

Programmazione Avanzata e parallela

Lezione 12

Luca Manzoni

Operazioni vettoriali

Eseguire la stessa operazione su più dato

- Cosa sono le operazioni vettoriali
 - Tassonomia di Flynn
 - Breve storia dei computer vettoriali
- Istruzioni vettoriali su architetture moderne
- Come utilizzare le istruzioni vettoriali
 - Intrinsics ed estensioni del compilatore
 - Linguaggi sviluppati appositamente

Istruzioni vettoriali

Motivazione

- Supponiamo di voler sommare due vettori \vec{u} e \vec{v} .
- Questo significa compiere una serie di operazioni della forma:
 $\vec{u}_1 + \vec{v}_1, \vec{u}_2 + \vec{v}_2, \vec{u}_3 + \vec{v}_3, \dots$
- Sarebbe comodo poter esprimere il concetto “esegui la stessa operazione (somma) su tutte le coppie di valori”
- Questo permetterebbe di leggere e decodificare una sola istruzione ed eseguirla su più dati...
- ...e potenzialmente eseguire alcune di queste operazioni in parallelo

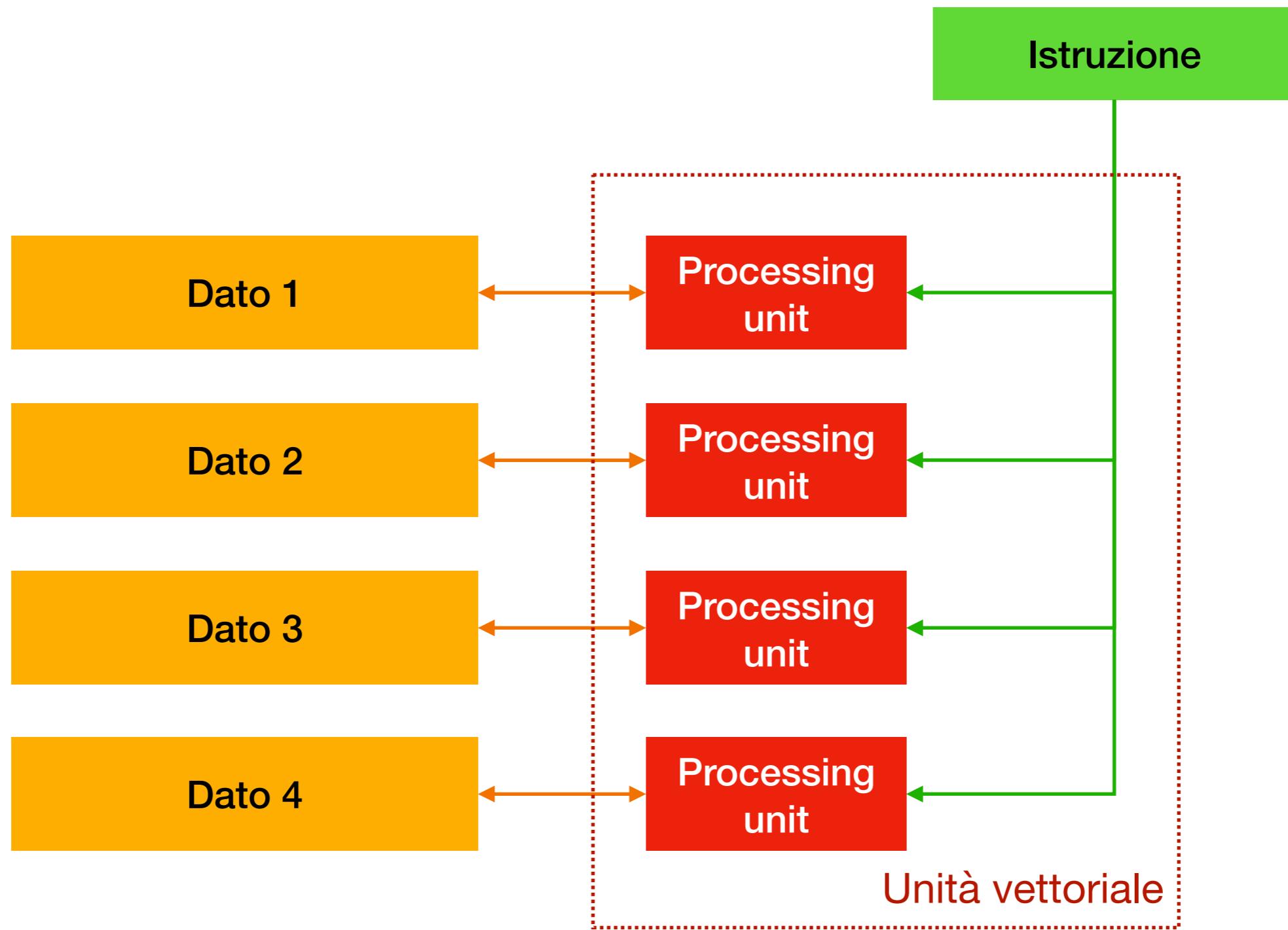
Istruzioni vettoriali

Motivazione

- Un metodo comune è quello di avere istruzioni che lavorano su k valori alla volta (e.g., da 2 a 16 valori)
- Un normale ciclo di somma degli elementi di due array può quindi essere scritto come un ciclo che somma 16 elementi per ciclo
- ```
for (int i = 0; i < n; i += 16) {
 // not a real instruction
 somma_16_elementi(&A[i], &B[i], &C[i]);
}
```

