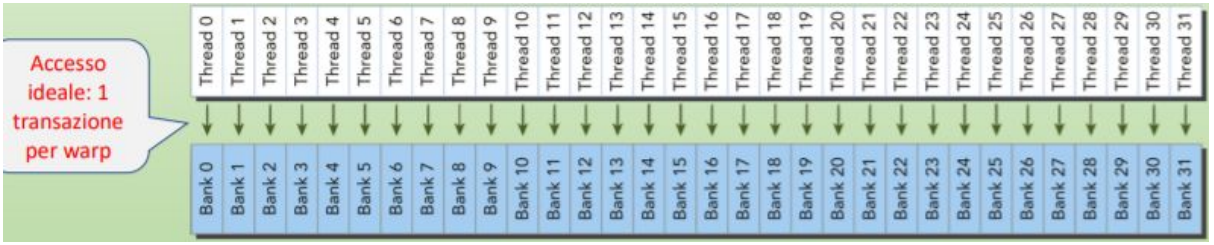
8.2.1 - Organizzazione SMEM

La SMEM della Fermi è suddivisa in blocchi da **4 byte** (chiamati word) o da 32bit.
Ogni word può contenere un int, un float, quattro char, …
Dati 32 bank, ogni word è memorizzata in bank distinti a gruppi di 32:

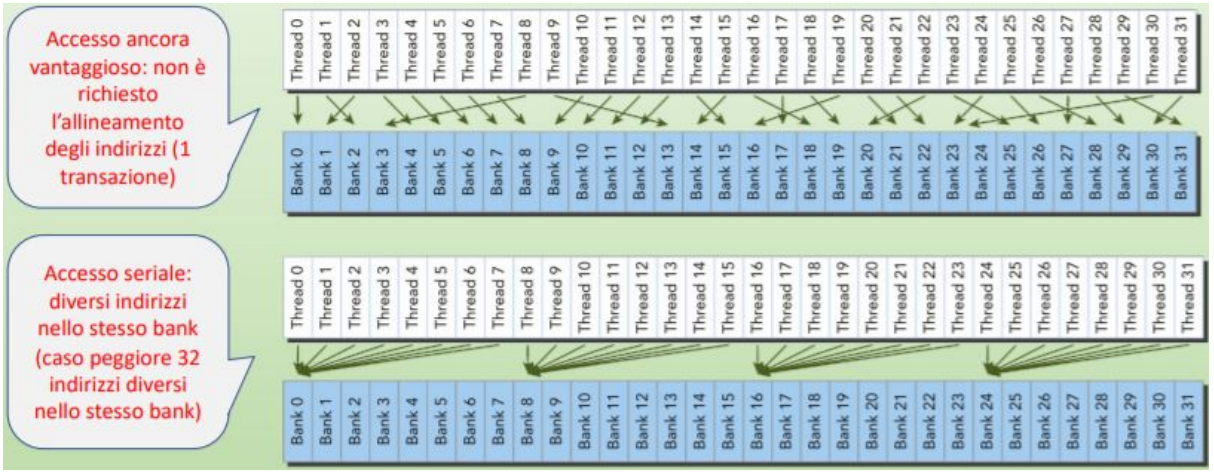


SMEM a runtime
Viene ripartita tra tutti i blocchi residenti in una SM, maggiore è la shared memory richiesta da un kernel, minore è il numero di blocchi attivi concorrenti. Il contenuto della shared memory ha lo stesso lifetime del blocco a cui è stata assegnata. L'accesso è per warp, quindi nel caso migliore c'è una transizione (contemporanea per tutti i 32 thread), mentre nel caso peggiore accedono tutti in posizioni diverse e ci sono 32 transizioni.

I moduli (bank) possono essere acceduti simultaneamente, quindi se l'operazione di load o store eseguita da un warp richiede al più di un accesso per bank, si può fare in una sola transizione.

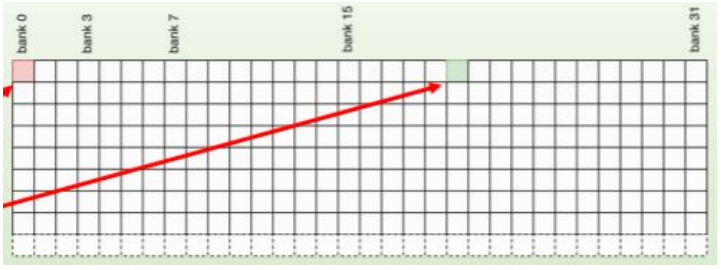


Conflitti: il conflitto avviene quando diversi indirizzi di shared memory insistono sullo stesso bank, cioè l'hardware effettua tante transizioni quante ne sono necessarie per eliminare i conflitti, cioè comporta una diminuzione della banda pari al numero di transazioni separate.



8.2.2 - Pattern di accesso

- Caso broadcast: un singolo valore letto da tutti i thread in un warp da un singolo bank
  Il primo caso: `float f = array_f[threadIdx.x*0]`
  il secondo caso: `float f = array_f[20]`



- Caso parallelo: un singolo valore letto da un singolo bank (caso migliore)

```
float f = array_f[threadIdx.x]
```

- Conflitto doppio: tutti i thread in un warp richiedono una word di indice doppio il threadidx.x (8 byte)

```
float f = array_f[threadIdx.x*2]
```

- Nessun conflitto: strutture di 3 word come terne di punti nello spazio

```
float f = array_f[threadIdx.x*3]
```

Configurazione SMSM/L1 cache
Ogni SM ha 64Kb di memoria on-chip, la shared memory e la L1 cache condividono questa risorsa hardware.
Se si usa la shared memory manualmente, è giusto seguire certi criteri:

- Si deve privilegiare la dimensione di 48kb a discapito della L1
- Da kepler in poi, se si usano molti registri, lo spilling ricade sulla cache L1
- Si può determinare il numero di registri usati dal kernel specificando nelle opzioni di compilazione `-Xptxas -v`.

CUDA fornisce due metodi per configurare la L1 cache e la shared memory

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

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

```
cudaFuncCachePreferNone:      no preference(default)
cudaFuncCachePreferShared:    prefer 48KB shared memory and 16 KB L1 cache
cudaFuncCachePreferL1:        prefer 48KB L1 cache and 16 KB shared memory
cudaFuncCachePreferEqual:     prefer 32KB L1 cache and 32 KB shared memory
```

.

Oppure si può configurare a runtime prima di lanciare il kernel:

```
cudaError_t cudaFuncSetCacheConfig(const void* func, cudaFuncCacheca cheConfig);
```

8.2.3 - Allocazioni statica e dinamica della SMEM

Una variabile shared memory può essere dichiarata sia locale a un kernel che globale in un sorgente (risulta globale a tutti i kernel lanciati dal sorgente).
Una variabile shared ha il qualificatore `__shared__`.

```
__global__ void mioKernel() {
    __shared__ int i;
    __shared__ float array_f[128];
    __shared__ int array_i[10][10];
    array_f[threadIdx.x] = 0;
    . . .
}
```

Il **metodo statico** richiede che la dimensione della memoria sia nota al momento di compilazione.
CUDA supporta dichiarazioni statiche di array multidimensionali 1D/2D/3D.

Se la dimensione non è nota al tempo di compilazione, è possibile **dichiarare una variabile dinamicamente** adimensionale con la keyword extern. Permette la dichiarazione di array 1D, può essere local (kernel scope) o globale (file scope).

```
extern __shared__ int array[];
```

Mentre per allocare la shared memory dinamicamente, bisogna indicare un terzo argomento nella chiamata del kernel.

```
kernel<<<grid, block, N * sizeof(int)>>>(...)
```

L'allocazione multipla, per dichiarare più array nella stessa area, necessita dell'uso esplicito degli offset

```
short array_s[128];
float array_f[64];
int array_i[256];
```

```
int main(void) {
    int nBytes = 128*sizeof(short)+64*sizeof(float)+256*sizeof(int);
    kernel<<<1, 1, nBytes>>>();  // param. dimensione shared memory
    cudaDeviceSynchronize();
    return 0;
}
```

```
extern __shared__ float array[];  // dynamic shared memory
__global__ void kernel() {
    short* array0 = (short*)array;          // offset inizio short
    float* array1 = (float*)&array0[128];   // offset inizio float
    int* array2 = (int*)&array1[64];        // offset inizio int
    for (int i = 0; i< 128; i++)
        array0[i] = 'a';
    for (int i = 0; i< 64; i++)
        array1[i] = 1.0;
    for (int i = 0; i< 256; i++)
        array2[i] = 1;
}
```

Uso della shared memory:

1) Carica i dati della device memory alla shared memory
2) Sincronizza i thread del blocco al termine della copia (per la coerenza dei dati)
3) Elabora i dati in shared memory
4) Sincronizza i thread per aspettare di avere tutti i risultati aggiornati
5) Si scrivono i risultati dalla device memory alla host memory

3/5/19 ---------- lezione 6

9 - Global memory

Una variabile può essere **statica**, dichiarandola con il qualificatore `__device__`, ma non può essere acceduta dall'host, può essere acceduta mediante l'istruzione `cudaGetSymbolAddress` che fornisce l'indirizzo invece di usare &.
Per copiare i dati da host a device `cudaMemcpyToSymbol` e `cudaMemcpyFromSymbol`

```
__device__ int devCount;   // static global var

/* kernel that uses the global var */
__global__ void incGlobalVariable() {
   devCount++; // gloabl var change
}
```

Mentre per avere variabili **dinamiche** si allocano da host

```
// dynamic global var
float *dev = (float *) malloc(nbytes);

// free memory
cudaFree(dev);
```

Riassunto dei qualificatori delle variabili

QUALIFIER	VARIABLE NAME	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

9.1 - Pinned Memory

A causa della virtualizzazione della memoria da parte dell'host, la ram potrebbe subire uno swapping delle pagine coinvolte nella comunicazione con la gpu. Per impedire ritardi a causa del page fault, si adotta la tecnica della *pinned memory*, quindi la porzione di memoria dichiarata page-locked o pinned serve per effettuare un trasferimento sicuro sul device e non subisce swap.

