Memorie e Cache

Sistemi di Elaborazione Accelerata, Modulo 1

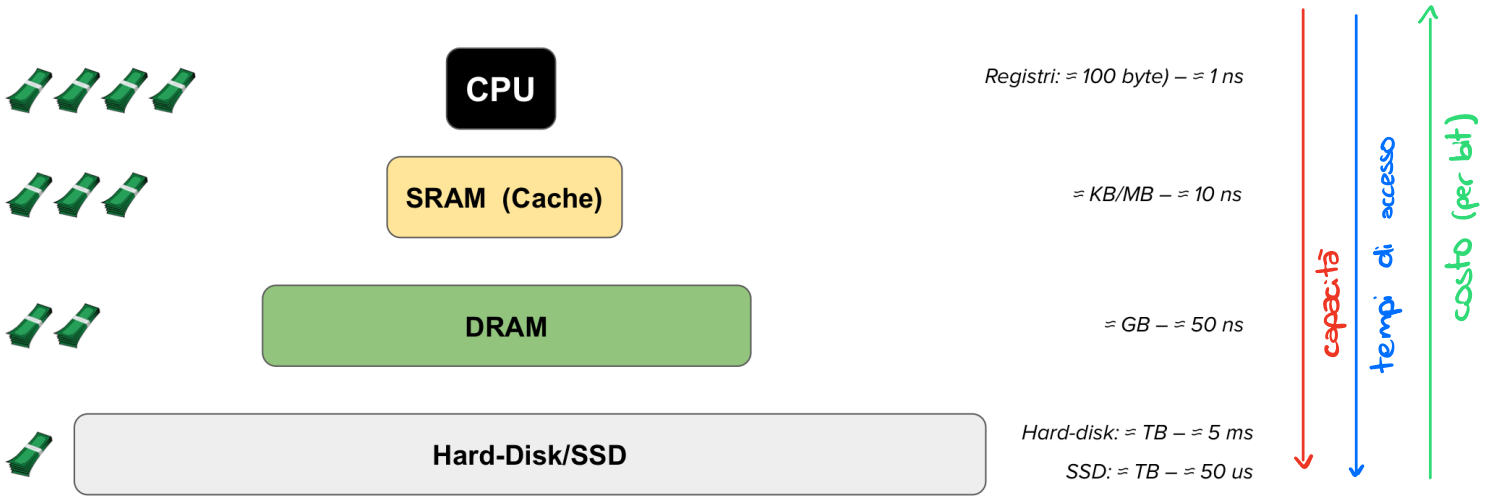
A.A. 2025/2026

Stefano Mattoccia

Università di Bologna

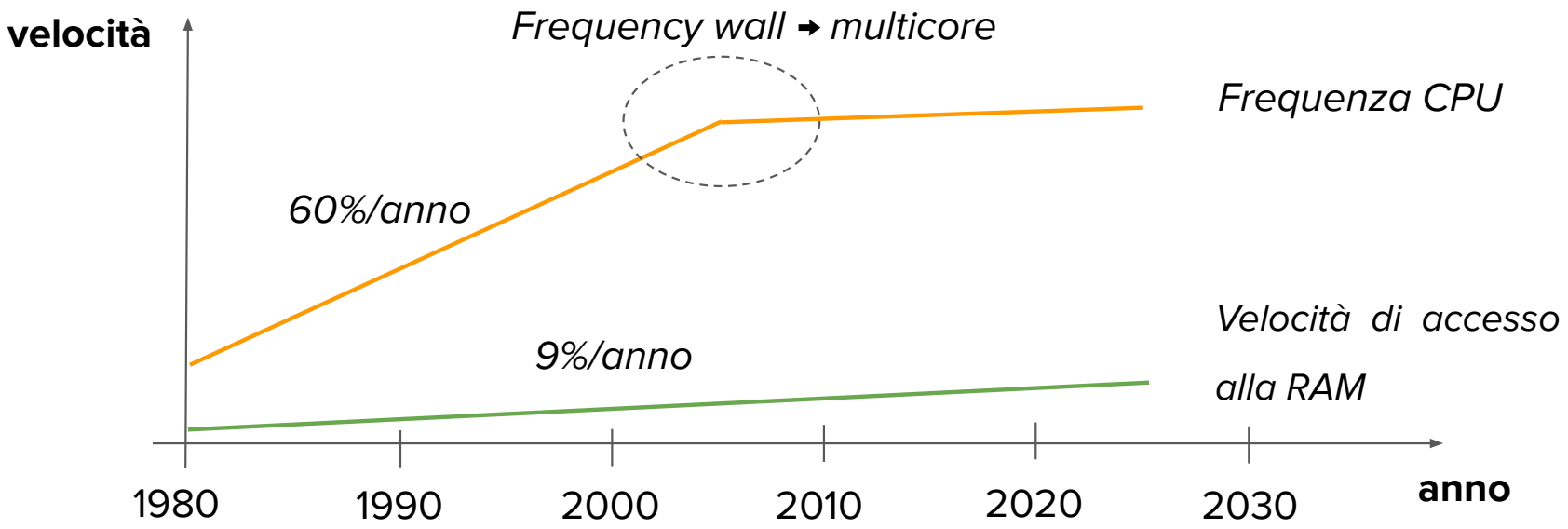
Gerarchia delle memorie

- In un sistema di elaborazione sono presenti varie tecnologie di memorie con diverse caratteristiche quali: capacità, tempo di accesso,e costo per bit



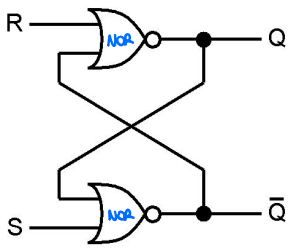
CPU vs Memoria: confronto sulle prestazioni

- Idealmente, sarebbe auspicabile che la CPU potesse accedere alla memoria alla stessa velocità con la quale necessità di accedere ai dati
- Tuttavia, il divario tra la frequenza delle CPU e il tempo di accesso alla memoria (eg, DRAM) è elevato e aumentato significativamente per un lungo periodo (molti dati)



Static RAM (SRAM) e Dynamic RAM (DRAM) 1/2

- Esistono due tecnologie per realizzare dispositivi di memoria RAM
  - Static RAM SRAM
    - L'elemento base per la memorizzazione di un bit è un latch (eg, 6 transistor)



si può realizzare (nello stesso modo) anche a NAND

Tabella della verità

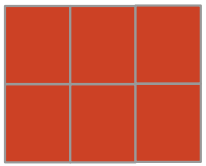
S	R	Q	
0	0	Q	Hold
0	1	0	Reset
1	0	1	Set
1	1	0	Non valido

  - Dynamic RAM DRAM
    - L'elemento base per la memorizzazione di un bit è un condensatore (e 1 transistor)

Condensatore di piccola capacità, poche cariche, si scarica velocemente → richiede refresh

Static RAM (SRAM) e Dynamic RAM (DRAM) 2/2

- Le DRAM hanno una capacità molto maggiore a parità di area (eg, 1/6 transistor)

