

GPU Computing

Laurea Magistrale in Informatica - AA 2019/20

Docente **G. Grossi**

Lezione 1 – Introduzione al corso

Sommario

- Dalla CPU alla GPU
- Motivazioni
- Esempi di applicazioni
- Che cos'è una GPU
- Architetture parallele / CUDA
- Argomenti del corso
- Primo programma CUDA

Dalla tradizione (la CPU)...

Central Processing Unit

- Fonte primaria per il calcolo
- Potente a suff. per applicazioni general purpose
- Potenza da sempre guidata dalla legge di Moore
- Tecnologia consolidata
- Limiti...



•••

Aspetti critici

- ✓ Limite all'aumento del clock, passato in 30 anni da **1 MHz** a **1 – 4 GHz** (1000 volte più veloce)
- ✓ La potenza dinamica dissipata cresce col **cubo della frequenza di clock** (power wall) inducendo seri problemi di dissipazione del calore :

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

con **alpha** la prob. di switch, **C_L** capacità, **V** tensione (lineare in **f**) e **f** frequenza di clock

- ✓ Legge di Moore: in anni recenti ha subito una battuta d'arresto con l'impossibilità di ulteriore aumento dell'integrazione
- ✓ Multi-core per sopperire al declino del singolo core!
- ✓ Inoltre...

**Limite intrinseco
dimensionale alla freq.
di clock:**

$$c \approx 3 \cdot 10^5 \text{ km/sec} = 3 \cdot 10^9 \text{ dm/sec (vel. luce)}$$

$$f = 4 \text{ GHz} = 4 \cdot 10^9 \text{ sec}^{-1} \text{ (freq. clock)}$$

$$s = c \cdot t = c \cdot f^{-1} = 3 \cdot 0.25 = 0.75 \text{ dm (spazio percorso!)}$$

...all'efficienza (la GPU)

Graphics Processing Unit

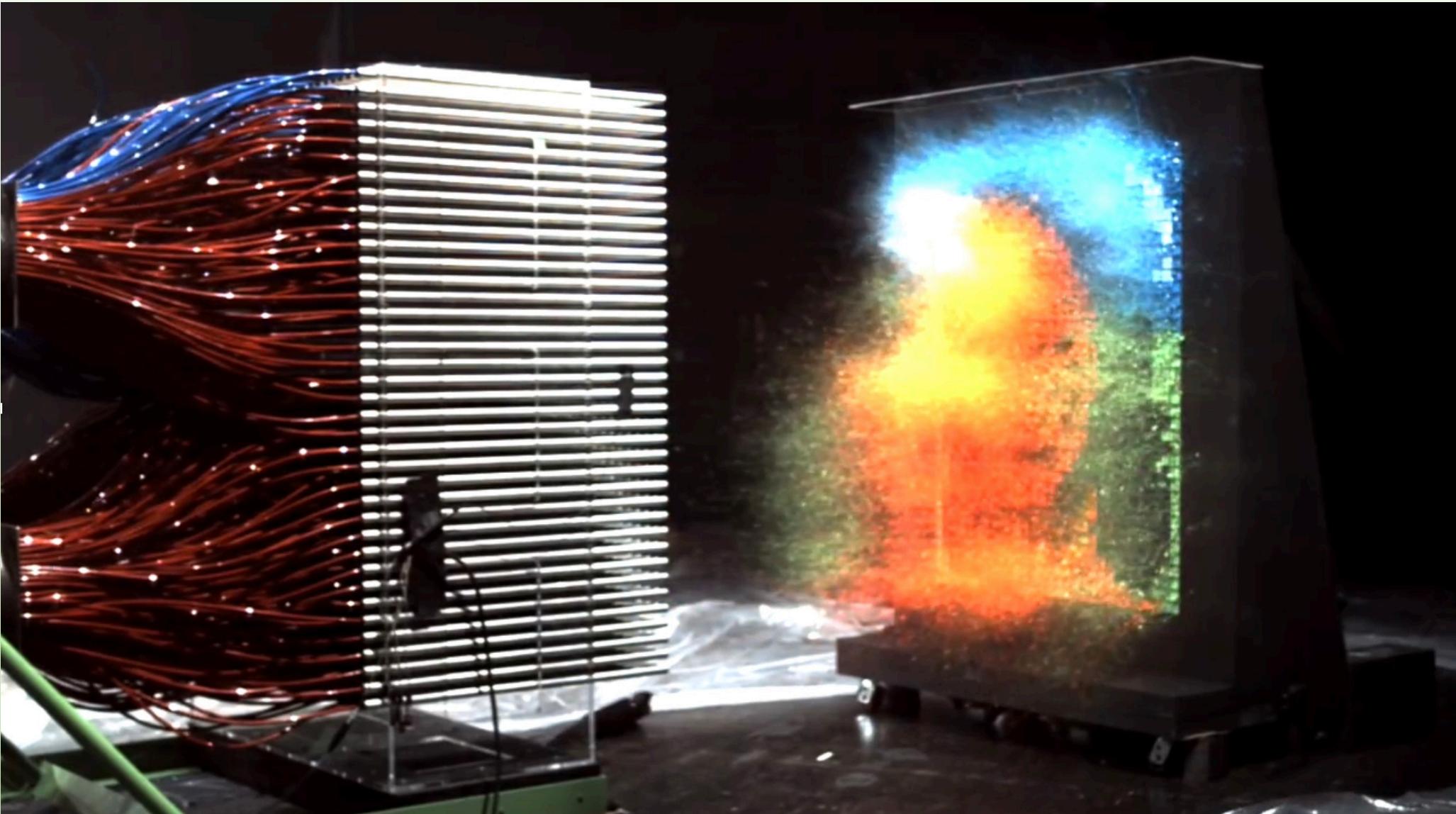
- Nasce per la grafica!
- Molto più potente per “applicazioni grafiche” rispetto alla CPU
- Tecnologia in continua evoluzione



Aspetti di interesse

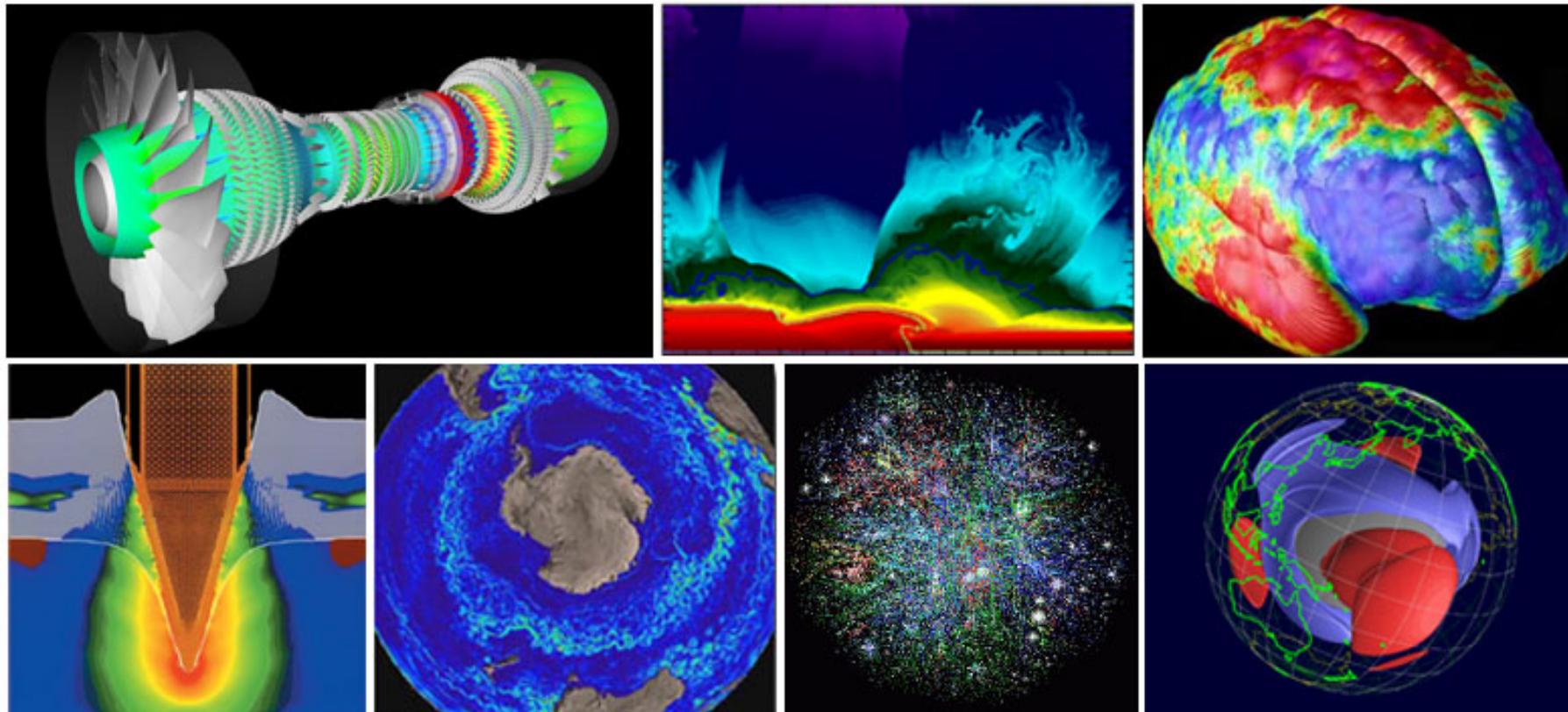
- Migliaia di core vs 2,4,8 della CPU
- Utile solo per problemi altamente parallelizzabili (velocità 10x, 100x ...)
- Introduce il paradigma GP-GPU (General Purpose – GPU)

CPU vs GPU demo!



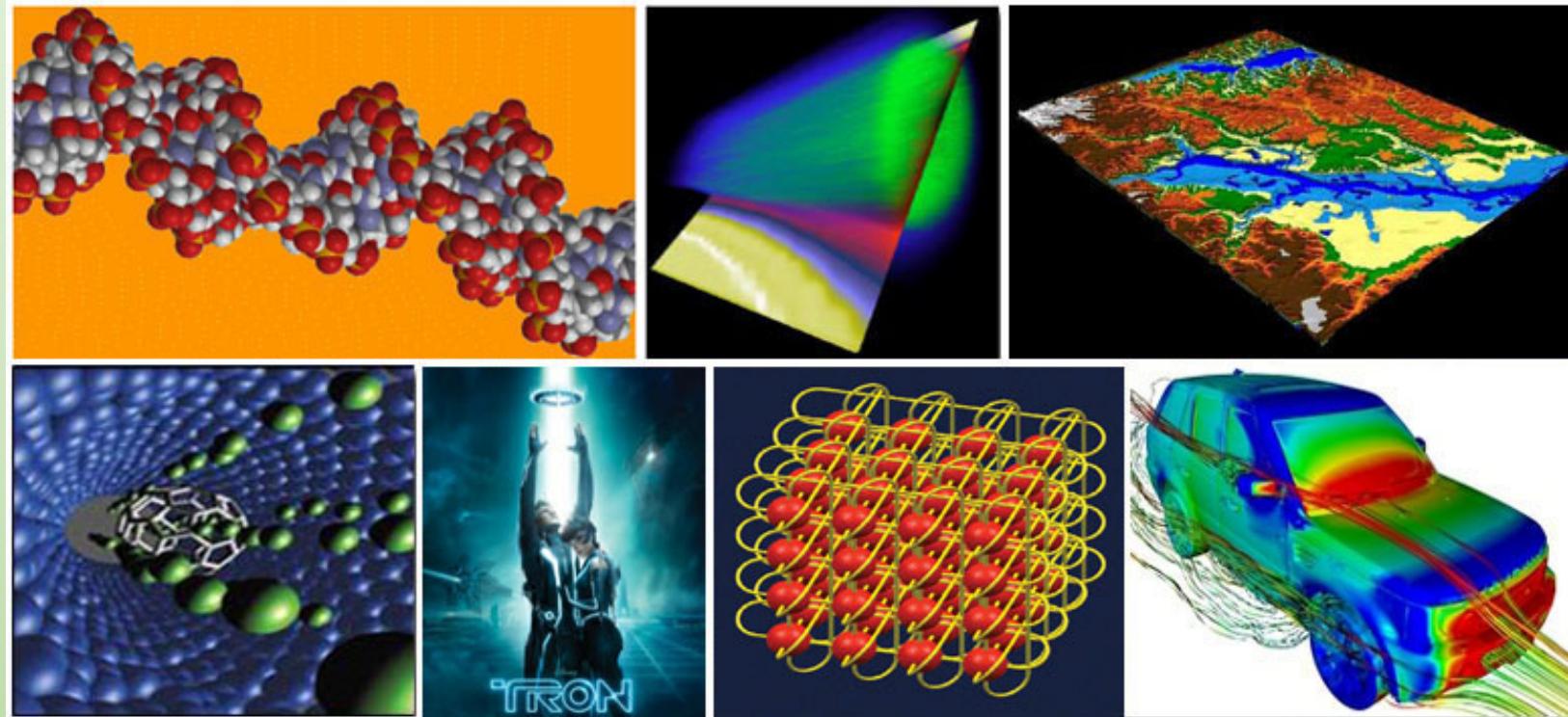
Perché parallel computing?

- ✓ Atmosphere, Earth, Environment
- ✓ Physics - applied, nuclear, particle, condensed matter, high pressure, fusion, photonics
- ✓ Bioscience, Biotechnology, Genetics
- ✓ Chemistry, Molecular Sciences
- ✓ Geology, Seismology
- ✓ Mechanical Engineering - from prosthetics to spacecraft
- ✓ Electrical Engineering, Circuit Design, Microelectronics
- ✓ Computer Science, Mathematics
- ✓ Defense, Weapons