Si alloca la memoria pinned
`cudaError_t cudaMallocHost(void **devPtr, size_t count);`
si dealloca la memoria pinned
`cudaError_t cudaFreeHost(void *devPtr);`

9.1.2 - Zero-copy memory

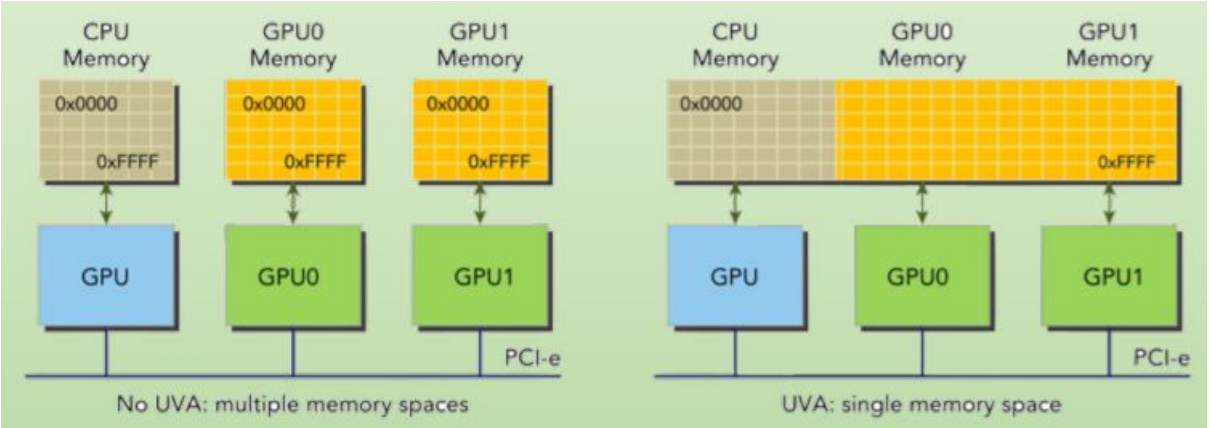
Un blocco di memoria host pinned può essere *mappata* nello spazio di indirizzamento del device, in modo da permettere al kernel di accedere alla memoria dell'host direttamente dalla gpu. Si usa il flag `cudaHostAllocMapped` nell'istruzione `cudaHostAlloc()`
Il blocco allocato ha due indirizzi, host e device. Vantaggi e svantaggi:

- Sfrutta la memoria host quando è insufficiente quella device
- Evita il trasferimento esplicito tra host e device
- Considerando che la memoria è condivisa, le applicazioni devono sincronizzare gli accessi usando stream o eventi per evitare read-after-write o write-after-read o write-after-write
- Si possono avere problemi con le operazioni atomiche.

9.1.3 - Unified Virtual Addressing (UVA)

La memoria host e le memorie device possono condividere un singolo spazio di indirizzamento, la memoria è più semplice e basta un solo puntatore per accedere ai dati (non è necessario fare `cudaMemcpy`). Per allocare la memoria si usano le chiamate `cudaMallocManaged(pointer, size, flag)` che sostituisce `cudaMalloc`.
`cudaMemAttachHost` se si vuole condividere con un solo host e `cudaMemAttachGlobal`

se si vuole condividere con altre GPU. Il qualificatore `__managed__` con `__device__` permette di dichiarare variabili device accessibili dall'host con scope globale.



The CPU cannot access any unified memory as long as GPU is executing a **cudaDeviceSynchronize()** call is required for the CPU to be allowed to access unified memory

```
__device__ __managed__ int x, y = 2; // Unified memory

__global__ void mykernel() {         // GPU territory
   x = 10;
}

int main() {                         // CPU territory
   mykernel <<<1,1>>> ();
   y = 20;                           // ERROR: CPU access concurrent with GPU
   return 0;
}
```

The GPU has exclusive access to unified memory when any kernel is executed on the GPU, and this holds even if the kernel does not touch the unified memory

```
__device__ __managed__ int x, y = 2; // Unified memory

__global__ void mykernel() {         // GPU territory
   x = 10;
}

int main() {                         // CPU territory
   mykernel <<<1,1>>> ();
   cudaDeviceSynchronize();
   y = 20;                           // Now the GPU is idle, so access to "y" is OK
   return 0;
}
```

9.2 - Migliorare le prestazioni

Memory latency: tempo necessario a soddisfare una richiesta dati in memoria
Memory bandwidth: il tasso con il quale la device memory viene acceduta da una SM, si misura in byte per unità di tempo.

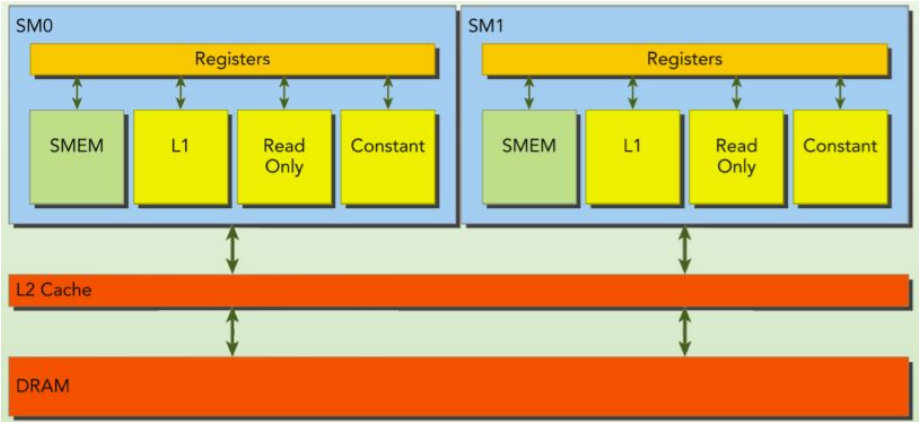
effective bandwidth(GB/s) = $\frac{(bytes\ read + written\ bytes) \cdot 10^{-9}}{time\ elapsed}$

Per migliorare le prestazioni occorre che le istruzioni vengano eseguite a livello warp e gli accessi in memoria dipendono dalle operazioni svolte nel warp. Per un dato indirizzo si esegue un'operazione di loading o storing, cooperativamente i 32 thread presentano una singola richiesta di accesso che viene servita da una o più transazioni in memoria.

Schema delle cache:

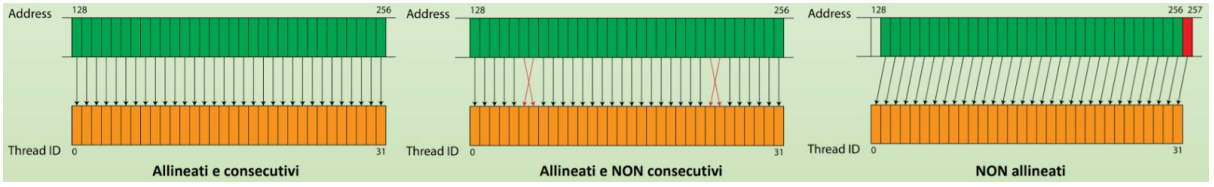
- Per global memory si intende lo spazio logico acceduto dai kernel, spazio fisico la DRAM. Le richieste vengono eseguite dalla DRAM o dalle memorie on chip delle SM

- Le transazioni sono da 32 o 128 byte e tutti gli accessi passano attraverso la L2 cache, possono passare anche dalla L1. Se vengono usate entrambe gli accessi sono a 128byte, se solo la L2, sono solo a 32byte.
- Per le architetture che usano la cache L1 per accessi alla global, le cache L1 possono essere abilitate o disabilitate solo a tempo di compilazione.
- se un thread in un warp chiede un valore allocato in 4-byte, comporta una richiesta di 128byte. Quindi per ogni richiesta si usano tutti i byte.



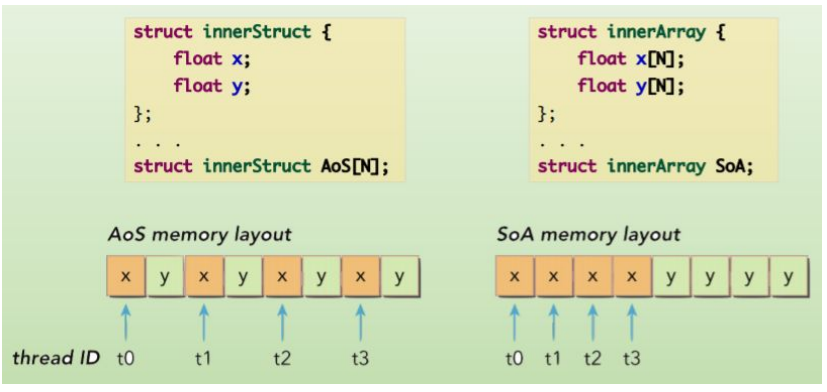
Accessi
Un accesso a memoria è **efficiente** se in una transazione si combinano accessi multipli a memoria tale che siano *allineati e coalescenti*.

- Un accesso è **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 L2 e 128 byte per L1)
- Un accesso è **coalescente** quando tutti i 32 thread di un warp accedono a un *blocco contiguo di memoria*



9.3 - Struct of arrays (SOA) - Array of structs (AOS)

SOA: una struttura composta da array
AOS: un array di strutture
Esempio memorizzazione di una immagine:
Con SOA posso usare una struct con 3 array (RGB) e ogni posizione indica un pixel
Con AOS posso usare un array di struct (che rappresenta il singolo pixel) di cui ognuna contenente 3 variabili per RBG.
Sono due tecniche per rappresentare lo stesso tipo di dati



---------- lezione 7

10 - Stream e concorrenza

10.1 - Concorrenza in CUDA

Tra **CPU** e **GPU** dato che sono dispositivi distinti e che operano indipendentemente tra loro
Grazie all'uso del DMA e della struttura degli SM, **Memcpy** e kernel processing possono operare concorrentemente.
Dalla compute capability 3.x è possibile eseguire fino a 32 kernel contemporaneamente.
Si possono usare più dispositivi gpu per migliorare ancora di più le prestazioni aumentando le unità computazionali (Multi-GPU programming).

Come descritto nel Cuda C programming guide, i comandi asincroni restituiscono il controllo al l'host chiamante prima che il device termini l'esecuzione del task, in quanto i comandi asincroni non sono bloccanti.
Le operazioni non bloccanti sono:
lanciare dei kernel, operazione di copia tra due indirizzi appartenenti alla device memory, operazione di copia da host a device di un blocco di memoria di dimensione minore o uguale a 64kb, operazione di set sulla memoria, infine copie asincrone dalla memoria.

Sincronismo tra GPU e CPU
su **GPU** il sincronismo è a livello di warp, infatti eseguono il codice è eseguito in maniera sincrona, ma warp diversi eseguono con arbitraria sovrapposizione, si può correggere con syncthreads, che impone una barriera di sincronizzazione su un blocco, ma non c'è modo di sincronizzare threads di blocchi differenti.

Sulla **CPU** le CUDA call sono sincrone (quindi bloccanti), il lancio di un kernel è asincrono, in entrambi i casi si forza la sincronizzazione dell'host con `cudaDeviceSynchronize` (blocca l'esecuzione dell'host finchè il device non ha completato tutti i task). Anche se vengono sovrapposte le operazioni, viene preservata la correttezza computazionale.

10.2 - Stream

Uno stream è una sequenza di comandi che vengono eseguiti in ordine, diversi stream sono indipendenti tra loro e potrebbero eseguire comandi in maniera disordinata o concorrente.

Uno stream è riferito a sequenze di operazioni CUDA asincrone che vengono eseguite sul device nell'ordine stabilito dal codice host.

Lo stream incapsula le operazioni, ne mantiene l'ordine fifo e indaga sullo status.
L'esecuzione di operazioni in uno stream è sempre asincrona rispetto all'host, le operazioni appartenenti a stream distinti non hanno restrizioni sull'ordine di esecuzione, quindi si possono eseguire in modo parallelo anche delle griglie diverse, generando un parallelismo a livello di griglia.