SRAM  *Area/bit*

DRAM  *Area/bit*

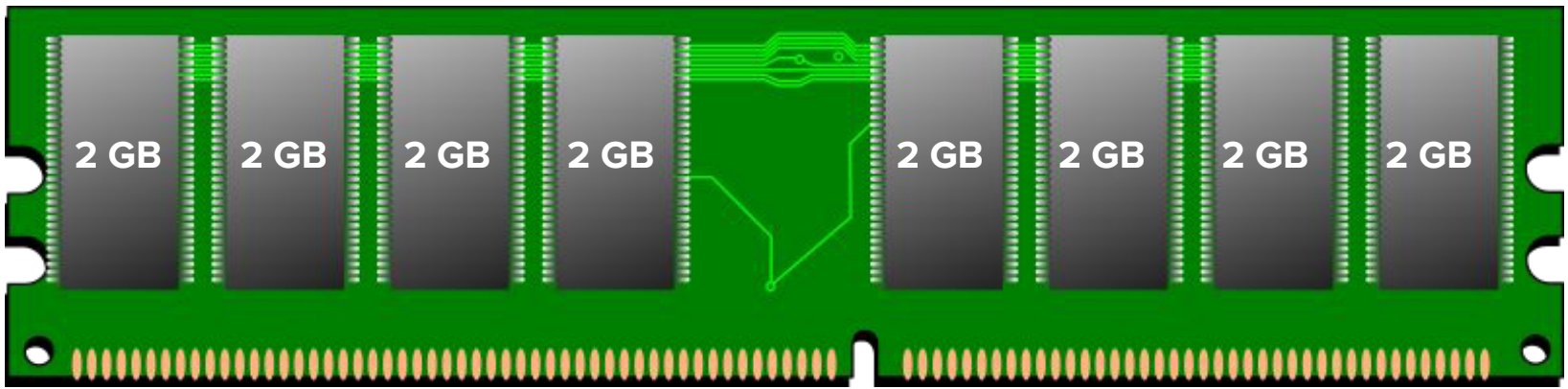
- Per questo motivo, il costo per bit è favorevole alle DRAM
- Tuttavia, le DRAM sono più lente nell'accesso ai dati rispetto alle SRAM (eg, 10x)
- Per quanto riguarda il consumo per bit: le DRAM risultano avvantaggiate nonostante sia indispensabile accedere a ciascun bit di informazione più volte al secondo per non perdere l'informazione memorizzata
- Tale periodico *refresh* è realizzato mediante un opportuno dispositivo al fine rigenerare continuamente la carica in ciascun condensatore

Considerazioni sulle memorie

- Idealmente, sarebbe desiderabile avere memorie capienti (come le DRAM) e veloci (come le SRAM)
- Tuttavia, le capienti DRAM, a causa dell'elevata latenza (eg, 50+ ns), hanno un tempo di accesso considerevole che induce notevoli ritardi (cicli di WAIT)
- Risulta quindi necessaria una strategia che consenta di superare questa limitazione
- Tale strategia si basa sull'utilizzo di una veloce memoria cache (SRAM) posta tra la CPU e la DRAM al fine di (cercare di) compensare l'elevato tempo di accesso delle DRAM
- Prima di descrivere come funziona una cache bene analizzare come sono organizzate le DRAM

Layout di una memoria (eg, DRAM)

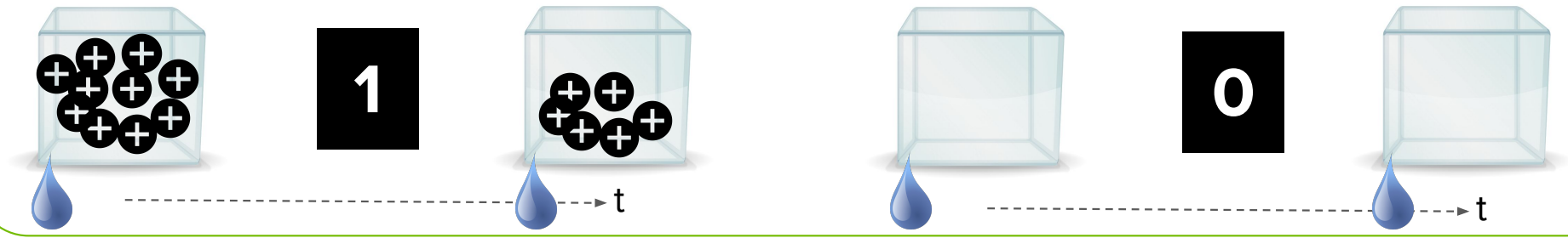
- Le moderne CPU hanno ampi bus dati (eg, 64 bit) e più di 32 linee di indirizzo (eg, 36)
- I moduli di memoria sono composti da multipli dispositivi a 8 bit al fine di incrementare la capacità di trasferimento dei dati
- Esempio, una ipotetica memoria da 16 GB potrebbe essere composta da 8 moduli di memoria (ciascuno con bus dati a 8 bit) al fine di eseguire trasferimenti a 64 bit



https://youtu.be/7J7X7aZvMXQ?si=5QPrKxwYVjURjbAt

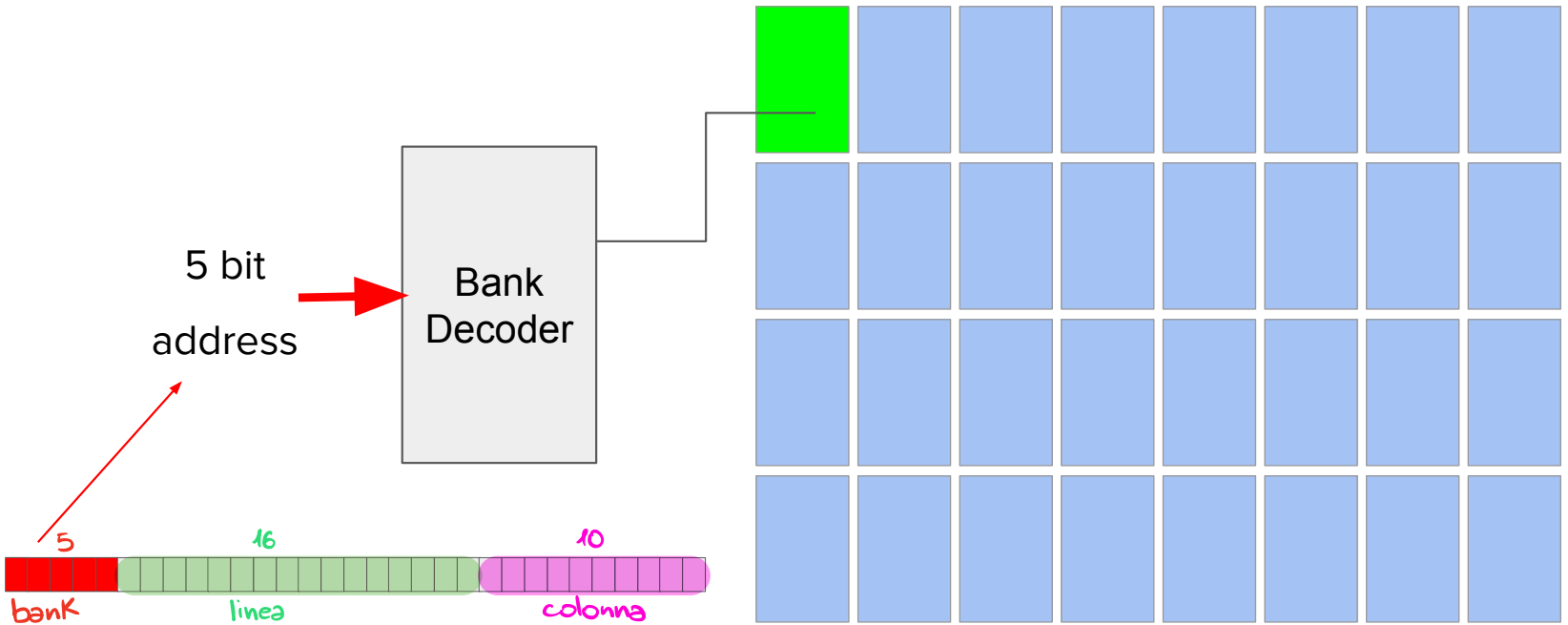
Elemento di memoria in una DRAM

- Come anticipato, l'elemento di memoria unitario (bit) in una DDR è composto da un transistor un condensatore
- Questa strategia consente di avere, a parità di area, una capacità molto più elevata rispetto a una SRAM
- Tuttavia, una SRAM risulta avere un tempo di accesso molto inferiore vs DRAM
- L'informazione memorizzata in una DRAM è la quantità di carica nel condensatore
- Tale carica, tende a svanire nel tempo ed è necessario un continuo *refresh* di ogni bit della memoria che viene realizzato attraverso un apposito circuito



