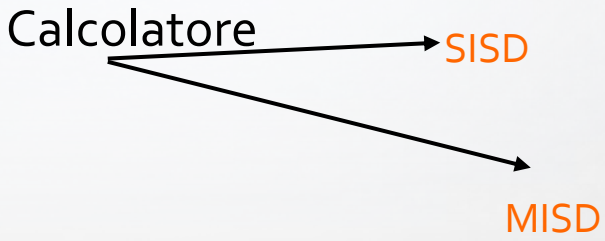
Architettura delle GPU
cenni alla programmazione CUDA

Prof. G. Laccetti
Corso di Parallel and Distributed Computing
Università degli Studi di Napoli Federico II
a.a. 2023/24

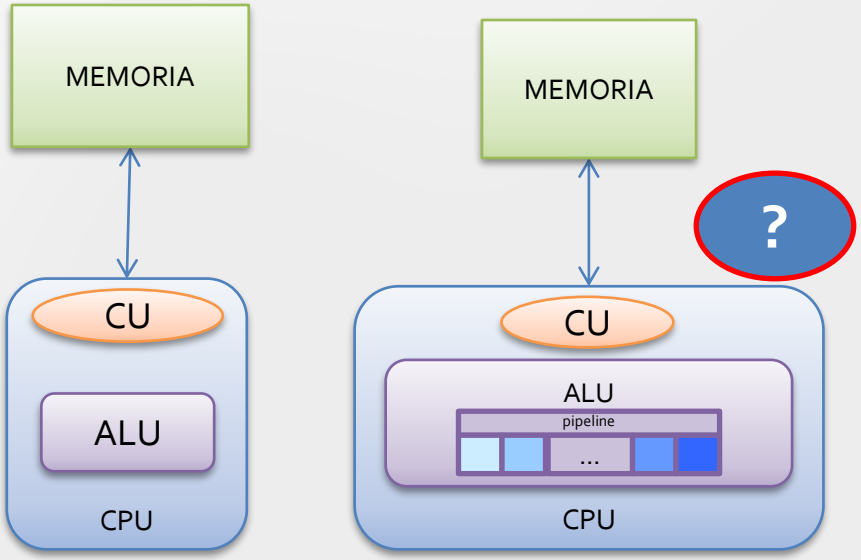
Premesse
Tassonomia di Flynn



Instruction Stream: sequenza di istruzioni eseguite dalla macchina.

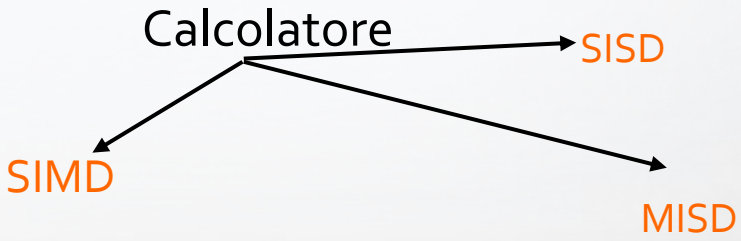
Data Stream: sequenza di dati richiesta dal flusso di istruzioni, compresi input e variabili d'appoggio.

[1]



- Single Instruction Stream / Single Data Stream (SISD)
- Single Instruction Stream / Multiple Data Stream (SIMD)
- Multiple Instruction Stream / Single Data Stream (MISD)
- Multiple Instruction Stream / Multiple Data Stream (MIMD)

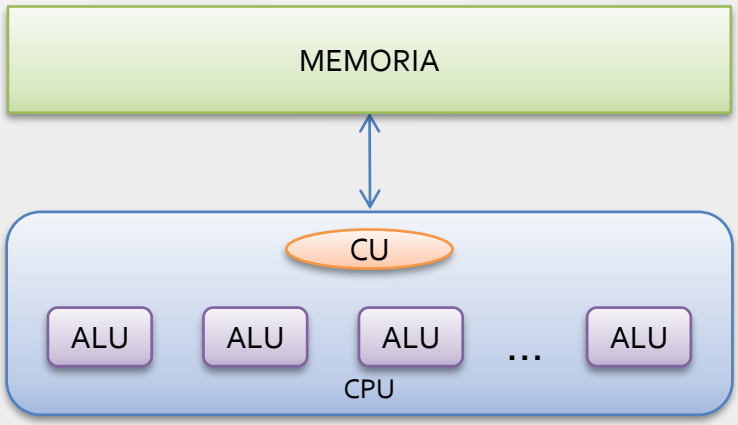
Premesse
Tassonomia di Flynn



Instruction Stream: sequenza di istruzioni eseguite dalla macchina.

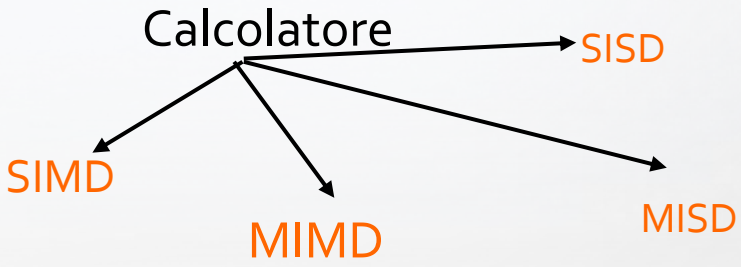
Data Stream: sequenza di dati richiesta dal flusso di istruzioni, compresi input e variabili d'appoggio.

[1]



- Single Instruction Stream / Single Data Stream (SISD)
- Single Instruction Stream / Multiple Data Stream (SIMD)
- Multiple Instruction Stream / Single Data Stream (MISD)
- Multiple Instruction Stream / Multiple Data Stream (MIMD)

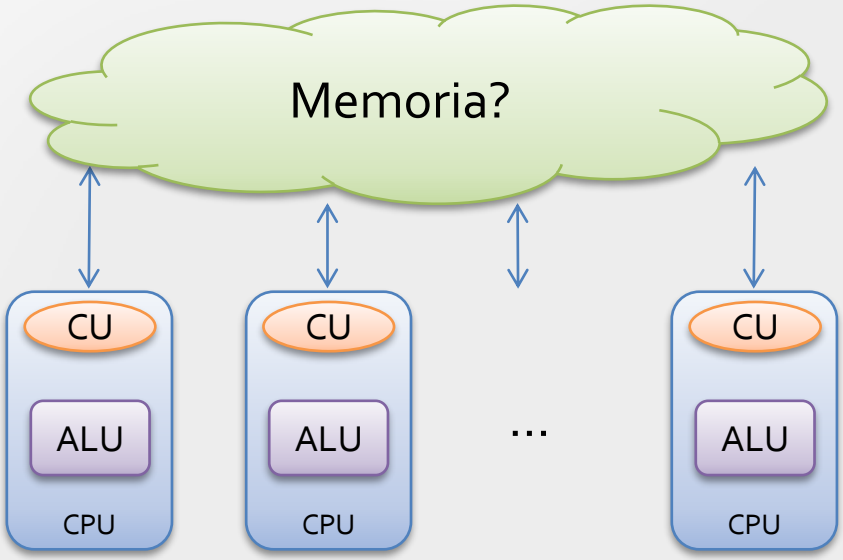
Premesse
Tassonomia di Flynn



Instruction Stream: sequenza di istruzioni eseguite dalla macchina.

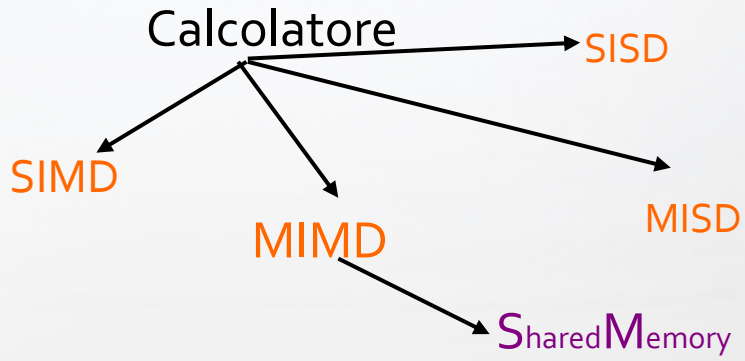
Data Stream: sequenza di dati richiesta dal flusso di istruzioni, compresi input e variabili d'appoggio.

[1]



- Single Instruction Stream / Single Data Stream (SISD)
- Single Instruction Stream / Multiple Data Stream (SIMD)
- Multiple Instruction Stream / Single Data Stream (MISD)
- Multiple Instruction Stream / Multiple Data Stream (MIMD)

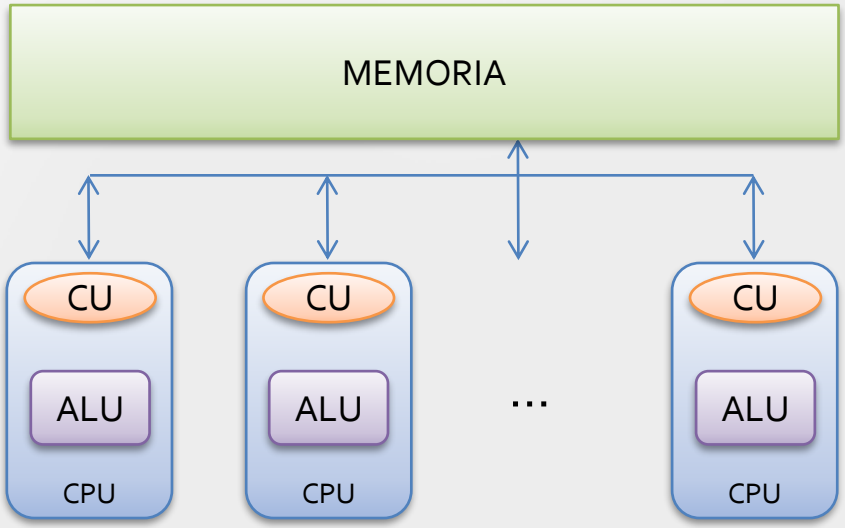
Premesse
Tassonomia di Flynn



Instruction Stream: sequenza di istruzioni eseguite dalla macchina.