10.2.1 - Tipi di stream

Null-Stream è lo stream di default, quando non viene dichiarato nessuno stream, implicitamente è un null-stream.

Non Null-Stream è lo stream che viene dichiarato esplicitamente. è utile per sfruttare il parallelismo a livello di stream, sovrapporre le computazioni/ trasferimenti concorrenti. Non sono

CUDA API stream create
Creazione di uno stream:
```
cudaError_t cudaStreamCreate(cudaStream_t *pStream);
```

Lancio del kernel
```
kernel_name<<<grid, block, sharedMemsize, pStream>>>(argument list);
```

Eliminazione di uno stream
```
cudaError_t cudaStreamDestroy(cudaStream_t pStream);
```

CUDA API stream async
Allocazione spazio su pinned, se il flag è 0, il comportamento è uguale
```
cudaError_t cudaMallocHost (void **ptr, size_t size);
cudaError_t cudaHostAlloc(void **pHost, size_t size, unsigned int flags);
```

Trasferimento asincrono su pinned memory
```
cudaError_t cudaMemcpyAsync(void*dst, const void*src, size_t count,
cudaMemcpyKind kind cudaStream_t stream);
```

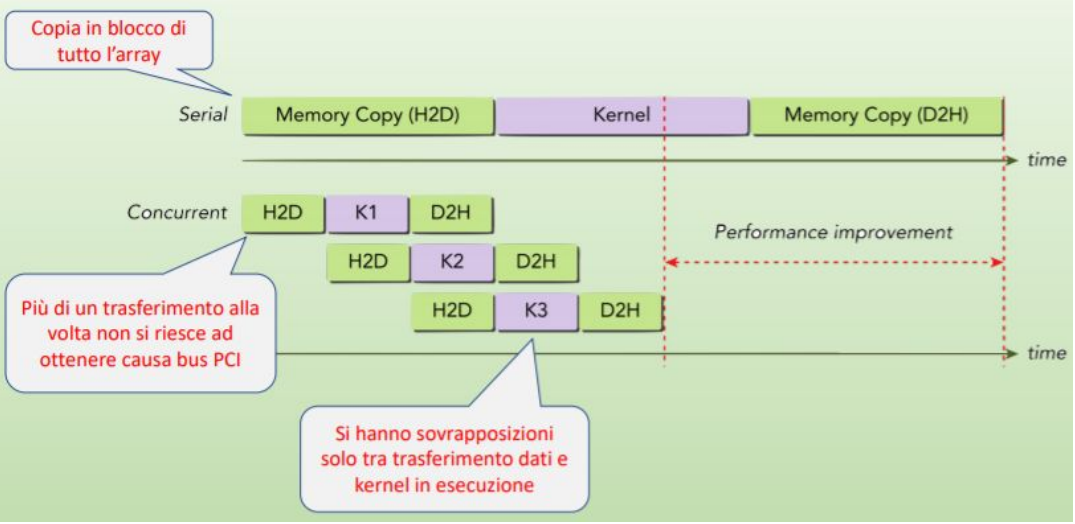
CUDA API stream synchronize
Blocco dell'host sullo stream, forza il blocco dell'host finchè lo stream non è terminato
cudaError_t cudaStreamSynchronize(cudaStream_t stream);

cudaDeviceSynchronize blocca l'host finchè tutti gli stream non sono terminati

Controllo stream completato
```
cudaError_t cudaStreamQuery(cudaStream_t stream);
```

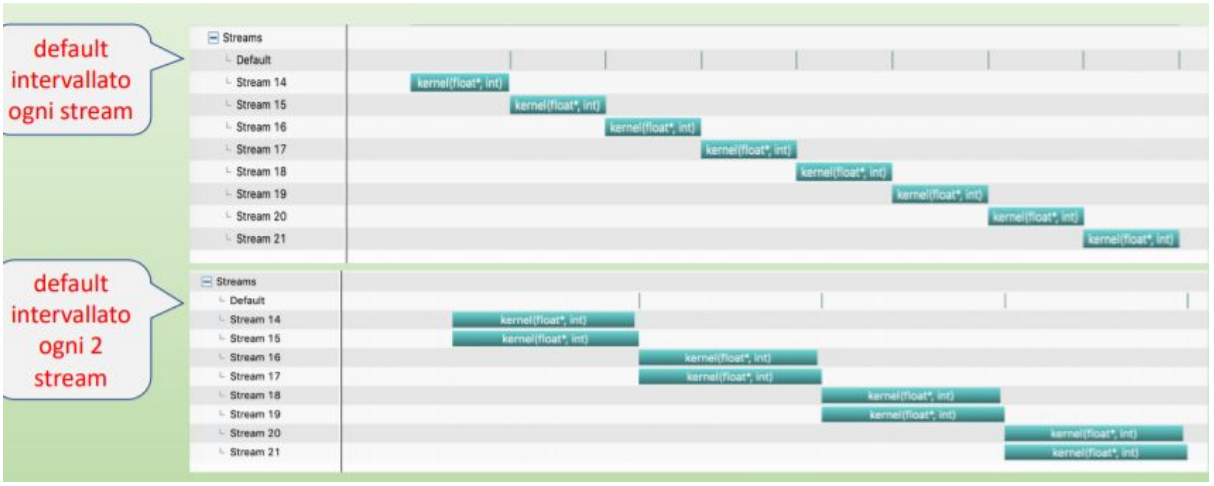
Per utilizzare le sovrapposizioni:

1. il device deve poter eseguire una copia di dati in parallelo, è specificato nel campo *deviceOverlap* della struct *cudaDeviceProp* (tutti i device con compute capability > 1).
2. il kernel e il trasferimento dati devono appartenere a diversi *non-null stream*
3. la host memory deve essere pinned



PRIMA DI CUDA 7:
Nel primo caso si verifica la serializzazione di tutti gli stream, poco efficiente, perchè viene fatta una sincronizzazione implicita.
DOPO CUDA 7
Consente di assegnare ad ogni host-thread un default stream e farlo eseguire in modo concorrente come non-null stream

Multi-stream



per-thread default stream

10.2.2 - Sincronizzazione rispetto Null-Stream

Lo NULL-stream si utilizza per il lancio di kernel crea sincronizzazione.
Il NULL-stream è bloccante per l'host (a parte il lancio del kernel)
Se i NON-NULL-stream non sono bloccanti per l'host o tra loro, possono essere sia sincroni che asincroni rispetto al NULL-stream.
I NON-Null-stream possono essere blocking, il NULL-stream è bloccante, cioè l'esecuzione di operazioni nel NULL-stream bloccherà le operazioni sullo stream che si sta creando fino al loro completamento, Il NULL stream non si sovrappone agli stream.
Oppure Non-blocking stream, lo stream null non è bloccante.

Comportamento tra due stream NULL: sono eseguiti sequenzialmente

```
kernel<<<blocks,threads>>>();
kernel<<<blocks,threads>>>();
```

Comportamento tra uno stream NULL e un not-NULL: sono eseguiti ancora sequenzialmente perchè di default lo stream creato è bloccante rispetto al NULL, quindi non può essere eseguito mentre è in esecuzione il NULL stream.

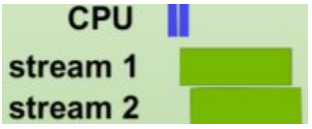
```
cudaStream_t stream1;
cudaStreamCreate(&stream1);
kernel<<<blocks,threads>>>();
kernel<<<blocks,threads,0,stream1>>>();
cudaStreamDestroy(stream1);
```

Comportamento tra Stream NULL e not-NULL non bloccante: i due stream vengono eseguiti sovrapposti

```
cudaStream_t stream1;
cudaStreamCreateWithFlags(&stream1,cudaStreamNonBlocking);
kernel<<<blocks,threads>>>();
kernel<<<blocks,threads,0,stream1>>>();
cudaStreamDestroy(stream1);
```

Comportamento tra due stream not-NULL: i due stream vengono eseguiti sovrapposti anche senza flags, di default non sono bloccanti tra loro

```
cudaStream_t stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);
kernel<<<blocks,threads,0,stream1>>>();
kernel<<<blocks,threads,0,stream2>>>();
cudaStreamDestroy(stream1);
cudaStreamDestroy(stream2);
```



10.3 - CUDA Event

Un evento è un **marker** all'interno di uno stream associato a un punto del flusso delle operazioni, serve per controllare se l'esecuzione di uno stream ha raggiunto un determinato punto del flusso per la sincronizzazione inter-stream.
Si usa per due scopi:

- Sincronizzare l'esecuzione di stream
- Monitorare il progresso del device

Le API CUDA permettono di inserire eventi in qualsiasi punto dello stream e effettuare query sugli stream per sapere lo stato dello stream. Eventi sullo stream di default sincronizzano tutte le precedenti operazioni su tutti gli stream, infatti sono considerati avvenuti solo quando tutte le operazioni precedenti su tutti gli stream sono state completate.

Creazione di un evento

```
cudaEvent_t event;
cudaError_t cudaEventCreate(cudaEvent_t *event);
```

Distruzione di un evento

```
cudaError_t cudaEventDestroy(cudaEvent_t event);
```

Sincronizzazione

Registrazione di un evento su uno stream (0 è per quello di default), segna l'evento

```
cudaError_t cudaEvenRecord(cudaEvent_t event, cudaStream_t stream);
```

Blocco dell'host fino a che non si verifica un evento

```
cudaError_t cudaEventSynchronize(cudaEvent_t event);
```

Check di un evento senza bloccare

```
cudaError_t cudaEventQuery(cudaEvent_t event);
```

Bloccare lo stream finchè non occorre un evento (attende). Sincronizzazione esplicita

```
cudaError_t cudaStreamWaitEvent(cudaStream_t stream, cudaEvent_t event);
```

Impone allo stream passato per argomento di attendere uno specifico evento prima di eseguire ogni altra operazione nello stream successivo alla chiamata `cudaStramWaitEvent`. L'evento può essere allo stesso o a un differente strea.