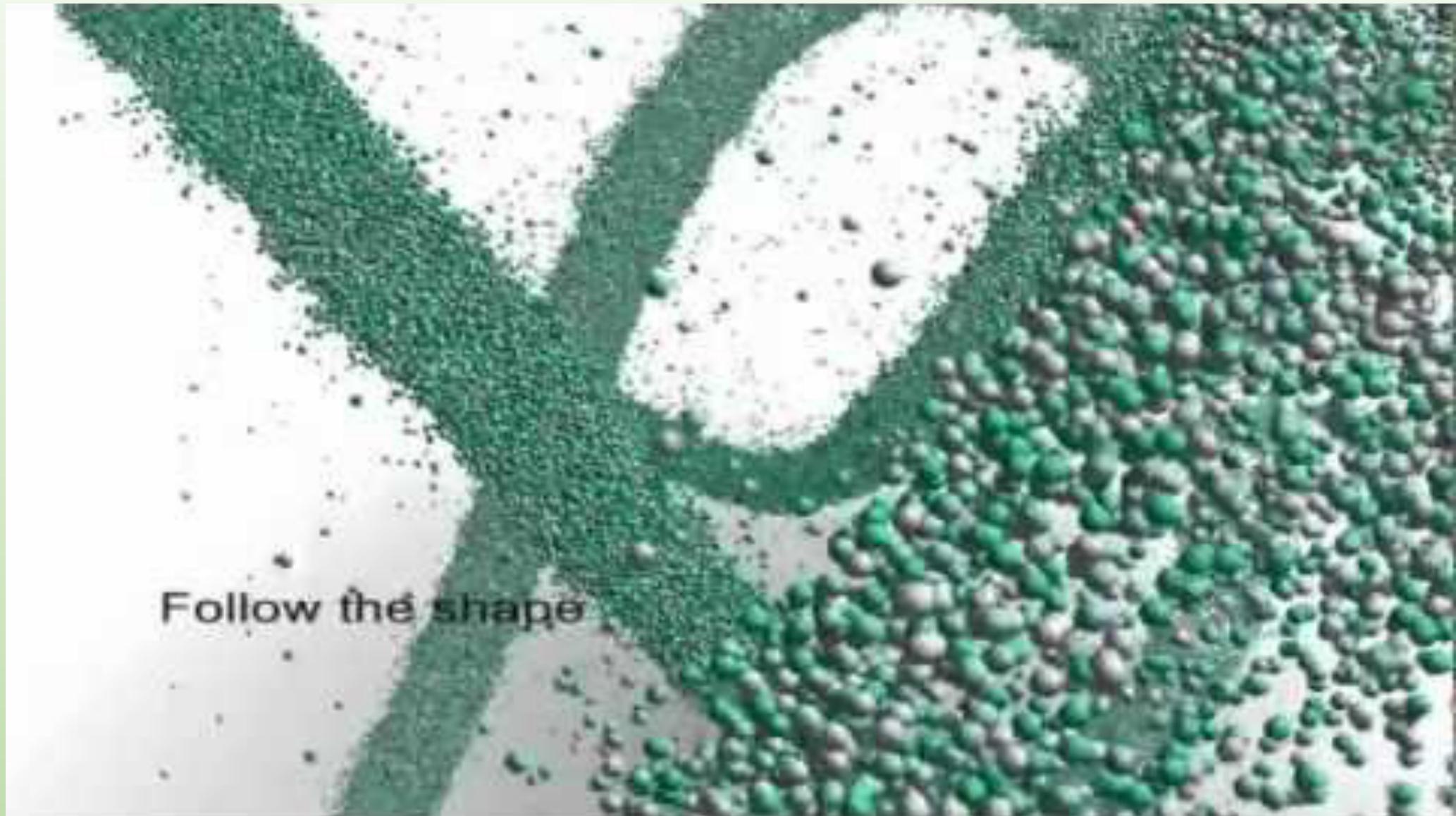


App. industriali e commerciali

- ✓ "Big Data", databases, data mining
- ✓ Oil exploration
- ✓ Web search engines, web based business services
- ✓ Medical imaging and diagnosis
- ✓ Pharmaceutical design
- ✓ Financial and economic modeling
- ✓ Management of national and multi-national corporations
- ✓ Advanced graphics and virtual reality, particularly in the entertainment industry
- ✓ Networked video and multi-media technologies
- ✓ Collaborative work environments



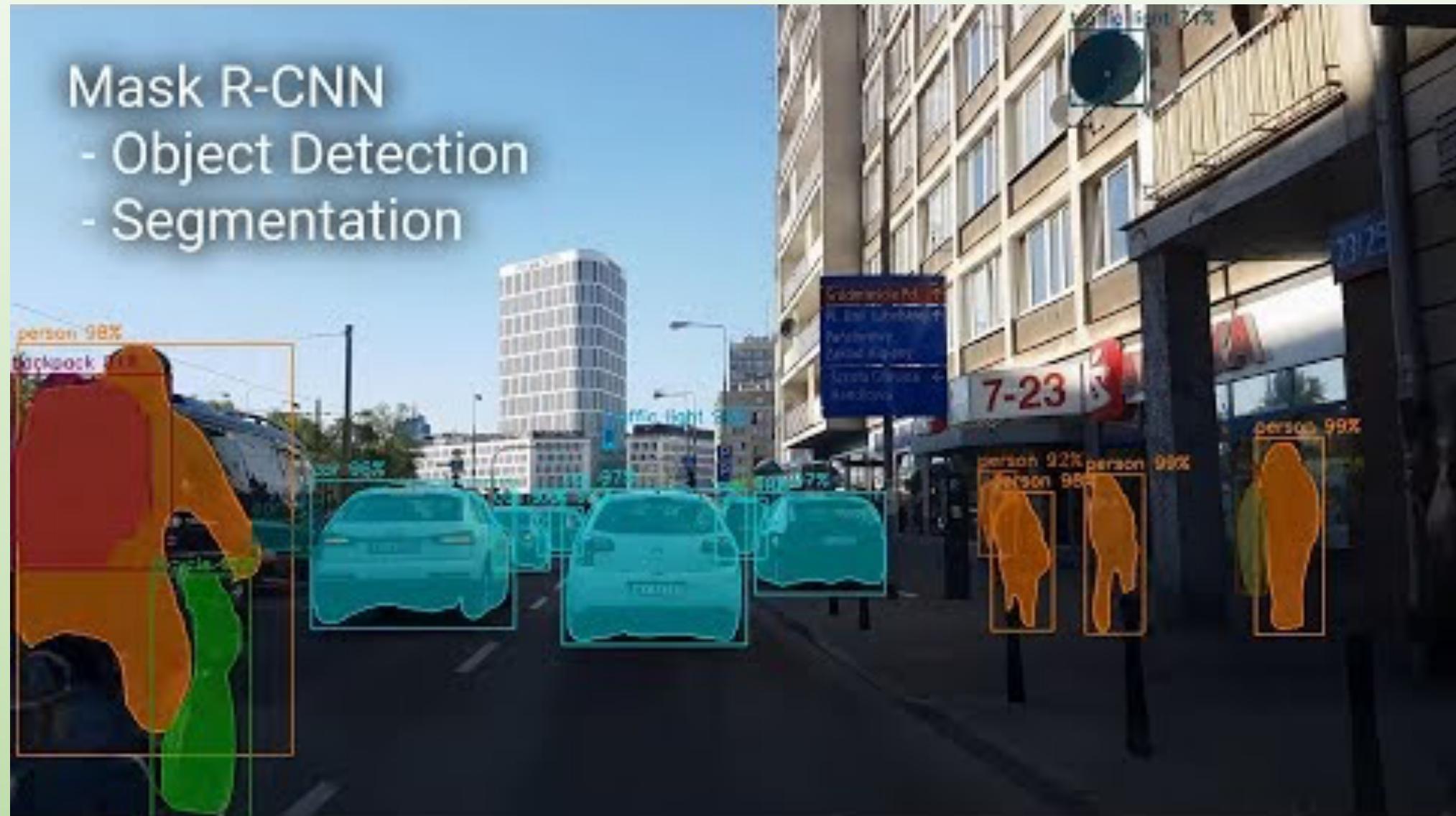
Massive Simulation of Distributed Behavioral Models



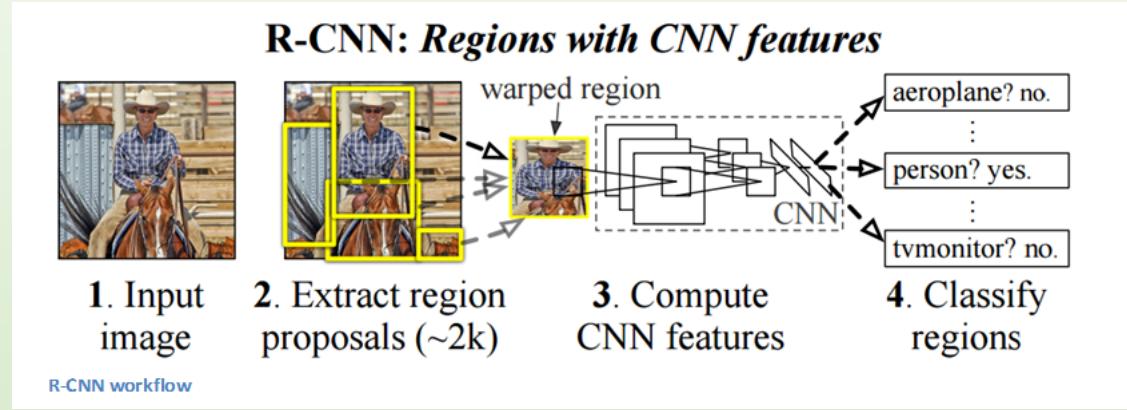
Real-time ray tracing



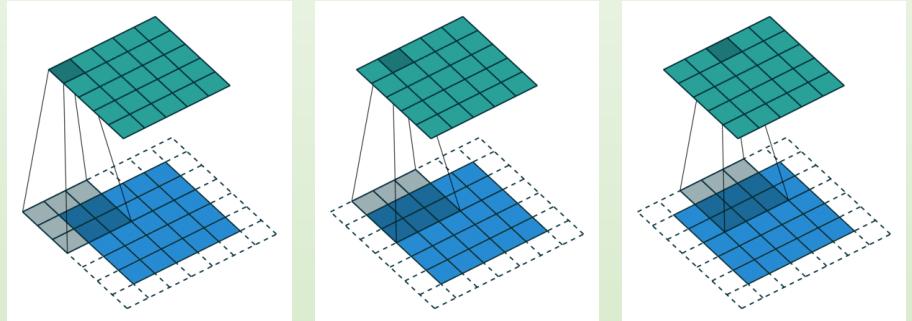
Deep Learning: Object detection and segmentation



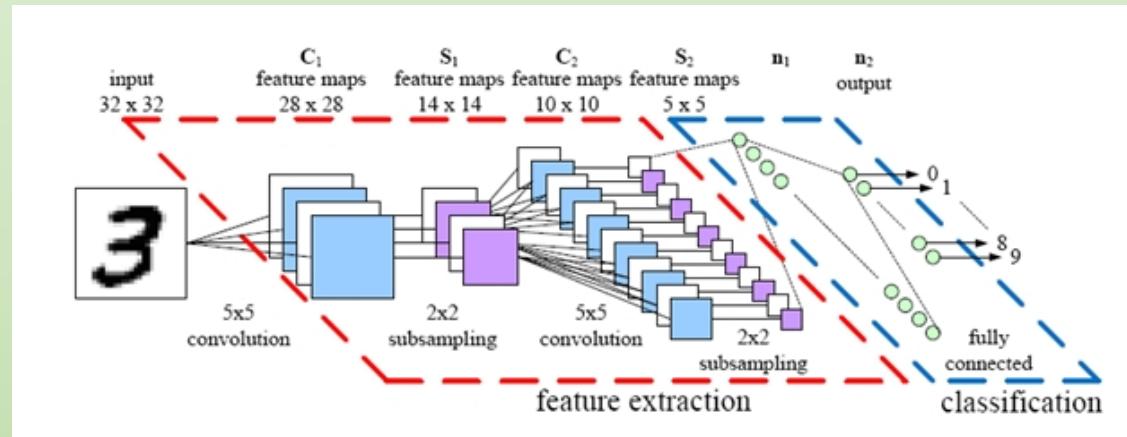
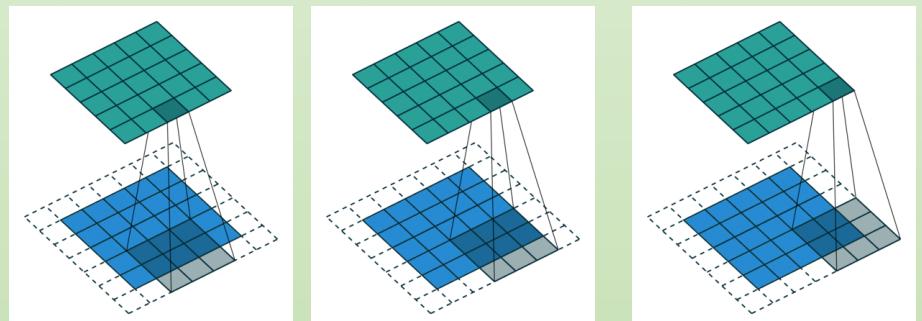
Convolutional Neural Networks (CNN)



convoluzione: prodotto ripetuto molte volte!



...



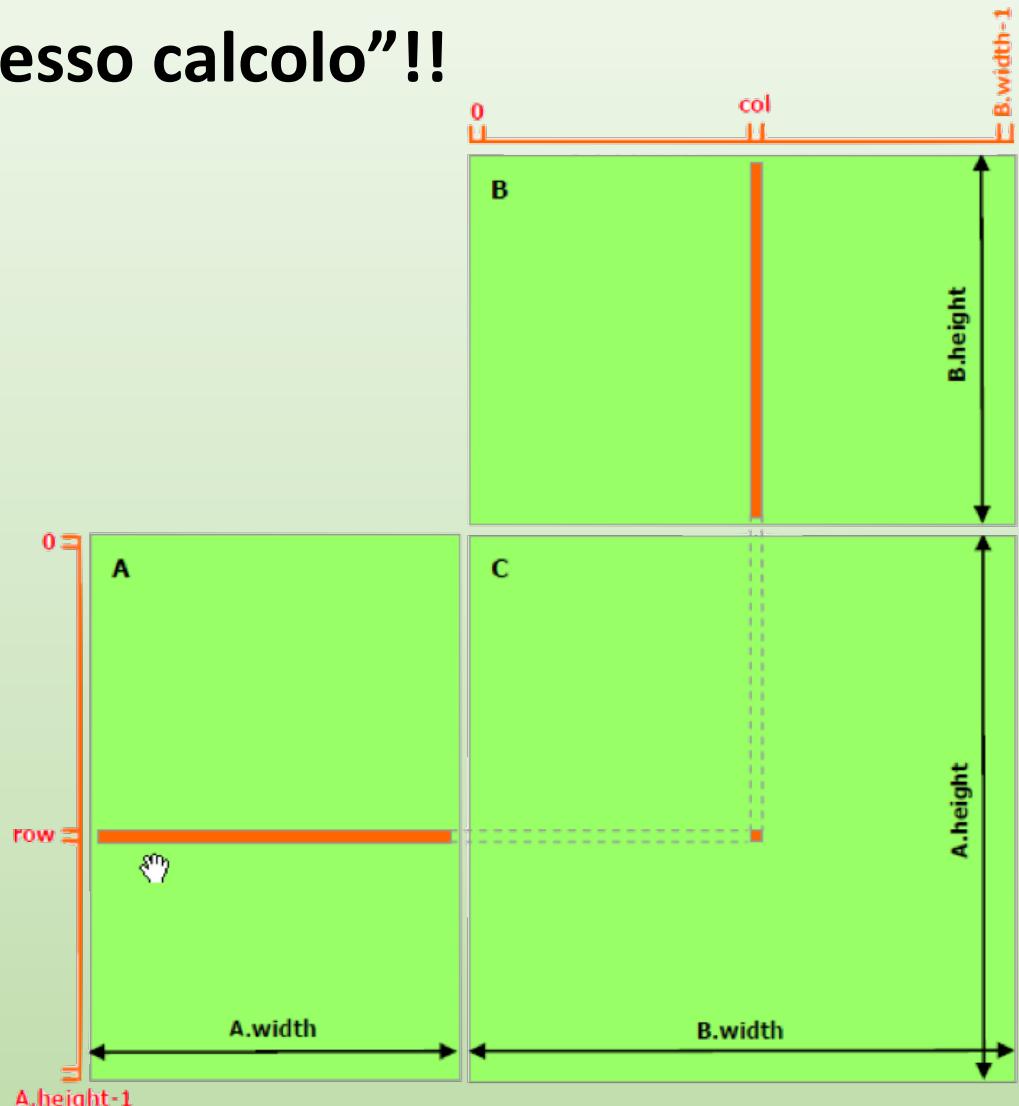
GPUs - Motivazioni

Grandi porzioni di calcoli sono lo “stesso calcolo”!!