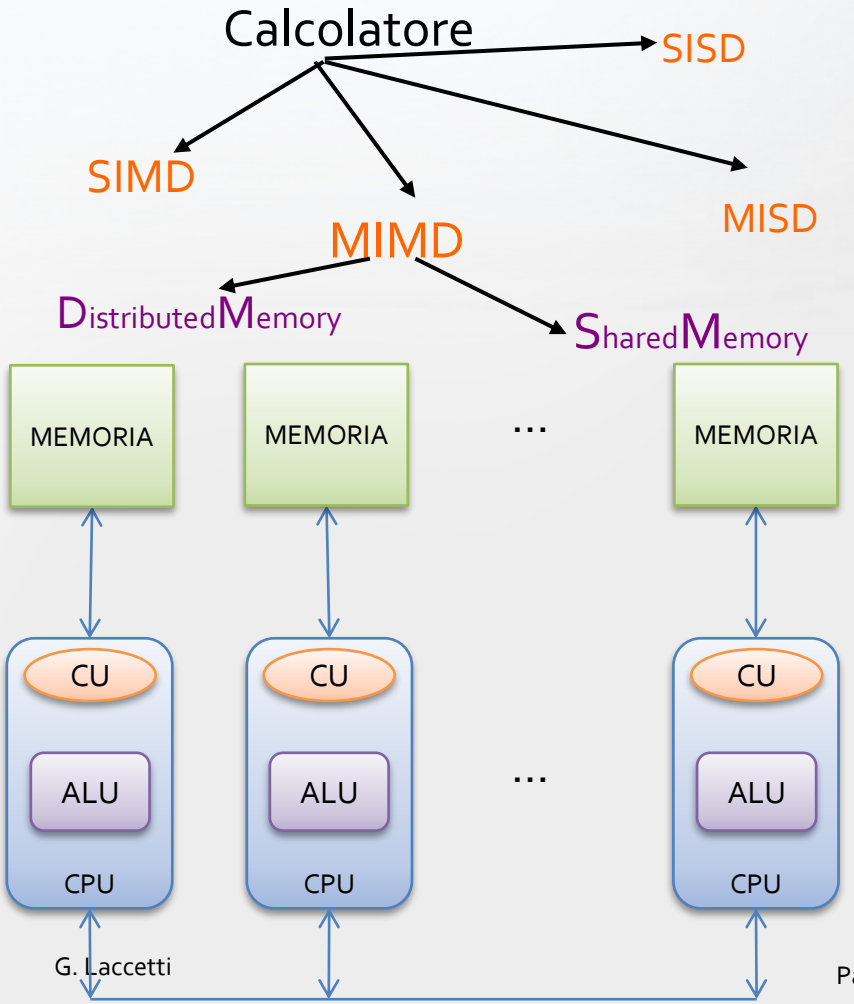
Data Stream: sequenza di dati richiesta dal flusso di istruzioni, compresi input e variabili d'appoggio.

[1]



- Single Instruction Stream / Single Data Stream (SISD)
- Single Instruction Stream / Multiple Data Stream (SIMD)
- Multiple Instruction Stream / Single Data Stream (MISD)
- Multiple Instruction Stream / Multiple Data Stream (MIMD)

Premesse
Tassonomia di Flynn



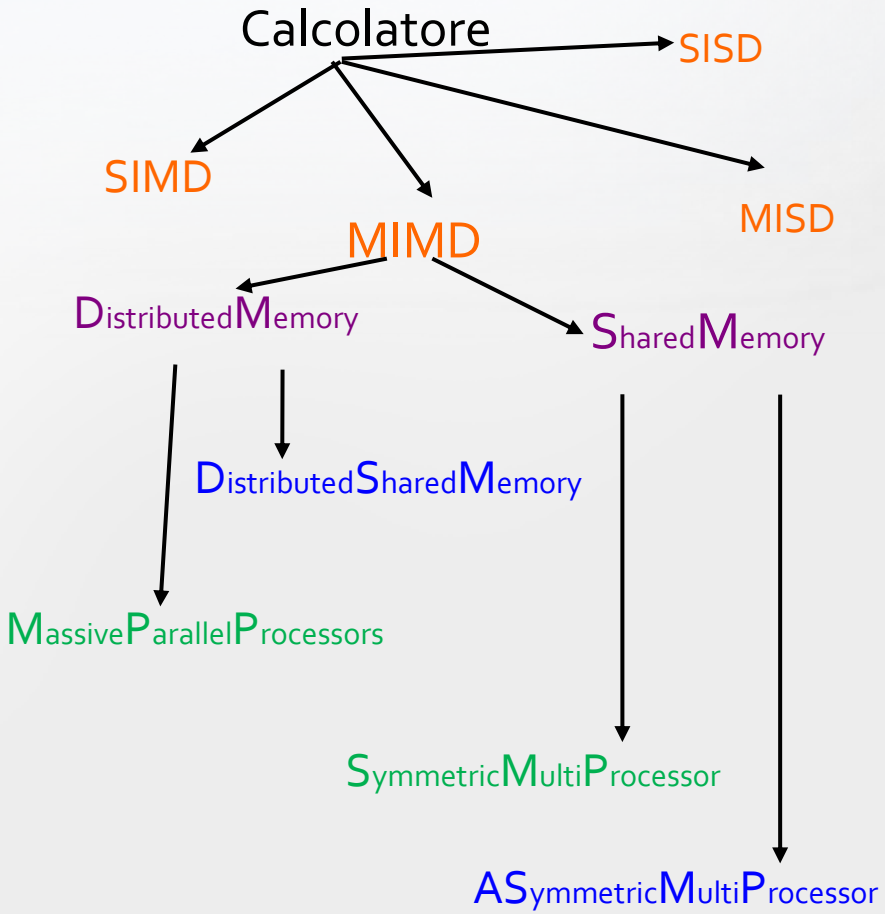
Instruction Stream: sequenza di istruzioni eseguite dalla macchina.

Data Stream: sequenza di dati richiesta dal flusso di istruzioni, compresi input e variabili d'appoggio.

[1]

- Single Instruction Stream / Single Data Stream (SISD)
- Single Instruction Stream / Multiple Data Stream (SIMD)
- Multiple Instruction Stream / Single Data Stream (MISD)
- Multiple Instruction Stream / Multiple Data Stream (MIMD)

Premesse
Tassonomia di Flynn



Instruction Stream: sequenza di istruzioni eseguite dalla macchina.

Data Stream: sequenza di dati richiesta dal flusso di istruzioni, compresi input e variabili d'appoggio.

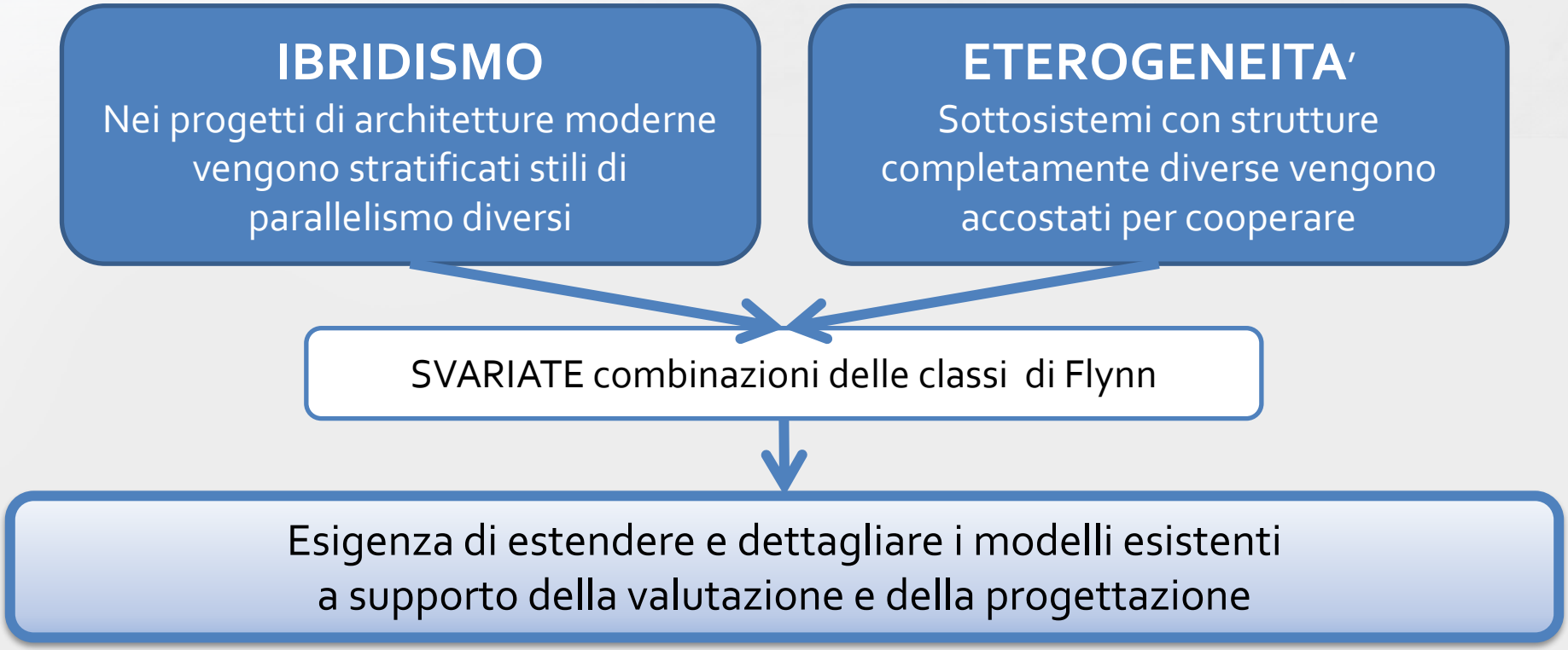
[1]

- Single Instruction Stream / Single Data Stream (SISD)
- Single Instruction Stream / Multiple Data Stream (SIMD)
- Multiple Instruction Stream / Single Data Stream (MISD)
- Multiple Instruction Stream / Multiple Data Stream (MIMD)

Premesse

Sistemi di calcolo moderni

- I sistemi attuali sono generalmente ibridi rispetto alla tassonomia di Flynn e molto spesso eterogenei