Per misurare il tempo trascorso tra due eventi in millisecondi

```
cudaError_t cudaElapsedTime(float*ms, cudaEvent_t start, cudaEvent_t stop);
```

Esempio

```
// create two events
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

// record 'start' event on the default stream
cudaEventRecord(start);

// execute kernel
kernel<<<grid, block>>>(arguments);

// record 'stop' event on the default stream
cudaEventRecord(stop);

// wait until the stop event completes
cudaEventSynchronize(stop);

// calculate the elapsed time between two events
float time;
cudaEventElapsedTime(&time, start, stop);

// clean up the two events
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

Quattro tipi di sincronizzazione

1. Livello di device: `cudaDeviceSynchronize`
2. Livello di stream: `cudaStreamSynchronize`
3. Internamente ad uno stream con eventi: `cudaEventSynchronize`
4. Tra stream mediante gli eventi: `cudaStreamWaitEvent`

---------- lezione 8

11 - Modelli Paralleli

Per *computer parallelo* si intende una collezione di processori interconnessi in modo tale da potersi coordinare e scambiarsi i dati.
Per *modello parallelo* si intende il framework che può essere usato per descrivere e analizzare algoritmi paralleli. Si è interessati a semplificare nella descrizione e analisi il sistema creato, senza rinunciare l'indipendenza dell'architettura.
Implementabilità è la capacità di realizzare il sistema con le performance teorizzate.

I modelli usati sono

- Direct acyclic graph (DAG)

Rappresenta bene il parallelismo nel flusso di dati. I nodi con solo un arco in uscita rappresenta l'input ad un nodo, i nodi con solo archi in entrata rappresentano gli output. Il resto dei nodi sono i task. Gli archi tra i nodi raooresebtabi i flussi dei dati in ingresso o in uscita a delle operazioni.

- Shared-Memory

Produce un estensione della ram (PRAM) in parallelo. La memoria risulta avere una dimensione infinito, il numero di processori non è limitato e ogni processore accede alla memoria in un ciclo di clock. Tutti i processori comunicano mediante la memoria e le operazioni sono considerate sincrone. Tutti i processori eseguono lo stesso algoritmo in maniera sincrona.

- Network

11.1 - Parallel pattern design

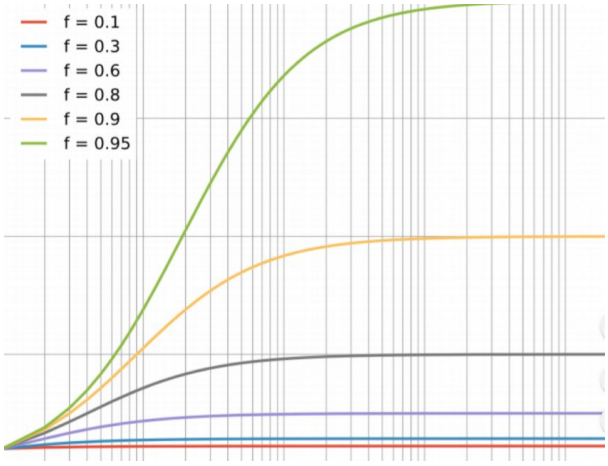
Fare parallel pattern design si intende decomporre una computazione complessa in task più piccoli. Bisogna identificare quali task sono parallelizzabili e coordinandoli tra loro.
Le possibilità sono due granularità differenti, cioè *avere tanti piccoli task* o *avere pochi ma grandi task*.

Un DAG è un task-dependency graph, un grafo orientato aciclico che rappresenta l'ordine della computazione e la sequenza temporale tra le operazioni.
Indici:

- *Massimo grado di concorrenza*: la quantità massima di task eseguiti in parallelo.
- *Grado medio di concorrenza*: quanti task in media sono eseguiti in parallelo.
- *Cammino critico*: il cammino più lungo nel grafo, si misura con la somma degli archi percorsi.
- *Lavoro totale*: quantità di nodi presenti nel grafo.

Scalabilità
Si riferisce a

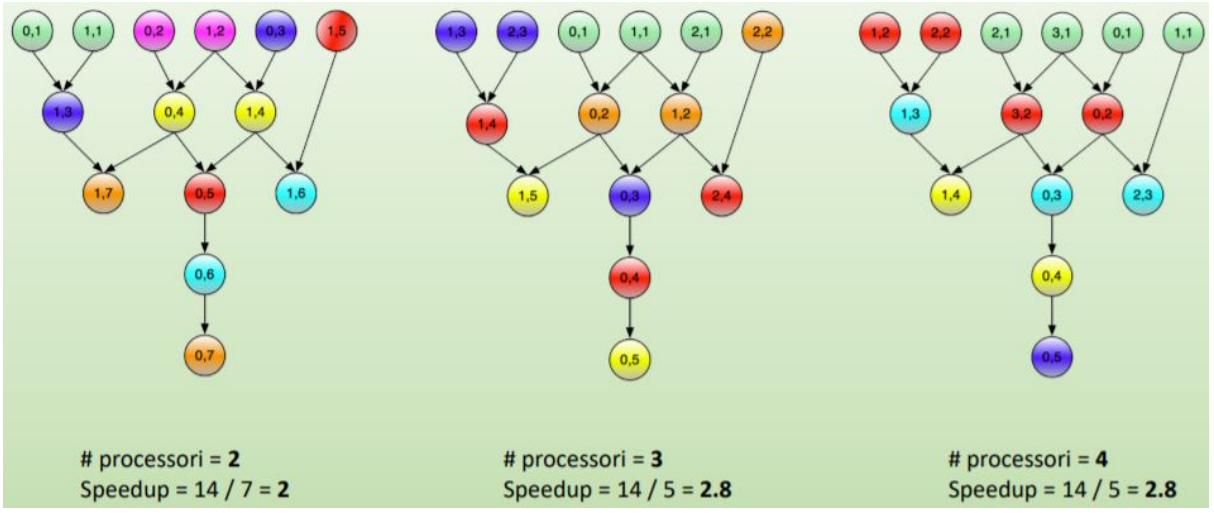
1. parallelismo: si considerano problemi di grandi dimensioni che contengono molte operazioni indipendenti tra loro
2. speedup: si ottiene con l'uso di molti processori, t_1 è il tempo di esecuzione con un processore e t_p è il tempo di esecuzione con p processori → $S_p = \frac{t_1}{t_p}$



La legge di *Amdahl* permette di modellare lo speedup in caso più realistico, cioè in cui solo una porzione f di calcoli siano parallelizzabili usando p processori.
Si assume che un certo carico di lavoro abbia ugual tempo di computazione su ogni processore e si abbia un completamento di una parte sequenziale di peso $(1-f)$.

$$\rightarrow S_{p\,(Amdahl)} = \frac{t_1}{\frac{t_1}{t_p} \cdot (f+(1+f)\cdot p)}$$

La funzione è superiormente limitata da $\frac{1}{1-f}$ per ogni p. Si assume che sia p sia f siano indipendenti, che non è necessariamente vero.



Scan o prefix-sum
L'operazione considera un operatore binario associativo $\oplus$ e un array di **n** elementi $[a_0, a_1, ..., a_{n-1}]$ e restituisce l'array $b = [a_0, (\oplus\ a_1 \oplus a_2), (a_0 \oplus a_1 \oplus ... \oplus a_{n-1})]$

Esempio: dividere un oggetto di 1mt in n parti di dimensioni differenti tra loro.
L'array [3, 5, 2, 7, 28, 4, 3, 0, 8, 1] è il numero di parti e ogni valore corrisponde alla dimensione di una parte. L'array risultante sarà b = [3, 8, 10, 17, 45, 9, 52, 52, 60, 61].
in cui *8* = 3 + 5, *45* = 3 + 5 + 2 + 7 + 28, *61* = 3 + 5 + 2 + 7 + 28 + 4 + 3 + 0 + 8 + 1.
Il metodo è utile per fare l'operazione di divisione in maniera parallela, indipendente dalle divisioni precedenti. Ogni tagliente taglierà esattamente a n cm.

Sorting
Operazioni molto usate, indipendentemente dalla piattaforma o dalla struttura dati da ordinare. Si valuta la complessità computazionale in base al numero di numero di operazioni elementari da fare (*confronti* e *scambi*). La complessità di un algoritmo di sorting si misura come funzione del numero di *n* elementi della sequenza, ci sono algoritmi semplici di complessità $O(n^2)$ e algoritmi evoluti di complessità $O(n \cdot log_2 n)$.

Quicksort
L'algoritmo prevede di dividere ricorsivamente un vettore di n elementi e ordinare i sottovettori per poi riunirli. L'operazione di divisione è chiamata *partizionamento*.
Algoritmo:

1. Se la cardinalità dell'insieme è 1, l'insieme è già ordinato
2. Altrimenti
    a. Si sceglie un elemento pivot nell'insieme e si ridistribuiscono gli elementi dell'insieme in due sottoinsiemi, disponendo nel primo insieme gli elementi minori del pivot, nel secondo insieme gli elementi maggiori o uguali ad un pivot.
    b. applicare nuovamente ai due sottoinsiemi
3. Fine

Il partizionamento:
Scelto un pivot, due indici (uno posizionato in testa all'array e l'altro nella coda) iniziano a scalare, i viene incrementato finchè non trova un elemento maggiore o uguale al pivot, j viene fatto decrementare fino a quando non trova un elemento minore o uguale al pivot.
Dopo aver trovato due elementi che corrispondono, il contenuto negli indici viene scambiato (il contenuto nella posizione i va in j e viceversa). Si prosegue finchè i due indici non arrivano alla posizione del pivot. Il processo termina quando gli indici si sono invertiti (j<i).

Divide et impera
Il metodo per risolvere questi problemi è detto divide et impera, si procede in 3 passi

1. suddivisione del sottoproblema
2. risoluzione del caso base (cioè risoluzione dei sottoproblemi)
3. combinazione delle soluzioni

Questa tecnica porta a soluzioni di costo $O(n \cdot log_2 n)$

Ordinamento bitonico - bitonic mergesort