Prodotto interno tra vettori riga e colonna:

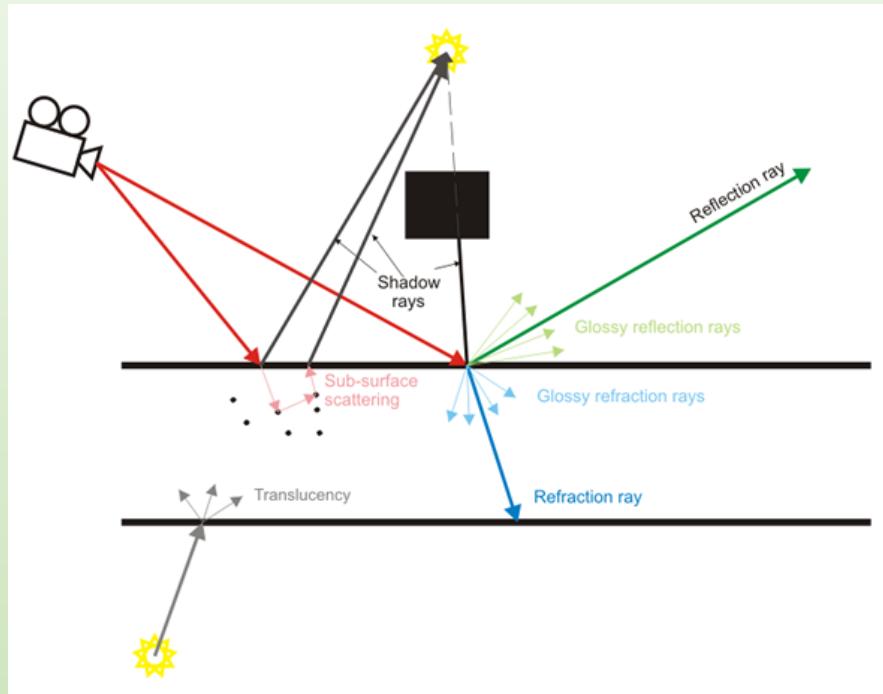
```
for all entry C(i,j):  
    for all k = 1:num_col(A)  
        calculate s = s + A(i,k)*B(k,j)  
    store s in C(i,j)
```

Complessità del prodotto matriciale: $\mathcal{O}(n^3)$



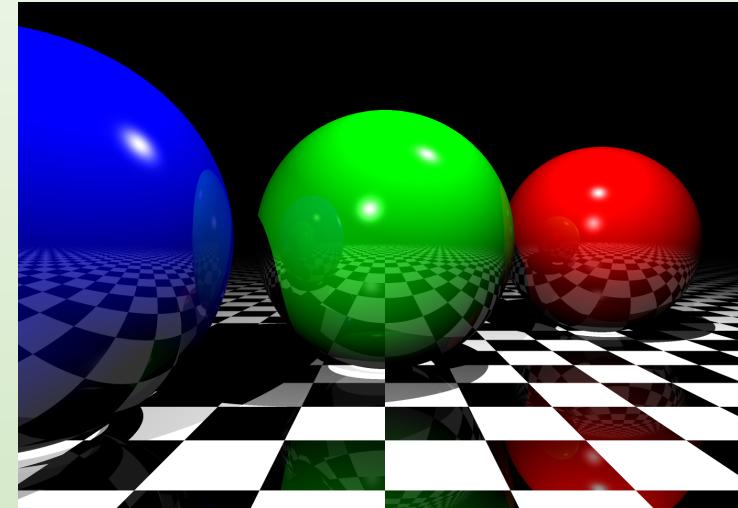
GPUs - Motivazioni

Grandi porzioni di calcoli sono lo “stesso calcolo”!!



RayTracing

```
for all pixels (i,j):  
    Calculate ray point and direction in 3d space  
    if ray intersects object:  
        calculate lighting at closest object  
        store color of (i,j)
```

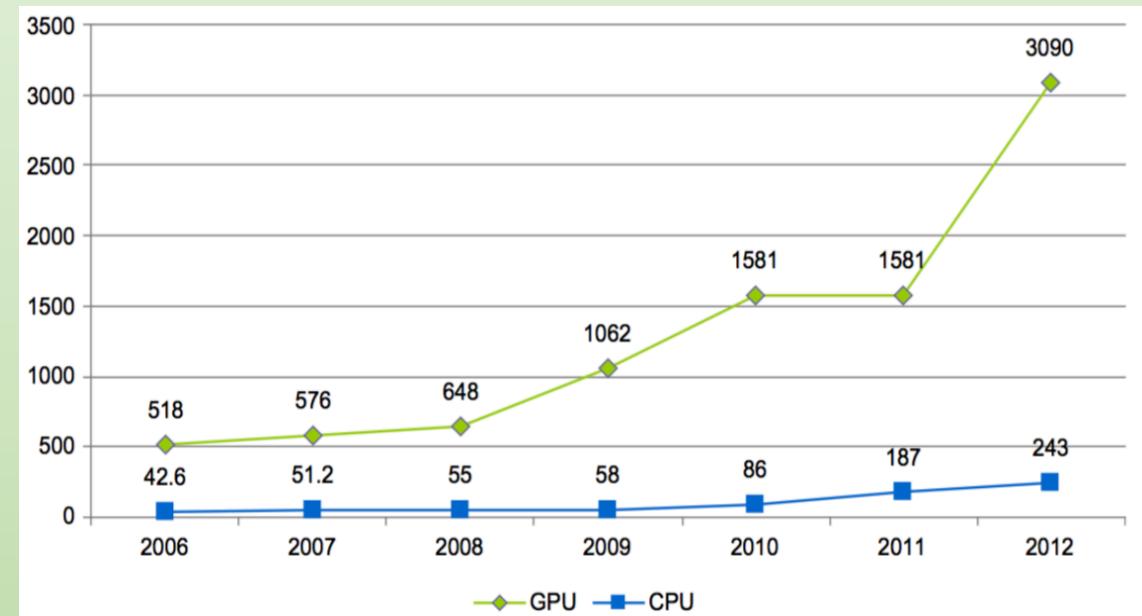


Un raggio colpisce una superficie, genera fino a tre nuovi tipi di raggi:
riflessione, rifrazione ed ombra

- Un raggio riflesso continua nella direzione della riflessione a specchio su di una superficie lucida
- Interagisce con altri oggetti della scena: il primo oggetto che colpisce sarà quello visto nel riflesso presente sull'oggetto originario.
- Il raggio rifratto viaggia attraverso il materiale trasparente in modo simile, con l'aggiunta che può entrare o uscire da un materiale

GPU ieri

- ✓ Nel 2007, NVIDIA ha visto la possibilità di introdurre le GPU in questo mainstream della programmazione general purpose, definendo una easy-to-use programming interface
- ✓ La divergenza in termini di potenza computazionale tra CPU e GPU si è iniziata a vedere quando, nel 2009, le GPU hanno infranto la barriera dei 1000 GFLOPS o 1 TFLOPS
- ✓ CUDA (Compute Unified Device Architecture)
 - General-purpose parallel computing platform for NVIDIA GPUs
- ✓ OpenCL (Open Computing Language)
 - General heterogenous computing framework



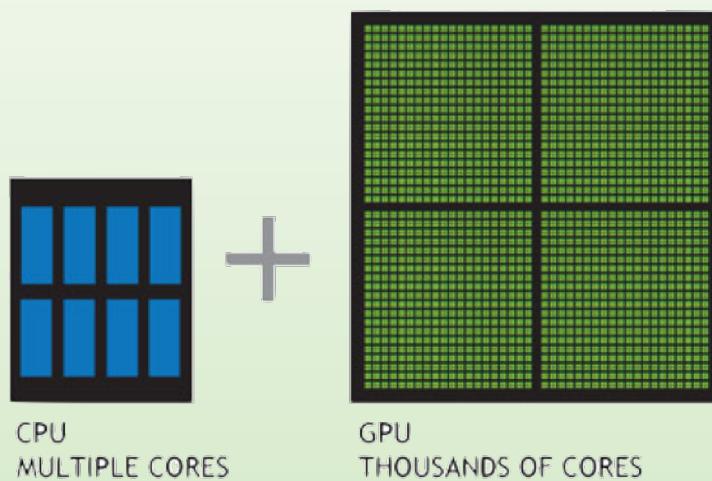
GPU oggi

- **General-Purpose computing on GPUs (GP-GPU)**
- **Parallel computing**
- **Heterogeneous architectures**
- **Heterogeneous applications**
- **Big data**

GP-GPU

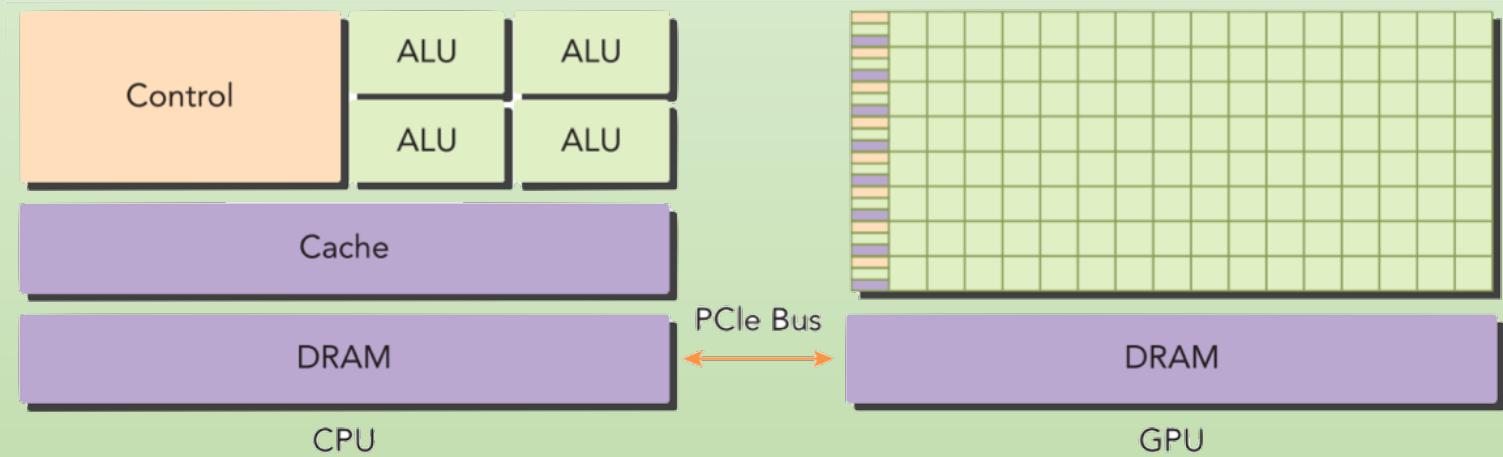
- ✓ **General-purpose GPU (GP-GPU)** si riferisce all'uso della GPU (graphics processing unit) per eseguire computazioni scientifiche, multimediali, ingegneristiche... a carattere generale
- ✓ **Modello eterogeneo**: implica l'uso di CPU e GPU insieme dando vita a un modello calcolo di co-processing
- ✓ **Separazione**: la parte sequenziale dell'applicazione esegue su CPU mentre la parte parallela (quella a maggiore intensità computazionale) viene accelerata dalla GPU
- ✓ **Trasparente**: dal punto di vista utente, l'applicazione esegue nel complesso più velocemente avvalendosi delle elevate prestazioni della GPU
- ✓ **Mapping**: una function sulla GPU implica riscrivere la function per esporla al parallelismo della GPU aggiungendo le annotazioni "C" che spostano dati ed eseguono una function (kernel) sulla GPU
- ✓ **Problema**: massimizzare la potenza di calcolo minimizzando l'energia consumata

Heterogeneous architectures

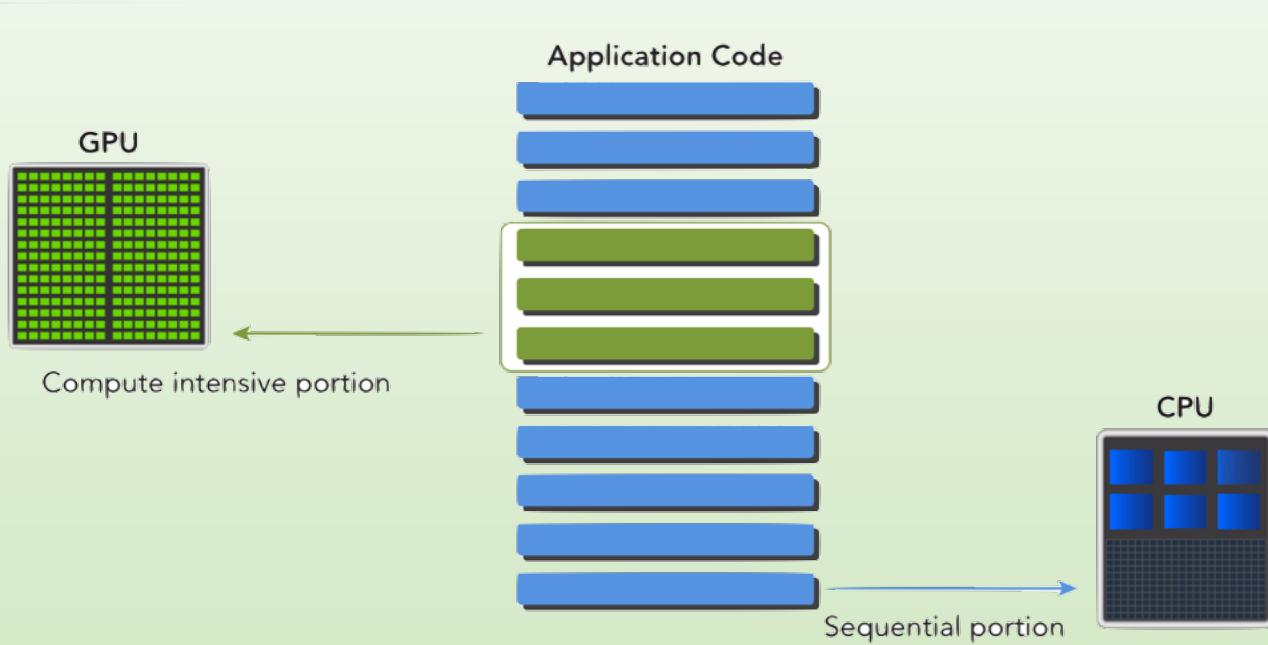


- ✓ Differenze di esecuzione dei task
- ✓ **CPU** = pochi core ottimizzati per l'elaborazione sequenziale
- ✓ **GPU** = architettura massicciamente parallela che consiste di migliaia di core che cooperano in modo efficiente per trattare una molteplicità di task concorrentemente

- ✓ La GPU è un coprocessore e non una piattaforma standalone, vengono anche chiamati **acceleratori**
- ✓ La CPU è chiamata **host** e la GPU **device**
- ✓ Opera congiuntamente con la CPU attraverso il **bus PCI-Express**
- ✓ La GPU necessita di **trasferimenti diretti di memoria** da parte della CPU
- ✓ CPU "orchestra" la **sincronizzazione**



