Programmazione Parallela ed Eterogenea

Programmazione Avanzata e Parallela
2022/2023

Alberto Casagrande

La Società dell'Informazione

La tecnologia ci consente di raccogliere grandi quantità di dati

Negli ultimi 5 anni anni abbiamo prodotto più dati di quanto fatto nei precedenti 50 anni

- Smart Devices e IOT
- Sequenziatori
- ...

Abbiamo bisogno di computer più potenti

L'Architettura di Von Neumann

I computer sono dotati di:

- un unità di calcolo
- una memoria per memorizzare
  - i programmi
  - i dati su cui i programmi operano

La CPU

È l'unità centrale di elaborazione

Essa è dotata di:

- **registri** per memorizzare/operare su istruzioni e dati
- **unità algebrico-logiche (ALU)** per svolgere operazioni
- **unità di controllo (CU)** per gestire/organizzare le operazioni

Il *clock* sincronizza le sue componenti e le operazioni svolte

Ciclo Fetch-Decode-Excute

La CPU ripete ciclicamente tre operazioni di base:

- **Fetch:** preleva un'istruzione dalla memoria e la carica in un registro
- **Decode:** decodifica l'istruzione e preleva i dati dalla memoria
- **Execute:** esegue l'istruzione

Aumentare la Potenza di Calcolo

Tre strade:

1. **Aumentare la frequenza di *clock***
    - aumenta il calore prodotto dal processore
    - genera disturbi elettronici
2. **Aumentiamo l'efficienza aggiungiendo transistor** (legge di Moore)
    - transistor troppo piccoli
    - processori troppo grandi
3. **Esecuzione Parallela**

Concorrenza vs Parallelismo

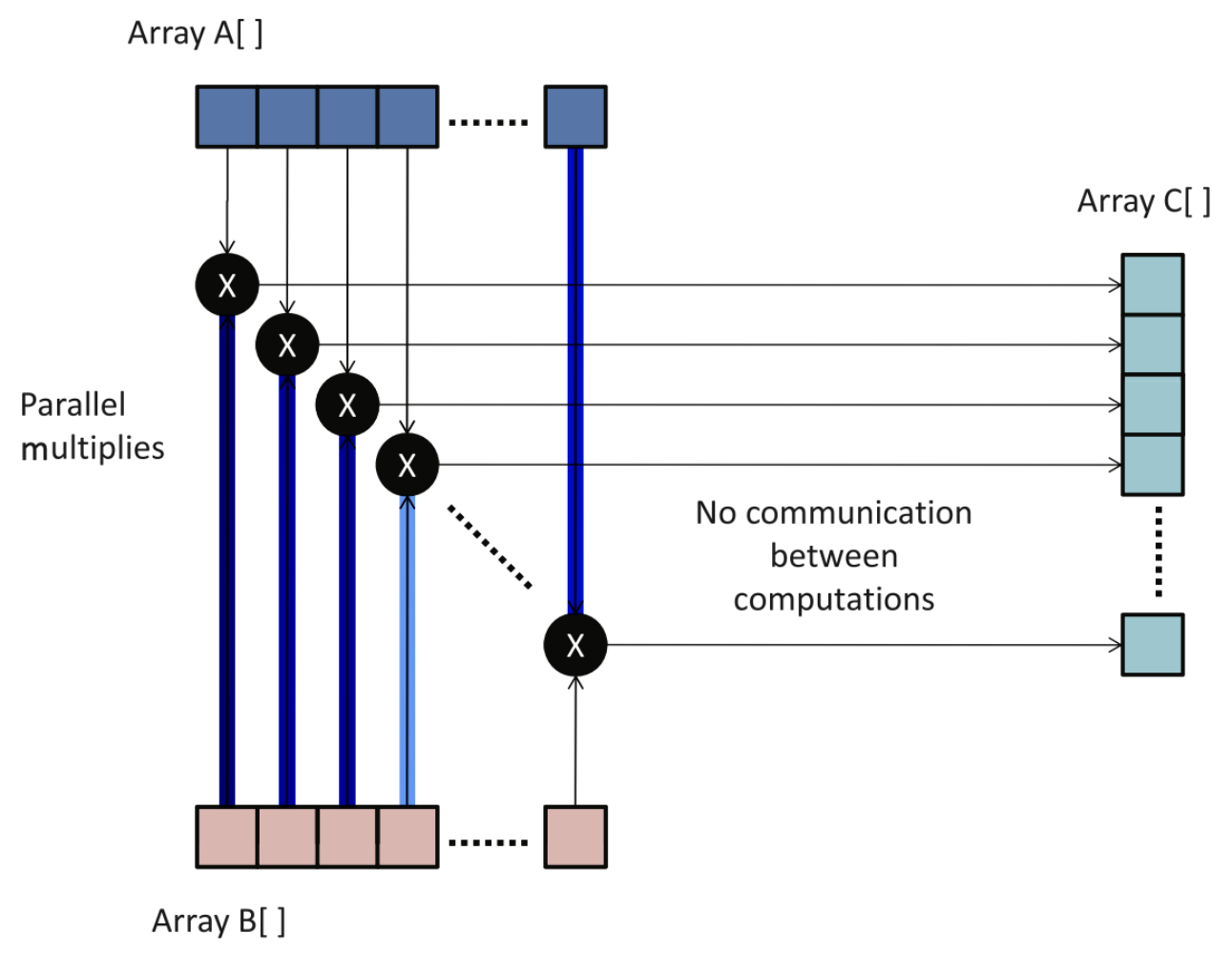
I **programmi concorrenti** hanno esecuzioni indipendenti e possono concorrere per una stessa risorsa, e.g., due programmi che cercano di stampare

I **programmi paralleli** sono eseguiti contemporaneamente

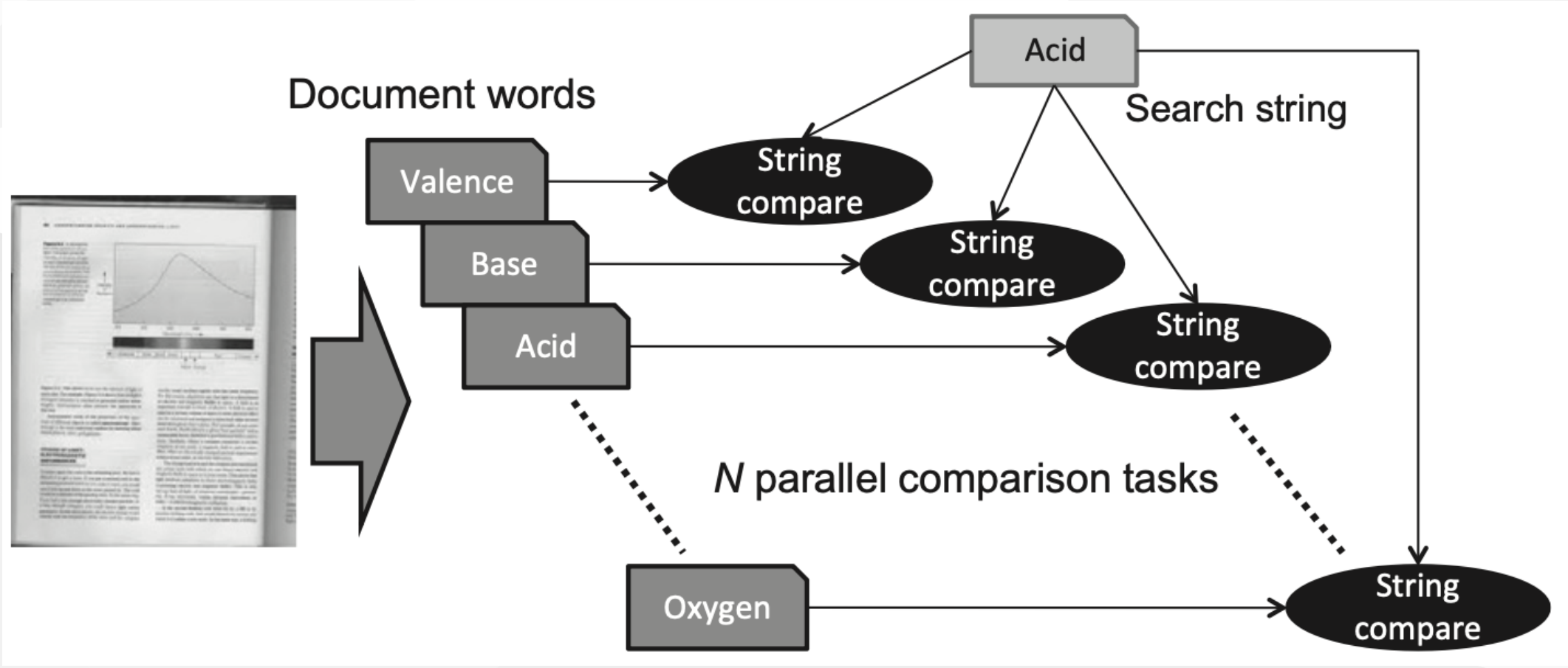
Parallelismo $\Rightarrow$ **Concorrenza**