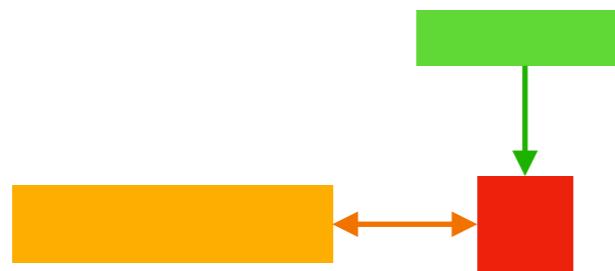
# Struttura di una unità vettoriale

## Schema generale

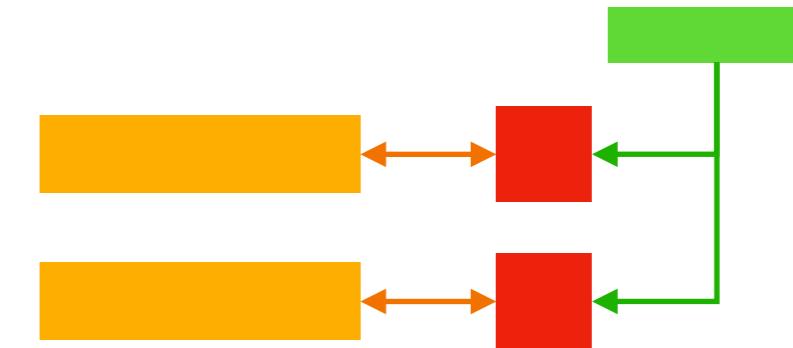


# Tassonomia di Flynn (1972)

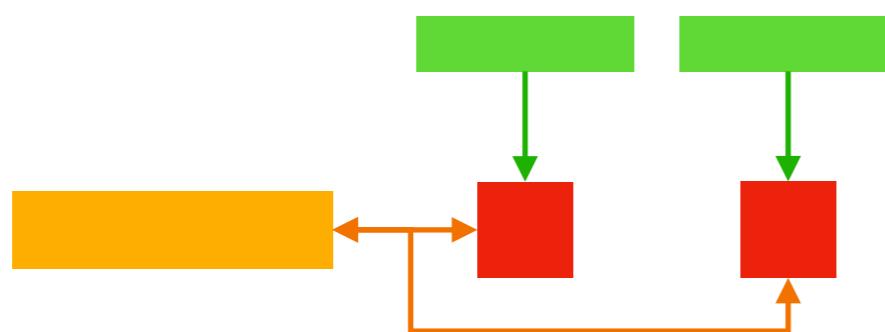
## Classificazione delle architetture



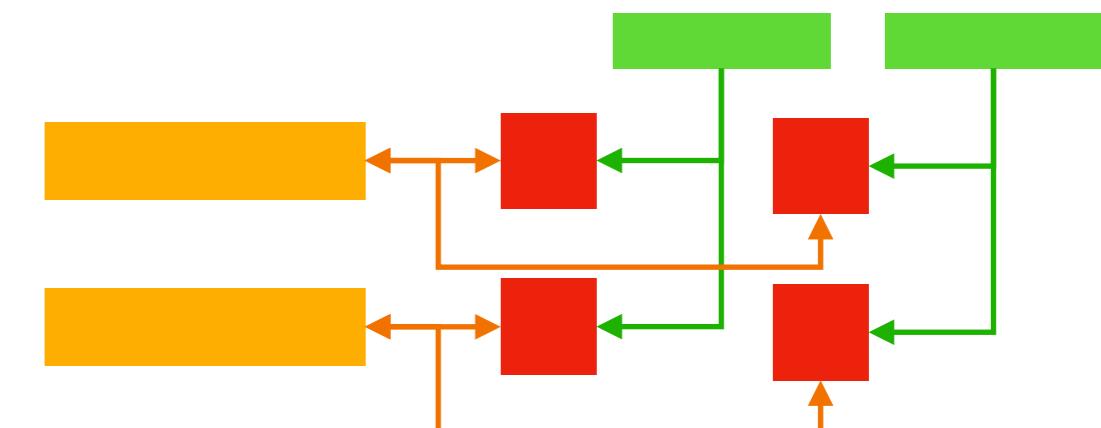
**SISD**  
Single Instruction Stream  
Single Data Stream



**SIMD**  
Single Instruction Stream  
Multiple Data Streams



**MISD**  
Multiple Instruction Streams  
Single Data Stream



**MIMD**  
Multiple Instruction Streams  
Multiple Data Streams

# Tassonomia di Flynn (1972)

## Classificazione delle architetture

- **SISD (Single Instruction Stream, Single Data Stream)**  
È il modello che pensiamo per il “normale” calcolatore con un singolo processore.  
Una singola sequenza di istruzioni che opera su un singolo flusso di dati (i.e., una operazione alla volta).
- **MISD (Multiple Instruction Streams, Single Data Stream)**  
Architetture poco presenti, usate per ridondanza (e.g, più processori che eseguono la stessa istruzione sugli stessi dati e si verifica che tutti i risultati coincidano).  
Le architetture systolic array sono classificate come MISD ma, per la loro struttura particolare la classificazione non è senza problemi.
- **MIMD (Multiple Instruction Streams, Multiple Data Streams)**  
Più unità di esecuzione operano su più flussi di dati.  
Ne fanno parte i sistemi con più core, i sistemi distribuiti, etc.

# Tassonomia di Flynn (1972)

## Classificazione delle architetture

- **SIMD (Multiple Instruction Streams, Single Data Stream)**  
Più flussi di dati su cui viene applicata la stessa operazione.  
Nell'articolo del 1972 Flynn opera una suddivisione aggiuntiva:
  - *Array Processors*. Ogni unità riceve la stessa istruzioni ma opera su memoria separata.
  - “*Pipelined*” Processors. Ogni unità riceve la stessa istruzione ma la memoria è comune (e le istruzioni operano su parti di essa).
  - *Associative Processors*. Ogni unità riceve la stessa istruzione locale ma compie una scelta locale se eseguirla o no. Nella terminologia moderna sono “*predicated instructions*”

# Storia delle istruzioni vettoriali

## Computer vettoriali



### Cray 1 (1975)

Rende comune l'idea di  
“processore vettoriale”

Esempi precedenti:

CDC Star-100, TI ASC  
(ma le istruzioni lavoravano  
sulla memoria, non su  
registri appositi)