Applicazioni ibride

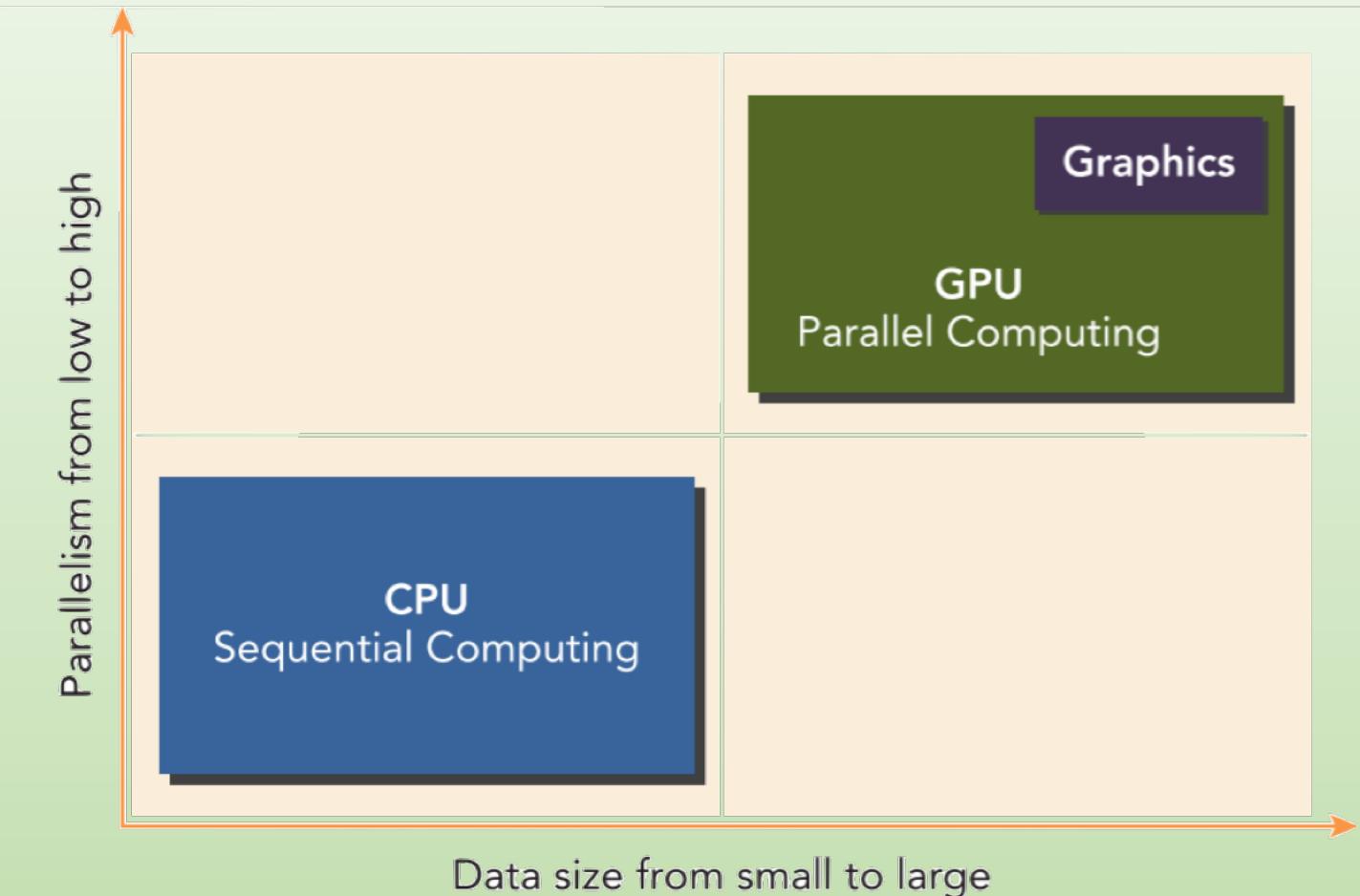


- ✓ Le applicazioni eterogenee sono caratterizzate da:
 - Codice host
 - Codice device
- ✓ Il codice host esegue su CPU e il codice device esegue su GPU
- ✓ Le GPU sono usate per accelerare le esecuzioni di codice parallelo su dati paralleli

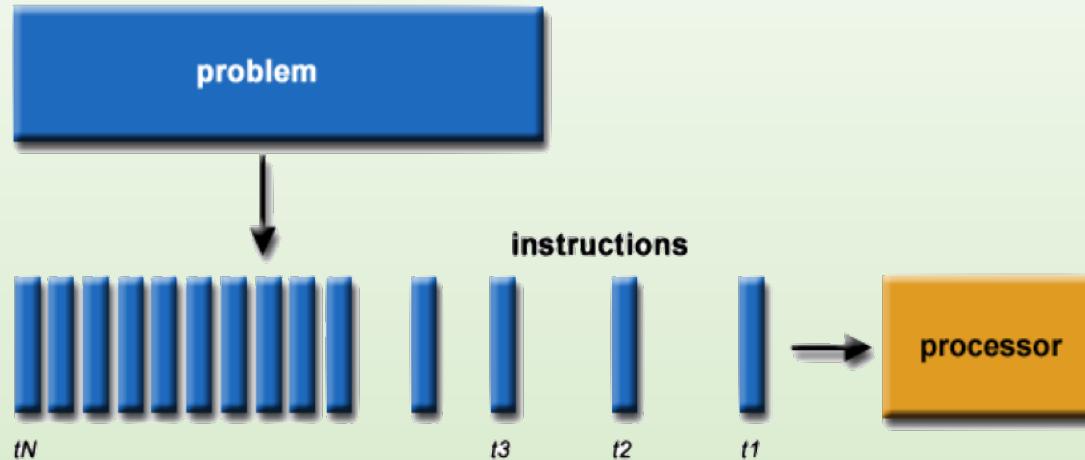
- ✓ Il codice CPU è responsabile della **gestione dell'ambiente**, dell'**IO** e della gestione dei **dati per il device** stesso, prima di caricare task intensivi sul device
- ✓ La GPU è usata per accelerare l'esecuzione di questa porzione di codice basandosi sul **parallelismo dei dati**
- ✓ La CPU è ottimizzata per sequenze di operazioni in cui il controllo del **flusso** è **impredicibile**
- ✓ le GPU lo sono per carichi dominati da **semplice flusso** di controllo

Parallelismo dei dati vs big data

- ✓ La **CPU** è ottimizzata per sequenze di operazioni in cui il controllo del **flusso** è **impredicibile**
- ✓ La **GPU** lo sono per carichi dominati da **semplice flusso** di controllo
- ✓ Come mostra la figura ci sono due dimensioni che differenziano lo scope di applicabilità di CPU e GPU: **livello di parallelismo** e **dimensione dei dati**
- ✓ La GPU è usata per accelerare l'esecuzione di quella porzione di codice basandosi sul **parallelismo dei dati**

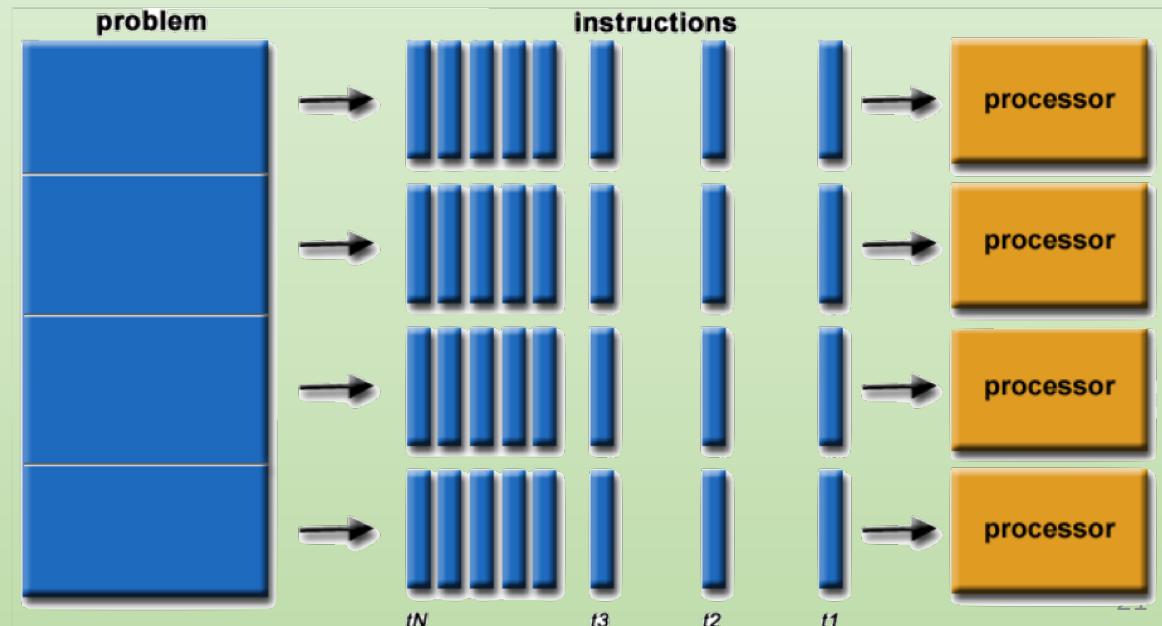


Parallelismo delle istruzioni



- ✓ Il problema è suddiviso in una serie discreta di istruzioni
- ✓ Un'istruzione è eseguita dopo la precedente... una dopo l'altra
- ✓ Sono tutte eseguite su un singolo processore
- ✓ Solo un'istruzione è eseguita ad ogni istante di tempo

- ✓ Il problema è suddiviso in parti che possono essere eseguite concorrentemente
- ✓ Ogni parte è successivamente ridotta a una serie di istruzioni
- ✓ Le istruzioni di ogni parte sono eseguite simultaneamente su differenti processori
- ✓ Un meccanismo di controllo/coordinamento viene impiegato alla fine per raccogliere i risultati parziali



Parallel computing

- ✓ **Scopo:** incrementare la velocità delle computazioni
- ✓ **Vantaggio:** molti calcoli vengono svolti in simultaneamente, secondo il principio che problemi di grandi dimensioni possono essere suddivisi in problemi di piccole dimensioni (task) che possono essere risolti simultaneamente/indipendentemente (es. prodotto matrici)
- ✓ **Prospettiva:** il programmatore deve gestire direttamente le risorse di calcolo (per es. core, memoria, I/O)
- ✓ Esistono due tipi fondamentali di parallelismo nelle applicazioni:
 - **Parallelismo dei task:** si verifica quando molti task possono operare indipendentemente in parallelo (funzioni distribuite su diversi core)
 - **Parallelismo dei dati:** si ha quando si può operare su più dati allo stesso tempo (dati distribuiti tra thread multipli)
- ✓ Due sono gli aspetti principali del parallel computing, legati anche alle tecnologi esistenti:
 - **Architetture di calcolo parallelo**
 - **Programmazione parallela**

Architetture parallele -1-

✓ Parallelismo a livello di bit

- **Word size:** 4,16,32,63 bit per avere sempre più accuratezza nelle operazioni floating point e spazi di indirizzamento sempre maggiori (2^{64} byte)
- **Esempi:** i primi mainframe, workstation, tutti i PC moderni single-core

✓ Parallelismo via pipelining

- **Idea:** sovrapposizione di esecuzioni di più istruzioni mediante pipeline
- **Stadi di esecuzione:** *fetch, decode, execute, write back*
- **Vantaggio:** diversi stadi possono operare assieme in parallelo (una istruzione per ciclo di clock)
- **Grado di parallelismo:** numero di stadi di pipeline, dato anche dal numero di pipeline presenti (parallelismo a livello di istruzione)
- **Problema:** la dipendenza dei dati spesso limita il grado di parallelismo
- **Necessità:** una ridotta dipendenza tra i dati (*data dependency*, quando una istruzione richiede dati prodotti da una precedente istruzione)

Architetture parallele -2-

✓ Parallelismo dato da molteplici unità funzionali

- **Unità funzionali:** tra loro indipendenti come le ALU (arithmetic logical unit), FPU (floating point unit), load/store unit o branch unit
- **Parallelismo:** differenti istruzioni indipendenti possono essere eseguite in parallelo da diverse unità funzionali
- **Data dependency:** per processori superscalari la dipendenza può essere determinata a runtime dinamicamente e lo scheduling invia le istr. alle varie unità dinamicamente

✓ Parallelismo a livello di processo o di thread

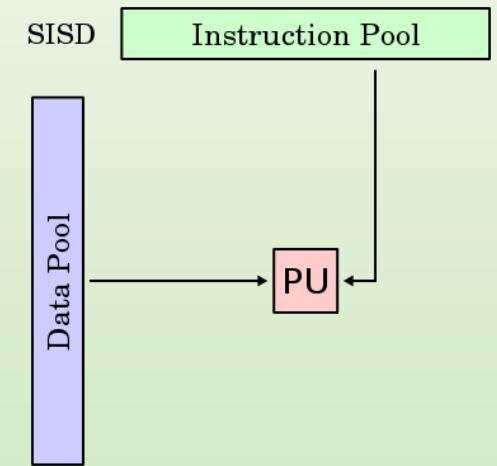
- **Approccio alternativo:** impiegare la logica integrata per porre all'interno del chip diverse unità indipendenti detti core
- **Flusso di controllo:** in luogo a un singolo flusso di controllo sequenziale di istruzioni se ne hanno diversi (uno per core o per thread!)
- **Programmazione parallela:** coordinare flussi, accessi multipla a memoria, condividere caches coordinare sincronizzazione e concorrenza

Tassonomia di Flynn -1-

Parallel computer = collezione di unità di elaborazione che comunicano e cooperano per risolvere velocemente un problema di grandi dimensioni

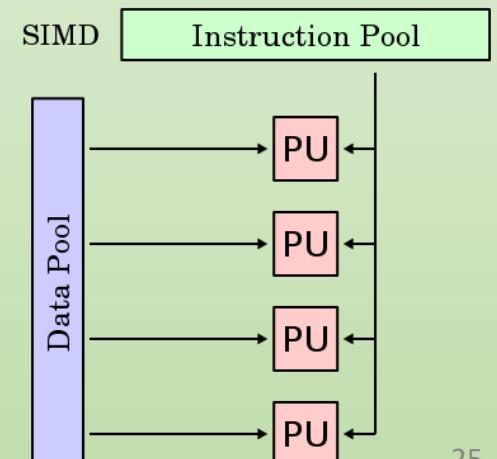
✓ SISD (Single Instruction Single Data)

- **Una unità** di computazione che accede a programma e dati
- **Nessun parallelismo**: le operazioni vengono eseguite sequenzialmente, su un dato alla volta (la classica architettura di von Neumann)



✓ SIMD (Single Instruction Multiple Data)

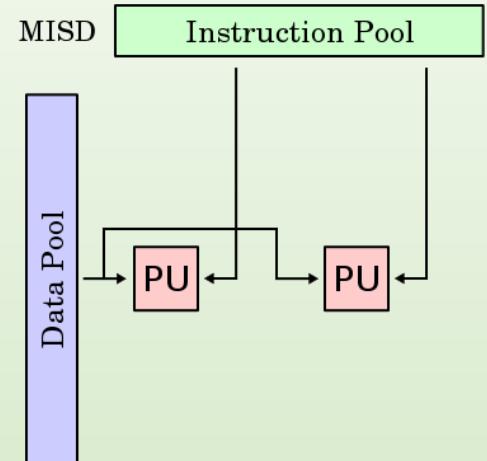
- **Molte unità** di computazione con accesso a memoria privata (condivisa o distribuita) per i dati , memoria globale unica per le istruzioni
- **Stessa istruzione**: ad ogni passo un processore centrale invia un'istruzione alle unità che leggono dati privati
- **Applicazioni** con alto grado di parallelismo ne traggono grande beneficio (multimedia, computer graphics, simulazioni, etc.)