Concorrenza $\not\Rightarrow$ **Parallelismo**

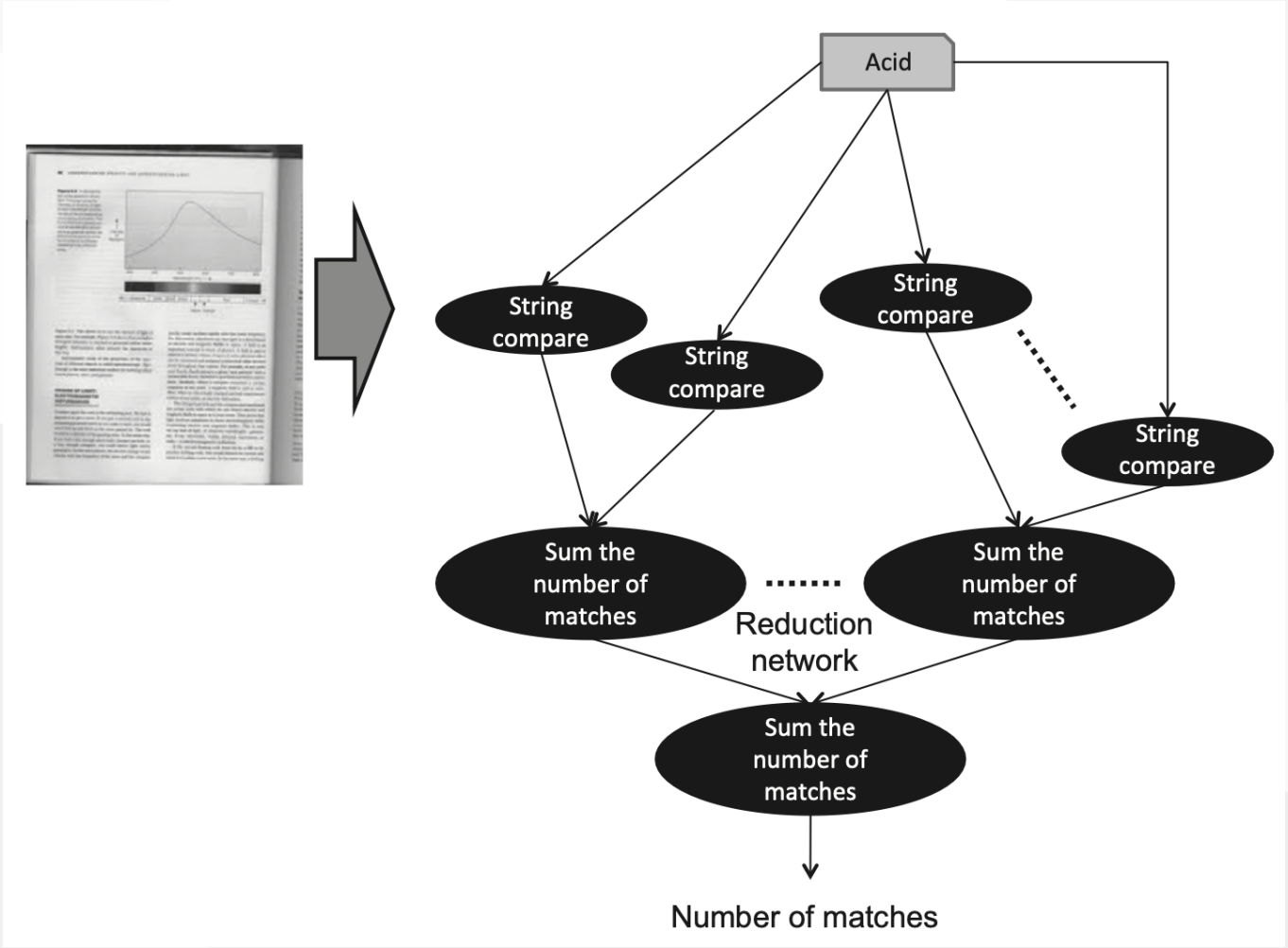
Un Esempio di Programma Parallelo



Un Esempio Più Complicato



Un Esempio Più Complicato (Cont'd)



Memoria Condivisa vs Comunicazione

Le computazioni parallele devono scambiarsi informazioni

- **Memoria Condivisa:**
  - meccanismi di sincronizzazione
  - principio di località
- **Comunicazioni:**
  - gestione esplicita della comunicazione (e.g., *MPI*)
  - consente hardware distribuito
- **Memoria Virtuale Condivisa**
  - semplifica la gestione della comunicazione

Granularità di Parallelismo (*Chuncking*)

È la quantità di lavoro indipendente svolto da ciascuna unità parallela

Dipende dall'algoritmo da implementare

- **Granularità Fine:**
  - basso carico per unità
  - richiede un basso costo di comunicazione
- **Granularità Grossolana:**
  - alto carico per unità
  - non facilmente automatizzabile

Architetture Parallele e Concorrenti

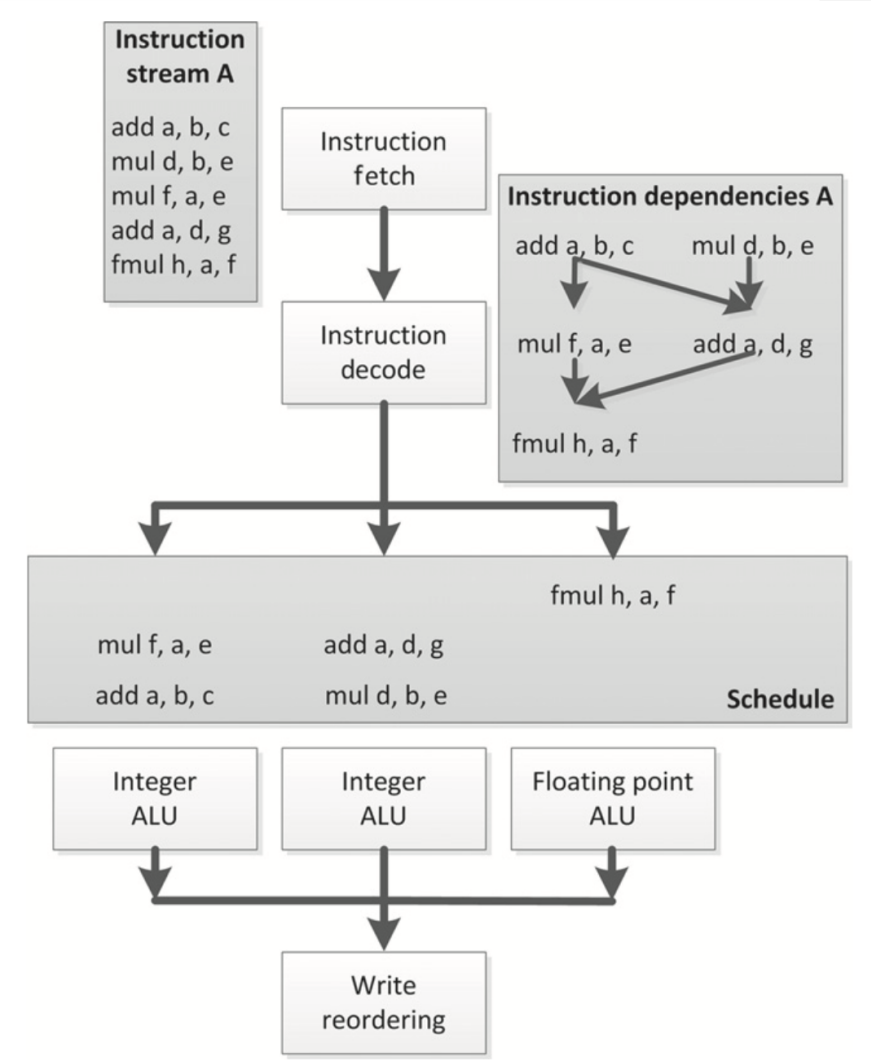
Su quale tipo di macchina posso eseguire un programma parallelo

Ogni architettura avrà i suoi vantaggi e svantaggi

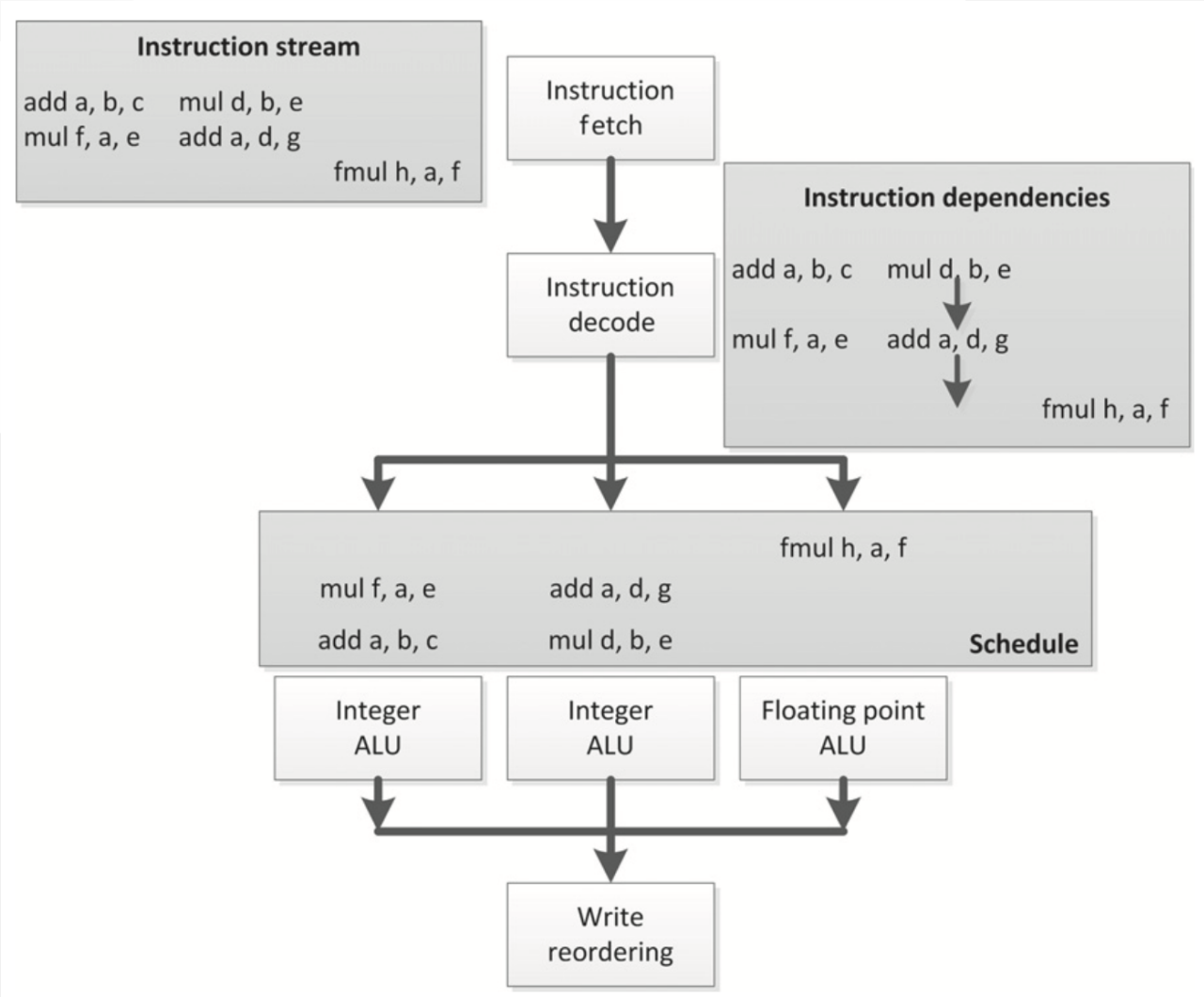
È importante conoscerle per:

- sviluppare algoritmi paralleli
- scegliere la granularità più adatta

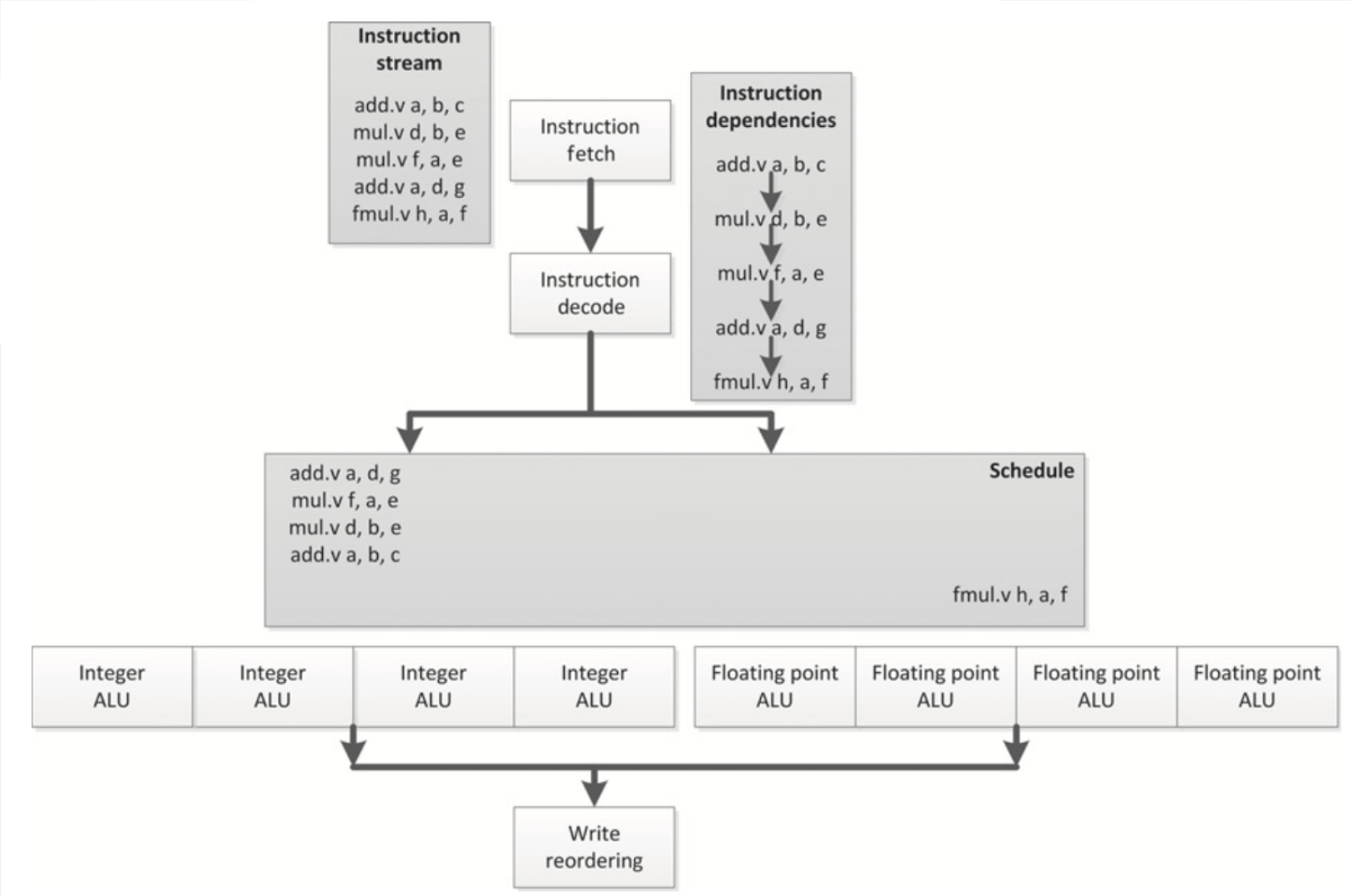
Esecuzione Superscalare



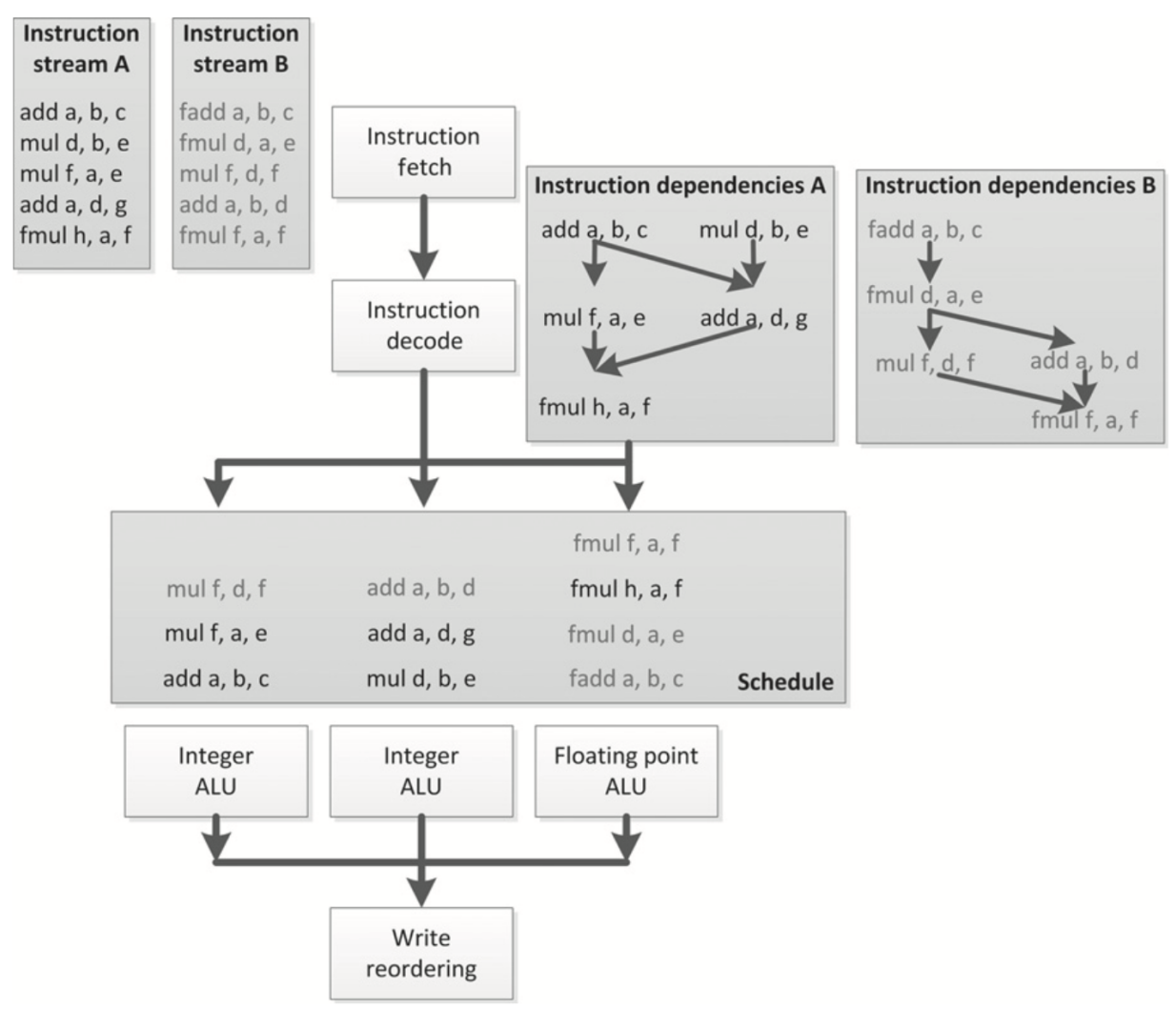
Very Long Instruction Word (VLIW)