### Connection Machine CM-1 (1985)

Esempio di macchina con 65536  
processori a un bit che potevano  
eseguire la stessa operazione su  
dati diversi (i.e., 65536 operazioni in  
parallelo)

# Storia delle istruzioni vettoriali

## Estensioni vettoriali

- Dagli anni '90 le diverse architetture hanno acquisito istruzioni e registri per lavorare su vettori di dati
- Generalmente consistono di un insieme di registri che devono essere implementanti per tenere più dati dello stesso tipo e un insieme di istruzioni per lavorare su di essi
- Nel caso di ARM ci sono state diverse estensioni:
  - NEON. Registri da 64 e 128bit
  - SVE (Scalable Vector Extension). Permette implementazioni con registri da 128 a 2048bit, con implementazioni esistenti fino a 512bit

# Storia delle istruzioni vettoriali

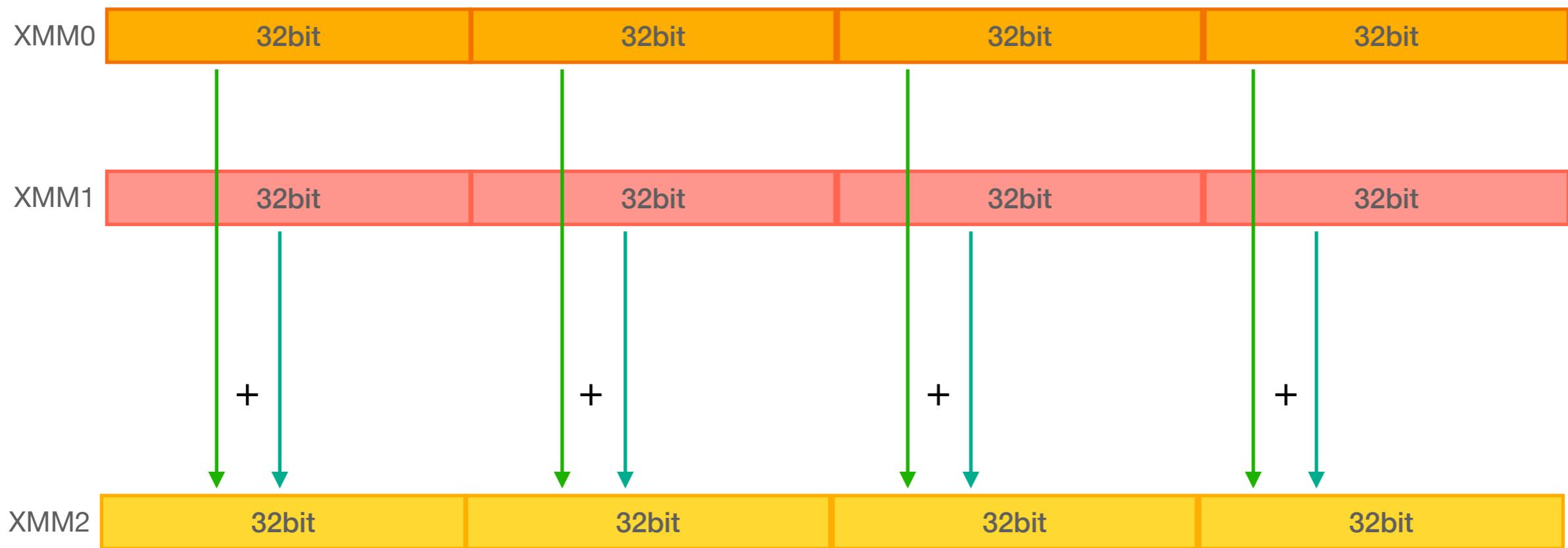
## Estensioni vettoriali

- Nel caso di x86 e x86\_64 ci sono state diverse estensioni:
  - MMX (1997), registri vettoriali di 64bit
  - SSE1 (1999), SSE2 (2000), SSE3 (2004), SSE4.1 (2007), SSE 4.2 (2008), registri di 128 bit
  - AVX (2011), AVX2 (2013), registri di 256 bit
  - AVX512 (2016), registri da 512bit
  - AVX10 (proposto ad Agosto 2023), uniforma diverse delle inconsistenze di AVX512