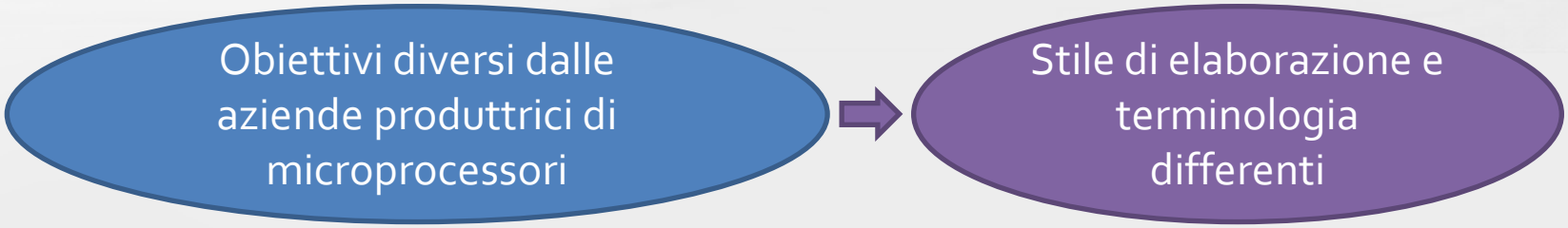
GPU
Evoluzione

- L'industria dei videogiochi ha investito moltissimo nel miglioramento dell'elaborazione grafica.
  - Nei primi anni '80 furono introdotte componenti specializzate nel rendering grafico per alleggerire il compito computazionale delle CPU

Obiettivi diversi dalle aziende produttrici di microprocessori → Stile di elaborazione e terminologia differenti

GPU
Evoluzione

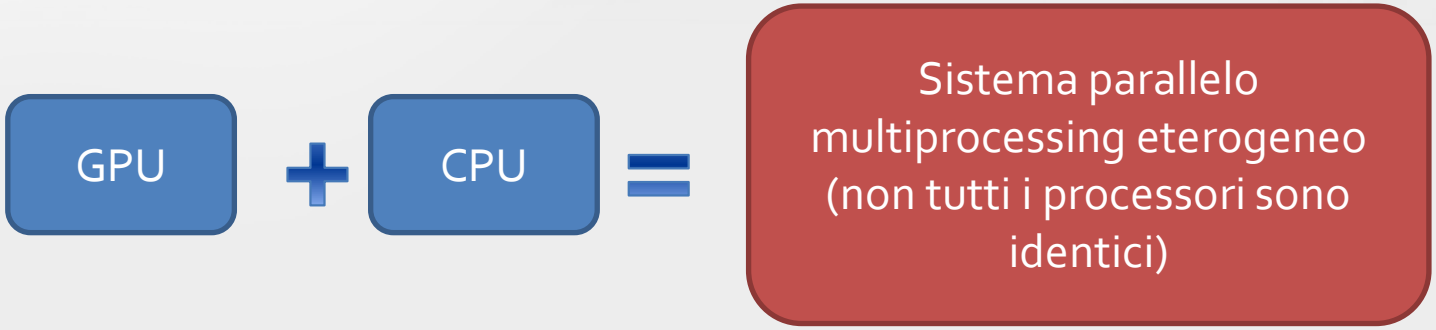
- L'industria dei videogiochi ha investito moltissimo nel miglioramento dell'elaborazione grafica.
  - Nei primi anni '80 furono introdotte componenti specializzate nel rendering grafico per alleggerire il compito computazionale delle CPU



  - PIPELINE GRAFICA: processo di **rendering** (generazione di un'immagine a partire da una descrizione di oggetti 3D) realizzato attraverso un hardware più o meno specializzato.

GPU
Evoluzione

- Le GPU, in quanto coprocessori ad integrazione della CPU, non hanno bisogno di eseguire tutti i compiti che sono eseguiti normalmente da una CPU.
  - Possono dedicare tutte le risorse alla grafica.
  - in un sistema che contiene sia una GPU che una CPU sarà la CPU ad eseguire i compiti per cui la GPU non è efficiente.

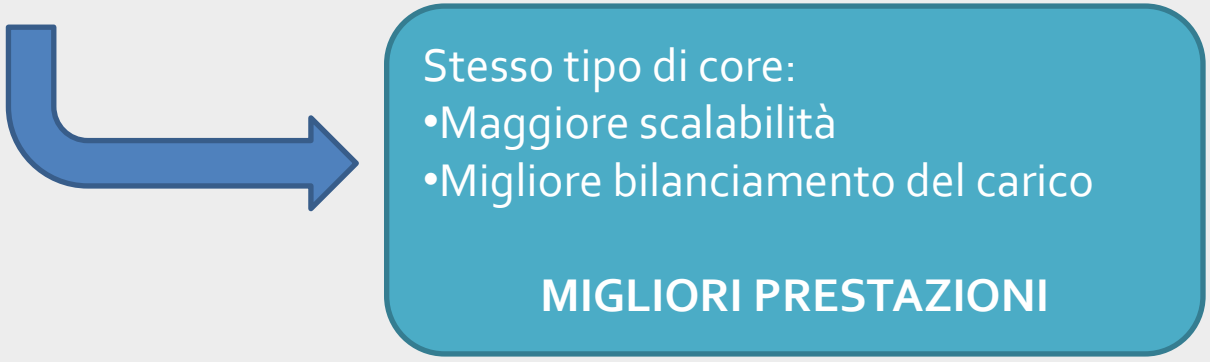


- Col tempo sono diventate sempre più programmabili e precise nei calcoli.

GPU
Evoluzione

- I dispositivi più moderni sono multiprocessori altamente paralleli e multithread, che nascondono l'elevata latenza di memoria sfruttando il parallelismo tra molti thread, per cui la memoria principale viene ottimizzata per massimizzare la larghezza di banda del trasferimento invece che per minimizzare la latenza.

- Vengono ormai progettate utilizzando processori di uso generico tra loro identici, diventando così più simili alle architetture multicore dei microprocessori convenzionali.



GPU
Evoluzione

- I dispositivi più moderni sono multiprocessori altamente paralleli e multithread, che nascondono l'elevata latenza di memoria sfruttando il parallelismo tra molti thread, per cui la memoria principale viene ottimizzata per massimizzare la larghezza di banda del trasferimento invece che per minimizzare la latenza.

- Vengono ormai progettate utilizzando processori di uso generico tra loro identici, diventando così più simili alle architetture multicore dei microprocessori convenzionali.

General Purpose GPUs
utilizzate per applicazioni generiche

GPU
Evoluzione

- I dispositivi più moderni sono multiprocessori altamente paralleli e multithread, che nascondono l'elevata latenza di memoria sfruttando il parallelismo tra molti thread, per cui la memoria principale viene ottimizzata per massimizzare la larghezza di banda del trasferimento invece che per minimizzare la latenza.

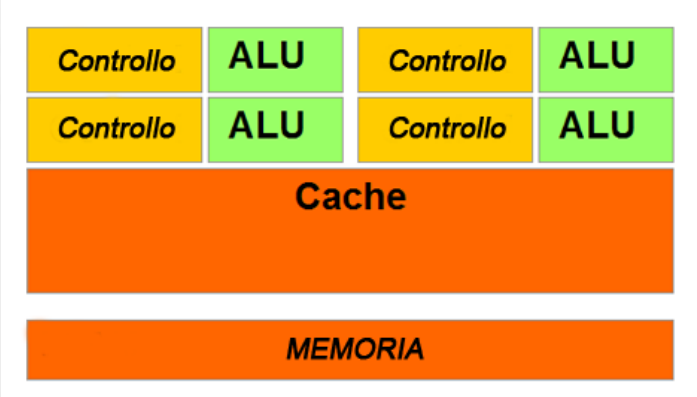
- Vengono ormai progettate utilizzando processori di uso generico tra loro identici, diventando così più simili alle architetture multicore dei microprocessori convenzionali.

Affiancate alla CPU
che mantiene il controllo
(da cui il termine
"acceleratore")

GPU vs CPU

CPU

- Molto spazio on-chip dedicato al controllo e alla cache
- Poche unità grafiche processanti, potenti e sofisticate
- Adatta a database, algoritmi ricorsivi, flussi di controllo non regolari



GPU

- Molte unità dedicate al calcolo
- Adatto a calcoli ripetitivi e su grandi quantità di dati
- La memoria on-chip evita il collo di bottiglia del bus di sistema

GPU vs CPU

Differenza secondo NVIDIA…

GPU vs CPU

Nella realtà…

- Non per tutti i problemi vale la superiorità della GPU
- Non per tutti i MODI di affrontare i problemi vale la superiorità della GPU
  - Anche nelle architetture più moderne la flessibilità non raggiunge quella delle CPU

- La cosa veramente importante è saper riconoscere il modo migliore di far collaborare le GPU e le CPU del sistema che abbiamo a disposizione.
  - Le GPU si utilizzano per ACCELERARE porzioni di codice con le caratteristiche giuste
  - Sono ormai disponibili in quasi tutti i sistemi, dai PC ai supercomputer, a costi relativamente contenuti
  - Il numero di core che forniscono continua a crescere

GPU
CUDA

- Con il GPGPU nascono anche linguaggi di alto livello adatti a programmare questi dispositivi.

- Alla fine degli anni 2000 NVIDIA introduce una particolare architettura di elaborazione in parallelo: **CUDA** (Compute Unified Device Architecture),
  - Con questa architettura nascono anche estensioni per i principali linguaggi di programmazione, soprattutto per C, che individuano un modello di programmazione parallela scalabile a memoria condivisa.
  - Attraverso CUDA/C lo sviluppatore può servirsi di alcune virtualizzazioni dell'architettura che semplificano molto la programmabilitàdella GPU.