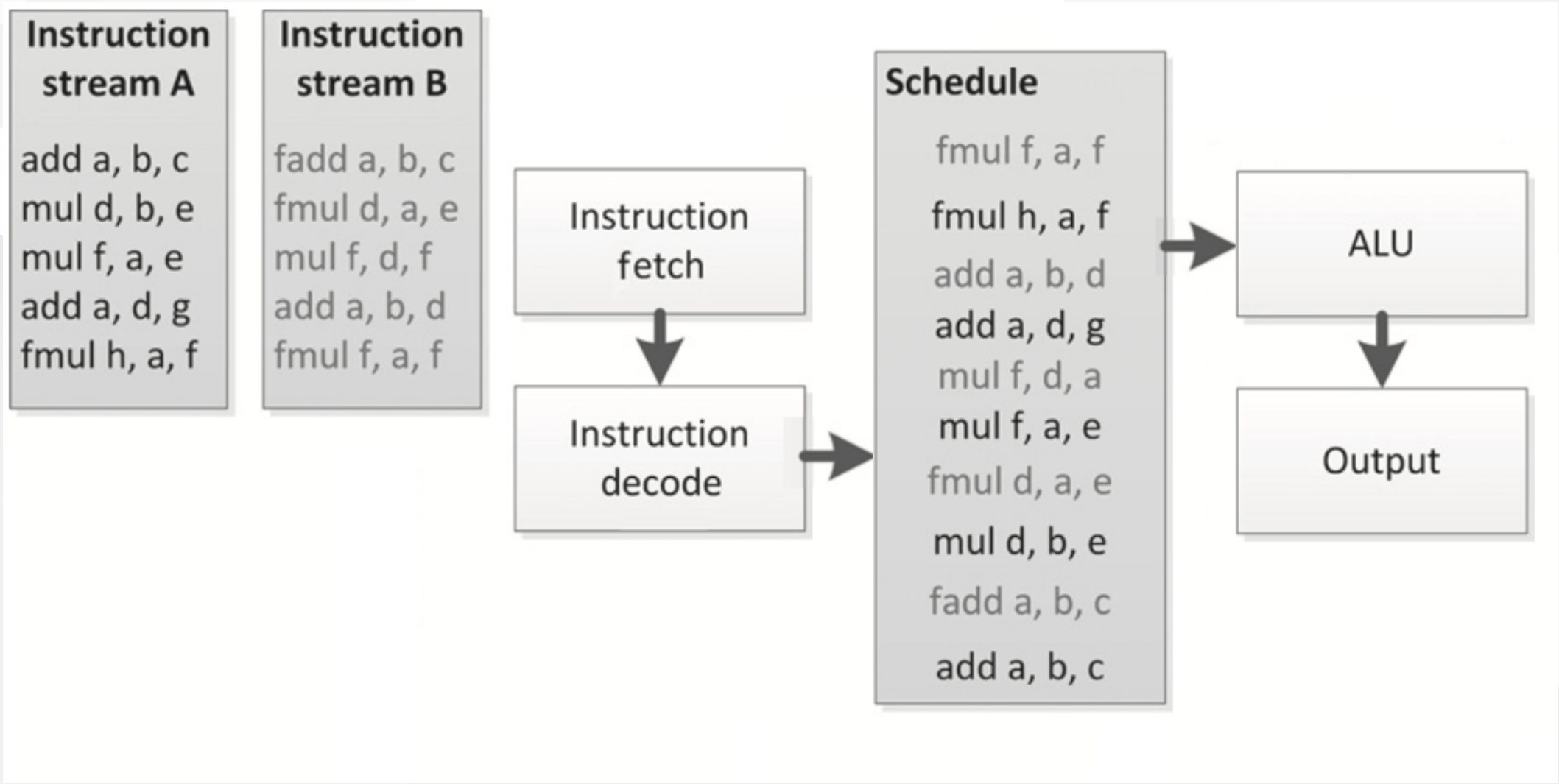
SIMD e *Vector Processing* (VLIW)



Simultaneous Multithreading (SMT)