Tassonomia di Flynn -2-

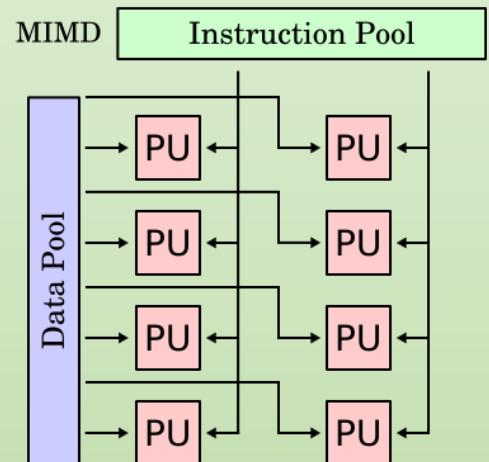
✓ MISD (Multiple Instruction Single Data)

- **Molte unità** di computazione con accesso a memoria privata di 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 dalla memoria privata
- **Parallelismo** a livello istruzione su medesimo dato (nessun sviluppo commerciale noto)



✓ 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 da altri
- **Esempi:** processori multicore, sistemi distribuiti, datacenter



Tassonomia di Flynn (sintesi)

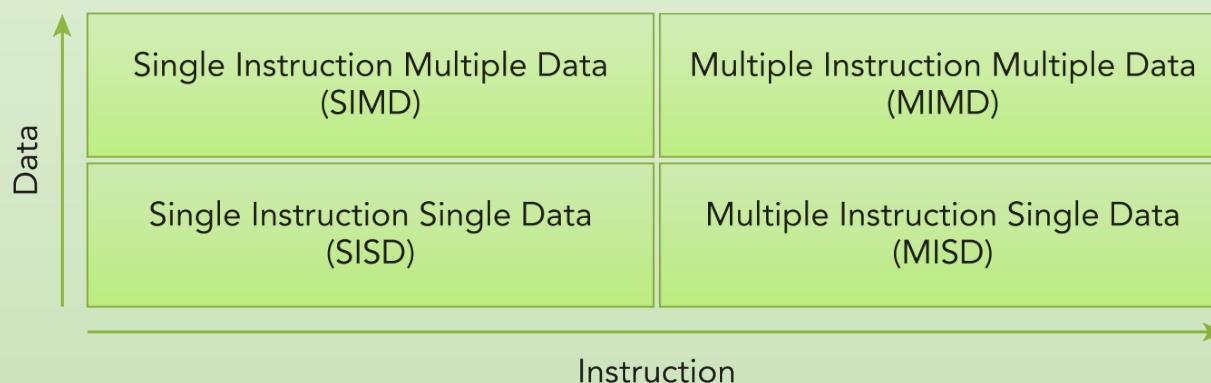
Classificazione sulla base di come istruzioni e dati fluiscono attraverso i core

Single Instruction Multiple Data (SIMD):

- architettura parallela con core multipli
- tutti i core eseguono nello stesso tempo la stessa istruzione, ma ognuna opera su dati diversi
- si continua a pensare sequenzialmente pur ottenendo uno speed-up dalle esecuzioni parallele sui dati
- grazie al compilatore si preoccupa dei dettagli non richiesti al programmatore

Multiple Instruction Multiple Data (MIMD):

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



Single Instruction Single Data (SISD):

- architettura seriale con un solo core (computer tradizionale)
- viene eseguita una istruzione alla volta e le operazioni sono eseguite su un dato per volta

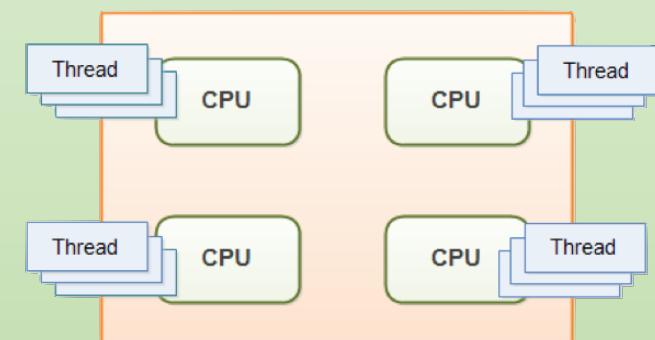
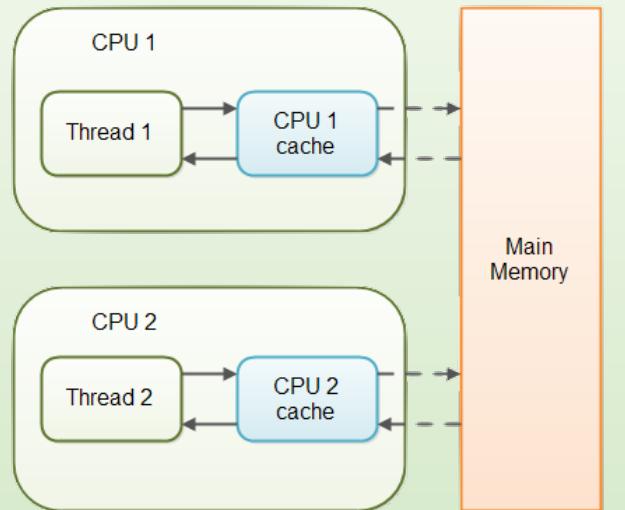
Multiple Instruction Single Data (MISD):

- architettura non comune dove ogni core opera sullo stesso stream di dati mediante diversi stream di istruzioni indipendenti

SIMT model (oltre Flynn)

✓ SIMT (Single Instruction Multiple Threads)

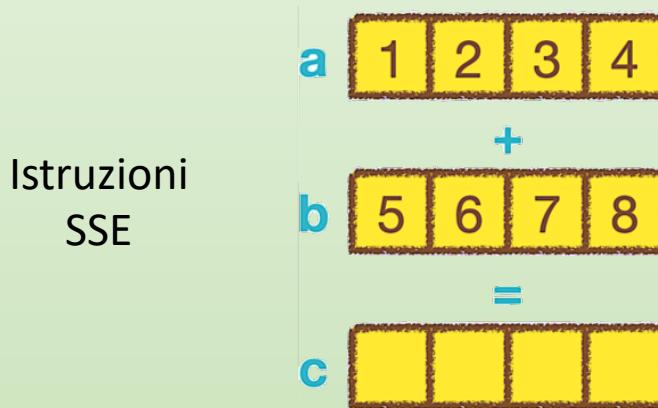
- **Multithreading**: in sistemi operativi multitasking rappresenta un diffuso modello di programmazione ed esecuzione che consente a thread multipli di coesistere all'interno del contesto di un processo in esecuzione
- **Modello SIMT** è usato in parallel computing dove si combina il modello SIMD con il multithreading
- **Condivisione** di risorse ma esecuzioni indipendenti che forniscono al programmatore una utile astrazione del concetto di esecuzione concorrente
- **Estensione** al modello di elaborazione che prevede che un processo abiliti esecuzioni parallele su sistemi multiprocessore o multicore
- **Implementato** su diverse GPU e fondamentale per GPGPU (introdotto d NVIDIA), per es. alcuni supercomputer combinano CPU con GPU



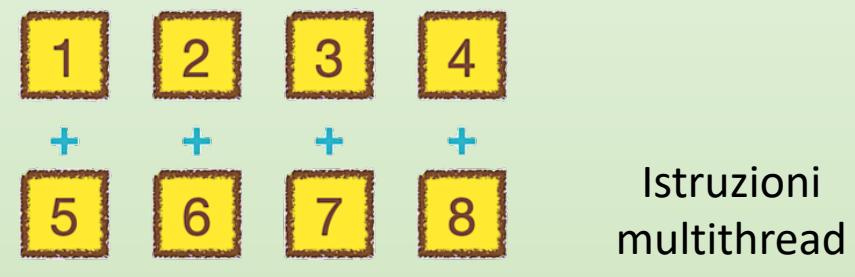
SIMT vs SIMD

Il modello **SIMT** prevede tre caratteristiche chiave che **SIMD** non contempla:

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



```
__m128 a = _mm_set_ps (4, 3, 2, 1);
__m128 b = _mm_set_ps (8, 7, 6, 5);
__m128 c = _mm_add_ps (a, b);
```

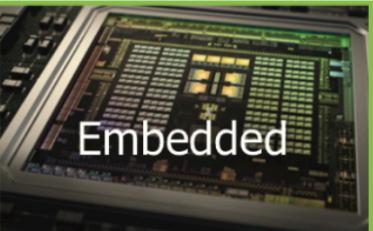


```
float a[4] = {1, 2, 3, 4};
float b[4] = {5, 6, 7, 8}, c[4];
{
    int id = ... ; // my thread ID
    c[id] = a[id] + b[id];
}
```

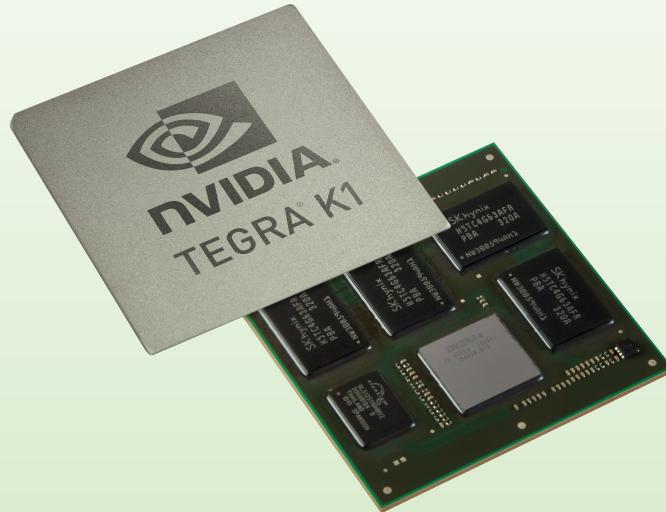
NVIDIA GPUs

Architetture e modelli delle GPU

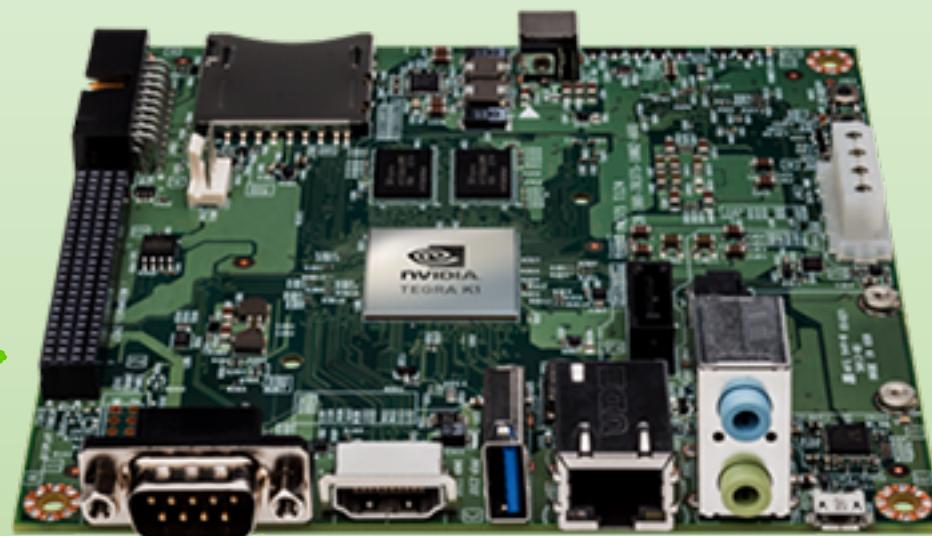
GPU NVIDIA

Volta Architecture (compute capabilities 7.x)				Tesla V Series
Pascal Architecture (compute capabilities 6.x)		GeForce 1000 Series	Quadro P Series	Tesla P Series
Maxwell Architecture (compute capabilities 5.x)	Tegra X1	GeForce 900 Series	Quadro M Series	Tesla M Series
Tensor Architecture (compute capabilities 3.x)	Tegra K1	GeForce 700 Series GeForce 600 Series	Quadro K Series	Tesla K Series
	 Embedded	 Consumer Desktop/Laptop	 Professional Workstation	 Data Center

NVIDIA Tegra K1



sistemi embedded

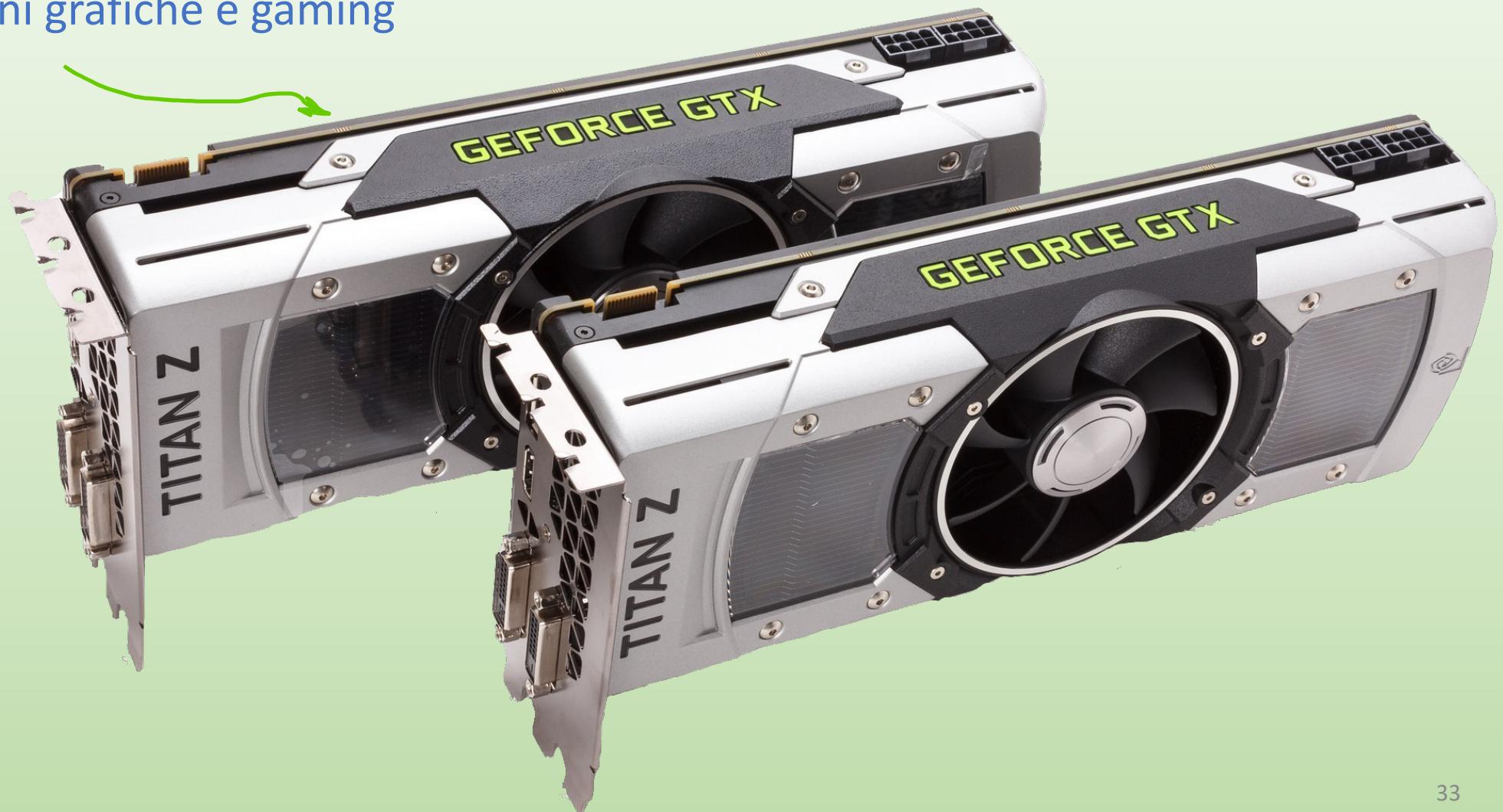


mobile e device



NVIDIA GeForce GTX Titan Z

applicazioni grafiche e gaming



NVIDIA Quadro 6000

grafica professionale
e visualizzazione



NVIDIA Tesla K40

datacenter e
parallel computing



Evoluzione HW

COMPUTE CAPABILITIES

NVIDIA uses a special term, *compute capability*, to describe hardware versions of GPU accelerators that belong to the entire Tesla product family. The version of Tesla products is given in Table 1-2.