# Registri vettoriali

## Esempio

Registro vettoriale da 128bit (i nomi sono quelli dei registri SSE)



Possiamo eseguire (in parallelo) l'operazione di somma su 4 valori da 32 bit

# Registri vettoriali

## Usarli ed accedervi

- Ogni estensione definisce un insieme di registri (xmm0-15, ymm0-15, zmm0-15 per SSE, AVX e AVX512) e operazioni che possono lavorare su di essi
- Generalmente queste operazioni assumo che ogni registro abbia un certo numero di valori dello stesso tipo (e.g., 4 float, 8 interi di 16 bit, etc)
- È però scomodo utilizzare direttamente l'assembly per poter sfruttare questi le istruzioni vettoriali
- Servono diversi modi per utilizzare le istruzioni

# Registri vettoriali

## Usarli ed accedervi

- **Opzione 1: ignorare la loro esistenza**

Le prestazioni che si ottengono possono però essere molto migliori (si stanno eseguendo più operazioni per lo stesso numero di cicli)

- **Opzione 2: affidarsi al compilatore**

Gli ottimizzati di molti compilatori forniscono dei sistemi di auto-vettorizzazione (o vettorizzazione automatica) del codice che coprono diversi casi comuni (e.g., somma degli elementi di due vettori)

- *Problema:* non sono perfetti, è facile cambiare poco e perdere la vettorizzazione

# Registri vettoriali

## Usarli ed accedervi

- **Opzione 3: Intrinsics del compilatore**  
Usare delle estensioni del compilatore che espongono alcune istruzioni come funzioni
  - Problema: il codice non è portabile ad altre architetture
- **Opzione 4: Estensioni vettoriali dei compilatori / librerie**  
Alcuni compilatori (GCC) espongono un tipo vettore generico utilizzabile per (parte) delle operazioni vettoriali.  
Alcune librerie espongono funzioni che poi sono trasformate nelle istruzioni vettoriali specifiche
- **Opzione 5: linguaggi con estensioni apposite per SIMD**  
Per esempio “Intel® implicit SPMD program compiler” (<https://ispc.github.io/>)

# Usare le intrinsics

## Caso di AVX

- Se vogliamo usare le intrinsics (nel nostro caso per AVX) dovremmo:
  - Importare le librerie necessarie con `#include <immintrin.h>`
  - Utilizzare tipi di dato e funzioni apposite (e.g., `_m256` come “8 float di 32 bit” o `_mm256d` per “4 double di 64 bit”)
  - Compilare con `-mavx` come opzione o una opzione di march che lo implica (il compilato sarà eseguibile solo su un sistema che supporta AVX)

# Intrinsics

## Esempio di istruzioni

8 valori float da 32 bit in  
un singolo registro di 256 bit

`__m256 __mm256_add_ps(__mm256 x, __mm256 y)`

Somma di 8 valori floating point da 32 bit

`__mm256 __mm256_loadu_ps(float * b);`

Carica 8 valori floating point a 32 bit  
dall'indirizzo puntato

`void __mm256_storeu_ps(float * v, __mm256 x);`

Salva gli 8 valori floating point a 32 bit  
all'indirizzo puntato

# Estensioni vettoriali

## Per GCC

- GCC ci fornisce anche delle estensioni vettoriali generiche (quindi il codice sarà comparabile su più architetture)
- Possiamo definire tipo con la sintassi simili:
  - `typedef int v4si __attribute__ ((vector_size (4 * sizeof(int))));`
  - `typedef int v8si __attribute__ ((vector_size (8 * sizeof(int))));`
  - `typedef int v8f __attribute__ ((vector_size (8 * sizeof(float))));`
- In pratica diamo un nome a un vettore di n elementi dello stesso tipo, sarà compito del compilatore capire che istruzioni vettoriali usare