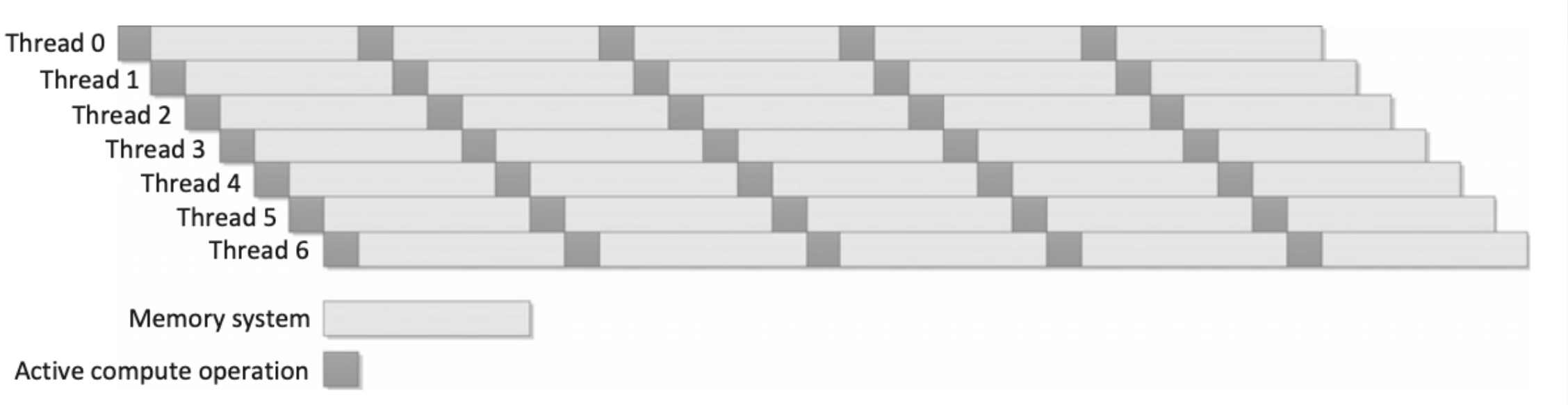


Temporal (Time-Sliced) Multithreading

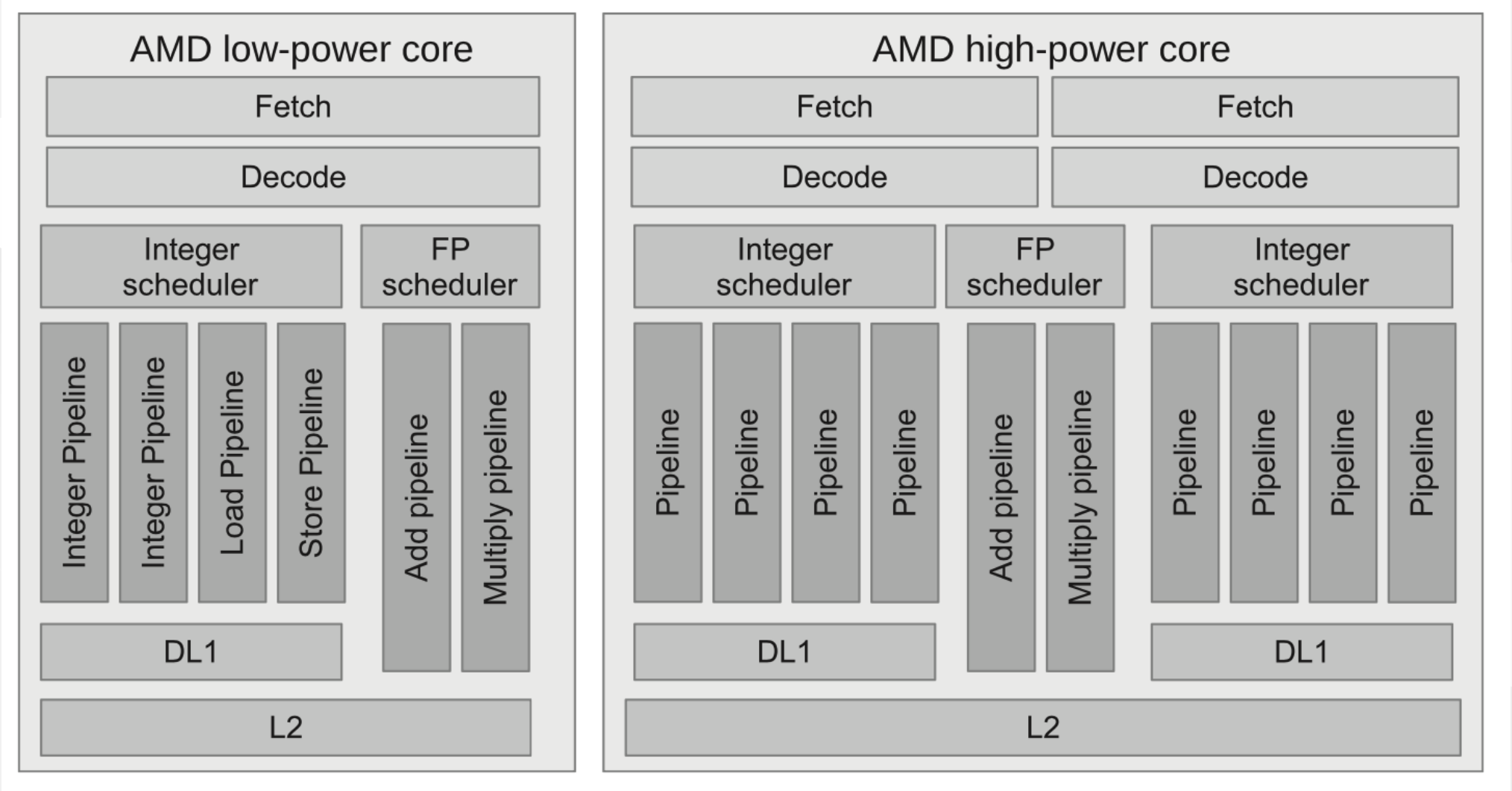


Latenza vs Portata di Calcolo

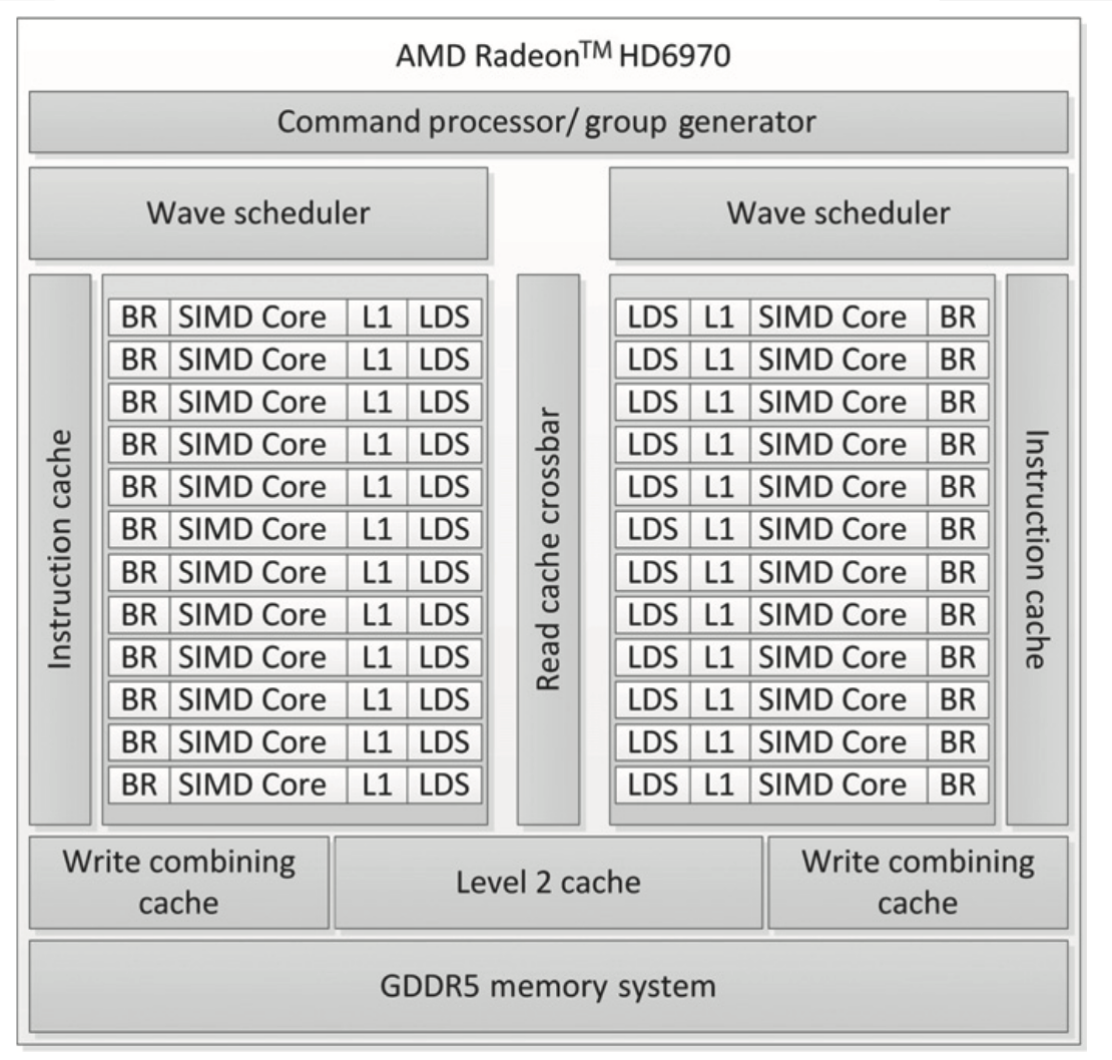
Il multithreading temporale aumenta il *throughput* a scapito della latenza



Multicore Architectures



Multicore Architectures (Cont'd)



La Cache

Sfrutta schemi di accesso

- spaziali: accessi ad aree di memoria vicine, e.g., `A[0]`, `A[1]`, ...
- temporali: accessi alla stessa area vicini nel tempo

La *cache* serve a:

- minimizzare la latenza delle operazioni
- ridurre l'utilizzo di un canale di traferimento dati (*bus*) limitato

Integrazione CPU-GPU

Le CPU e le GPU possono essere integrate nello stesso *die* per:

- contenere la latenza
- diminuire il consumo
- ridurre lo spazio

I processori AMD, Intel e Apple Silicon adottano questa integrazione

Programmi Paralleli e Hardware Eterogeneo

Date le tante differenze nell'hardware

- modelli di calcolo
- differenze nell'insieme di istruzioni
- specificità nelle versioni dell'hardware

Scrivere programmi paralleli, potrebbe essere un incubo

Vorremmo astrarre l'hardware e nascondere i dettagli

OpenCL

È un ambiente di sviluppo per hardware eterogeneo

Supporta CPU, GPU, FPGA, DSP, etc di molti produttori

Consiste in:

- un'API C/C++ per controllare i dispositivi (e.g., trasferimenti dati, selezione del dispositivo da usare, etc.)
- un linguaggio basato su C99 (*binding* C++, Python, etc.) per i compiti eseguiti dai dispositivi

CUDA vs OpenCL

CUDA è l'architettura hardware di NVIDIA (al momento solo GPU) e un'ambiente di sviluppo per essa

OpenCL è un ambiente per lo sviluppo parallelo che è:

- adatto a hardware eterogenei
- supportato da molti produttori
- uno standard open-source

Il Mio Hardware È Supportato?

`clinfo` restituisce informazioni in merito

```
Number of platforms                   1
  Platform Name                       Apple
  Platform Vendor                     Apple
  Platform Version                    OpenCL 1.2 (Sep 30 2022 01:38:14)
  Platform Profile                    FULL_PROFILE
  Platform Extensions                 cl_APPLE_SetMemObjectDestructor
  Platform Name                       Apple
Number of devices                     1
  Device Name                         Apple M1
  Device Vendor                       Apple
  Device Vendor ID                    0x1027f00
  Device Version                      OpenCL 1.2
  Driver Version                      1.2 1.0
  Device OpenCL C Version             OpenCL C 1.2
  Device Type                         GPU
```

La Specifica OpenCL

Quattro modelli:

1. ***Platform Model***: un *host* che coordina almeno un *device* che esegue i *kernel*
2. ***Execution Model***: definisce l'host e i device di calcolo. Specifica come interagiscono e il modello di concorrenza.
3. ***Kernel Programming Model***: mappa il modello di concorrenza nell'hardware
4. ***Memory Model***: Astrae la memoria usata dai kernel. Può definire la memoria virtuale condivisa tra i *device* e l'*host*

Platform Model

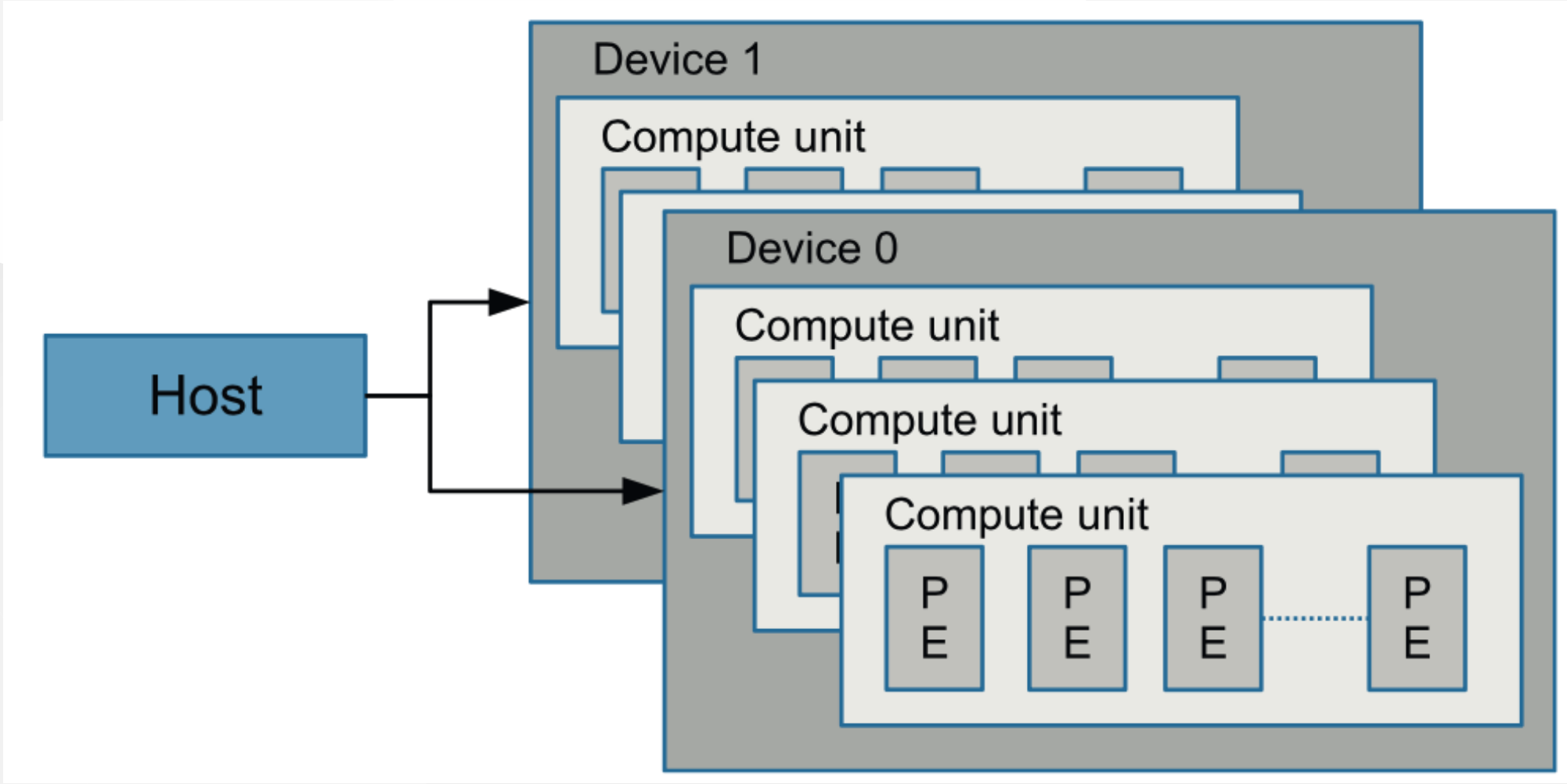
Prevede:

- Un *host*
- Uno o più *device*

Ogni *device* è diviso in ***compute unit***

Ogni *compute unit* contiene diversi ***processing elements***

Platform Model (Cont'd)