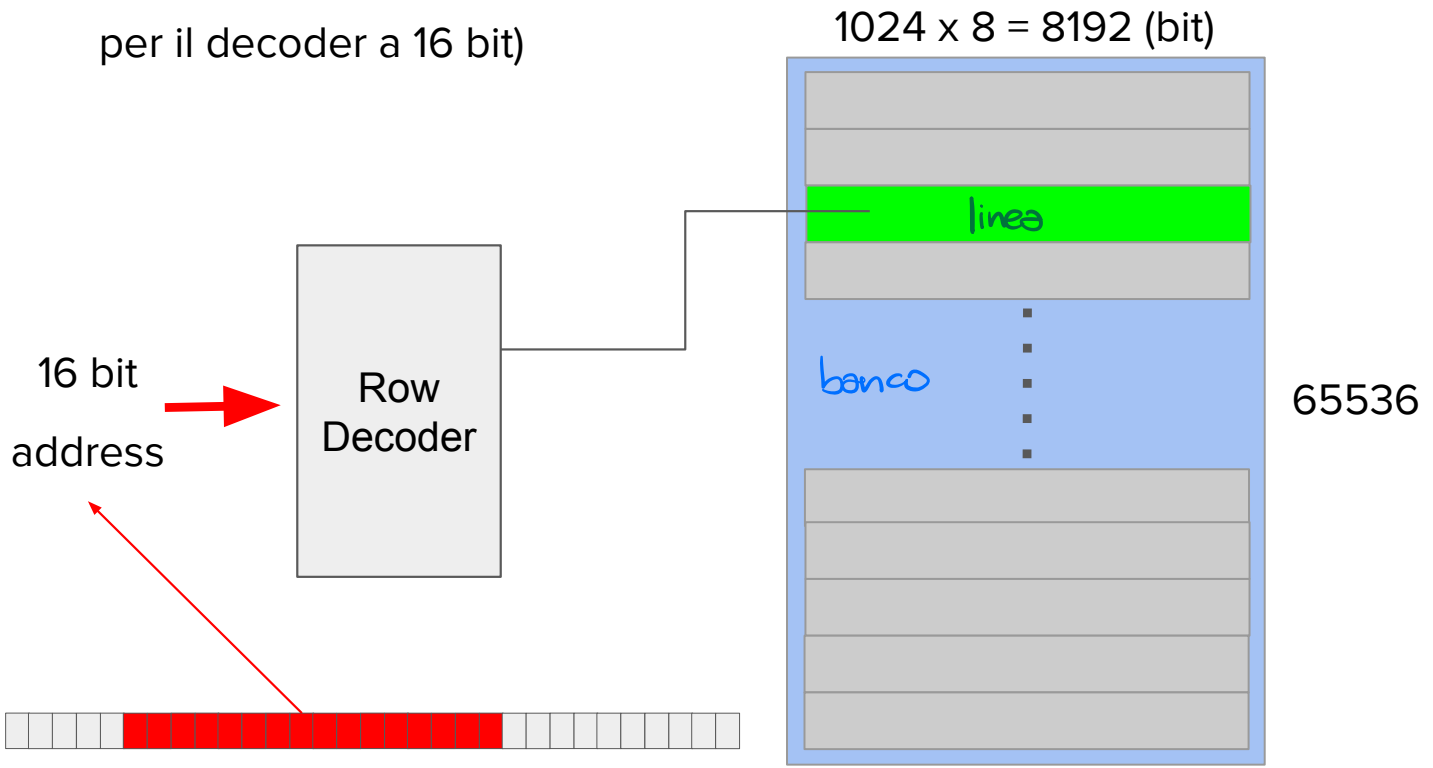
Organizzazione di una DRAM (2 GB) 1/4

- La memoria da 2 GB è organizzata in banchi (bank), ad esempio 32
- Ogni banco è identificato mediante i **5 bit** più significativi dell'indirizzo a **31 bit**
- Al suo interno la decodifica (di **II livello**) è organizzata nel modo seguente



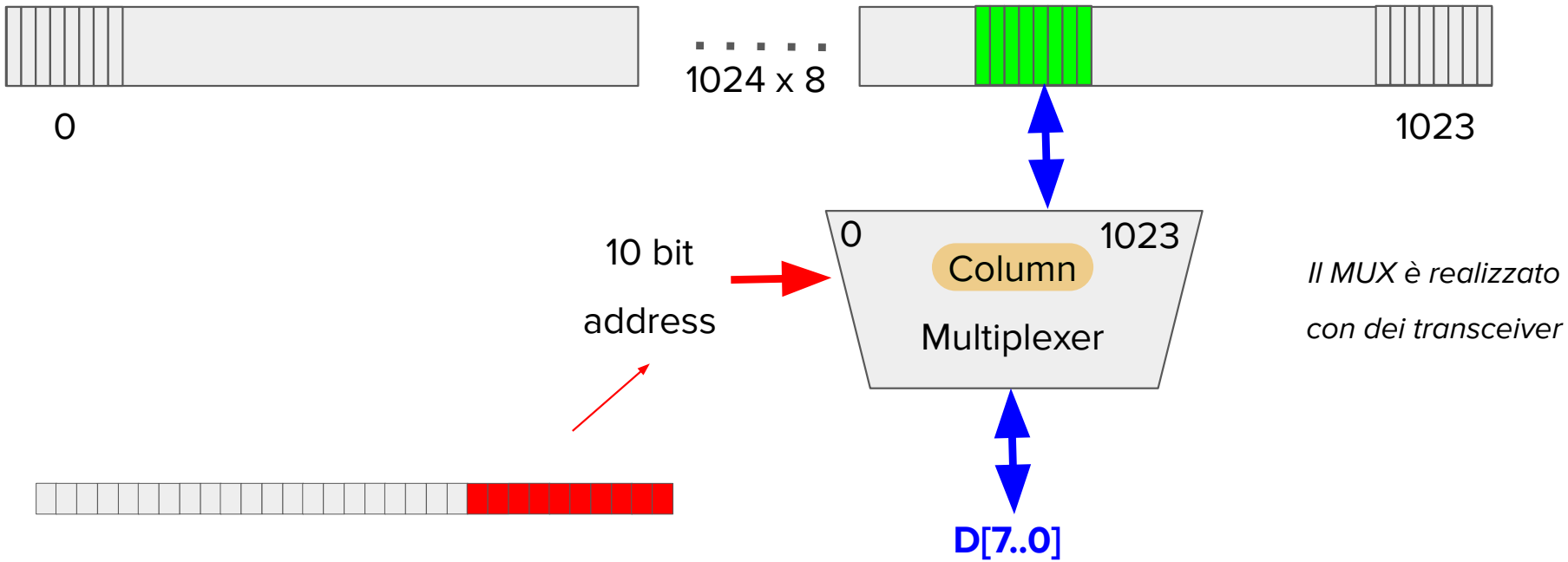
Organizzazione di una DRAM (2 GB) 2/4

- Ogni banco è organizzato in righe (16 bit successivi dell'indirizzo a 31 bit)
- Individuare e predisporre l'accesso ai dati nella linea è la fase più lenta (non solo per il decoder a 16 bit)