Ordinamento parallelo basato su **sorting network** (rete di comparatori), che hanno complessità temporale $O(log^2 n)$ e il numero di comparatori usati $O(N \ log \ N)$.
Data una sequenza bitonica, cioè i numeri sono ordinati tc:
$s = < a_1, a_2, ..., a_{n-1} >$ esiste $0 < i < n-1$ (un indice all'interno dell'insieme s)
tale che: $a_0 \leq a_1 \leq a_2 \leq ... \leq a_i$ e $a_{i+1} \geq a_{i+2} \geq ... \geq a_{n-1}$
Quindi le due sequenze si dicono bitoniche.

Se la sequenza è tale che n è una potenza di 2,
$a_0 \leq a_1 \leq ... \leq a_{\frac{n}{2}-1}$ and $a_{\frac{n}{2}} \geq a_{\frac{n}{2}+1} \geq ... \geq a_{n-1}$

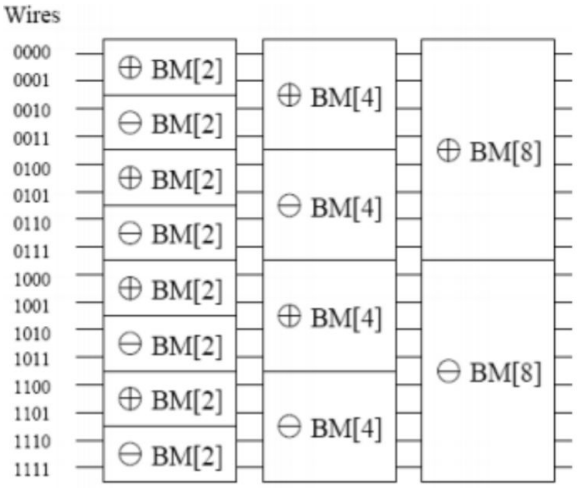
Allora le sequenze s1 e s2 sono ancora bitoniche (*bitonic split*)
$s1 = < min\{a_0, a_{\frac{n}{2}-1}\}, min\{a_1, a_{\frac{n}{2}+1}\}, ..., min\{a_{\frac{n}{2}-1}, a_{n-1}\} >$

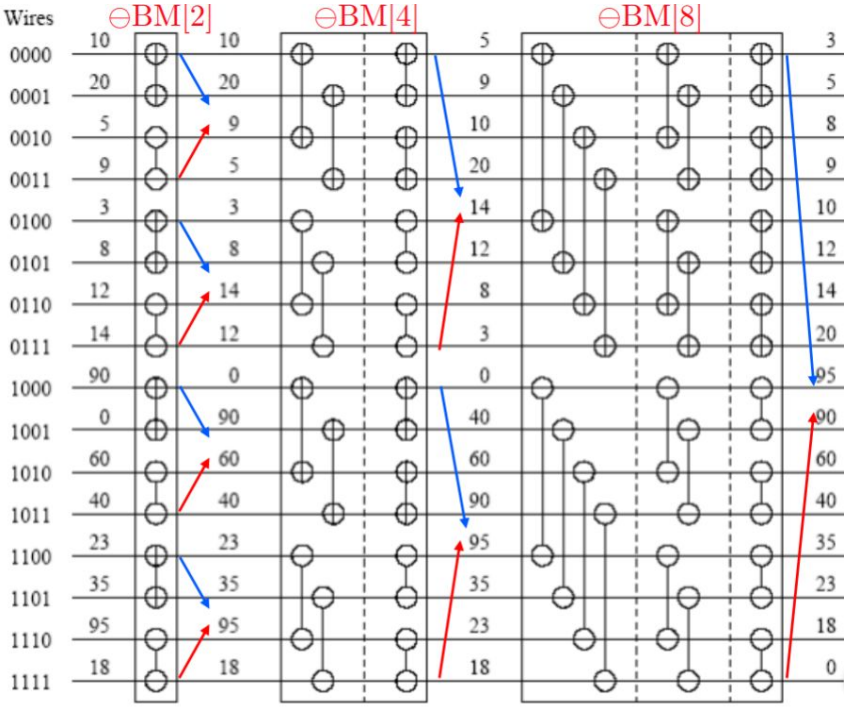
$$s2 = \ < max\{a_0,\ a_{\frac{n}{2-1}}\},\ max\{a_1,\ a_{\frac{n}{2+1}}\},\ ...,\ max\{a_{\frac{n}{2-1}},\ a_{n-1}\} >$$

Dopo log n -1 passi la sequenza avrà solo due elementi da ordinare.
Il bitonic merging network con n input produce una *sequenza crescente* con il comparatore $\oplus BM[n]$ e una *sequenza decrescente* con il comparatore $\ominus BM[n]$.

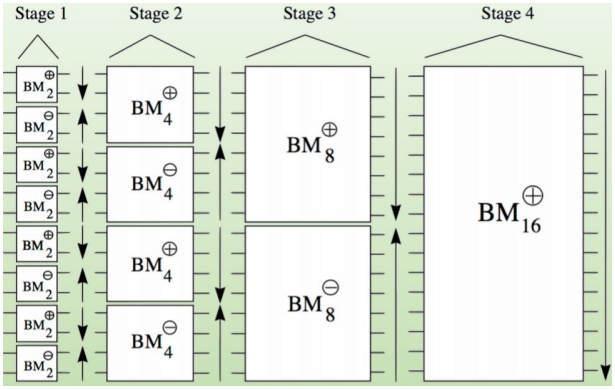
1. Costruzione di una sequenza bitonica

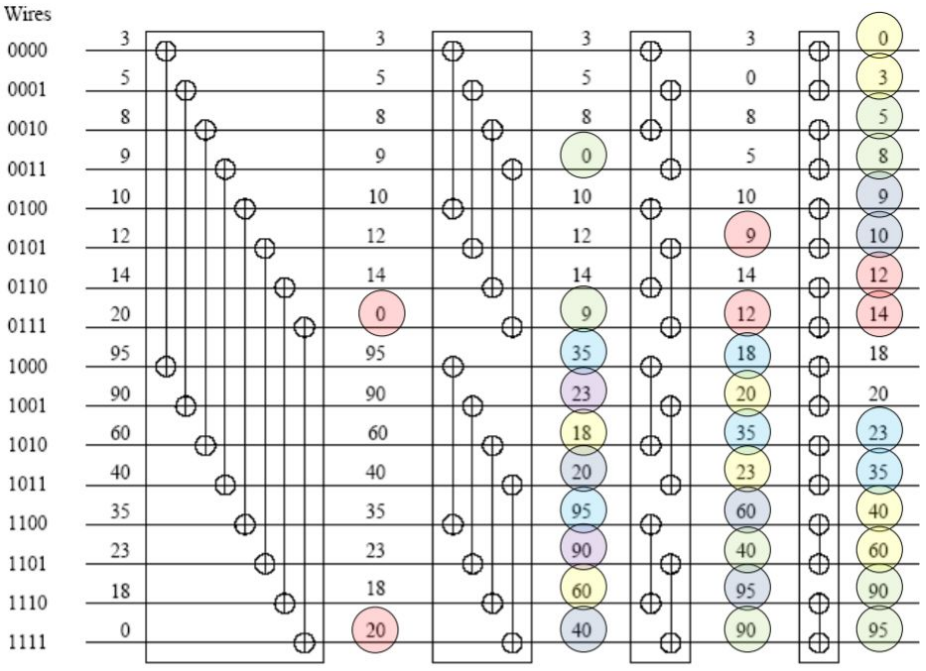
Costruisco la sequenza bitonica ordinando i primi due elementi con $\oplus BM[2]$ e i successivi due elementi con $\ominus BM[2]$. Una sequenza di lunghezza 2 è sempre bitonica.



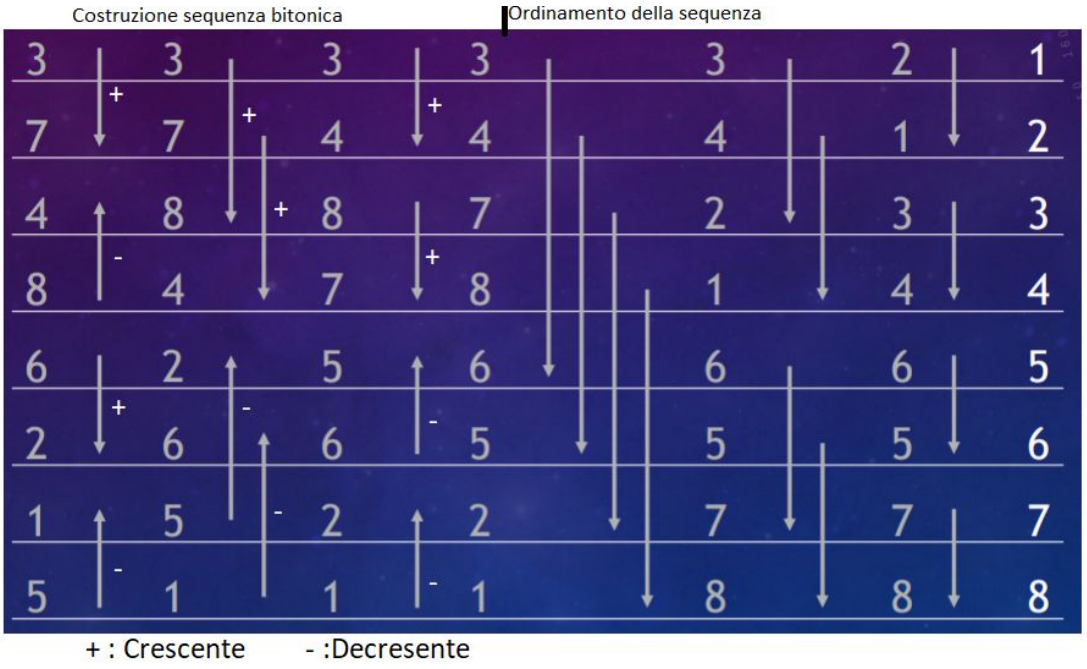


2. Ordinamento





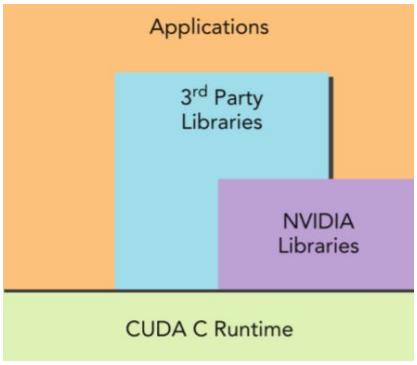
Esempio Passo per passo



$Log_2 n$ colonne $\frac{n}{2}$ comparatori $O(log^2 n)$ costo dell'ordinamento

---------- lezione 9

12 - Librerie CUDA



Le libreria cuda sono una raccolta di implementazioni accelerate dalla GPU, queste offrono un buon bilanciamento tra usabilità e prestazioni. Molte delle API sono di proposito simili a quelle della libreria standard in modo da facilitare il porting. Nessun costo di mantenimento è a carico degli sviluppatori della libreria, compito svolto dagli esperti in CUDA programming → maintainer della libreria.
Alcune delle librerie sono sviluppate direttamente da NVIDIA.

Tutte si appoggiano sopra il livello RUNTIME al pari delle applicazioni sviluppate dagli utenti.

Workflow:

1. *Creare un **handle** specifico della libreria* per ogni libreria che si vuole usare, manterrà il contesto in cui opera e gestisce i dati
2. *Allocare la **device memory** per gli **input** e **output** alle funzioni della libreria*, se gli input non sono in un formato già supportato dalla libreria, bisogna convertirli nel formato specifico di uso della libreria.
3. ***Popolare** con i **dati** nel formato specifico*
4. ***Configurare** le computazioni per l'esecuzione*

5. *Eseguire la **chiamata** della funzione desiderata che avvia la computazione*
6. ***Recuperare** i **risultati** dalla device memory*
7. *Se necessario, **convertire** i dati nel formato desiderato*
8. ***Rilasciare le risorse** CUDA allocate per la libreria*

- **cuBLAS**

Libreria per **Basic Linear Algebra Subprograms** (**BLAS**, tutte le operazioni sono pensate per essere usate su *vettori* o matrici *dense*.
Come la corrispondente per la CPU (BLAS) a libreria è divisa in diverse classi, basate sui tipi su cui operano.

1. cuBLAS livello 1 contiene operazioni sui vettori, come la somma tra vettori
2. cuBLAS livello 2 contiene operazioni tra matrici e vettori
3. cuBLAS livello 3 contiene operazioni tra matrici

Per operazioni con dimensionalità superiore c'è la libreria LAPACK. Sia BLAS che LAPACK furono scritte originariamente in fortran, storicamente usa il *column-major order* per l'indexing e lo storage degli array.
Ogni operazione è svolta asincronamente, quindi il programmatore deve attendere il completamento prima di accedere ai dati.
I dati trattati sono dai floating point a singola o doppia precisione infine i numeri complessi.

Operare con cuBLAS

1. ***Creare l'handle*** di cuBLAS usando **cublasCreateHandle**
2. ***Allocare memoria nella device memory*** per input e output con **cudaMalloc**
3. ***Popolare*** la memoria allocata con gli input usando **cublasSetVector** e **cublasSetMatrix**
4. ***Eseguire*** **cublasSgemv** per avviare le operazioni
5. **Recuperare** i dati dalla device memory usando **cublasGetVector** e **cublasGetMatrix**
6. Distruggere l'handler con **cublasDestroy** e liberare la memoria con **cudaFree**

- **cuRAND**

Fornisce dei generatori di numeri randomici semplici ed efficienti. I generatori possono essere

a. *Pseudorandom*: la sequenza di numeri soddisfa molte delle proprietà statistiche di una vera sequenza casuale. La sequenza è generata da un algoritmo deterministico del tipo $A_{n+1} = (Z \cdot A_n + I)\ mod(M)$.
b. *Quasirandom*: la sequenza di punti n-dimensionali uniformemente generati da un algoritmo deterministico (non si formano cluster nello spazio di generazione, sono equispaziati tra loro)

cuRAND si compone di due parti, una libreria per l'host (curand.h) e una per il device (curand_kernel.h).

- Per l'host:

curandCreateGenerator() crea un generatore
curandSetRandomGeneratorSeed() setta il seed per il generatore
curandGenerate_____() fornendo opportuni parametri e in base alla distribuzione scelta, permette di fare la generazione.

- Per il device: necessaria questa libreria perchè bisogna trasferire i dati sul device, quindi anche se guadagna in prestazioni, è necessario passare i dati.

curandGenerate() è molto più veloce della versione CPU

Operare con cuRAND

1. Preallocare un set di oggetti *cuRAND state objects* nella device memory per ogni thread che usa un generatore random.
2. Facoltativamente pre-allocare la device memory per salvare valori random generati da cuRAND.
3. Inizializzare lo stato di tutti i *cuRAND state objects* nella device memory con cudaKernelcall.
4. Eseguire il kernel cuda che chiama cuRAND device function.
5. Facoltativamente trasferire i valori random all'host, sempre se è stato preallocato dello spazio per memorizzare i valori nello step 2.

- **Metodo Monte Carlo**

Metodo usato in coppia con le librerie per accelerare la computazione e la risoluzione di problemi in cui la generazione di tanti numeri pseudorandom.
Estraggo valori a caso in un intervallo o punti in un'intorno di un punto nello spazio; controllo se verificano una proprietà e segno con due contatori quanti la verificano e quanti no; in base a quei contatori, faccio rapporti (tra i contatori e altri dati) o proporzioni per avere un'approssimazione del risultato che cercavo.

- **cuFFT**