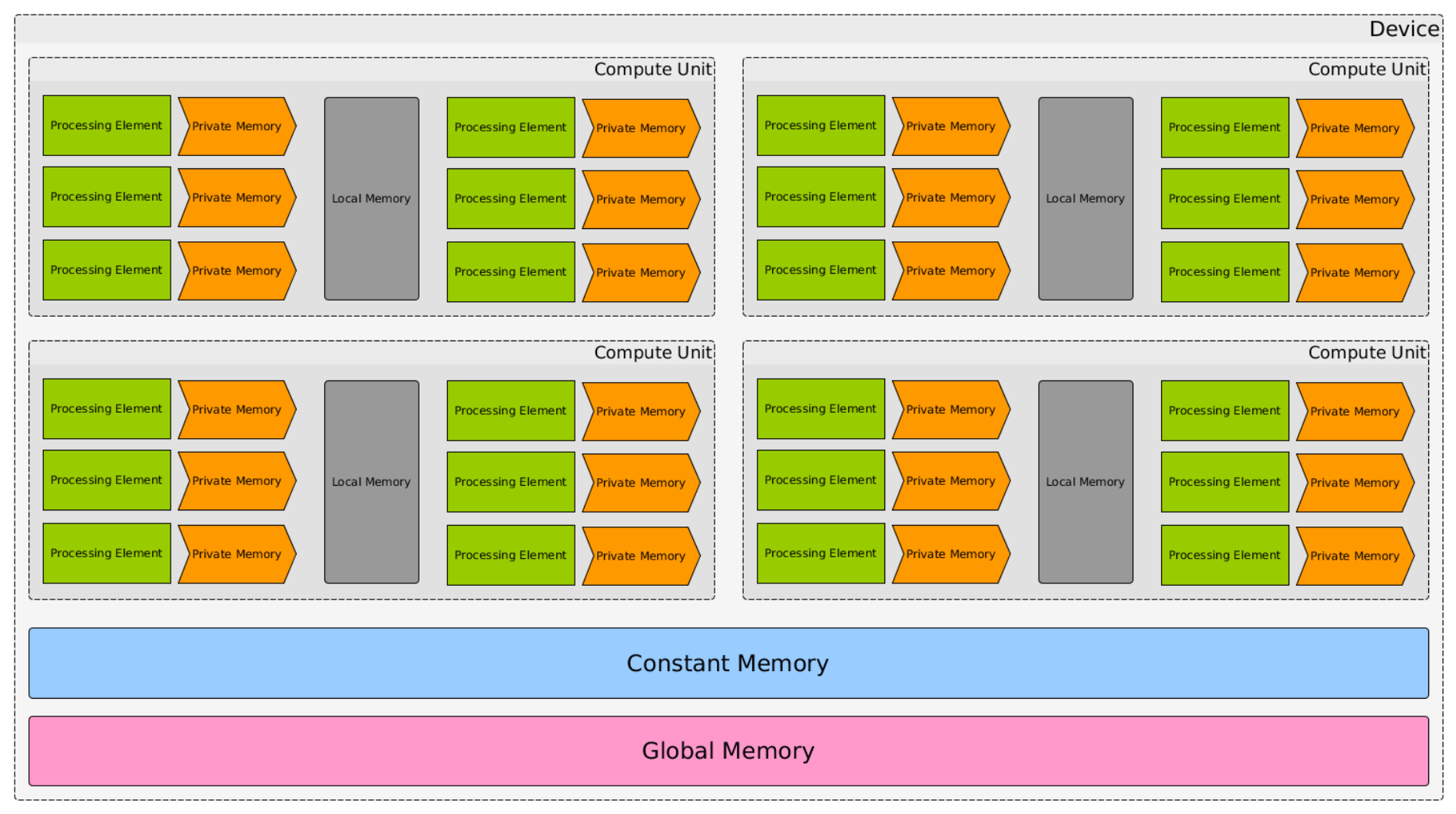
Memoria del *Platform Model*

Quattro tipi di memoria:

- **Globale**: condivisa da tutti i *device*, ma "lenta"
- **Costante**: più veloce della globale. Può essere usata per il passaggio di parametri
- **Locale**: associata a ciascuna *compute unit*, ma condivisa tra i *PE*
- **Privata**: la più veloce, ma associata a ciascuna *PE*

Solo la memoria globale è persistente

Memoria del *Platform Model* (Cont'd)



Execution Model

I programmi sono eseguiti sull'*host* e inviano lavoro ai device

- **Work-Item**: unità di lavoro di base dei *device*
- **Kernel**: il codice che viene eseguito su un *work-item*
- **Programma**: è una collezione di *kernel* e funzioni per l'host
- **Contesto**: è l'ambiente in cui si svolgono i *work-item* e il modo in cui è organizzato il lavoro

Dai Programmi Scalari...

Supponiamo di voler calcolare `a+b`

```
void vecadd(int* C,              // array di output
            int* A, int* B,      // array di input
            int n) {
  for (int i=0; i<n; ++i) {
    C[i] = A[i] + B[i];
  }
}
```

Il ciclo può essere spezzato nei singoli *work-item*

```
C[i] = A[i] + B[i];    // dove i è l'identificatore del work-item
```

... A Quelli OpenCL

```
__kernel      // definisco un kernel
void vecadd(__global int* C,                        // array di input
            __global int* A, __global int* B) {     // array di output

  int wid = get_global_id(0);      // ottengo l'id del work-item
  C[wid] = A[wid] + B[wid];
}
```

Il kernel verrà eseguito da n *work-item*

Ciascun *work-item* verrà eseguito da un *processing element*

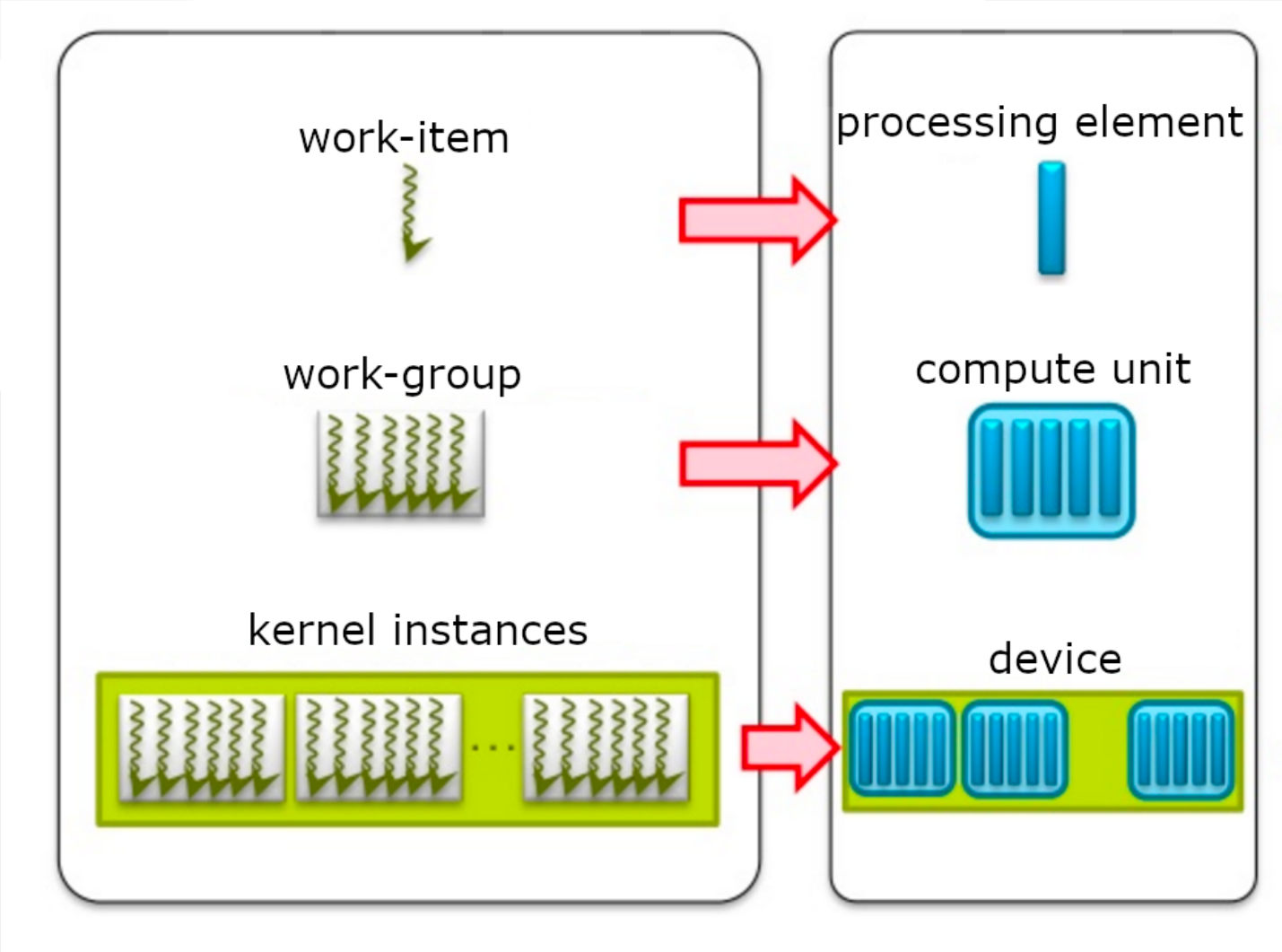
I *Work-Item* Possono Comunicare?