Organizzazione di una DRAM (2 GB) 3/4

- Individuata la linea, i dati (8 bit/modulo) possono essere acceduti rapidamente
- Inoltre, individuata la linea, è possibile eseguire veloci accessi in sequenza (*burst*)
- Gli accessi burst sono fondamentali per riempire le memorie cache (*line-fill*)



Organizzazione di una DRAM (2 GB) 4/4

- E’ importante osservare che, la priorità, è reperire il byte/dato richiesto (in verde) e inviarlo alla CPU il prima possibile, al fine di consentire alla CPU di proseguire il proprio lavoro

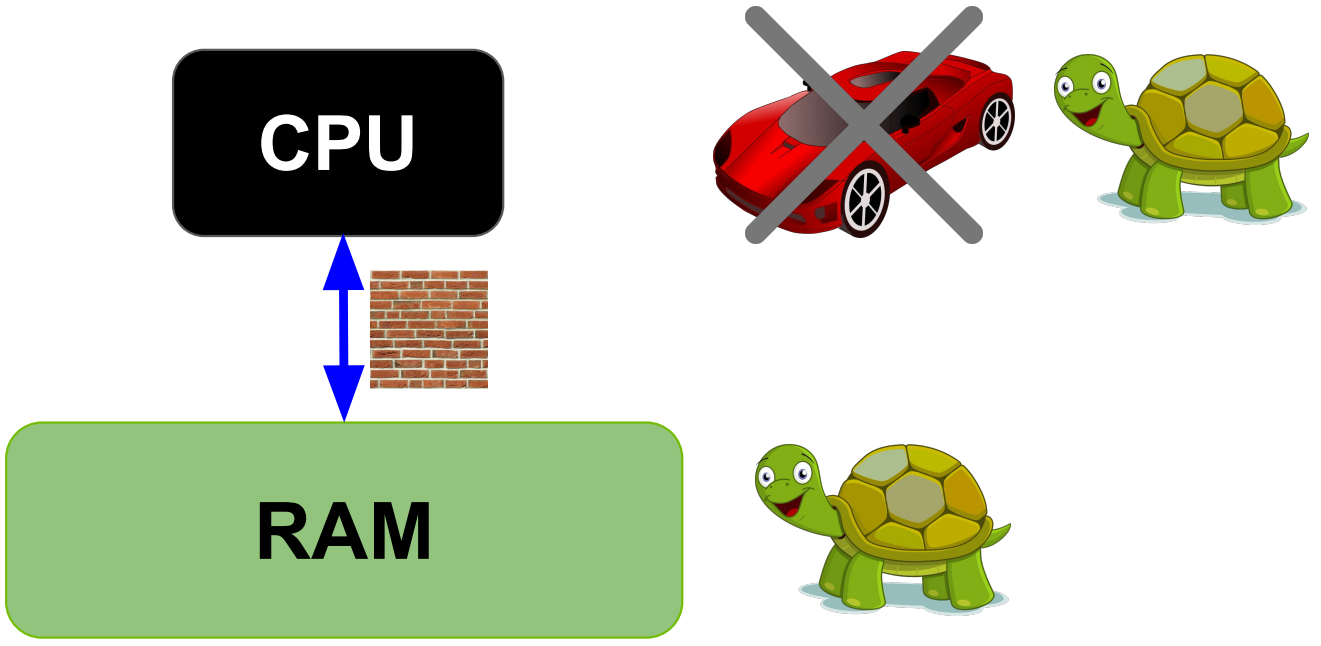
- Successivamente, in *background* e senza l’intervento diretto della CPU, sarà eseguito un line-fill degli altri dati appartenenti alla line di cache nella quale è contenuto il byte/dato verde
- Questa strategia consente di rifornire la CPU con i dati richiesti appena possibile, in modo da non rimamere bloccata ulteriormente in attesa di quel dato, e, in background, di riempire una linea di cache sfuttando il principio di località discusso in seguito ipotizzando l’utilizzo di dati in prossimità a breve

Introduzione alle memorie cache

- Con riferimento alle memorie DDR, una volta che è stata individuata la linea (row open), è possibile accedere a tutti i dati in essa contenuti molto rapidamente (poiché le fasi precedenti non sono più necessarie)
- Questo, consente di eseguire veloci accessi *burst* a dati multipli senza dover eseguire nuovamente la decodifica di tutti gli indirizzi (ma solo di quelli che identificano i dati desiderati all'interno della linea generabili in modo automatico)
- Tale caratteristica è fondamentale per eseguire velocemente riempimenti di linee intere (line-fill) nelle cache
- In seguito, vedremo come sia possibile sfruttare questa caratteristica e la tipica modalità di accesso ai dati per velocizzare mediante il *caching* gli accessi alla lenta memoria centrale DRAM mediante l'utilizzo di più veloci SRAM