- Nel 2008, AMD ed NVIDIA, insieme ad altre industrie del settore, hanno investito sulla realizzazione di OpenCL, linguaggio che segue la filosofia e i principi di CUDA/C ma concepito come *open-source*
  - dal 2009 OpenCL è supportato dai prodotti di entrambe le case

GPU
CUDA

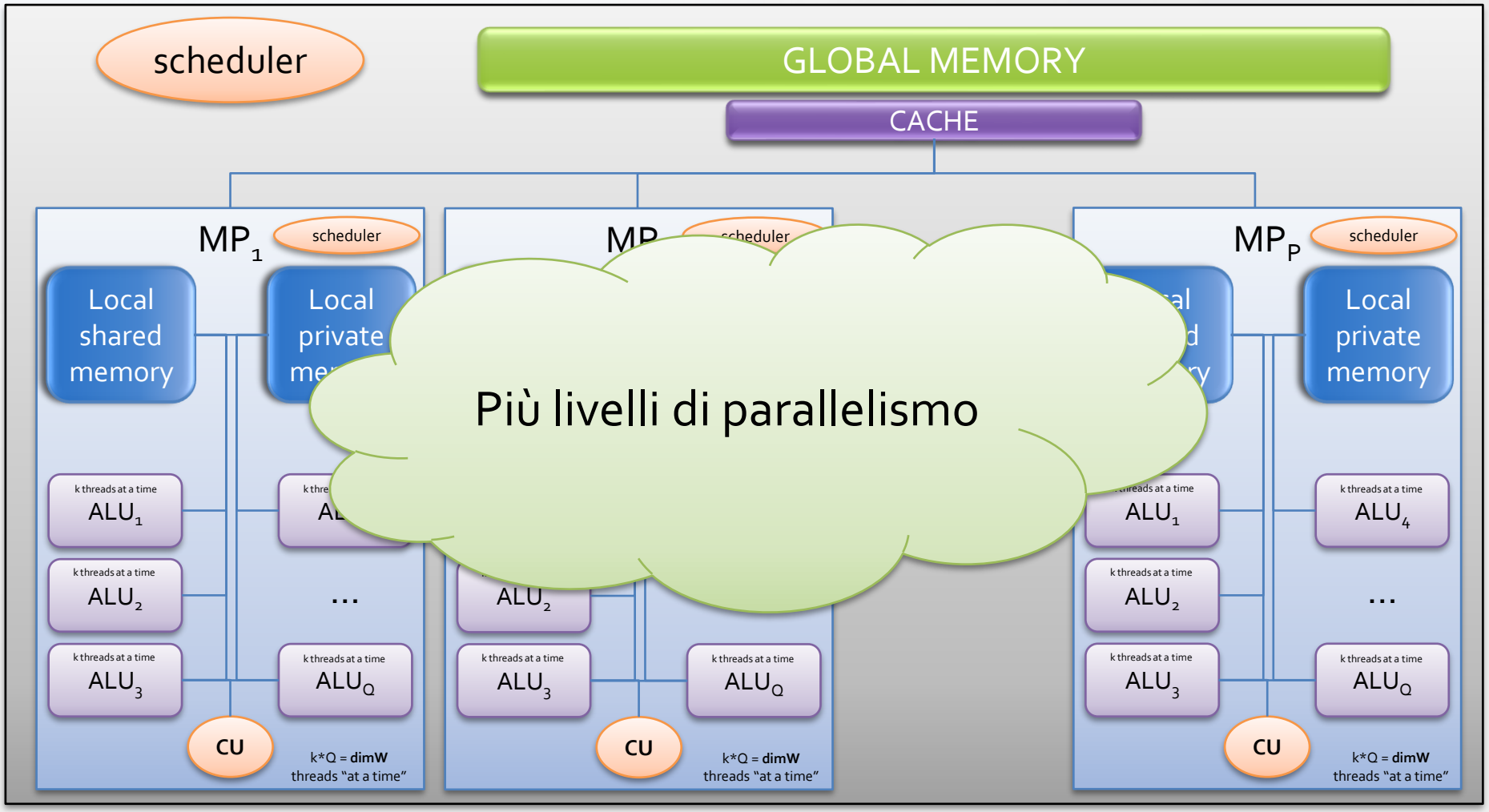
- Approfondire CUDA e CUDA/C è un buon modo per approcciare lo studio delle GPU.
  - L'architettura NVIDIA G80 ha segnato una tendenza nelle scelte progettuali, non radicalmente diverse da quelle fatte da AMD (ATI), che caratterizzano ormai questo tipo di dispositivi.
  - Il modello CUDA è applicabile anche ad altre architetture di elaborazione parallela a memoria condivisa, come le CPU multicore.
  - Le astrazioni introdotte da CUDA rispecchiano esattamente le caratteristiche dell'hardware delle GPU NVIDIA, che riuniscono più livelli di parallelismo: tali astrazioni guidano il programmatore nella suddivisione del problema e nella sincronizzazione del lavoro.

GPU
Architettura di una GPU

Cominciamo ora a modellare
un'architettura che rispecchi le caratteristiche delle più comuni GPU