Abbiamo bisogno di memoria condivisa tra i *work-item*

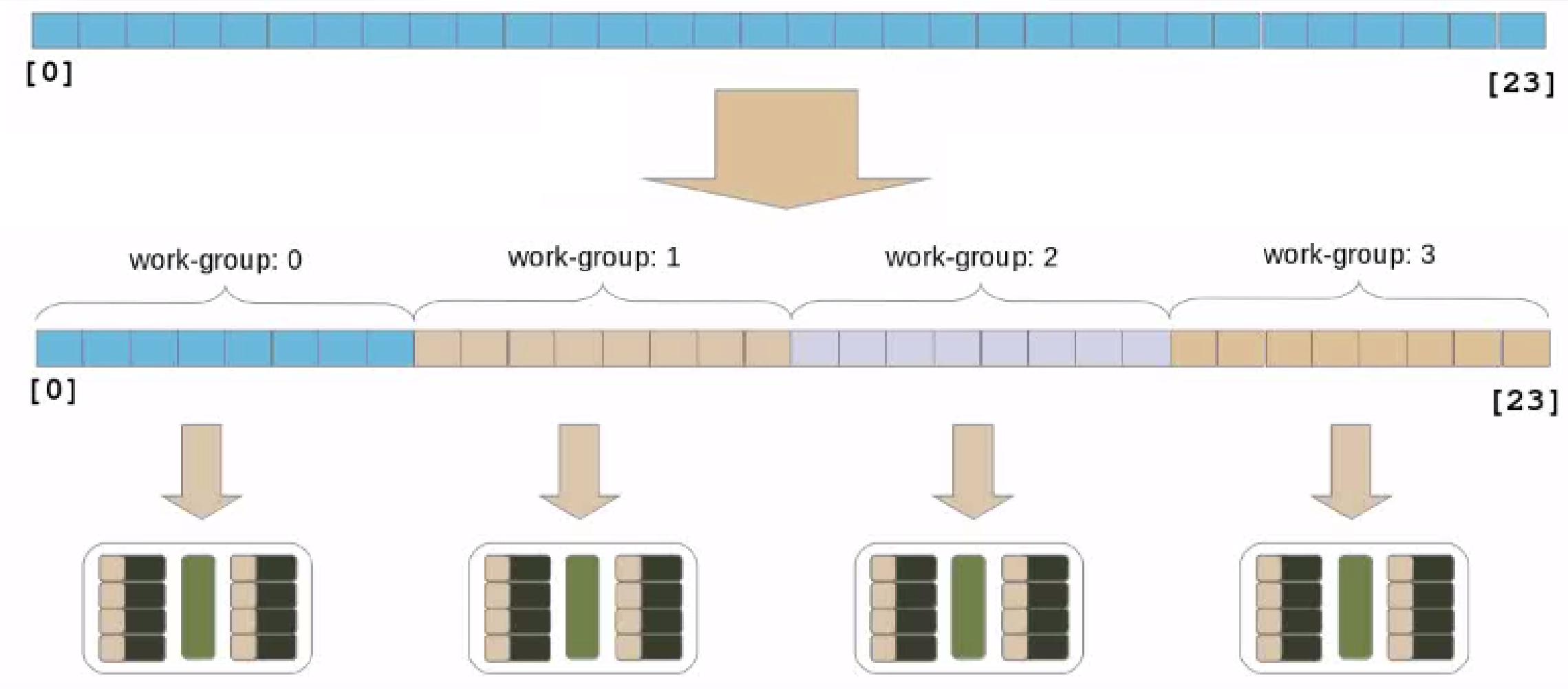
Raggruppiamo i *work-item* in ***work-group*** con memoria condivisa

OpenCL assegna ogni *work-group* a un *compute unit*

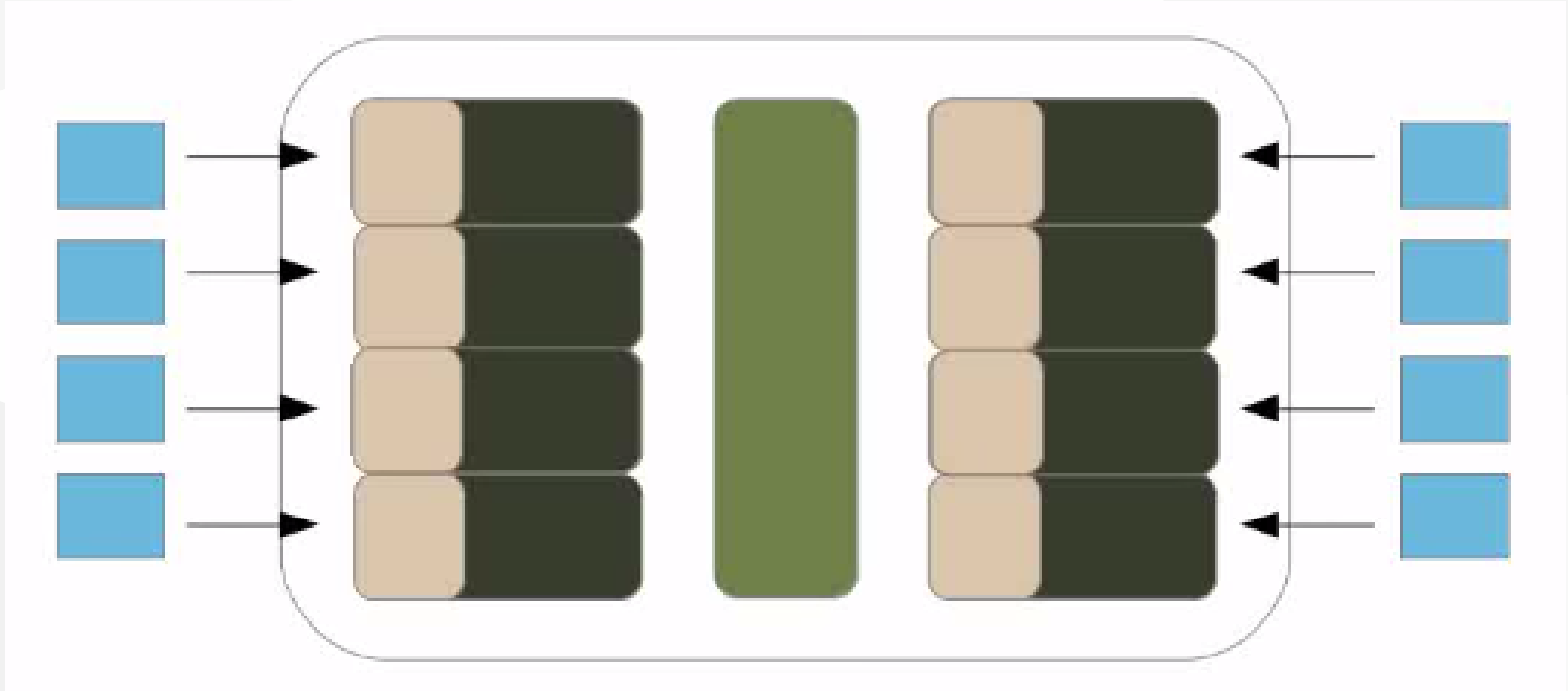
Software in Hardware



Dal Lavoro Globale alle *Compute Unit*

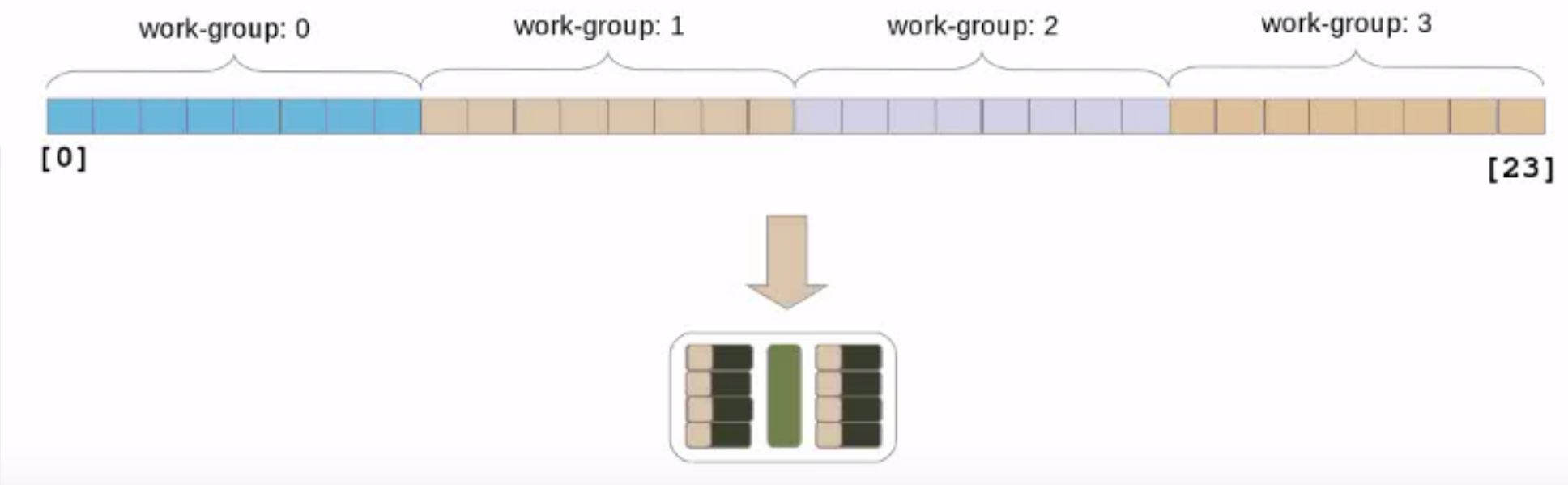


Dai *Work-Group* ai *Processing Element*



2^{10} Non Approssima Bene ∞!

E se il numero di *work-group* è maggiore dei *compute unit*?