Si occupa di svolgere la *trasformata di fourier*, la trasformazione che converte in un segnale dal dominio del tempo al dominio delle frequenze, la trasformata inversa è da considerarsi come l'operazione inversa.

Una trasformata di Fourier veloce (Fast Fourier Transform) riceve come input una sequenza di campioni presi da un segnale a intervalli regolari, mentre restituisce come output un insieme di componenti a livello di frequenze, che sovrapposte, creano il segnale generato dai campioni.

I dati supportati vanno dai reali ai complessi, dalla singola precisione alla doppia. I tipi sono identificati da dei flags:
1. C2R: input complesso → output complesso
2. R2C: input reale → output complesso
3. C2R: input complesso → output reale

La configurazione dell'operazione da svolgere è fatta mediante dei plan, gli handle usati dalla libreria, ciascun plan corrisponde ad un singola operazione da svolgere. I pln sono la base da cui derivano le allocazioni e i trasferimenti di memoria, nonchè i lanci dei kernel necessari al completamento dell'operazione.

Operare con cuFFT
1. **Creare** e configurare un **cuFFT plan**
2. Allocare memoria nella device memory con **cudaMalloc** per memorizzare i campioni e le frequenze in output dal calcolo.
3. Popolare la device memory con **cudaMemcpy** con i campioni dei segnali in input
4. Eseguire il plan usando la funzione **cufftExec***
5. Recuperare i risultati dalla memoria del device usando **cudaMemcpy**
6. Rilasciare le risorse CUDA usando **cudaFree** e le risorse cuFFT con **cufftDestroy**

---------- lezione 10

13 - Grafi

Un grafo è una struttura costituita da un insieme di vertici (o nodi) V, e un insieme di lati (edge) E, un lato è composto da due vertici dell'insieme V.
Un grafo *diretto* è un grafo con un verso per ogni edge, quindi ogni coppia di vertici che rappresenta il lato, è ordinata. mentre per un grafo *indiretto* l'ordine dei vertici nella coppia che compone il grafo non è importante.

Un percorso è una sequenza di vertici che corrispondono ai lati da attraversare per raggiungere un nodo specifico. Un percorso è detto semplice quando tutti i vertici sono distinti, mentre un percorso è detto un ciclo quando il nodo di partenza è anche il nodo di destinazione.

Un grafo è *connesso* se esiste un percorso che raggiunge ogni nodo per ogni coppia di vertici dell'insieme. Mentre un grafo è *completamente connesso* se esiste un lato per ogni coppia.
La connessione di un grafo è un parametro utile per la riduzione delle dimensioni da trattare. Un grafo di n nodi la complessità può aumentare fino a n^2 o potenze superiori.
Dividere un grafo originario in due alberi permette di suddividere la computazione in due porzioni ottenendo dei miglioramenti.

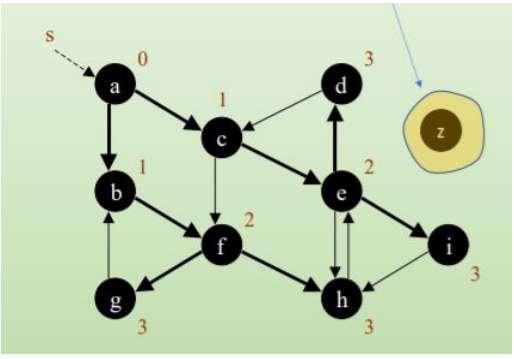
Un *sottografo* è un grafo costituito da un sottoinsieme dei vertici e lati.
Un *albero* è un grafo connesso e aciclico.
Una *foresta* è un insieme di *alberi*.
Un grafo si dice *sparso* se la cardinalità dell'insieme E (dei lati) è minore del quadrato della cardinalità dei vertici. Mentre un grafo è *denso* quando è uguale.

Un grafo è pesato quando ogni lato oltre ad avere due vertici ha anche un valore numerico. Il peso di un grafo è la somma di tutti i pesi dei lati.

Si può rappresentare un grafo attraverso una matrice di adiacenza (una matrice di dimensione |V| x |V|, simmetrica, un 1 nella matrice indica un lato tra i due vertici). Serve per rappresentare i grafi indiretti. Una matrice di incidenza invece è una matrice di dimensione |V|x|V| non simmetrica, in cui ogni 1 rappresenta un lato tra i due vertici.

Si possono usare anche delle liste per rappresentare un grafo, usando una lista linkata, cioè in ogni posizione c'è una lista che corrisponde ai vari nodi con cui ha lato.
Altrimenti un arraylist in cui ogni array punta ad un adjacency list

Ricerca in un grafo
Lo scopo è quello di esplorare i nodi del grafo e scoprire quali di questi nodi destinazione sono raggiungibili da un nodo di partenza, detto sorgente. Bisogna poter calcolare il cammino più breve (shortest path).



BFT: breadth-first tree
Si cerca a livelli nei figli, prima tutti i figli della root, poi tutti i figli di ogni figlio della root e così via.
Viene esplorato in maniera progressiva il grafo.

BFT Si presta molto bene per la parallelizzazione (al contrario del DFS-depth first search).
Per ogni nodo vengono definite delle strutture dati specifiche:

- colore del nodo (bianco: non visitato, grigio: visitato ma non processato, nero: visitato e processato).
- genitore del nodo (nullo se è root o se non si è ancora stati visitati)
- distanza dalla sorgente (inizialmente posta ad infinito, si assume in principio che nessun nodo sia raggiungibile)
- lista di adiacenza (indica i propri figli/nodi con la quale ha un nodo)

L'algoritmo normale viene eseguito in questo modo

1. Seleziona la sorgente e inizializziamo il colore a grigio e distanza 0
2. aggiungiamo la sorgente in una coda FIFO da processare
3. finchè la coda non è vuota, estraiamo un nodo u e
    a. per ogni nodo v nella lista di adiacenza se il suo colore è biaco, lo rendiamo grigio, aggiorniamo il suo genitore ad u e rendiamo la sua distanza dalla sorgente pari alla distanza di u +1, quindi lo aggiungiamo alla coda
4. Considerato come estratto dalla coda, il nodo u diventa nero

CUDA BFS
L'implementazione assume che una volta attraversato un livello, questo non venga più visitato, la frontiera di BFS corrisponde a tutti i nodi che vengono processati in un livello. Per evitare di mantenere una coda per ogni vertice, l'implementazione assegna un thread ad ogni vertice e mantiene due array di booleani, F_a per la frontiera e X_a per i nodi visitati.
L'array di vertici C_a memorizza il costo del cammino minimo per ciascun vertice data una sorgente s.

1. calcolato il thread id corrispondente al nodo a cui è assegnato, si verifica che esso sia appartenente alla frontiera, indicizzando l'array di booleani; se non lo è si ferma nella computazione.
2. Se appartiene alla frontiera, il suo valore viene passato a false, ma viene messo a true il suo esser visitato
3. Ciascuno dei nodi vicini al nodo indicizzato dal thread è memorizzato per indice in una struttura apposita (neighs); per ciascuno degli indici dei vicini (nid) del nodo in esame, se esso non è tra i visitati si aggiorna il costo del cammino, se esso non è tra i visitati, si aggiorna il costo del cammino fino ($C_a[nid] = C_a[tid] + 1$) e si pone nella frontiera ($F_a[nid] = true$).

Il kernel verrà chiamato finchè la frontiera non sarà nulla.

tempo di esecuzione CUDA BFS:

- inizializzazione O(V),
- accodamento O(V), considerando che ogni vertice è rimosso e aggiunto O(1)
- il processing O(E)

Algorithm 2. CUDA_BFS_KERNEL (V_a, E_a, F_a, X_a, C_a)