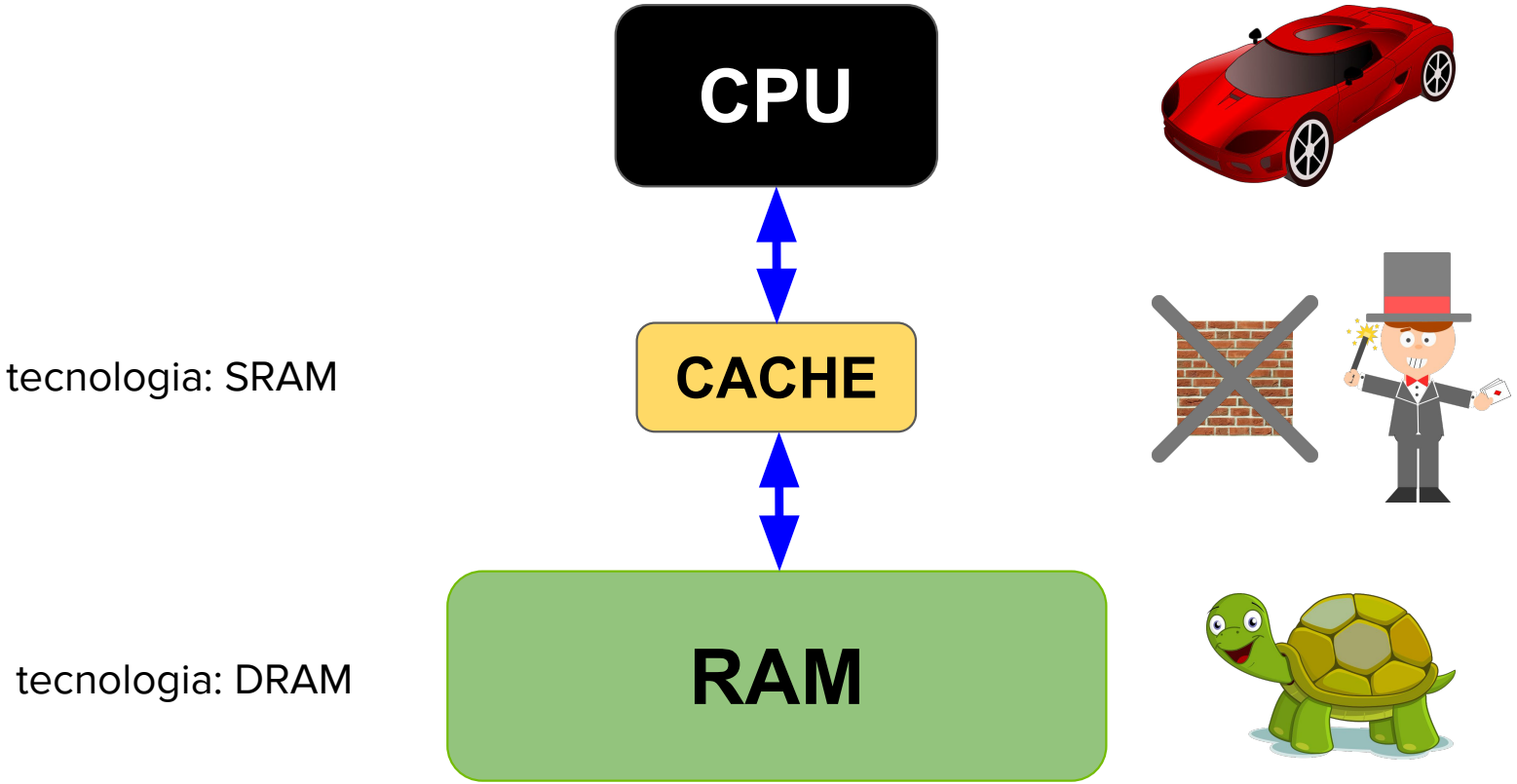
Memory wall

- Una CPU che non può rapidamente accedere ai dati deve rimanere in attesa (WAIT)
- Una memoria *lenta* riduce significativamente l'efficienza delle CPU



Cache

- Le cache sono memorie SRAM piccole e veloci (vs DRAM) poste tra la CPU e la RAM



Cache: principio di località spaziale e temporale

Al fine di emulare il comportamento ideale di una memoria, si sfruttano due osservazioni per inserire nella memoria cache un sottoinsieme dei dati presenti in memoria RAM:

- la **località spaziale**: se si accede ad un dato ad un determinato indirizzo di memoria, è presumibile che si acceda anche a dati contigui o in prossimità (e.g. prelievo codice, accesso ad elementi consecutivi di un array, etc)

- la **località temporale**: se si accede ad un determinato dato in memoria, è presumibile che, successivamente, si acceda al medesimo dato (e.g. codice di un loop, funzioni chiamate frequentemente, accesso ad elementi non consecutivi di un array, etc)

Accesso alla memoria in presenza di cache

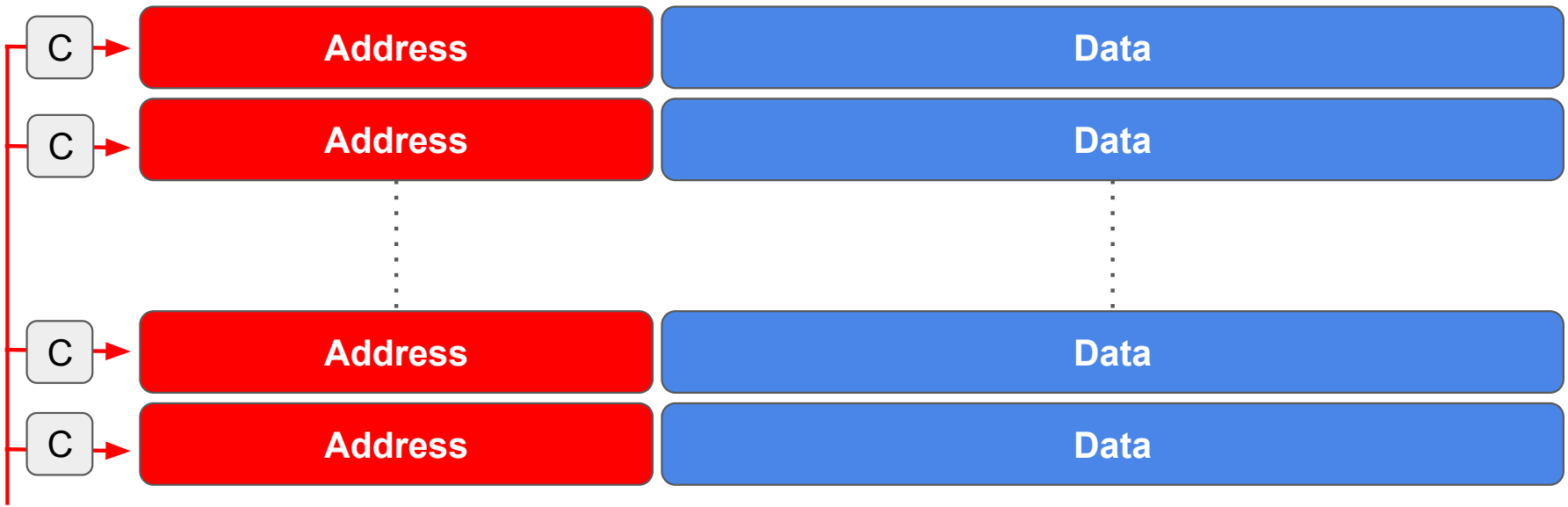
- Una cache è una memoria associativa (contiene indirizzi e dati) organizzata per linee; in ciascuna linea è presente l'indirizzo (unico associato alla linea) e i relativi dati
- Con le cache è anche presente un dispositivo, il controller della memoria, che esamina (indirizzi, segnali di controllo) le richieste di accesso alla memoria
- La richiesta è inviata a tutti i dispositivi di memoria, incluse le cache
- Se tale indirizzo è presente all'interno della cache (Hit), si esegue rapidamente l'accesso da tale dispositivo
- In caso contrario (Miss), si accede alla più lenta memoria di livello inferiore accedendo all'indirizzo richiesto; tuttavia, ipotizzando che sia valido il principio di località, si esegue un accesso burst per leggere un'intera linea della cache (line-fill)
- In caso di Hit, la velocità di accesso ai dati è quello della memoria veloce (ie, cache)

Organizzazione di una cache in linee

- Una cache è una memoria associativa organizzata per linee di lunghezza costante
- In ogni linea è presente l'indirizzo dei dati (contigui) presenti, esclusi i bit meno significativi che consentono di individuare ciascun byte

Cache Fully-associative 1/3

- La presenza o meno di un indirizzo è determinato confrontando l'indirizzo richiesto con quello presente in ciascuna linea
- Questo confronto, per essere veloce, deve avvenire contemporaneamente

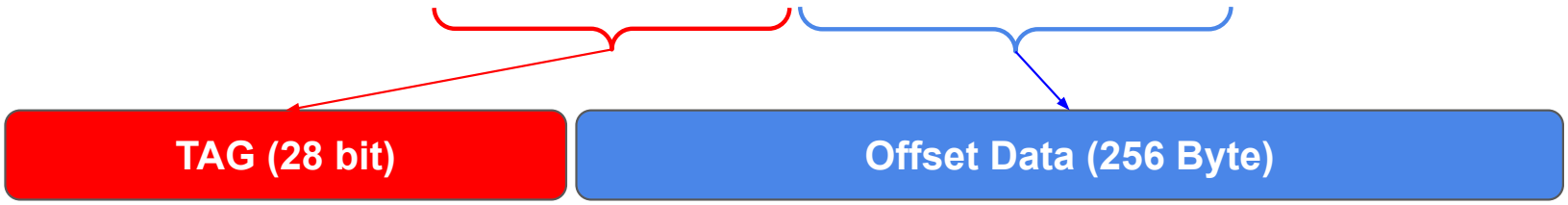


Indirizzo cercato (serve un comparatore per ogni linea)