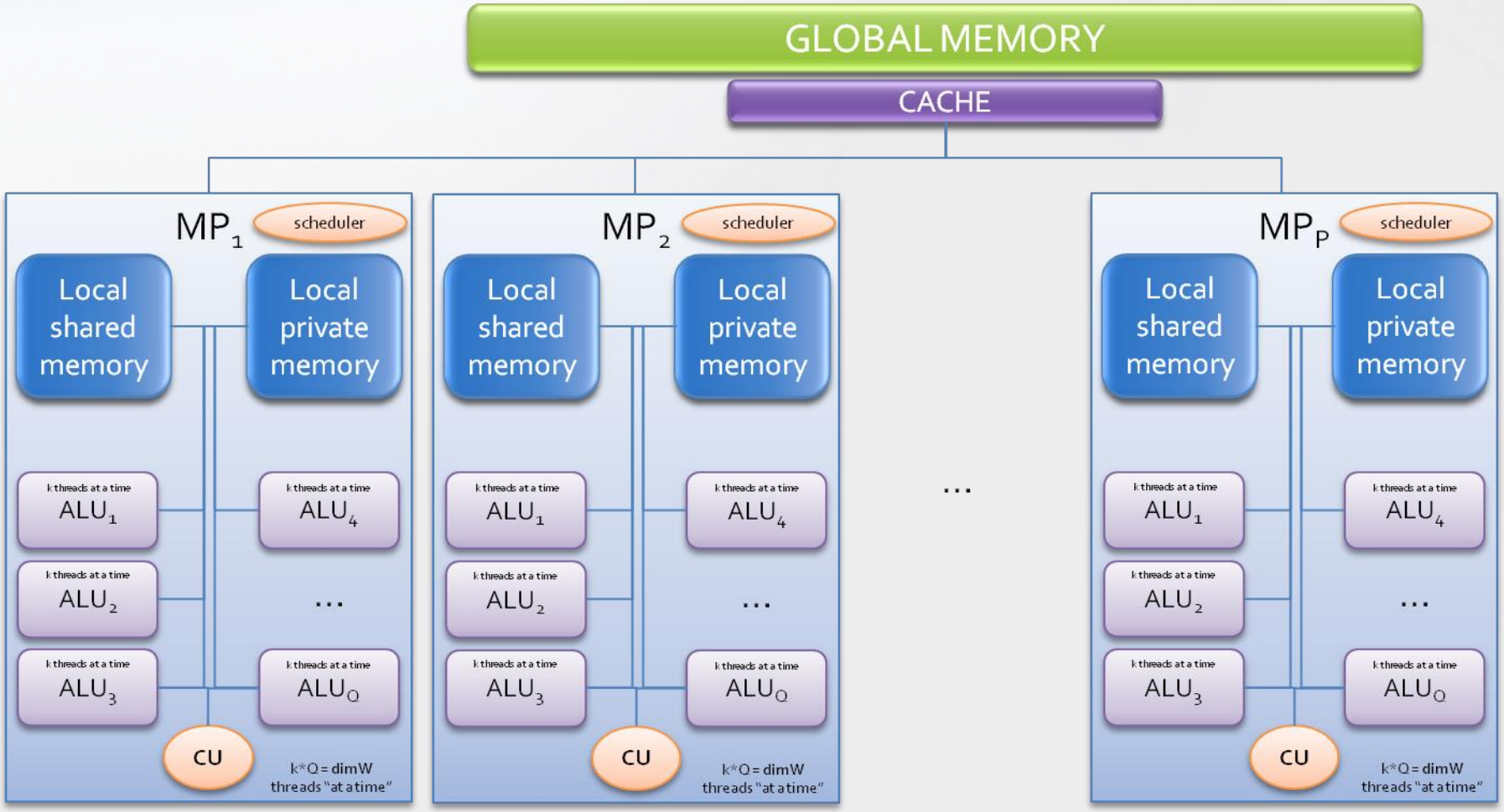
GPU

Architettura di una GPU: Più Livelli di Parallelismo



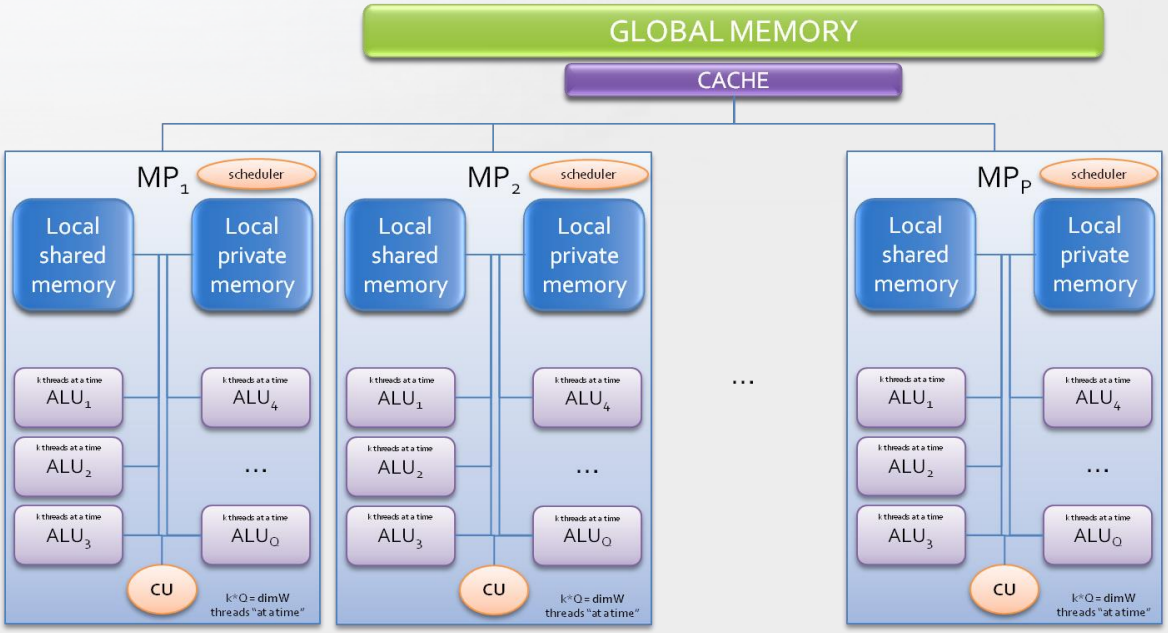
GPU

Architettura di una GPU: Più Livelli di Parallelismo



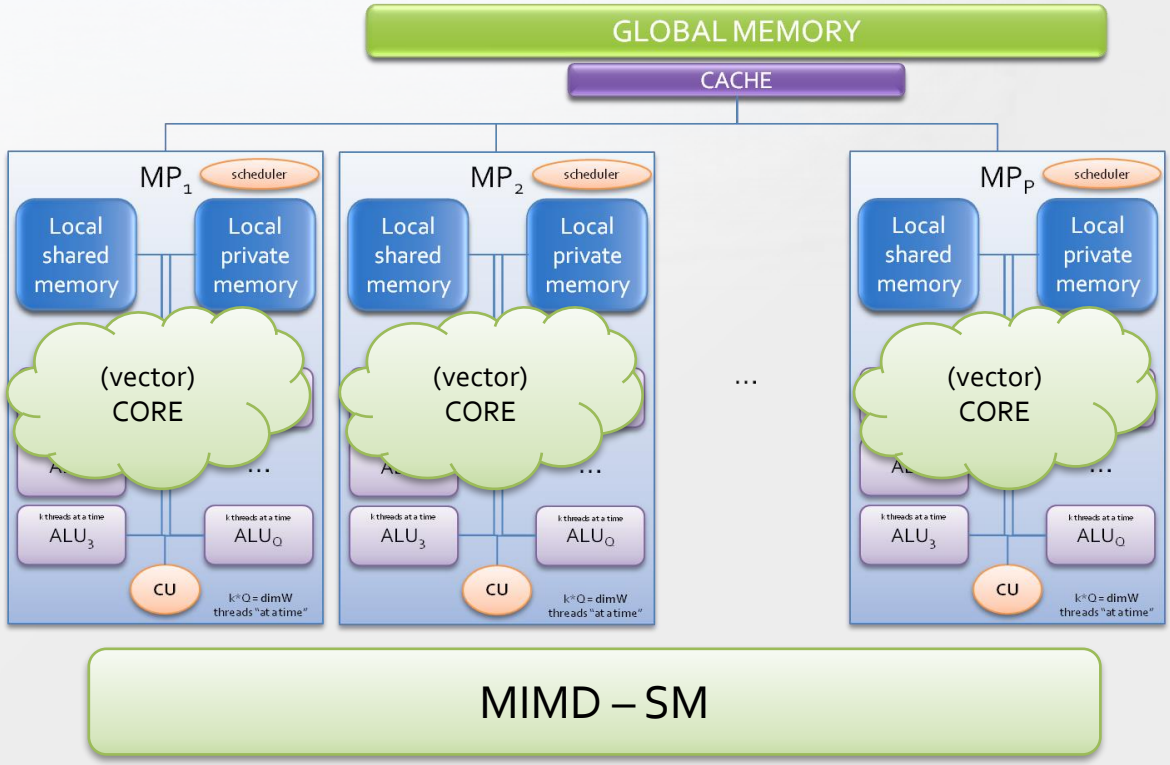
GPU

Architettura di una GPU: Più Livelli di Parallelismo



GPU

Architettura di una GPU: Più Livelli di Parallelismo



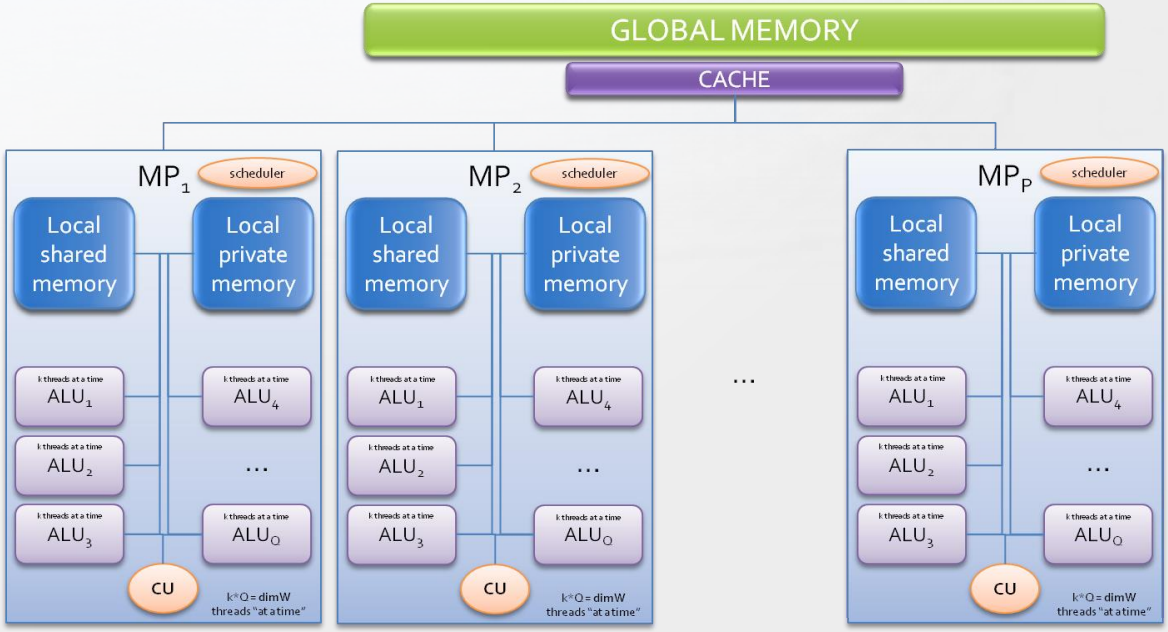
Livello più alto

P core che condividono una memoria (con eventuale cache):

macchina MIMD shared memory.