Devices with the same major revision number are of the same core architecture.

- Kepler class architecture is major version number 3.
- Fermi class architecture is major version number 2.
- Tesla class architecture is major version number 1.

GPU	COMPUTE CAPABILITY
Tesla K40	3.5
Tesla K20	3.5
Tesla K10	3.0
Tesla C2070	2.0
Tesla C1060	1.3

Lagrange

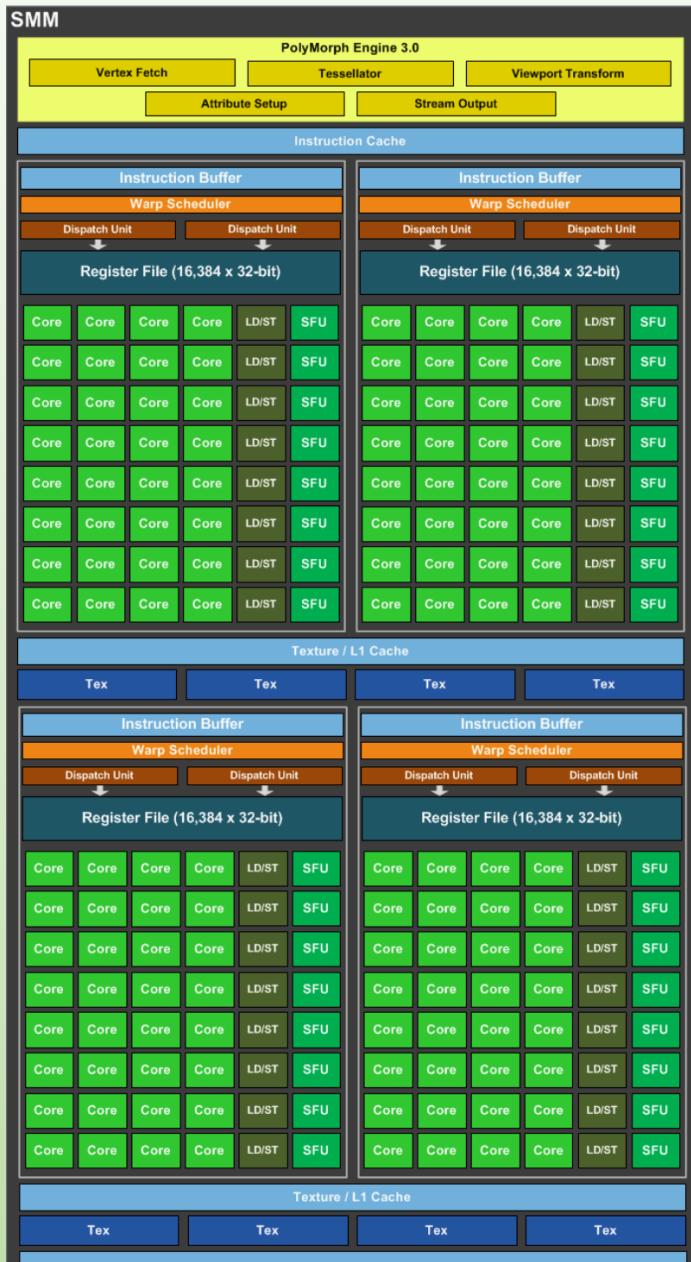


Architetture per GPU NVIDIA

- ✓ **Fermi** (2010): per applicazioni HPC, come elaborazione dati sismici, simulazioni biochimiche, modelli del clima e fluidodinamici, signal processing, finanza computazionale e data analysis
- ✓ **Kepler** (2012): migliora Fermi in termini potenza di calcolo per HPC e consumo energetico e aggiunge nuove features, come ad es. nuove metodi per ottimizzare e aumentare il carico di lavoro parallelo su GPU
- ✓ **Maxwell** (2014): architettura next-generation che estende il modello di programmazione precedente pur rimanendo compatibile con il codice sviluppato in precedenza, ovv. aumenta prestazioni di calcolo rispetto a Kepler, bandwidth molto superiore per trasferimenti in memoria
- ✓ **Pascal** (2016): oltre 5 TeraFLOPS in doppia precisione per carichi di lavoro HPC, deep learning (prestazioni 12 volte superiori nel training delle reti neurali), intelligenza artificiale
- ✓ **Volta** (2018): calcolo basato su tensori per l'AI

Maxwell SMM

- Four 32-core processing blocks each with a dedicated warp scheduler that can dispatch 2 instructions per clock
- Larger shared memory (dedicated to SM)
- Larger L2 cache (shared by SMs)



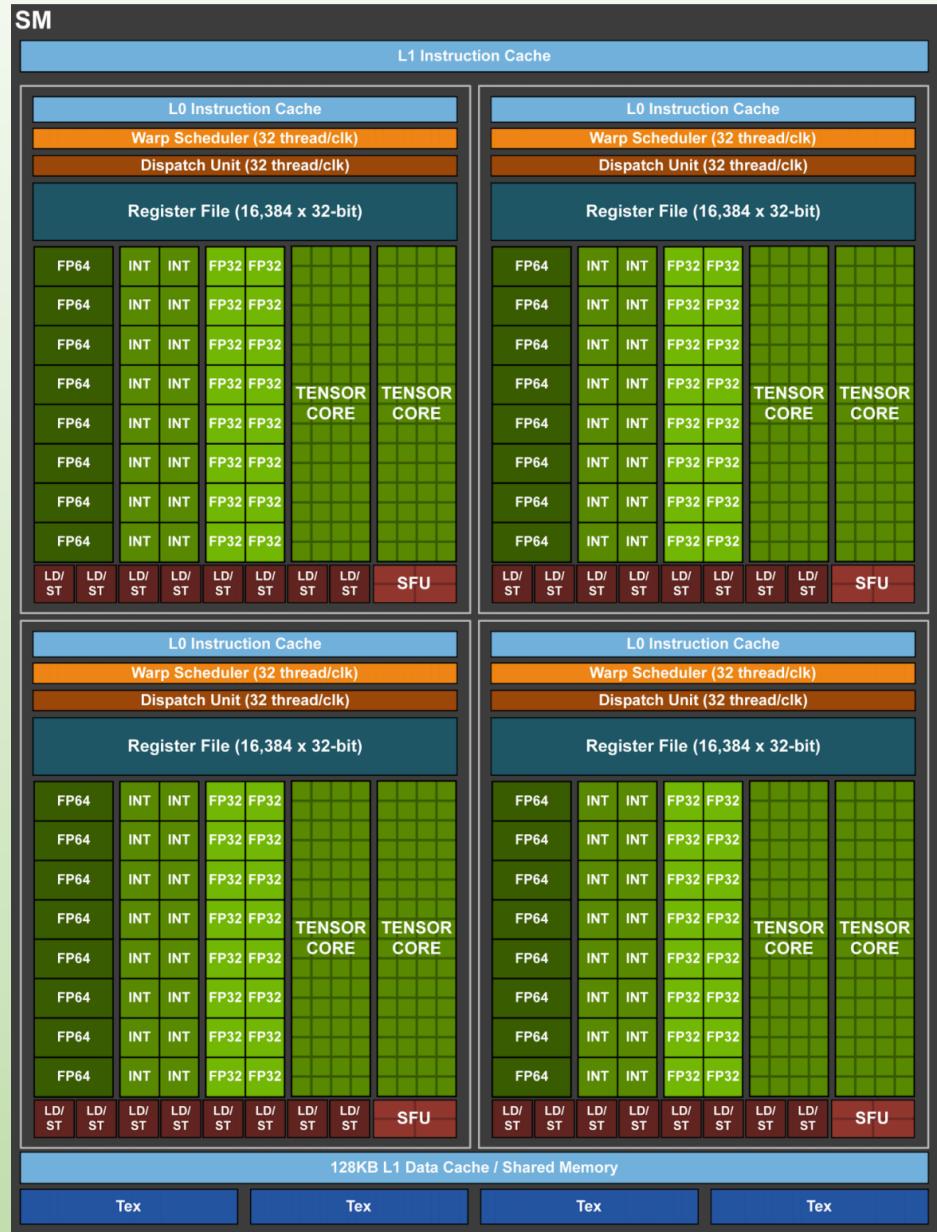
Pascal SM

- More SMs per GPU
- FP16 computation (2× faster than FP32)
- Less cores but same # registers: more registers per core
- Fast HBM2 memory interface
- Fast NVLink bus
- Unified memory: programs can access both CPU and GPU RAM



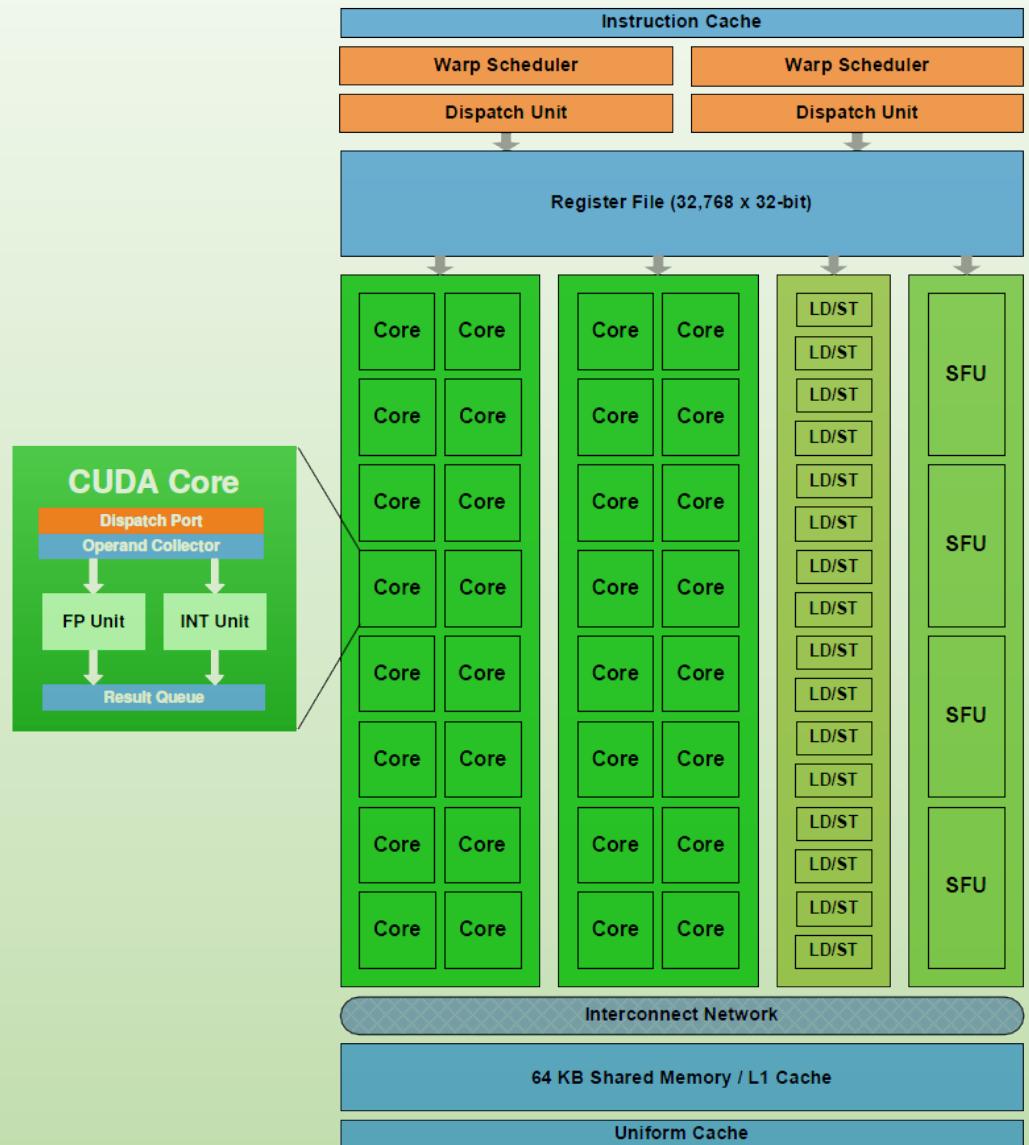
Volta SM

- New tensor cores
- Unified L1 / shared memory
- Independent FP32 and INT32 cores
- More SMs per GPU
- Larger L2 cache
- New L0 instruction cache (accessed directly from functional units)



Panoramica architettura GPU

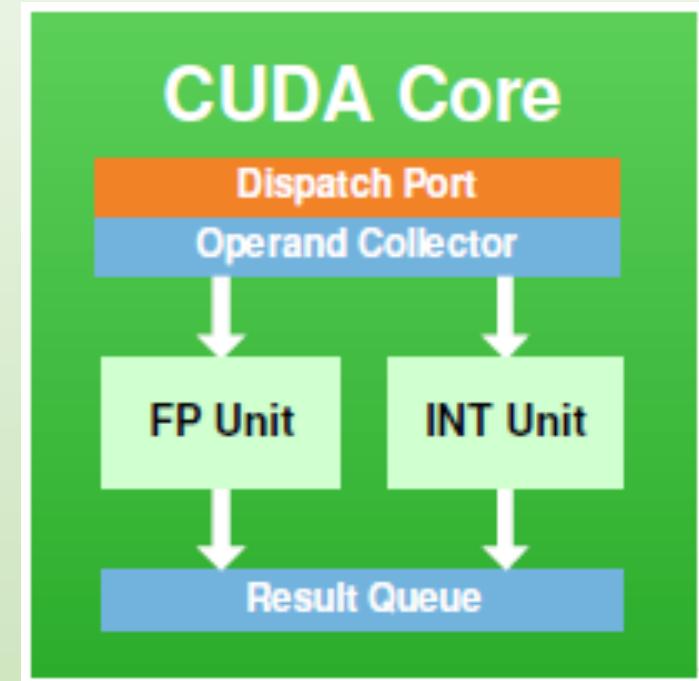
- L'architettura GPU è costruita attorno a un array scalabile di **Streaming Multiprocessors (SM)**
- Ogni SM in una GPU è progettato per supportare l'esecuzione **concorrente** di centinaia di thread
- Molteplici SM per GPU NVIDIA GPU eseguono thread in **gruppi of 32** chiamati **warp**
- Tutti i thread in un warp eseguono la **stessa istruzione allo stesso tempo**
- GPU HW differiscono in base alla “**compute capability**” (CC): più è alta meglio è!
- L'architettura **Maxwell** (e.g. GTX980) ha CC 5.2
L'architettura **Pascal** (e.g. GTX1080) ha CC 6.0-6.2. L'architettura (l'ultima) **Volta** ha CC 7.0



Cuda core

- It's a vector processing unit
- Works on a single operation
- It's the building block of SM
- As the process reduces them (e.g.