```
1: tid ← getThreadID
2: if F_a[tid] then
3:     F_a[tid] ← false, X_a[tid] ← true
4:     for all neighbors nid of tid do
5:         if NOT X_a[nid] then
6:             C_a[nid] ← C_a[tid]+1
7:             F_a[nid] ← true
8:         end if
9:     end for
10: end if
```

Problema del graph coloring o clustering di task indipendenti. Il problema del coloring può essere visto come un problema di individuazione di task indipendenti massimali.
Colorare i nodi in maniera tale che questi non siano adiacenti tra loro colori uguali.

Algoritmo sequenziale:

```
n = |V|
Choose a random permutation p(1),...,p(n) of numbers 1,...,n
U := V
for i = 1 to n do in sequence
    v := p(i)
    S := {colors of all colored neighbors of v}
    c(v) := smallest color not in S
    U := U − {v}
end do
```

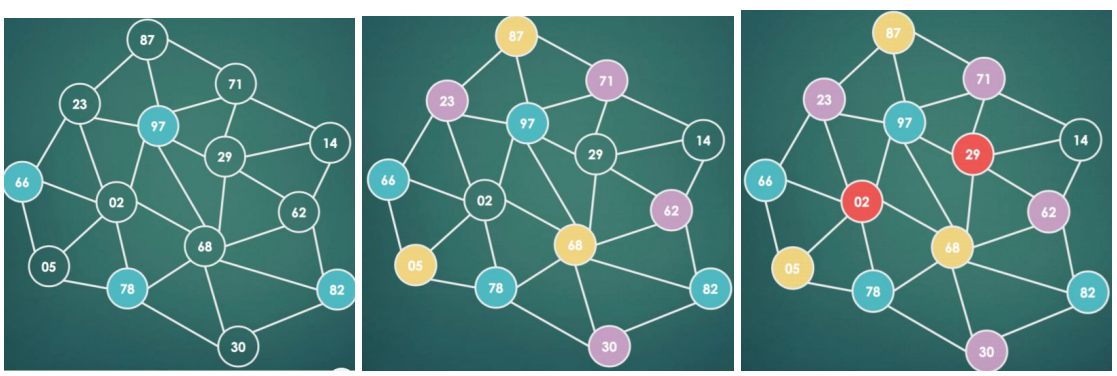
Algoritmo parallelo:

```
U := V
while ( |U| > 0 ) do in parallel
    Choose an independent set I from U
    Color all vertices in I
    U := U − I
end do
```

Jones-Plassmann Coloring

```
U := V
while ( |U| > 0 ) do
    for all vertices v ∈ U do in parallel
        I := {v such that w(v) > w(u) ∀ neighbors u ∈ U}
        for all vertices v' ∈ I do in parallel
            S := {colors of all neighbors of v'}
            c(v') := minimum color not in S
        end do
    end do
    U := U − I
end do
```

1. Assegna un numero a caso a tutti i vertici
2. in parallelo si prende ogni vertice, questo confronta se stesso con i suoi vicini, se è il più alto, si mette nell'insieme I.
3. In parallelo er ogni vertice presente nell'insieme I, si controllano i colori dei nodi vicini, si pongono in un insieme S, il colore non presente in S sarà il colore scelto.
4. Colora con il colore scelto e itera sui nodi non ancora colorati



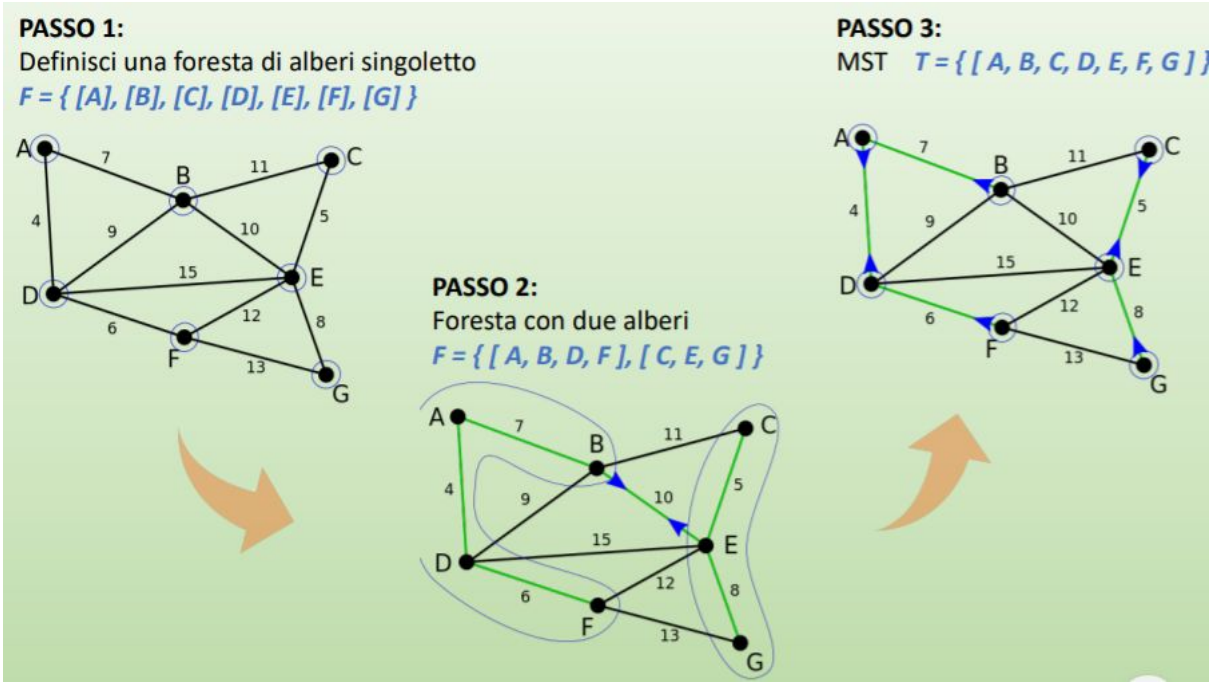
Minimum Spanning Tree (**MST**)
Da un grafo, bisogna generarne uno che ha il peso minimo connettendo tutti i nodi.
Sono basati su un processo di colorazione che mantiene un opportuno invariante.

Il problema, dato un grafo orientato e pesato G(V,E,w), si vuole determinare un albero di costo minimo tra tutti i possibili alberi generabili .

Algoritmo Boruvka
Input: un grafo G(V,E)

1. inizializza la foresta in modo che ogni foresta abbia un solo nodo.
2. finchè la dimensione della foresta |F|>1 (quindi ci sono nodi liberi)
    a. per ogni albero della foresta:
        i. crea un insieme di lati (ST) e inizializzalo
        ii. per ogni nodo dell'albero:
            - trova il lato con il peso minimo, tra v e u, con u non appartenente all'albero che si sta creando. Quindi considero tutti i lati del nodo e prendo solo quello più leggero.
    b. aggrega gli alberi in F connessi da un lato in ST
    c. quando tutti i nodi sono connessi, ma ci sono ancora più alberi nella foresta, i due alberi si collegano con il lato dal peso inferiore

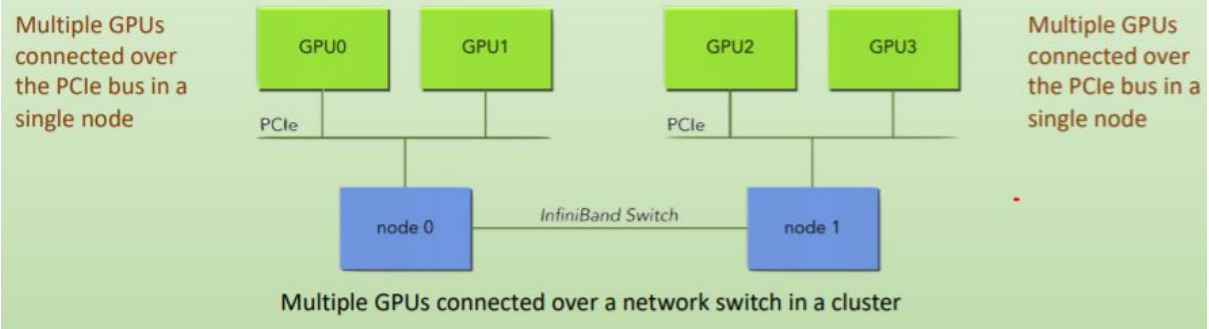
Output: un albero T che corrisponde al MST



infine ci sono anche gli algoritmi greedy: Algoritmo Kruskal Algoritmo Prim

13.2 - Multi-GPU programming

Ulteriore livello di parallelismo, permette di *gestire ed eseguire* kernel su GPU multiple interconnesse, *sovrapporre* computazioni e comunicazioni sulle diverse GPU, *sincronizzare* le esecuzioni sulle varie GPU usando stream ed eventi, *scalare* le applicazioni su cluster da GPU, *importante*, progettare correttamente la comunicazione inter-GPU.



Ogni thread che esegue sull'host può accedere ai diversi device

Determinare il numero di device utilizzabili:

```
cudaError_t cudaGetDeviceCount(int *count);
```

Determinare le proprietà del device, come capability o informazioni sull'hardware

```
cudaError_t cudaGetDeviceProperties(struct cudaDeviceProp *prop, int device)
```

Si può fissare quale GPU è destinata ad eseguire le operazioni dalla GPU, quindi imposta il device corrente con

```
cudaError_t cudaSetDevice(int id)
```

```
cudaSetDevice( 0 );
kernel<<<...>>>(...);
cudaMemcpyAsync(...);
cudaSetDevice( 1 );
kernel<<<...>>>(...);
```

La GPU può essere cambiata anche quando le chiamate async sono in esecuzione (kernels memcopies). è possibile accodare un gruppo di chiamate async a una GPU e poi passare ad un'altra GPU. Il codice viene eseguito concorrentemente su entrambe le GPU, anche stream ed eventi creati dalla stessa CPU thread vengono eseguiti su quel device.

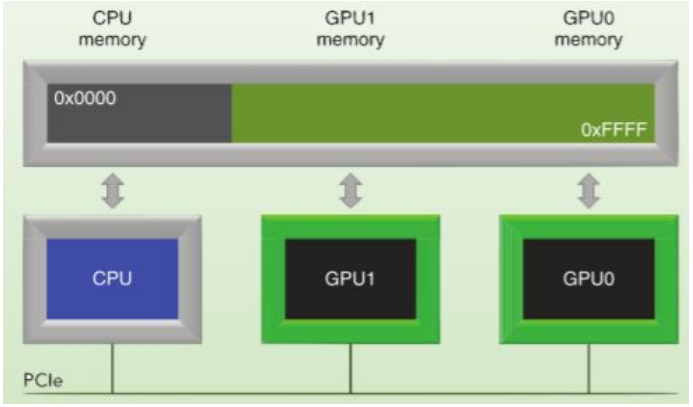
Una volta determinato il device corrente, tutte le operazioni CUDA saranno indirizzate a quel device, quindi le allocazioni di memoria dall'host saranno fisicamente sul device, definizione degli stream e di eventi o lancio del kernel, saranno eseguite sul device definito.

I kernel che eseguono applicazioni a 64bit su device con compute capability di almeno 2.0 possono accedere alla global memory di ogni GPU connessa sullo stesso nodo PCIe.
Da CUDA API 4.0 cn la comunicazione p2p è possibile instaurare comunicazioni inter-device.

- Accesso p2p: consente alle GPU connesse allo stesso nodo PIC di referenziare dati memorizzati su device memory di altre GPU (trasparente al kernel, i dati vengono trasferiti tramite PCI al thread richiedente)
- Trasferimento p2p: consente di copiare i dati direttamente tra GPU (se sullo stesso nodo) altrimenti la copia passa attraverso la memoria host (caso di GPU appartenenti a nodi diversi del cluster)

L'allocazione di dati tra CPU e GPU usa **UVA** (unified virtual address space)

- Ogni CPU e GPU gestisce il suo proprio spazio di indirizzamento
- I driver si occupano di determinare gli indirizzi dove risiedono i dati
- Una certa allocazione risiede solo su un device (un array non può essere ripartito)



Per abilitare e disabilitare il p2p:
Restituisce 1 in canAccessPeer se il device è in grado di accedere alla global memory di peerDevice.