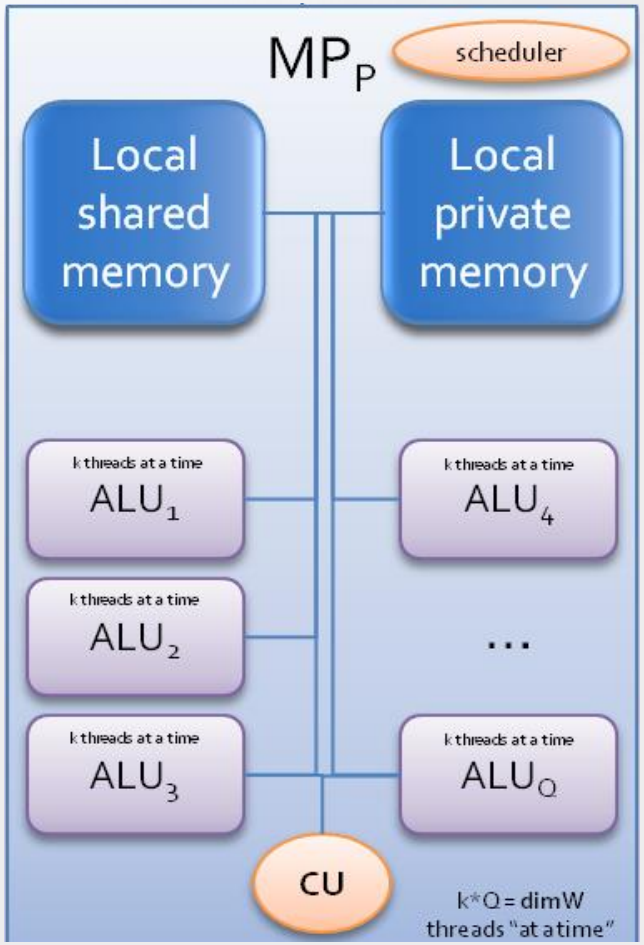
GPU

Architettura di una GPU: Più Livelli di Parallelismo



GPU

Architettura di una GPU: Più Livelli di Parallelismo



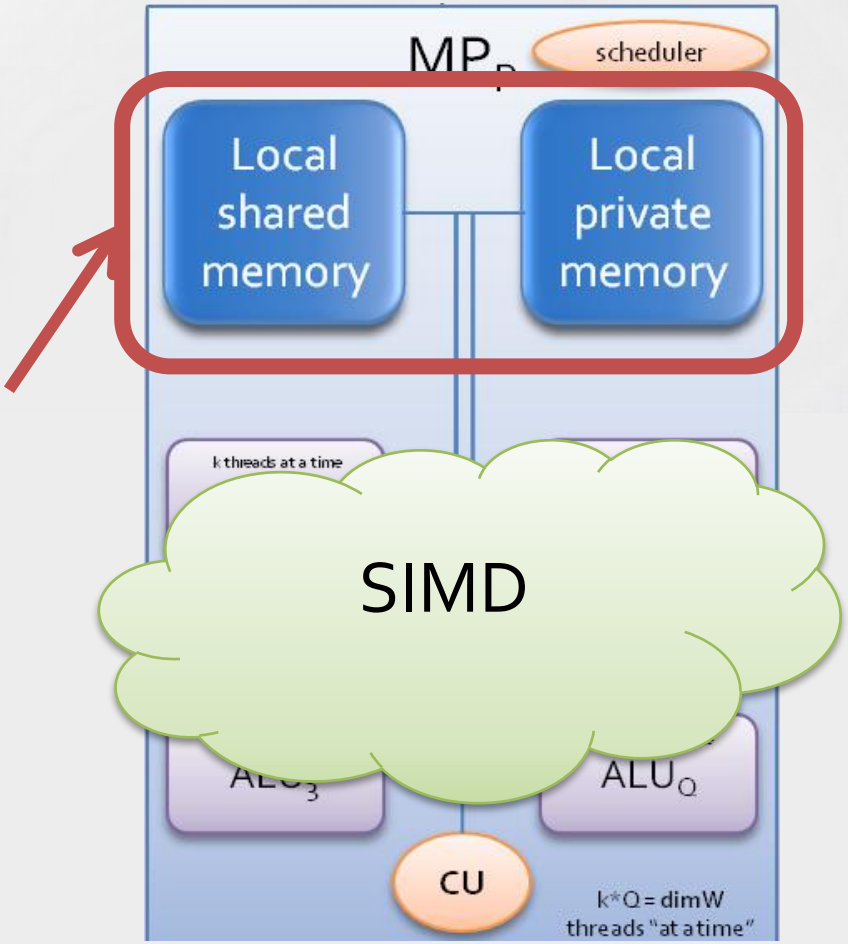
GPU

Architettura di una GPU: Più Livelli di Parallelismo

Livello intermedio

Ogni MP è un calcolatore SIMD con **Q** ALU,

Ogni MP ha una sua area di memoria locale



GPU

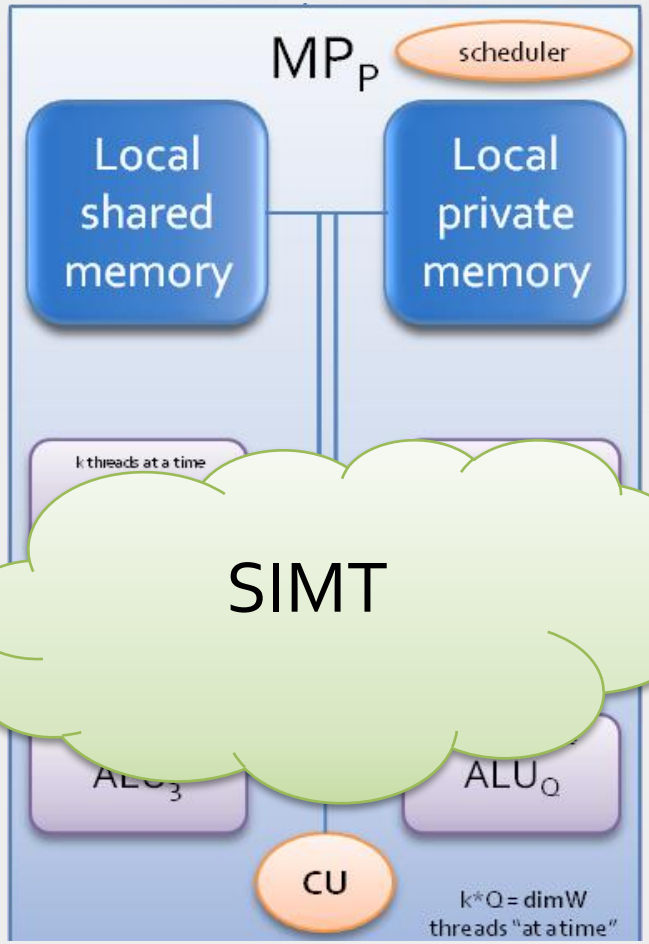
Architettura di una GPU: Più Livelli di Parallelismo

Livello intermedio

Ogni MP è un calcolatore SIMD con **Q** ALU,

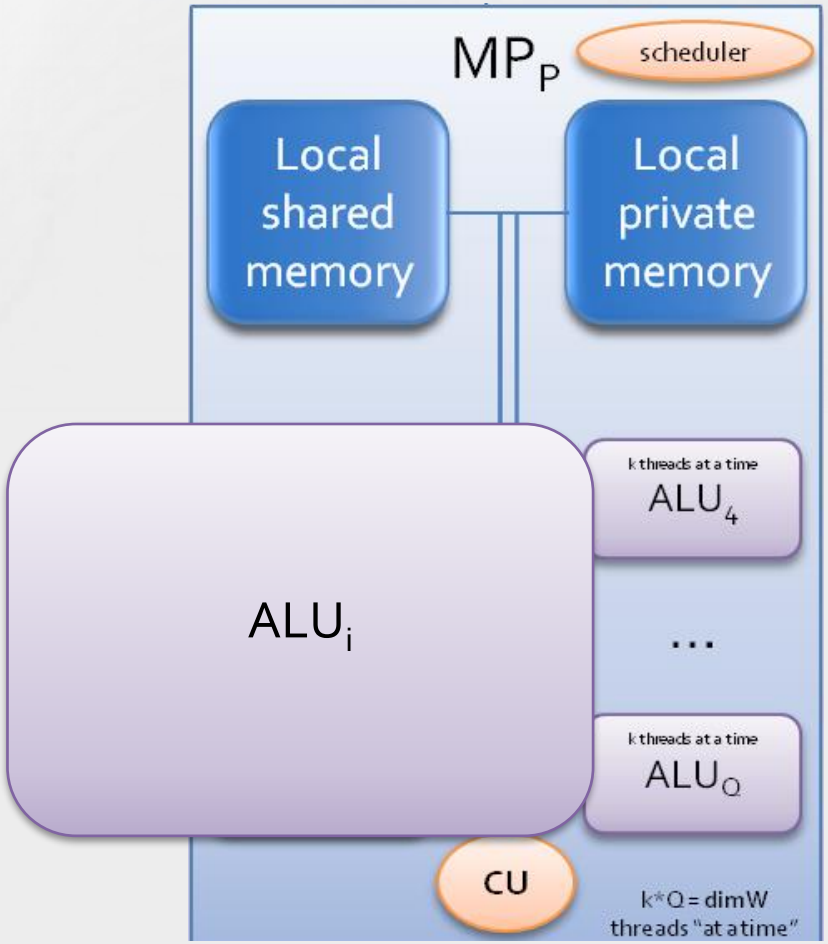
Commercialmente viene chiamata **SIMT**
Ovvero
Single Instruction Multiple Threads

Questo perché nella realizzazione fisica non si tratta di ALU ma di piccole e semplici CPU complete, e i thread possono avere comportamenti leggermente diversi tra loro (sono ammessi i branch) : eseguono la stessa porzione di codice, ma non sempre ESATTAMENTE le stesse istruzioni.



GPU

Architettura di una GPU: Più Livelli di Parallelismo

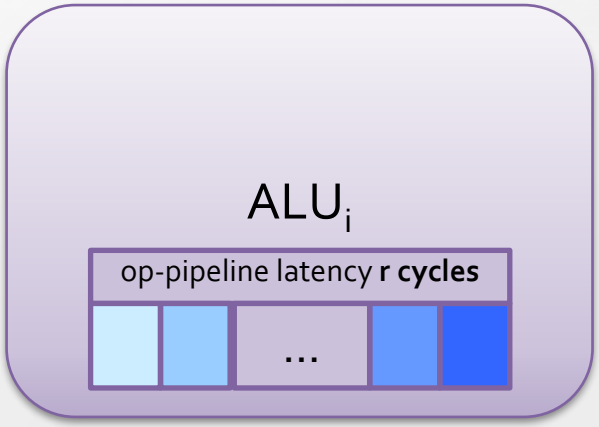


GPU

Architettura di una GPU: Più Livelli di Parallelismo

GPU

Architettura di una GPU: Più Livelli di Parallelismo



Livello più basso

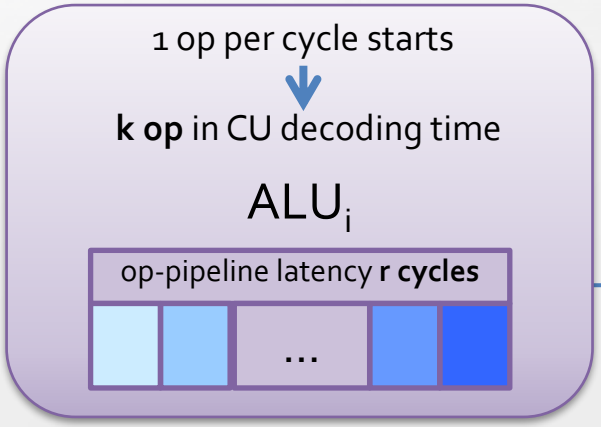
Ogni ALU è pipelined

latenza di pipeline di **r** cicli.

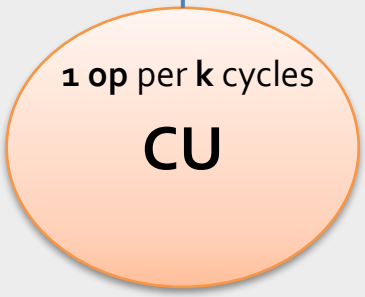
GPU

Quanti thread simultanei?

La CU dell'MP impiega **k<r** cicli per la decodifica di un'istruzione.
(Consideriamo che a un'istruzione corrisponda un'operazione).



Nella pipeline di ogni ALU può entrare 1 operazione per ciclo.

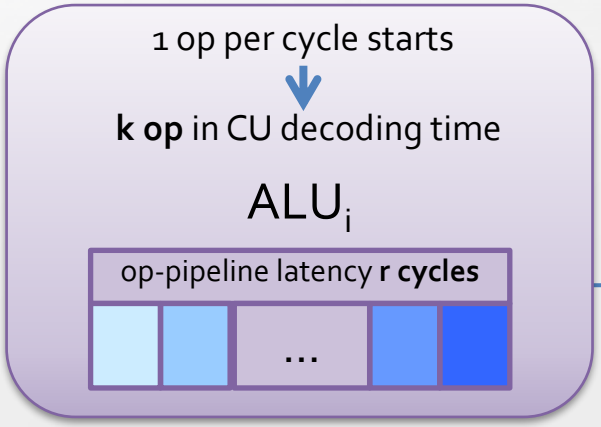


per ogni istruzione decodificata possono partire **k** operazioni

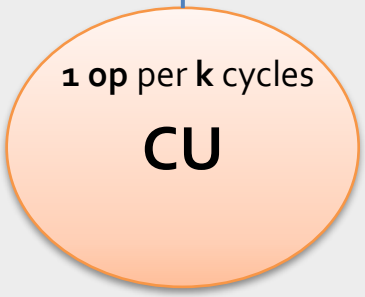
GPU
Quanti thread simultanei?

Se la CU impiega **k cicli a decodificare**, ogni k cicli entra in pipeline un'istruzione diversa.
Se ad ogni ciclo può partire un'istruzione in pipeline, allora tra una decodifica e l'altra escono dalla pipeline (**terminano**) k istruzioni già inserite.
E' come aver eseguito contemporaneamente k thread.

La CU dell'MP impiega **k<r** cicli per la decodifica di un'istruzione.
(Consideriamo che a un'istruzione corrisponda un'operazione).



Nella pipeline di ogni ALU può entrare 1 operazione per ciclo.