Cache Fully-associative 2/3

- Ogni linea contiene 2^K byte; pertanto, nella comparazione dovranno essere esaminati tutti gli indirizzi esclusi i K bit meno significativi ⇝ questi K bit sono usati (tutti) per indirizzare i singoli byte nella riga.
- L'indirizzo completo, escludendo tali bit meno significativi, è denominato **TAG**
- Esempio, CPU con 36 bit di indirizzo, cache da 128K con 512 linee da 256 byte (K=8)
  - Il TAG è composto dai 36-8 = 28 bit più significativi dell'indirizzo
  - Sono necessari 512 comparatori a 28 bit per confrontare i TAG
  - Ogni byte è localizzato (**Offset**) all'interno della linea mediante i K=8 bit meno significativi dell'indirizzo a 36 bit
  - Indirizzo completo: **A35**,**A34**,....,**A8**,**A7**,**A6**,**A5**,**A4**,**A3**,**A2**,**A1**,**A0**

↳ Ogni byte è indirizzabile
=> $256 = 2^8$
=> 8 bit (LSB) sono usati per "indirizzare" i singoli byte della linea



Cache Fully-associative 3/3

- In una cache *fully associative*, ogni indirizzo viene collocato in una linea libera senza altri vincoli (ie, ogni indirizzo può finire in qualsiasi linea di cache)
- Per questa ragione serve un comparatore (veloce e costoso) per ogni linea
- Cosa accade se non ci sono più linee libere? Per il principio di località (temporale), si dovrebbe eliminare la linea che da più tempo è stata inserita nella cache
- Tuttavia, tenere traccia di questo aspetto (counter + altri comparatori?) e l'elevato numero di comparatori per il confronto con ogni linea le rende poco utilizzate in favore di altre strategie come:
  - Direct-mapped (i bit meno significativi del TAG vincolano univocamente la linea)
  - Set associative, come le precedenti ma sono presenti più linee per ogni possibile opzione aumentando la flessibilità

cache fully associative
→ cioè sono raggruppate le linee sulla base dei loro indirizzi => è possibile scorrere solo il gruppo (set) di linee in cui potrebbe essere il tag
alla ricerca del tag

Cache Direct-mapped

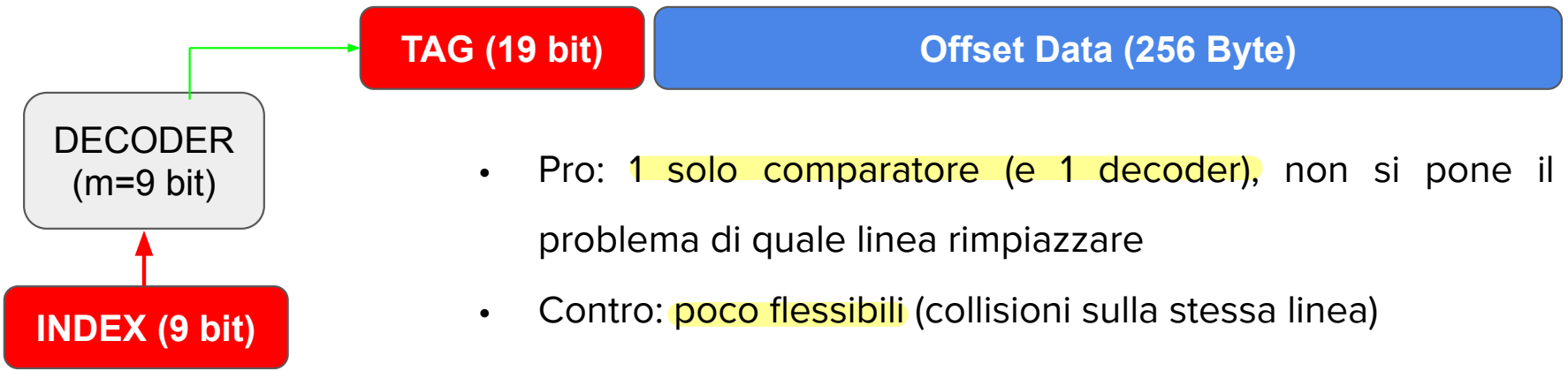
- Gli m bit meno significativi del TAG, denominati INDEX, determinano univocamente la linea corrispondente nella cache
- Indirizzi diversi con lo stesso INDEX collidono

ossia competono. Se un nuovo dato destinato ad un INDEX già occupato viene richiesto dalla CPU => il nuovo rimpiazza il vecchio

- Processore con 36 bit di indirizzo, 512 linee di cache da 256 byte (cache da 128 K)
- In tal caso, l'INDEX è composto dai m=9 bit meno significativi del TAG (19 bit)

$2^9 = 512$ (linee della cache)

- La decodifica dell'INDEX individua univocamente la linea di cache associata

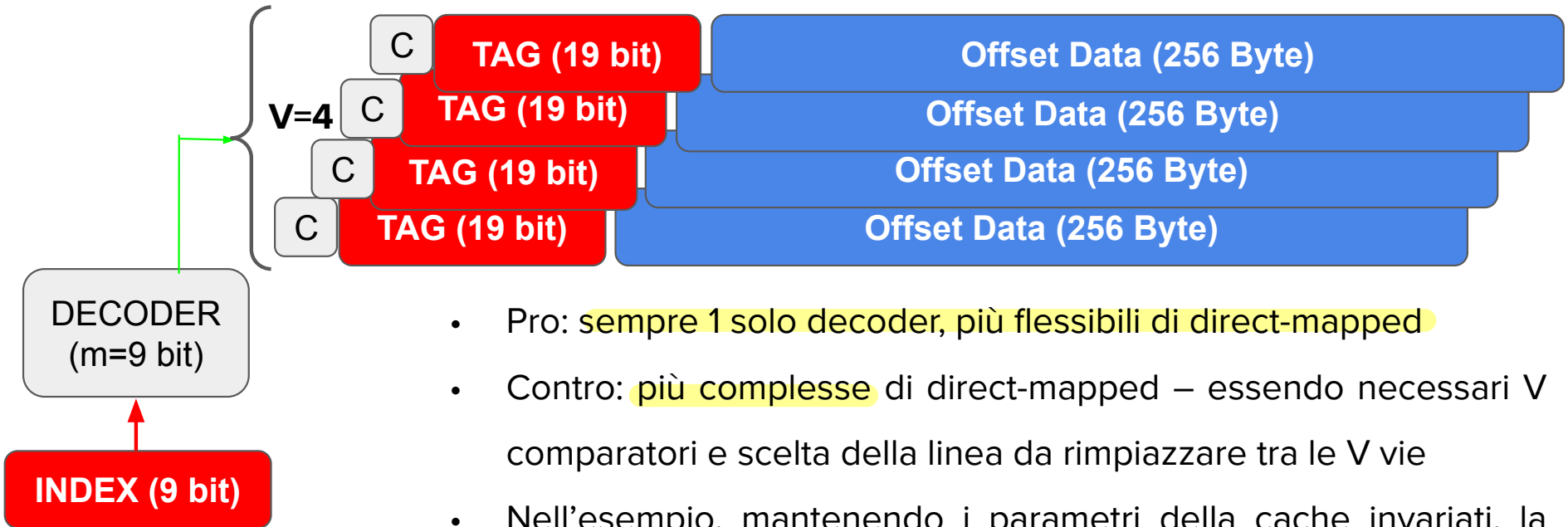


- Pro: 1 solo comparatore (e 1 decoder), non si pone il problema di quale linea rimpiazzare
- Contro: poco flessibili (collisioni sulla stessa linea)

Cache Set Associative a V vie

- dalle direct mapped prende l'associazione (analoga) set = index
- dalle fully associative prende il controllo (parallelo) da eseguire per le linee (vie) una volta individuato il set.