28nm) they increase in number per SM



CUDA zone

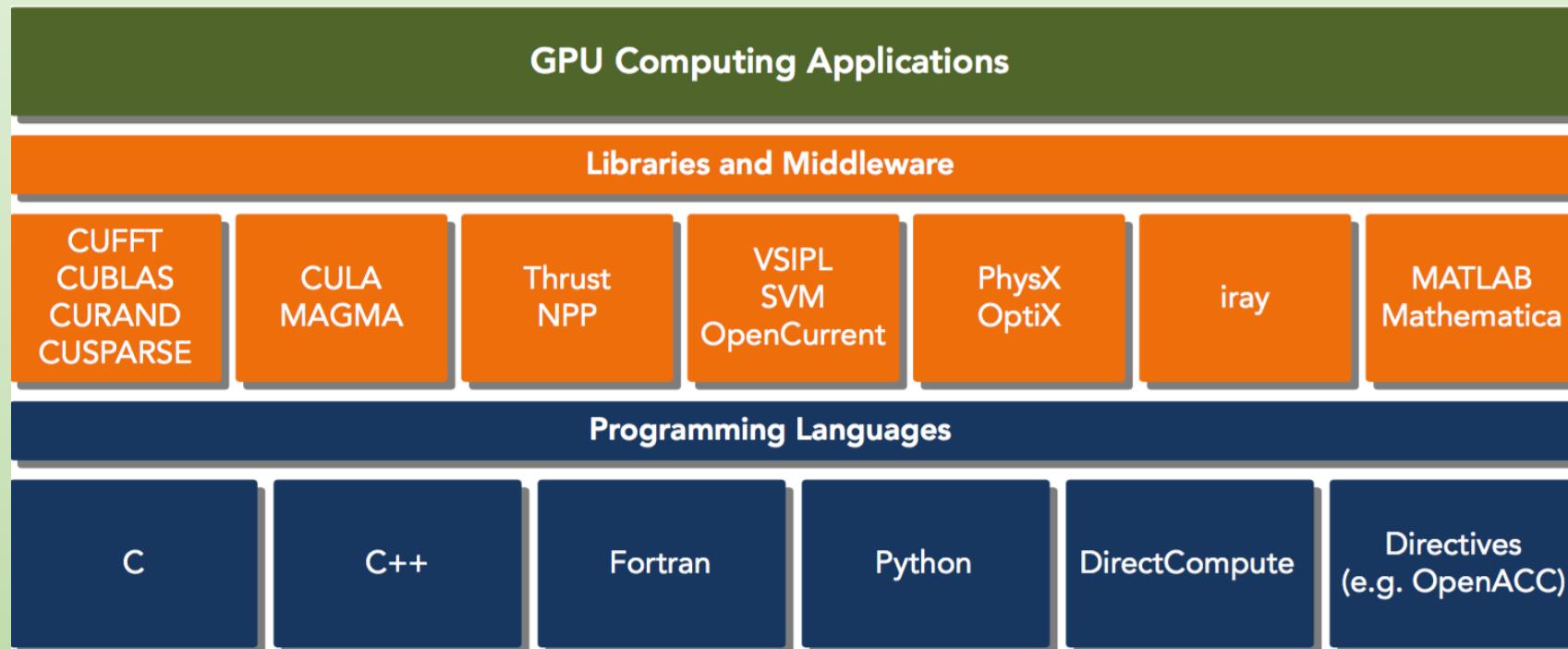
CUDA toolkit

CUDA: Compute Unified Device Architecture

- ✓ It enables a **general-purpose programming model** on NVIDIA GPUs
- ✓ Current CUDA SDK is 10.0
- ✓ Enables explicit GPU **memory management**
- ✓ The **GPU** is viewed as a **compute device** that:
 - Is a **co-processor** to the CPU (or host)
 - Has its own **DRAM** (global memory in CUDA parlance)
 - Runs many **threads** in parallel

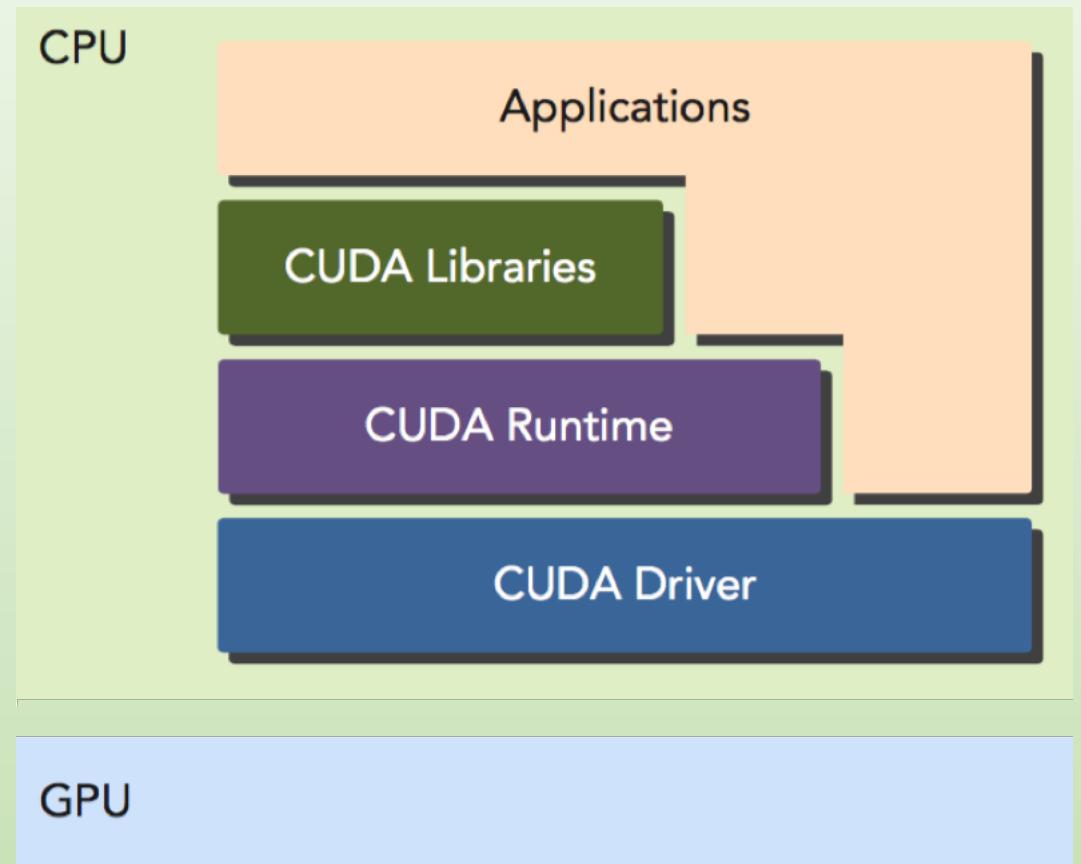
La piattaforma CUDA

- ✓ The CUDA platform is accessible through CUDA-accelerated libraries, compiler directives, application programming interfaces (API), and extensions to industry-standard programming languages, including C, C++, Fortran, and Python
- ✓ CUDA C is an extension of standard ANSI C with a handful of language extensions to enable heterogeneous programming, and also straightforward APIs to manage devices, memory, and other tasks.



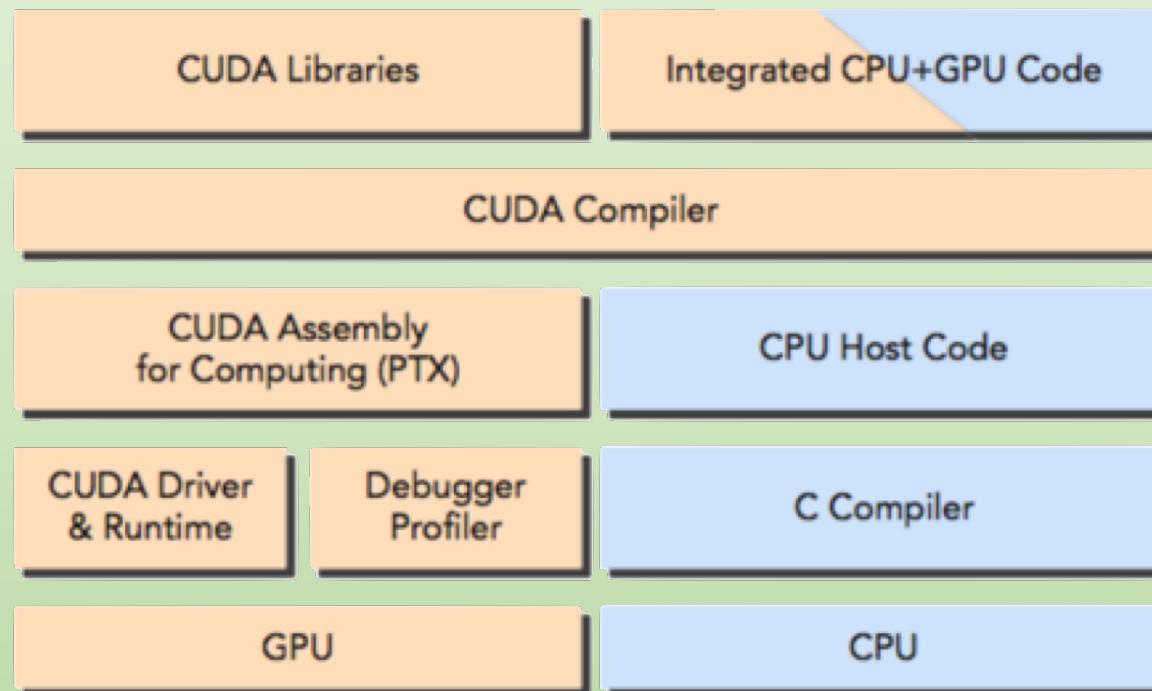
CUDA APIs

- ✓ CUDA fornisce due livelli di API per la gestione della GPU e l'organizzazione dei thread
 - ✓ **CUDA Driver API**
 - ✓ **CUDA Runtime API**
- ✓ Le **driver API** sono API a **basso livello** e piuttosto difficili da programmare ma danno un maggior controllo della GPU
- ✓ Le **runtime API** sono API ad **alto livello** implementate sulle driver API
- ✓ Ogni **funzione** delle runtime API è **suddivisa** in diverse **operazioni basilari** fornite dalle driver API

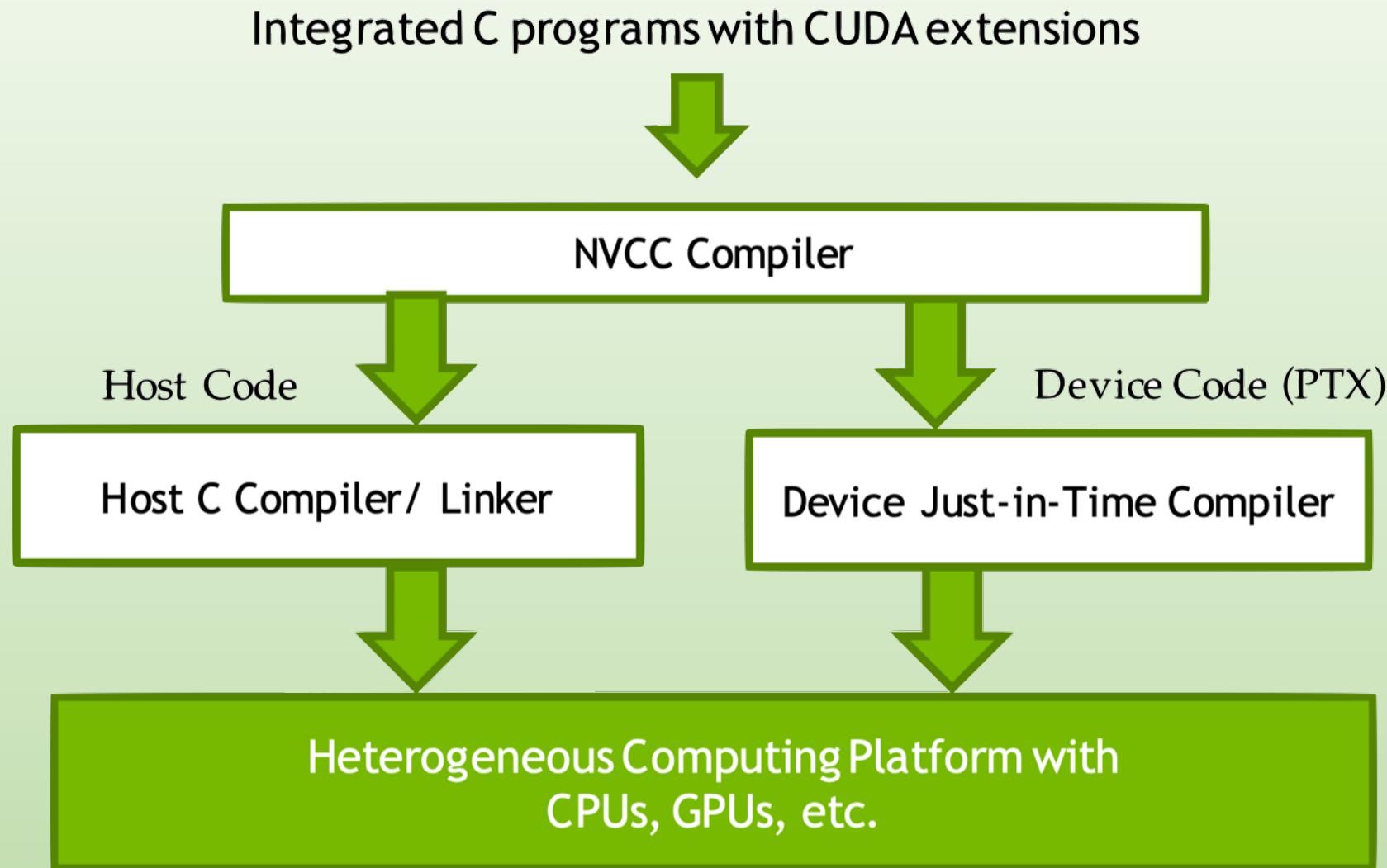


Un programma CUDA

- ✓ Un programma CUDA consiste di una mistura di due parti:
 1. il **codice host** eseguito su CPU
 2. il **codice device** eseguito su GPU
- ✓ Il compilatore **nvcc** CUDA separa il codice host da quello device durante la **compilazione**



Compilazione di un programma CUDA



Un semplice problema...