```
cudaError_t cudaDeviceCanAccessPeer(int*canAccessPeer, int device, int peerDevice)
```

Abilitare l'accesso p2p deve essere un'operazione esplicitata ed unidirezionale. Si abilita l'accesso p2p dal device corrente al device peerDevice

```
cudaError_t cudaDeviceEnablePeerAccess(int peerDevice, uint flag)
```

P2P rimane abilitato finchè non viene esplicitamente disabilitato (solo per 64bit)

```
cudaError_t cudaDeviceDisaablePeerAccess(int peerDevice)
```

Si possono usare le stesse funzioni di sincronizzazione basate su stream ed eventi in applicazioni multi-GPU.
Workflow

1. **Seleziona l'insieme di GPU** che l'applicazione userà
2. **Crea stream ed eventi** per ogni device
3. **Alloca le risorse** su ogni device come la device memory
4. **Lancia i task** su ogni GPU attraverso gli stream
5. **Usa gli stream ed eventi** per interrogare o sincronizzare il completamento dei task
6. **Libera** le **risorse** per tutti i device

Combinare p2p e UVA si hanno accessi trasparenti su qualsiasi device, se si usano troppo intensamente gli accessi p2p in combinazione con UVA, ci può essere un deterioramento delle prestazioni a causa dei piccoli trasferimenti attraverso il bus PCIe.

Se p2p e UVA sono abilitati entrambi, l'esecuzione di kernel su un device può deferenziare un puntatore a un buffer di memoria in un altro device.
Non si devono gestire separatamente i buffer di memoria o effettuare copie esplicite dalla memoria host, il sistema lo rende evitando di esplicitare le operazioni e semplificando il codice.

---------- Domande

- Differenze SIMD SIMT piú snippet codice esempio di comportamento SIMD su GPU

SIMD - Single Instruction Multiple data dalla Tassonomia di Flynn.
Si utilizzano molte unità computazionali che eseguono in parallelo la stessa istruzione, ma utilizzando dati differenti, quindi le unità accedono ad una memoria condivisa e recuperano i dati necessari alla computazione. Si usa nella applicazioni ad alto grado di parallelismo.

SIMT - Single Instruction Multiple Thread. Esclusa dalla tassonomia di Flynn
Usato per il parallel computing dove si combina il SIMD con il multithreading. Vengono condivise le risorse ma con esecuzioni indipendenti (thread). Ogni thread ha diverse caratteristiche, il proprio instruction address counter, il proprio register state e un execution path indipendente.

- Perchè il warp è fondamentale

Il warp è l'unità di computazione per ogni SM (streaming multiprocessor), infatti ogni SM contiene 32 core. Il Warp è l'insieme di thread che eseguono la stessa istruzione nello stesso momento. Se avessi 36 thread, verrebbero eseguiti in maniera meno ottimizzata perchè userei un warp da 32 e un warp da 32 usandone solo 4 di core.
Ogni warp ha un contesto in esecuzione a runtime, mantenuto on-chip, ha program counter e registri a 32 bit ripartiti tra i thread (ogni thread eseguito sul proprio cuda core)

- Sincronizzazione device level e block level, e quando ci puó essere deadlock [quasi letterale]

La sincronizzazione a livello di blocco consiste nell'attendere che tutti i thread di un blocco abbiano terminato la loro esecuzione. La sincronizzazione a livello di device consiste nell'attendere che tutti i task in esecuzione sui diversi blocchi siano terminati.
Il deadlock si può verificare in un caso in cui in un if (quindi già c'è divergenza) chiedo di bloccarmi in attesa dei task (con il comando _synchthreads), e nell'else faccio la stessa cosa. Dato che viene eseguito il primo ramo, al termine dell'esecuzione viene chiesto di bloccarsi per sincronizzarsi. Il secondo ramo non verrà eseguito e non raggiungerà mai lo stato di completamento del task. Bisogna evitare di mettere i sistemi di sincronizzazione negli if/else.

- Cos'è uno stream e che livello di concorrenza introduce il loro uso

Uno stream è una sequenza di comandi che vengono eseguiti in ordine. Diversi stream sono tra loro indipendenti e possono eseguire i comandi in maniera concorrente. Uno stream incapsula le operazioni, ne mantiene l'ordine fifo e permette di controllarne lo status.
Ci sono i NULL-stream (stream default) e i NON-NULL-stream.
Viene introdotto il livello di concorrenza tra stream. Precisamente tra Null-stream e NON-NULL-stream. Infatti due NULL-stream vengono eseguiti sequenzialmente, perchè bloccanti tra loro. Un NULL-stream e un NON-NULL-stream vengono eseguiti nuovamente sequenzialmente, per il NULL-stream è bloccante. Dichiarare il NON-NULL-stream non bloccante con `cudaStreamCreateWithFlags(stream, cudaStreamNonBlocking)`, permette di far eseguire parallelamente i due stream. Due NON-NULL-stream invece verranno eseguiti sempre parallelamente.

- Esempio di stream usage

- Usi sw della shared memory e descrizione architetturale [quasi letterale]

La shared memory è una memoria on-chip, quindi ad alta banda e minore latenza rispetto alla local o global memory. La shared memory è la memoria che permette la comunicazione tra i thread di uno stesso blocco, è richiesta una sincronizzazione tra i thread con _synchthread prima di accedere. Il miglior pattern di accesso è quando ogni thread accede con una transizione ad un bank distinto dagli altri thread. Un accesso allineato (transizione multiplo pari della granularità della cache (32 per L2 e 128 per L1) e coalescente (tutti i 32 warp accedono ad un blocco contiguo) l'accesso è efficiente al 100%. Le variabili devono avere qualificatore __shared__ per essere memorizzate direttamente in shared memory.ù

- Cos'è la memoria pinned e come funziona?

La memoria pinned si intende memoria dell'host che non subisce lo swapping della memoria da parte del sistema operativo. Infatti la porzione di memoria definita pinned memory o page lock, permette di fare un trasferimento sicuro sul device senza rischiare page fault.

- Come è fatta una merging network per ordinare sequenze?

sorting network:
Si basa sui comparatori. Si usano due operazioni
+: per creare una sequenza bitonica crescente
-: per creare una sequenza bitonica decrescente.
Una sequenza di due numeri è sempre bitonica. Quando si hanno n numeri da ordinare, si deve creare una sequenza bitonica da ordinare.

La sequenza bitonica si crea applicando alle coppie le funzioni + e - in maniera alternata. Si prende la prima sequenza lunga 2, si applica +, si prende la seconda sequenza lunga due e si applica - fino a terminare il numero di oggetti da ordinare.

Al secondo passaggio si crea una sequenza lunga 4 (il quadrato di 2) e si applica la funzione +, poi alla successiva sequenza lunga 4 si applica la funzione -. Si continua ad aumentare la dimensione delle sequenze a cui applicare le funzioni in modo alternato, la dimensione è data dalla potenza del passaggio(es passaggio 3, quindi il cubo di 2). In log_2 n si ottiene una sequenza bitonica.

Per ordinare una sequenza bitonica di dimensione 2^k si usa la funzione di comparazione +:
Al passo 1 si prendono i valori in posizione j e si comparano con il valore in posizione j+2^(k-1), con la funzione di comparazione +, quindi se il primo valore è maggiore del secondo, vengono scambiati di posto.

Al passo 2 si prendono i valori in posizione j e si comparano con il valore in posizione j+2^(k-2)
.... così fino a raggiungere una comparazione all'interno delle sequenze di due cifre.

- Come fare una prefix sum/parallel sum parallela work-efficient

Parallel sum prefix
Assumendo che il numero di valori è una potenza di 2, cioè $n = 2^k$
Al passo 1, viene fatta la somma per tutti i thread con idx > 0 $\rightarrow somma_i = x_{i-1} + x_i$
Al passo 2, per tutti i nodi con idx > 1 $\rightarrow somma_i = x_{i-3} + \ldots + x_i$
Al passo 3, per tutti i nodi con idx > 2 $\rightarrow somma_i = x_{i-5} + \ldots + x_i$
…

Parallel sum prefix efficient
Step 1: si aggiornano solo gli elementi dispari (quindi somma1 = 0+1, somma3 = 2+3…)
Step 2: si aggiornano solo gli elementi con indice $4 \cdot n - 1$ (3, 7, 11, 15)
(somma3 = somma1+somma3, somma7 = somma5+somma7…)
Step 3: si aggiornano solo gli elementi con indice $8 \cdot$

- Algoritmo di Boruvka per MST [quasi letterale]

L'algoritmo serve per creare da un grafo un MST-minimum spanning tree, cioè un grafo composto da archi di peso minimo che collegano tutti i nodi.

Si passa in input un grafo G.

Si prendono tutti i nodi e questi vengono trasformati in alberi composti da un nodo solo, la foresta è di dimensione uguale alla cardinalità dell'insieme dei vertici.

Per ogni albero si inizializza la struttura dati che memorizza i lati, per ogni nodo dell'albero si cerca un lato con il peso minimo tra u e v, con v che non appartiene all'albero che si sta considerando. Quindi si considera tra tutti i lati, il lato dal peso minimo.

Si aggregano in base alla struttura dati che raccoglie i lati. L'output è un albero T che corrisponde all'albero MST del grafo G messo in input.

- Passi da fare per l'uso delle librerie CUDA

Gestione delle librerie:
1- creare un handle specifico della libreria per ogni libreria che si vuole usare, serve per mantenere il contesto e gestire i dati
2- allocare device memory per input e output delle funzioni, in caso convertire gli input per renderli idonei al formato della libreria
3- popolare con i dati di input
4- configurare le computazioni per l'esecuzione
5- eseguire la chiamata
6- recuperare i dati dal device memory
7- se necessario convertire i dati nel formato necessario
8- rilasciare le risorse

- Mostrare come si possono usare piú GPU per distribuire computazione (es. prodotto di vettori) su device connessi allo stesso nodo PCIe [quasi letterale]

1. Selezionare l'insieme di GPU che verrà usata dall'applicazione
2. Creare stream ed eventi per ogni device
3. Allocare le risorse su ogni device, tipo la device memory
4. Lanciare i task su ogni GPU attraverso gli stream
5. Usare gli stream ed eventi per interrogare o sincronizzare i task
6. Liberare tutte le risorse

Utilizzare la comunicazione p2p e l'UVA tra i device