- Le cache Set-associative combinano le caratteristiche delle due strategie precedenti
- Come le direct-mapped l'INDEX individua univocamente un set
- Tuttavia, per ogni entry (set) esistono più vie/linee, in totale V (ie, 4 nell'esempio)



- Pro: sempre 1 solo decoder, più flessibili di direct-mapped
- Contro: più complesse di direct-mapped – essendo necessari V comparatori e scelta della linea da rimpiazzare tra le V vie
- Nell'esempio, mantenendo i parametri della cache invariati, la dimensione totale della cache aumenta di un fattore V (ie, 4)

Scelta della linea e altre informazioni

- La scelta della linea da rimpiazzare (problema inesistente nel caso directly-mapped) può essere eseguito in accordo a varie strategie. Tra le quali:
  - **Random**: semplice ma sorprendentemente efficace
  - **LRU** (Least Recently Used): basata su un contatore
  - . . .

- In realtà, ogni linea di cache contiene anche altre informazioni:
  - **Dirty bit**: indica se la linea è stata modificata, in caso non sia così non è necessario modificare i dati corrispondenti nel livello gerarchico più basso (RAM) nel caso sia necessario rimpiazzare quella linea (perché la RAM non viene aggiornata subito, alla modifica di un dato in cache.)
  - **Invalid bit**: indica se la linea contiene dati validi, utile per evitare di utilizzare dati mai inseriti nella cache e quindi non significativi => 1 = dato utile (caricato dalla RAM). 0 = ciarpame o dato non aggiornato.

Parametri che impattano sulla frequenza di miss (le 3 “C”)

Compulsory

Il primo accesso a qualunque blocco (linea) di memoria (cold start o first-reference misses)

Capacity

La cache non è in grado di contenere tutti i blocchi necessari all’esecuzione di un programma ovvero il *working set* è maggiore della capacità della intera cache. Ne deriva che un blocco viene scartato per essere poi richiamato subito dopo

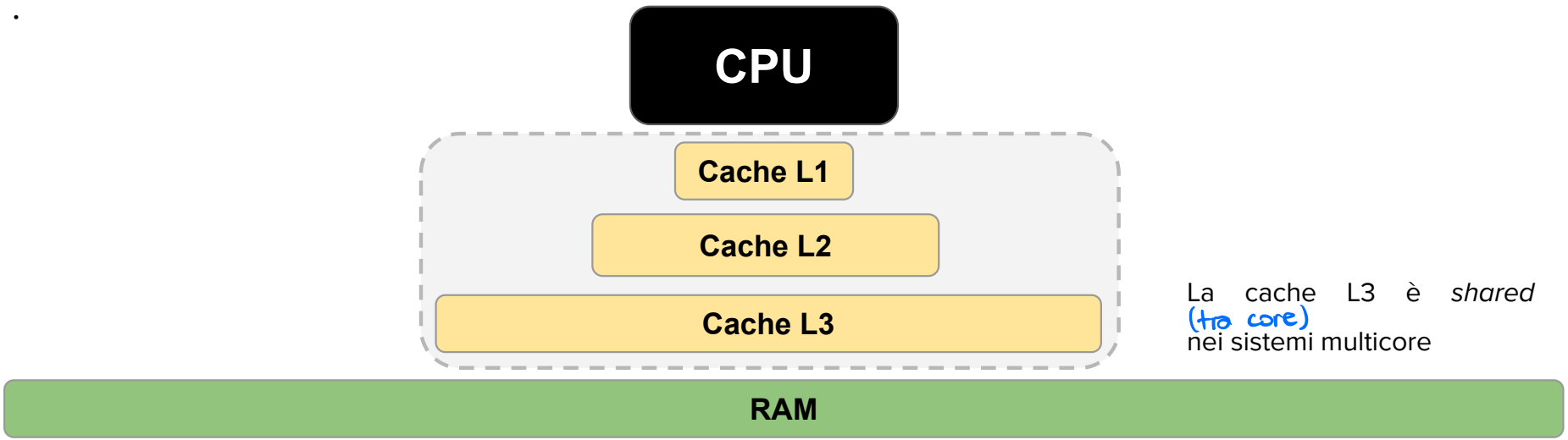
Conflict

Le restrizioni sulla allocazione in cache (se non fully associative): index uguale per un numero di blocchi contemporaneamente utilizzati superiore alla associatività (*collision misses*). Di fatto si è obbligati a scartare blocchi che potrebbero servire immediatamente dopo

Cache a più livelli

- In realtà, in un sistema reale sono presenti all'interno del chip più livelli di caching
- Il livello superiore, più veloce, contiene un sottoinsieme delle informazioni presenti nei livelli inferiori, più lenti
- Possono sorgere problemi di coerenza tra i dati (gestiti con appropriate strategie)

.