Somma di due vettori: C[*] <- A[*] + B[*]

Dal lato 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];
```

<- intrinsecamente sequenziale!

Un possibile miglioramento (rimedio al sequenziale puro):

- ✓ Crea # di thread uguale al numero di core del processore (2, 4, 8, ...)
- ✓ Indica le porzioni di A, B, C da assegnare ad ogni thread...
- ✓ In ogni thread:

```
for all i <nella regione relativa al thread>  
    C[i] <- A[i] + B[i] // molta attesa per le op. r/w in memoria...
```

- ✓ Sincronizza i thread (wait...)
- ✓ Leggermente più veloce con speedup di 2-8 x (all'incirca)!

Una semplice soluzione

Somma di due vettori: $C[*] \leftarrow A[*] + B[*]$

Dal alto GPU

- Alloca memoria per A, B, C sulla GPU
- Crea un “**kernel**” – ogni thread esegue una (o più) addizioni
- Specify the following kernel operation:

for all i <assegnati a questo specifico thred>

C[i] <- A[i] + B[i] // poca attesa per le op. read/write in memoria...

- Attiva ~20000 (!) thread
- Sincronizza i thread (wait...)

Soluzione GPU ->

```
/*
 * kernel: somma di vettori
 */
__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];
}
```

<- ID unico del thread

Questo corso

- ✓ Introduzione ai sistemi di calcolo eterogenei basati su CPU e GPU
- ✓ Programmazione parallela e il concetto di GPGPU (General Purpose GPU)
- ✓ Il modello di programmazione CUDA: API e linguaggio CUDA C
- ✓ Il modello di esecuzione CUDA: kernel, blocchi, thread e sincronizzazione
- ✓ Il modello di memoria di CUDA: globale, condivisa e costante
- ✓ Stream, concorrenza e ottimizzazione delle prestazioni
- ✓ Librerie di CUDA SDK accelerate da GPU
- ✓ Programmazione e computazione multi-GPU
- ✓ Pattern di parallelismo negli algoritmi
- ✓ Sviluppo e implementazione di applicazioni su GPU NVIDIA

Il mio primo programma CUDA

Hello world (a più voci!)

Accesso a Lagrange

Connettersi via **ssh** alla macchina **Lagrange** dotata di 3 schede NVIDIA: 2 **Tesla M2090** e 1 **Tesla M2050**

```
$ ssh grossi@cuda.lagrange.di.unimi.it
```

Controllare se il **compilatore CUDA** è presente con il comando Linux di sistema:

```
$ whereis nvcc
```

Controllare se le schede CUDA sono montate sul sistema con il comando Linux di sistema:

```
$ ls -l /dev/nv*
crw-rw-rw-. 1 root 195,    0  2 mar 11.29 /dev/nvidia0
crw-rw-rw-. 1 root 195,    1  2 mar 11.29 /dev/nvidia1
crw-rw-rw-. 1 root 195,    2  2 mar 11.29 /dev/nvidia2
crw-rw-rw-. 1 root 195, 255  2 mar 11.29 /dev/nvidiactl
crw-rw-rw-. 1 root 245,    0  2 mar 11.29 /dev/nvidia-uvm
```

Hello World in C (ma compilare con nvcc)

1. Creare un file sorgente con estensione **.cu**
2. Compilare il sorgente usando il compilatore CUDA **nvcc**
3. Eseguire il programma (che contiene il codice kernel ed eseguibile)

Hello world in C ->

```
#include <stdio.h>

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

Salvare il codice nel file **hello.cu** e compilare con **nvcc** (la procedura è simile a **gcc** o altri compilatori – fornire parametri di compilazione ed elenco sorgenti) ed eseguire:

```
$ nvcc hello.cu -o hello
$ hello
Hello World from CPU!
```

Hello world in CUDA

Hello world in CUDA ->

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

- ✓ Il qualificatore **`__global__`** dice al compilatore che la function è chiamata dalla CPU ma eseguita in GPU
- ✓ L'invocazione del kernel : **`helloFromGPU <<< 1, 10 >>>()`**;
- ✓ La **tripla parentesi angolare** **`<<< * , * >>>`** denota una **chiamata dal codice host al codice device** che ne attiva l'esecuzione
- ✓ Un **kernel** viene eseguito da un **array di thread** e tutti i thread eseguono lo **stesso codice**
- ✓ I parametri all'interno della **triplice parentesi angolare** sono i **parametri di configurazione** che specificano quanti thread verranno eseguiti dal kernel
- ✓ In questo esempio, verranno eseguiti **10 GPU thread!**

Listato finale

```
#include <stdio.h>

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

int main(void) {
    // hello from GPU
    printf("Hello World from CPU!\n");
    helloFromGPU <<<1, 10>>>();
    cudaDeviceReset();
    return 0;
}
```

- ✓ La funzione **cudaDeviceReset()** distrugge e ripulisce tutte le risorse associate al device corrente nel processo corrente (non indispensabile!... vedere successivamente)

- ✓ La compilazione richiede lo switch **-arch sm_20** per generare un codice eseguibile su architettura **Fermi capability 2.0**

```
$ nvcc -arch sm_20 hello.cu -o hello
$ ./hello
Hello World from CPU!
Hello World from GPU!
```

In sintesi

Struttura di un programma CUDA

1. Allocare la memoria GPU
2. Copiare i dati da memoria CPU
a memoria GPU
3. Invocare il kernel CUDA
4. Copiare i dati da memoria GPU
a memoria CPU
5. Ripulire le memorie GPU

Tool di sviluppo CUDA

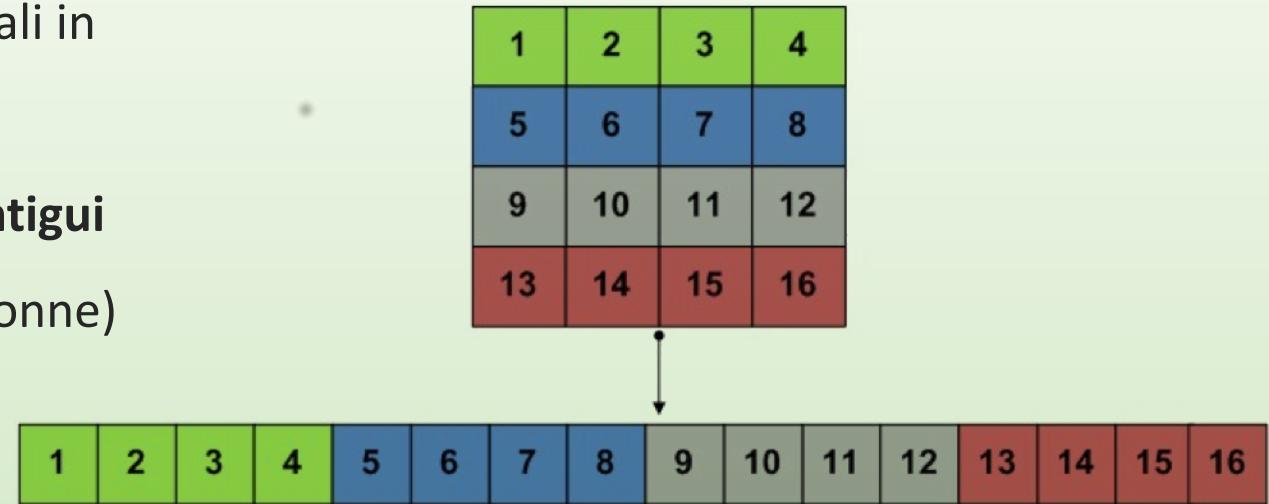
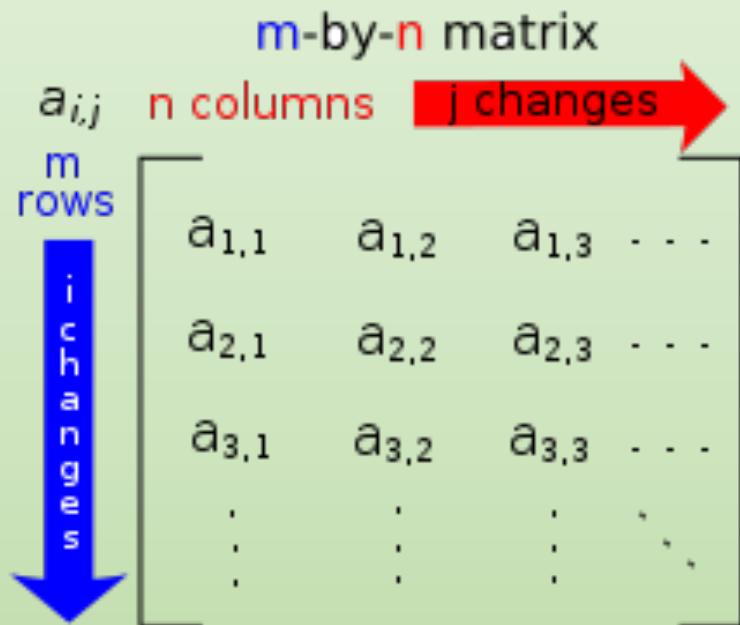
1. Ambiente integrato di sviluppo
NVIDIA Nsight™
2. Debugger a linea di comando CUDA-GDB
3. Profiler grafico o a linea di comando
per analisi di prestazioni
4. Analizzatore CUDA-MEMCHECK per
la memoria
5. Tool per la gestione del device GPU

Array multidimensionali in C

Esercitazione su prodotti di matrici “linearizzate”

Allocazione dinamica della memoria

- ✓ C organizza i dati di array multidimensionali in **row-major order** ("linearizzati")
- ✓ Elementi **consecutivi** delle **righe** sono **contigui**
- ✓ Esempio: matrice **$n \times m$** (**n** righe e **m** colonne)



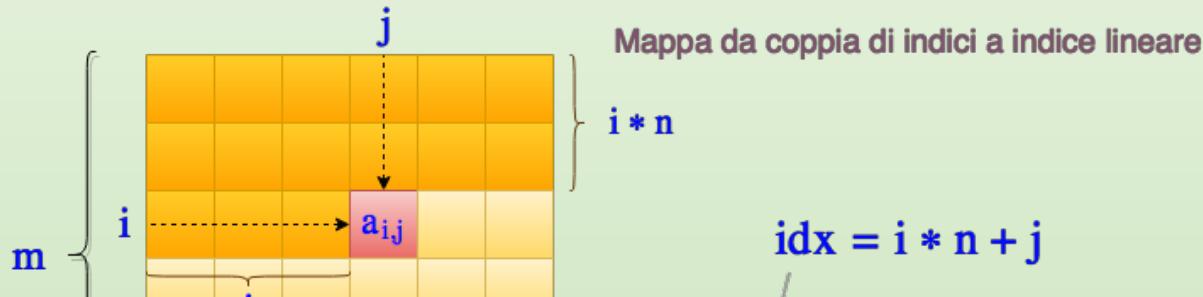
```
// azzeramento della matrice generata dinamicamente
A = (int *) malloc(n * m * sizeof(int));

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

Allocazione dinamica e indicizzazione

- ✓ Allocazione **fisica** di dati in memoria e accesso **logico** alle strutture dati in C

Matrice: organizzazione logica



Mappa da coppia di indici a indice lineare

$i * n$

$idx = i * n + j$

Matrice: organizzazione "linearizzata" in memoria fisica



- ✓ Codice C per l'azzeramento di dati in una matrice di n righe e m colonne con accesso tramite **indice linearizzato**

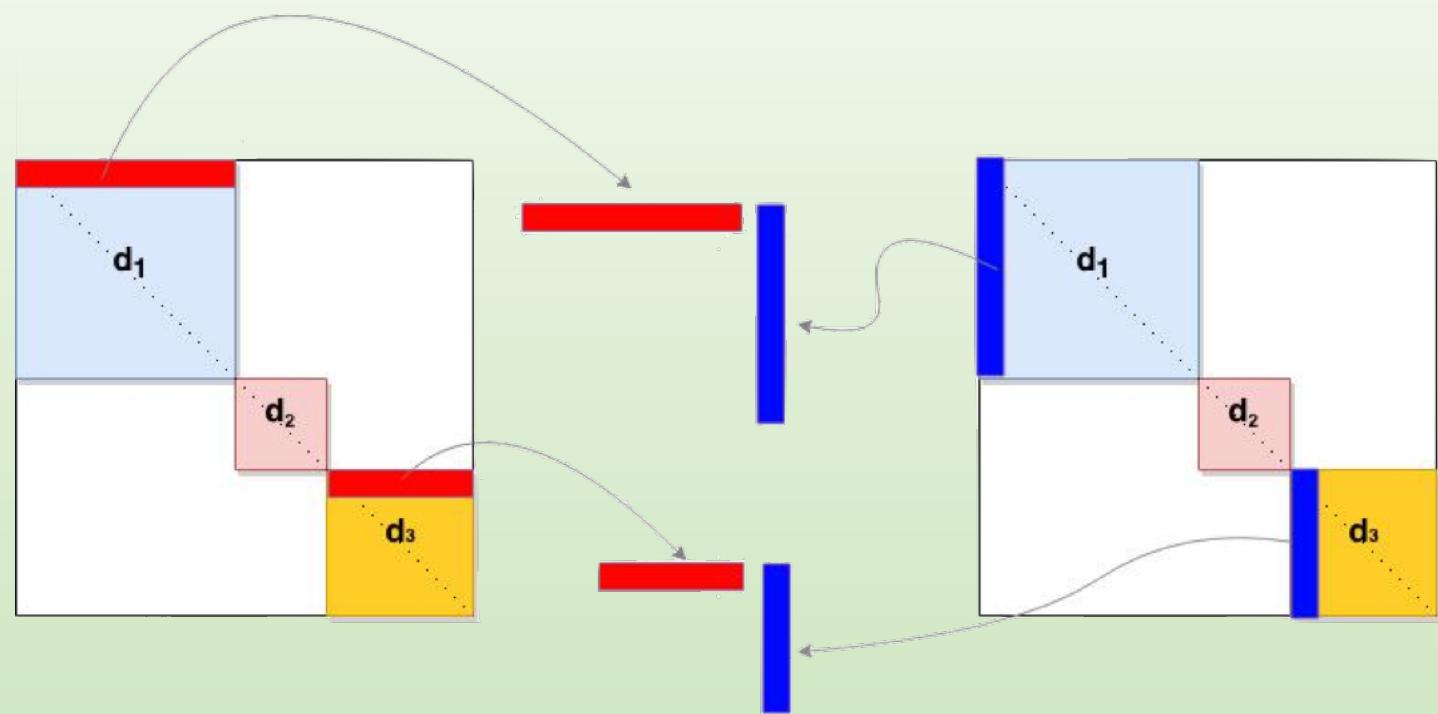
```
// azzeramento della matrice generata
// dinamicamente

A = (int *) malloc(n * m * sizeof(int));

for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
        idx = i * n + j; // indice linearizzato
        A[idx] = func(i,j); // funz di i e j
    }
}
```

Esercizio: Prodotto di matrici diagonali a blocchi

- ✓ L'algoritmo naïve per una matrice quadrata esegue in tempo $O(n^3)$
- ✓ L'algoritmo di Strassen, basato sul prodotto efficiente di matrici, esegue in tempo $O(n^{2.8})$
- ✓ Algoritmi più efficienti arrivano a $O(n^{2.37})$ sfruttando il prodotto di matrici $k \times k$
- ✓ Il tempo di esecuzione per il prodotti di matrici diagonali a blocchi d_1, d_2, \dots, d_k è $\sum_{i=1}^k d_i^3$



Sviluppare un programma C che implementi in modo efficiente il prodotto di **matrici diagonali a blocchi** ammettendo di conoscere le dimensioni (contenute in un array) dei blocchi componenti la matrice

Risultato

```
pascal[~]->mat_blk_prod
```

```
matrix A:
```

```
4 4 4 4  
4 4 4 4  
4 4 4 4  
4 4 4 4  
    2 2  
    2 2  
    1  
    3 3 3  
    3 3 3  
    3 3 3
```

```
matrix B:
```

```
4 4 4 4  
4 4 4 4  
4 4 4 4  
4 4 4 4  
    2 2  
    2 2  
    1  
    3 3 3  
    3 3 3  
    3 3 3
```

```
matrix C:
```

```
64 64 64 64  
64 64 64 64  
64 64 64 64  
64 64 64 64  
    8 8  
    8 8  
    1  
    27 27 27  
    27 27 27  
    27 27 27
```

Esercizio: prodotto di Kronecker

Date le matrici:

$$A \in \mathbf{R}^{n \times m}, \quad B \in \mathbf{R}^{p \times q}$$

Calcolare:

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} A_{11}\mathbf{B} & A_{12}\mathbf{B} & \cdots & A_{1m}\mathbf{B} \\ A_{21}\mathbf{B} & A_{22}\mathbf{B} & \cdots & A_{2m}\mathbf{B} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1}\mathbf{B} & A_{n2}\mathbf{B} & \cdots & A_{nm}\mathbf{B} \end{bmatrix}$$

Sviluppare un programma C che implementi in modo efficiente il prodotto di Kronecker tra due **matrici diagonali a blocchi**

Riferimenti bibliografici

Testi generali:

1. Jason Sanders, Edward Kandrot, CUDA by examples: an introduction to general-purpose GPU programming, Pearson Education, 2011
2. Dispensa (online) di parallel computing:
<http://www.cse.unt.edu/~tarau/teaching/parpro/papers/Parallel%20computing.pdf>

NVIDIA docs:

1. CUDA Programming: <http://docs.nvidia.com/cuda/cuda-c-programming-guide/>
2. CUDA C Best Practices Guide: <http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/>