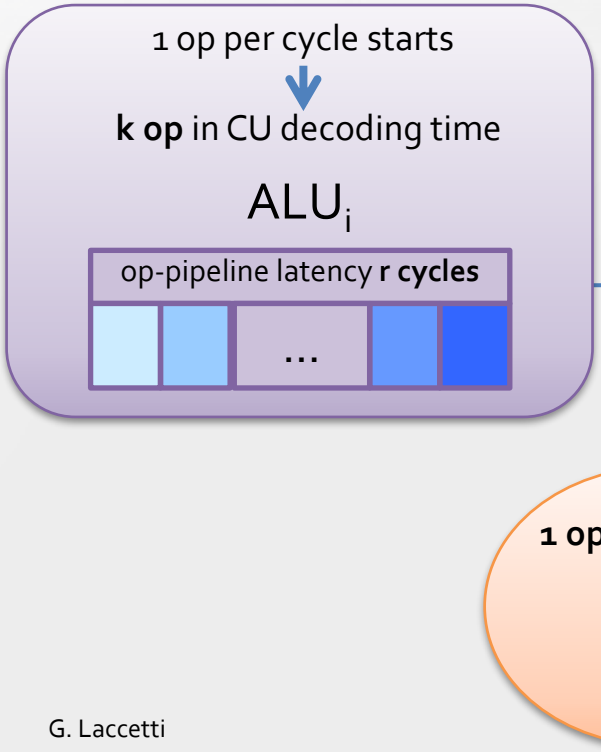


per ogni istruzione decodificata possono partire **k** operazioni

GPU
Quanti thread simultanei?

k si suppone minore di r (numero di cicli perché la pipeline vada a regime) perché se la CU ci mette più della latenza di pipeline a decodificare, la pipeline non è mai a regime.

La CU dell'MP impiega **k<r** cicli per la decodifica di un'istruzione.
(Consideriamo che a un'istruzione corrisponda un'operazione).

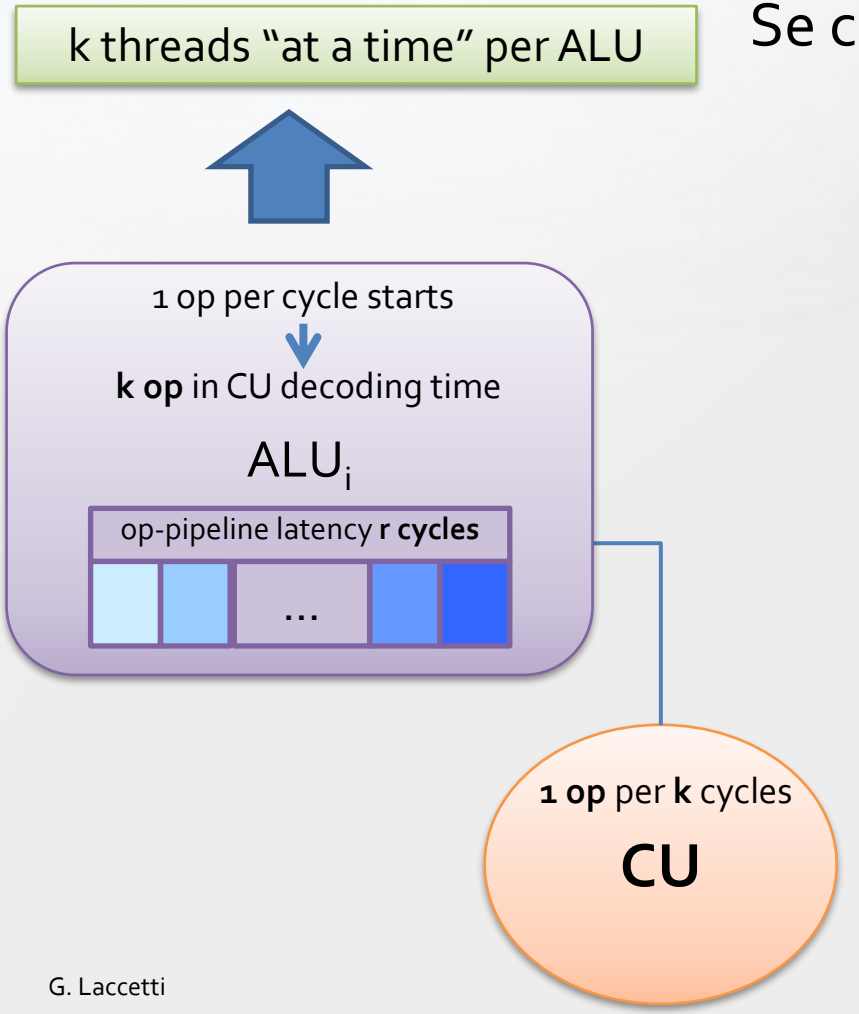


Nella pipeline di ogni ALU può entrare 1 operazione per ciclo.

per ogni istruzione decodificata possono partire **k** operazioni

GPU

Quanti thread simultanei?



Se consideriamo che ogni operazione sia svolta da un thread diverso

1 operazione ⟶ 1 thread

per ogni istruzione decodificata
k thread eseguono l'operazione corrispondente
su ogni ALU

GPU
Quanti thread simultanei?

GPU
Quanti thread simultanei?

GPU

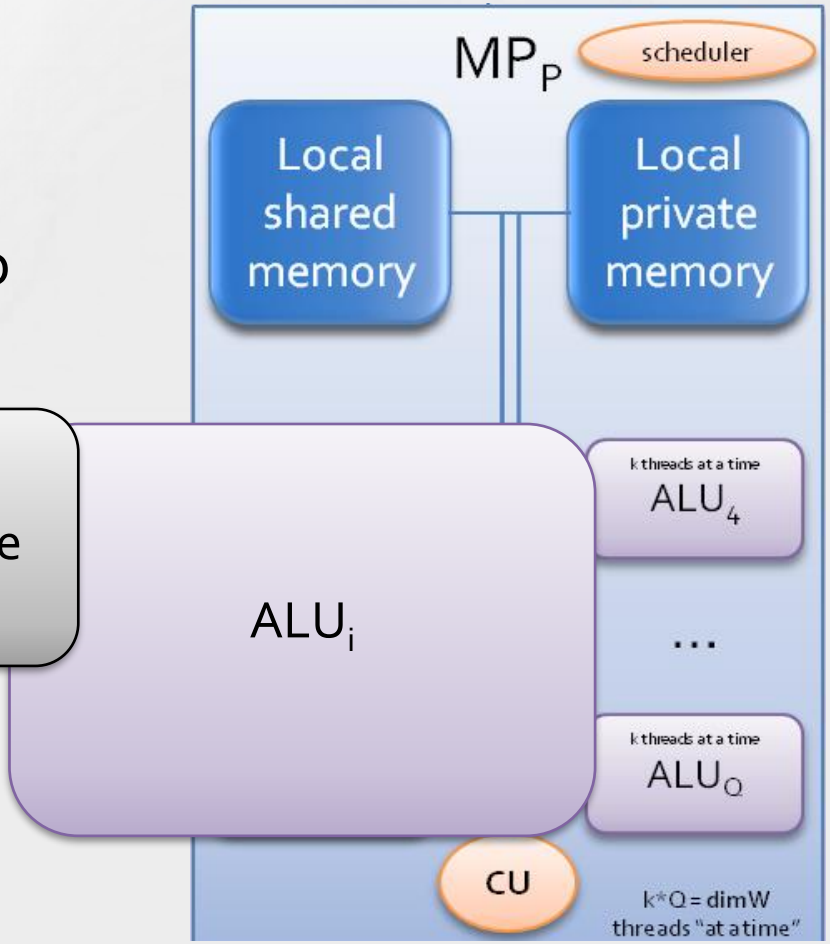
Quanti thread simultanei?