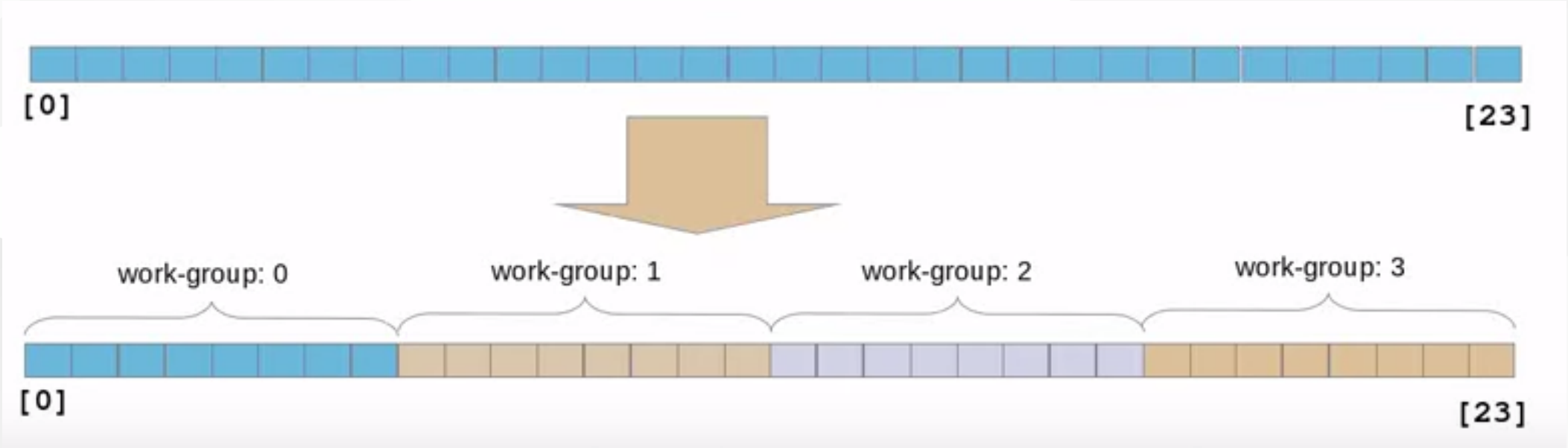
I *work-group* sono schedulati sulle *compute unit* automaticamente

Dimensione Globale e del *work-group*

Possiamo allora distinguere tra:

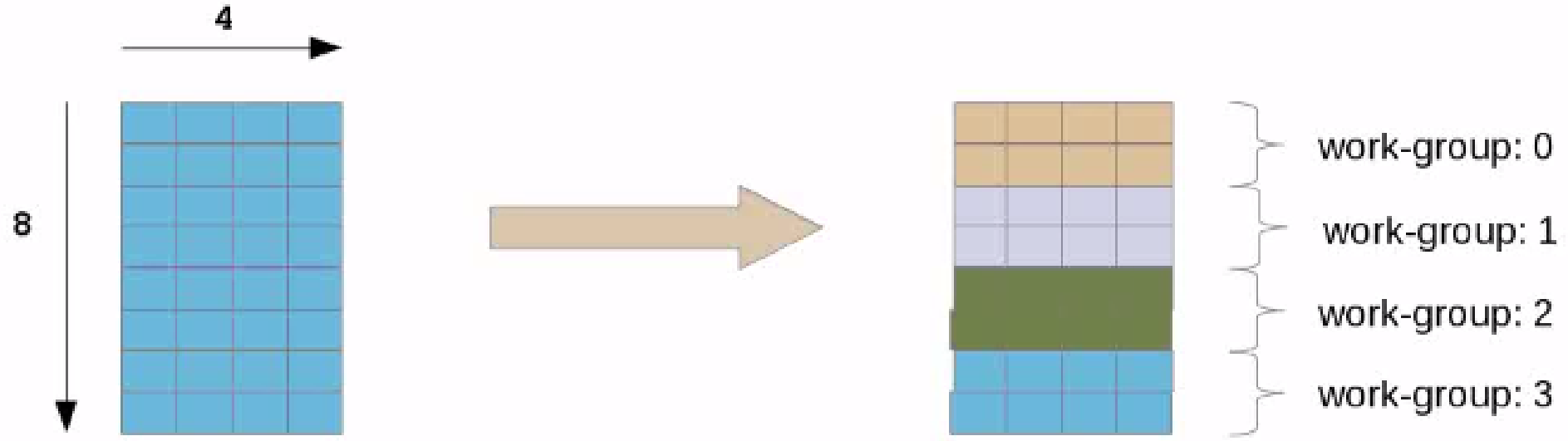
- **dimensione globale**: è la dimensione dell'input del problema
- **dimensione del *work-group***: è il numero di *work-item* attribuiti ad ogni *work-group*

Abbiamo Diviso i *Work-Group* in 1D ...



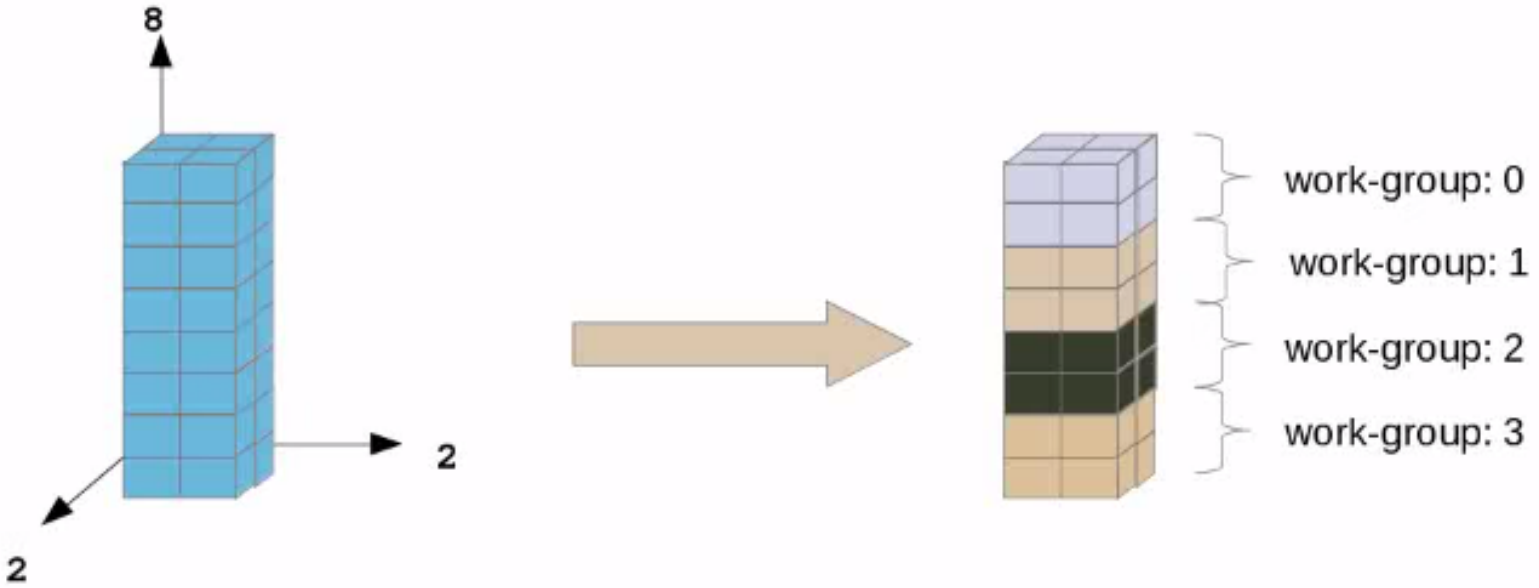
… Ma Possiamo in 2D…

```
global work size:  (8, 4)
work-group size:  (2, 4)
```



… E in 3D

```
global work size:  (2,2,8)
work-group size:  (2,2,2)
```



Kernel Programming Model

Definisce:

- *work-item* e *work-group*
- *kernel*
- il numero N-dimensionale dei *work-group* (*NDRange*)

Kernel Eterogenei

Ci sono molti tipi di device diverso (e.g., GPU NVIDIA, CPU AMD, CPU ARM)

Non possiamo avere un binario ottimizato per ciascuno di essi

I *kernel* vengono (normalmente) compilati al tempo di esecuzione

Sono ottimizzati per la specifica architettura su cui sono eseguiti

Compilare i *kernel*

1. i sorgenti sono memorizzati come array di caratteri
2. i sorgenti vengono trasformati in `cl_program` con una chiamata a `clCreateProgramWithSource()`
3. `clBuildProgram()` compila i `cl_program` e segnala eventuali errori
4. `clCreateKernel()` crea `cl_kernel` a partire da un `cl_program` compilato

Code di Comandi

L'*execution model* dice che i *device* assolvono a **comandi**

L'*host* invia comandi ai *device* tramite **code di comandi**

I comandi includono:

- trasferimento dati, e.g., `clEnqueueReadBuffer()`
- sincronizzazione tra *work-item*
- l'esecuzioni di *kernel*, e.g., `clEnqueueNDRangeKernel()`

I Parametri dei *kernel*

Come passarli se i *kernel* sono accodati?

Vanno copiati nella memoria del device

Viene fatto automaticamente usando `clSetKernelArg`

```
cl_int
clSetKernelArg (
   cl_kernel kernel,          // kernel a cui si vuole passare un parametro
   cl_uint arg_index,         // indice del parametro nel vettore dei parametri
   size_t arg_size,           // dimensione in byte del parametro
   const void *arg_value)     // puntatore all'area di memoria del parametro
```

Memory Model

Gli oggetti memorizzabili sono del tipo `cl_mem`

- **buffer**: analoghi agli array
- **image**: per sfruttare le ottimizzazioni dei *device*. Non si può accedervi direttamente, e.g., `read_imagef()`, `write_imagei()`
- **pipe**: sono code FIFO. Per leggere e scrivere dati si usano `read_pipe()` and `write_pipe()`

I Passi di un Programma OpenCL

1. Definire un'astrazione dell'hardware

- Scoprire le piattaforme e i relativi *device*
- Creare un contesto
- Creare una coda di comandi per *device*

2. Definire gli oggetti in memoria

- Creare i buffer per i dati
- Copiare i buffer nei *device*

I Passi di un Programma OpenCL (Cont'd)

3. Definire i *kernel*

- Creare i `cl_program` e compilarli
- Creare i `cl_kernel`

4. Eseguire i *kernel*

- Associare i parametri ai *kernel*
- Accordarli sugli opportuni *device*

I Passi di un Programma OpenCL (Cont'd 2)

4. Leggere i dati
5. Rilasciare le risorse

```
clReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseCommandQueue(cmdQueue);
clReleaseMemObject(bufA);
clReleaseMemObject(bufB);
clReleaseMemObject(bufC);
clReleaseContext(context);
```