Ogni MP ha Q ALU

Per ogni istruzione sull'MP partono **k*Q** thread

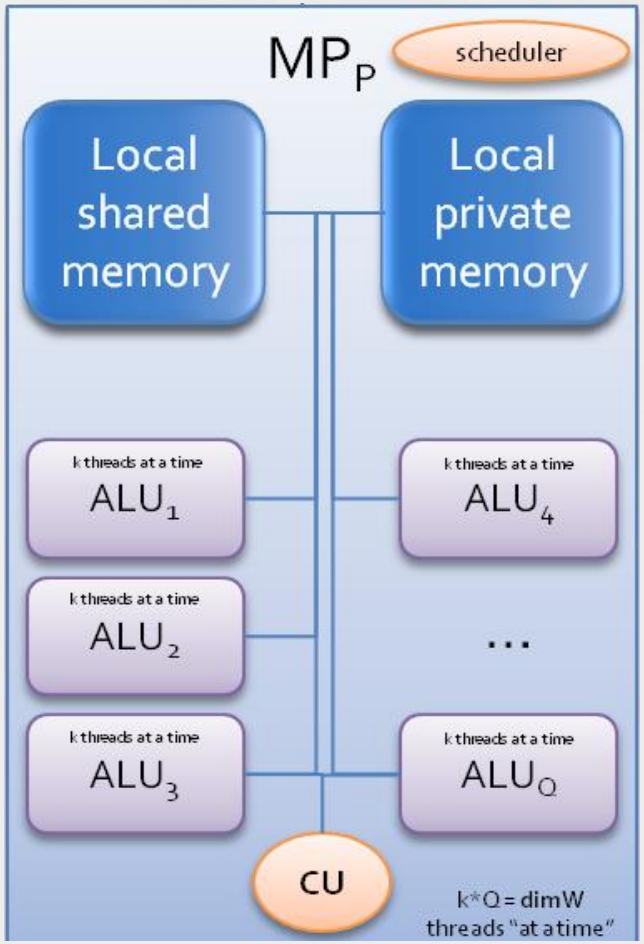
Warp

gruppo di **k*Q** thread simultaneamente eseguiti da un MP. **k*Q=dimW**



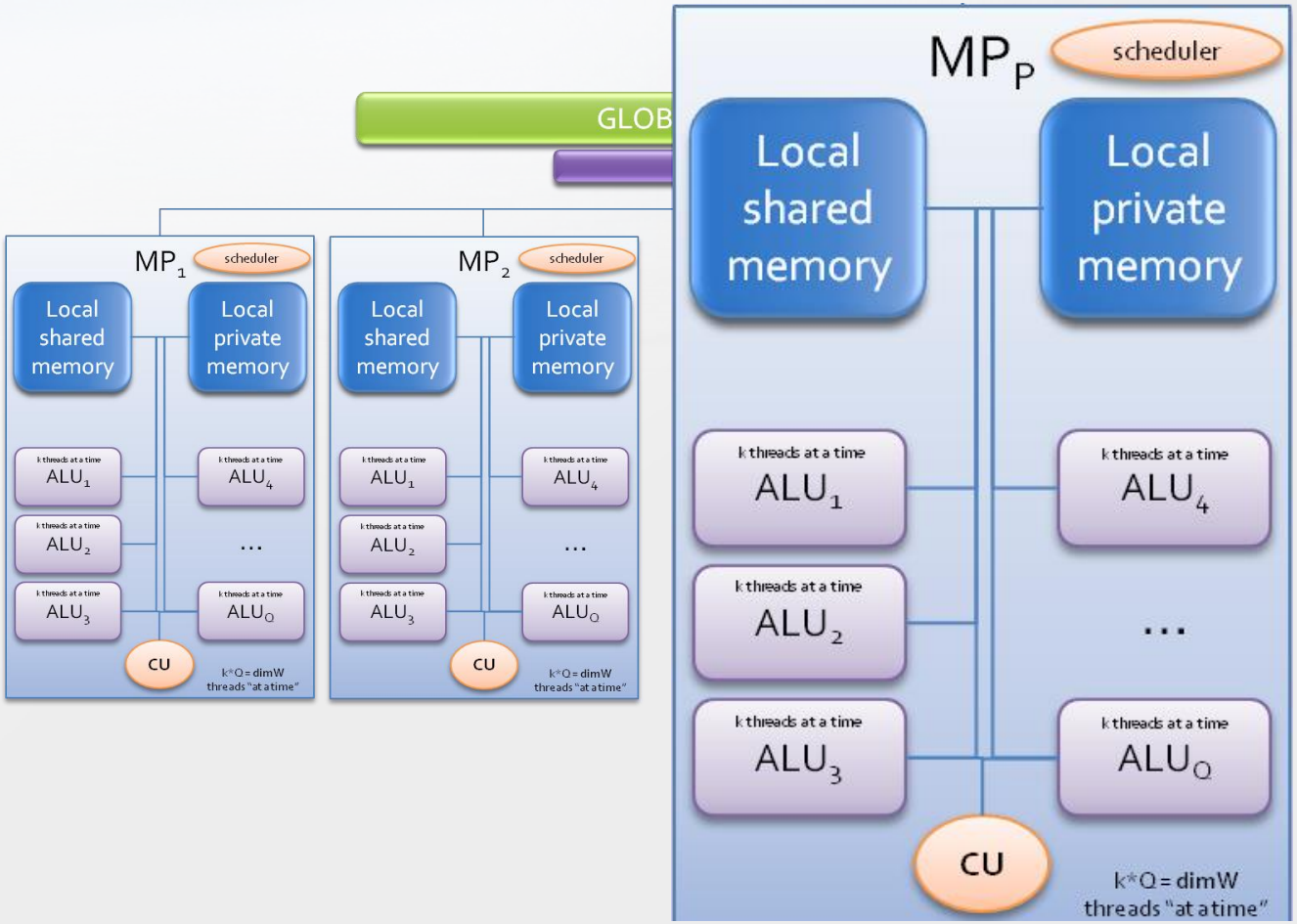
GPU

Quanti thread simultanei?



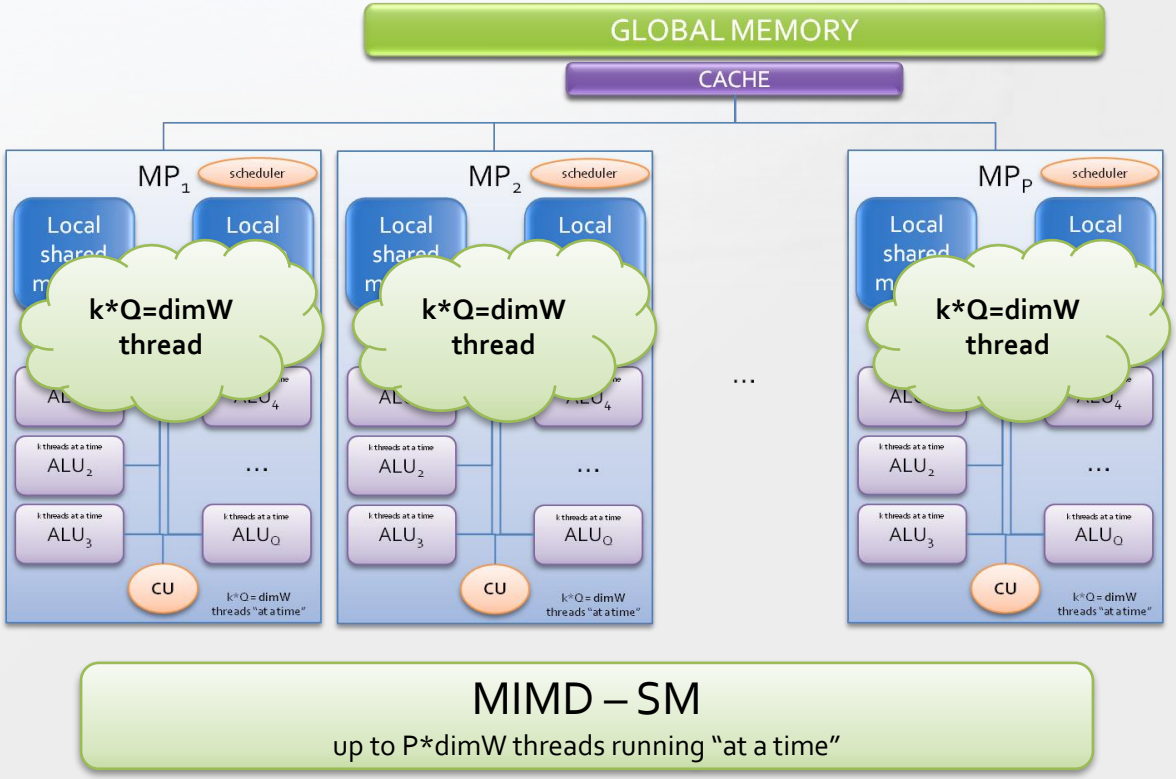
GPU

Quanti thread simultanei?



GPU

Quanti thread simultanei?



MIMD – SM

up to P*dimW threads running "at a time"

Ogni GPU ha P MP

Per ogni istruzione sulla GPU partono fino a **P*dimW=P*k*Q** thread conteporaneamente

GPU

Tempo di calcolo

P core con Q unità
Pipeline con k segmenti
k thread per ogni unità

Ogni thread esegue istruzioni tutte dipendenti tra loro.

Algoritmo puramente parallelo
T tempo di esecuzione sequenziale

IDEALE!!

Tempo d'esecuzione algoritmo per GPU

$$T_{GPU}(P \cdot Q \cdot k, N) = \frac{T(N)}{P \cdot Q \cdot k} \cdot t_{calc}$$

N dimensione del problema

A questo tempo va aggiunto quello di accesso alla memoria

GPU

Tempo di calcolo

P core con Q unità
Pipeline con k segmenti
k thread per ogni unità

Ogni thread esegue istruzioni tutte dipendenti tra loro.

Algoritmo puramente parallelo
T_c tempo di esecuzione concorrente

IDEALE!!

Tempo d'esecuzione algoritmo per GPU

$$T_{GPU}(P \cdot Q \cdot k, N) = \frac{T_c(N)}{P \cdot Q \cdot k} \cdot t_{calc}$$

N dimensione del problema

Lanciando l'esecuzione di un numero di thread molto maggiore di P*Q*k (possibilmente multiplo) per la **stessa dimensione N,** il tempo di calcolo teoricamente aumenta, ma quello d'esecuzione totale **NO**: si copre la latenza di memoria

GPU

Tempo di calcolo

P core con Q unità
Pipeline con k segmenti
k thread per ogni unità

Ogni thread esegue istruzioni tutte dipendenti tra loro.

Algoritmo puramente parallelo
T_c tempo di esecuzione concorrente

IDEALE!!

Tempo d'esecuzione algoritmo per GPU

$$T_{GPU}(P \cdot Q \cdot k, N) = \frac{T_c(N)}{P \cdot Q \cdot k} \cdot t_{calc}$$

N dimensione del problema

Lanciando l'esecuzione di un numero di thread molto maggiore di P*Q*k (possibilmente multiplo) **aumentando** proporzionalmente **la dimensione N,** il tempo di calcolo teoricamente aumenta, ma quello d'esecuzione totale **NO**: si copre la latenza di memoria

GPU

Tempo di calcolo

P core con Q unità
Pipeline con k segmenti
k thread per ogni unità

Ogni thread esegue istruzioni tutte dipendenti tra loro.

Algoritmo puramente parallelo
T_c tempo di esecuzione concorrente

IDEALE!!

Tempo d'esecuzione algoritmo per GPU

$$T_{GPU}(P \cdot Q \cdot k, N) = \frac{T_c(N)}{P \cdot Q \cdot k} \cdot t_{calc}$$

N dimensione del problema

Risolviamo un problema di dimensione maggiore nello stesso tempo d'esecuzione
Cioè le possibilità di **scalabilità** sono maggiori di quello che ci si aspetterebbe dal numero di unità

GPU

Tempo di calcolo

P core con Q unità
Pipeline con k segmenti
k thread per ogni unità

Ogni thread esegue istruzioni tutte dipendenti tra loro.

Algoritmo puramente parallelo
T_c tempo di esecuzione concorrente

IDEALE!!

Tempo d'esecuzione algoritmo per GPU

$$T_{GPU}(P \cdot Q \cdot k, N) = \frac{T_c(N)}{P \cdot Q \cdot k} \cdot t_{calc}$$

N dimensione del problema

ASSUNZIONE FONDAMENTALE: Algoritmo perfettamente parallelo

Il carico deve essere molto ben bilanciato tra le unità: ottime performance in applicazioni che prevedono le stesse operazioni su dati diversi (come appunto il rendering)

Compilare ed eseguire sorgente CUDA

- Per compilare basta digitare nella shell:

nvcc -o *nome-eseguibile* nome-codice.cu

- Una volta compilato il codice, basta lanciare l'eseguibile come di consueto

./nome-eseguibile

Installare CUDA in locale

- Recarsi a questa pagina e seguire le istruzioni a video:

https://developer.nvidia.com/cuda-downloads

- Tutta la documentazione CUDA è disponibile a questa pagina

https://docs.nvidia.com/cuda/