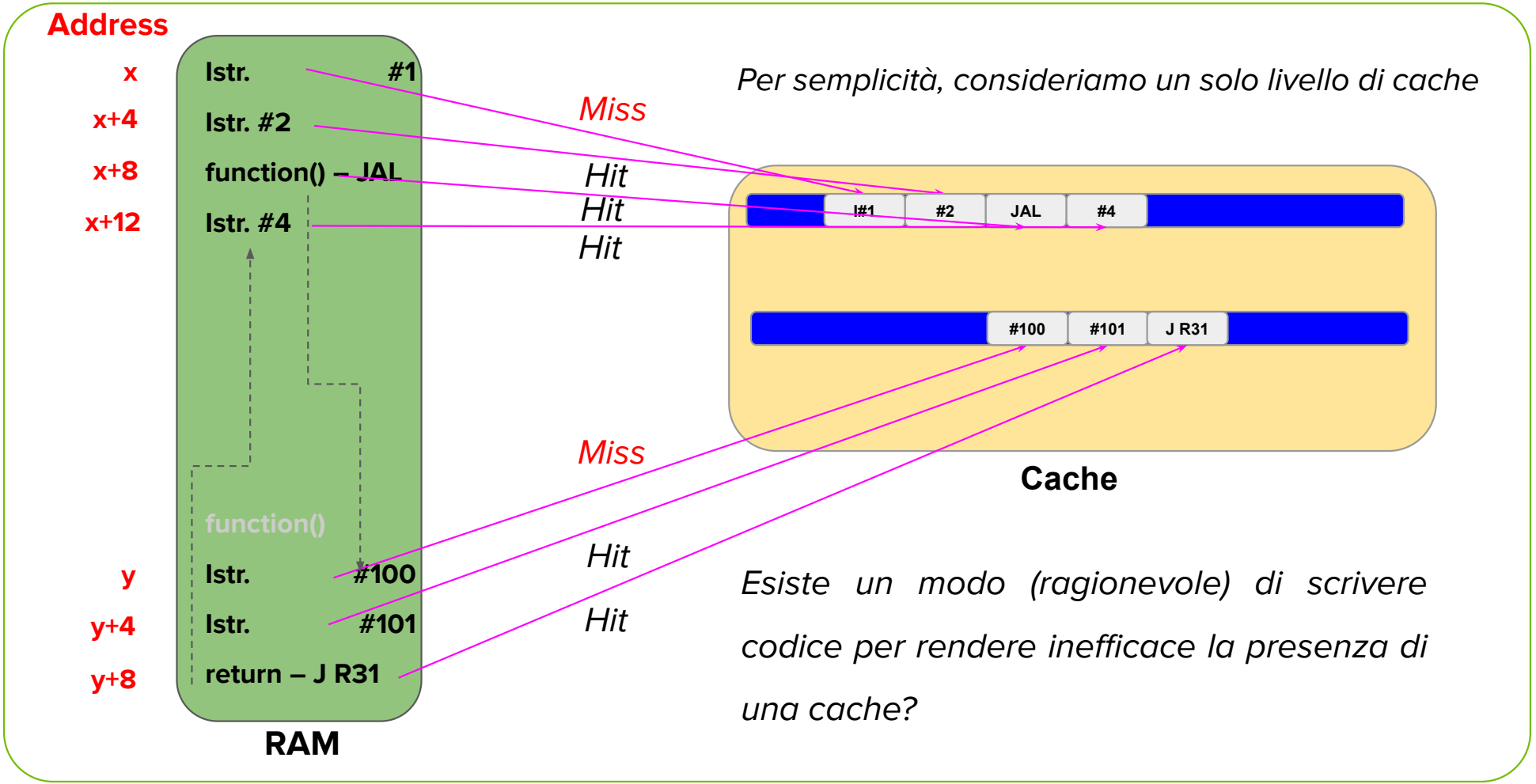


La cache L3 è *shared* (tra core) nei sistemi multicore

Caso di studio I: *fetch* del codice 1/2

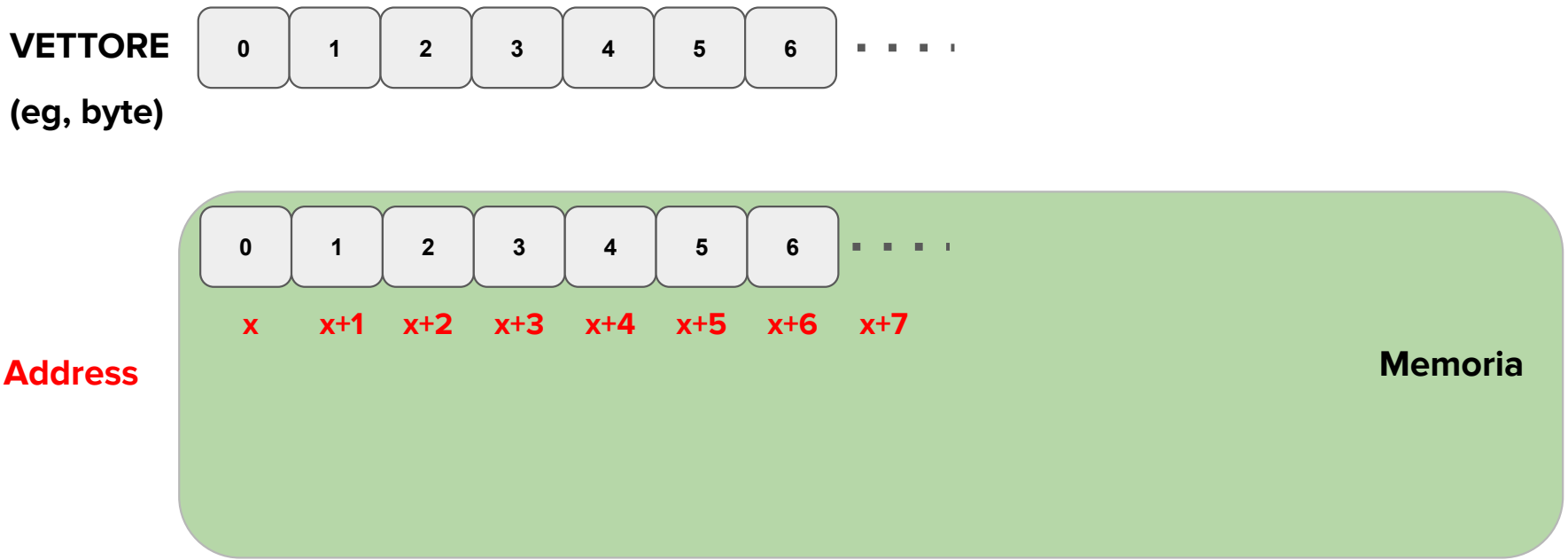
- Il codice che deve essere eseguito da un sistema di elaborazione è letto dalla memoria a indirizzi consecutivi
- Questo regolare comportamento varia solo in presenza di salti (condizionati o incondizionati come chiamate a funzioni), interrupt o altre circostanze che portano a eseguire codice a indirizzi diversi
- Tuttavia, anche nel caso precedente (eg, chiamata a una funzione), si cambierà l'indirizzo nel quale effettuare il fetch (ie, modifica del PC) si procederà con la lettura del codice della funzione a indirizzi consecutivi
- Per le ragioni evidenziate, il principio di località (spaziale e temporale) risulta valido nell'accesso al codice
- TIpicamente, esiste una cache dedicata al codice separata da quella dei dati

Caso di studio I: *fetch* del codice 2/2



Come sono memorizzati i vettori in memoria

- Elementi consecutivi di un vettore sono memorizzati a indirizzi consecutivi di memoria
- L'allineamento dei dati in memoria è cruciale per ottenere le migliori prestazioni
- In alcuni casi, eg DLX, è indispensabile fare accessi allineati alla memoria



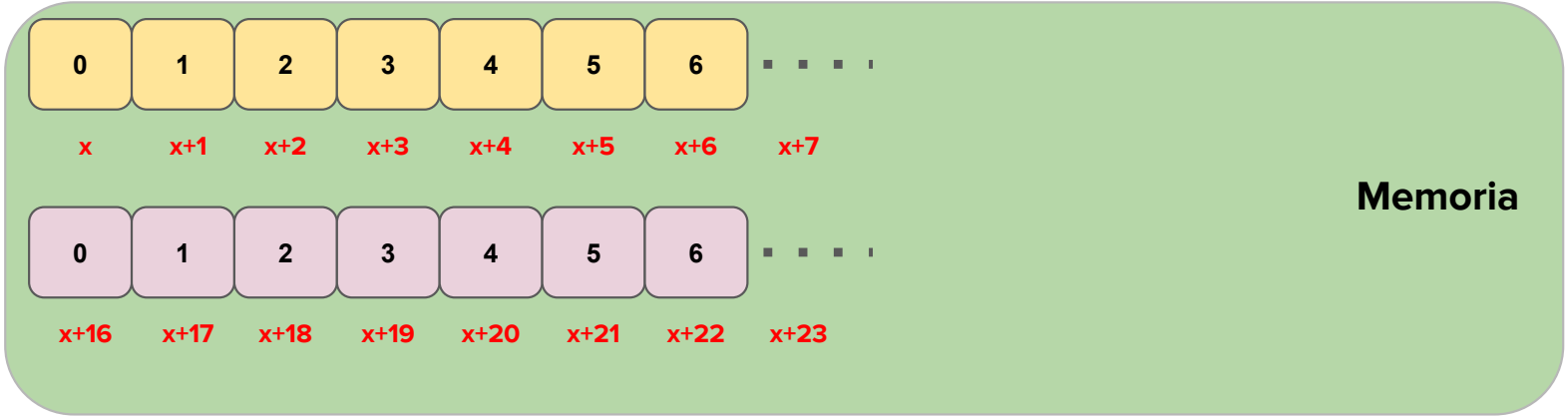
Come sono memorizzati le matrici in memoria

- Nel caso di matrici: righe: partendo dalla prima riga, gli elementi di ciascuna riga sono memorizzati a indirizzi consecutivi (*raw major order*)

Matrice

(eg, byte)

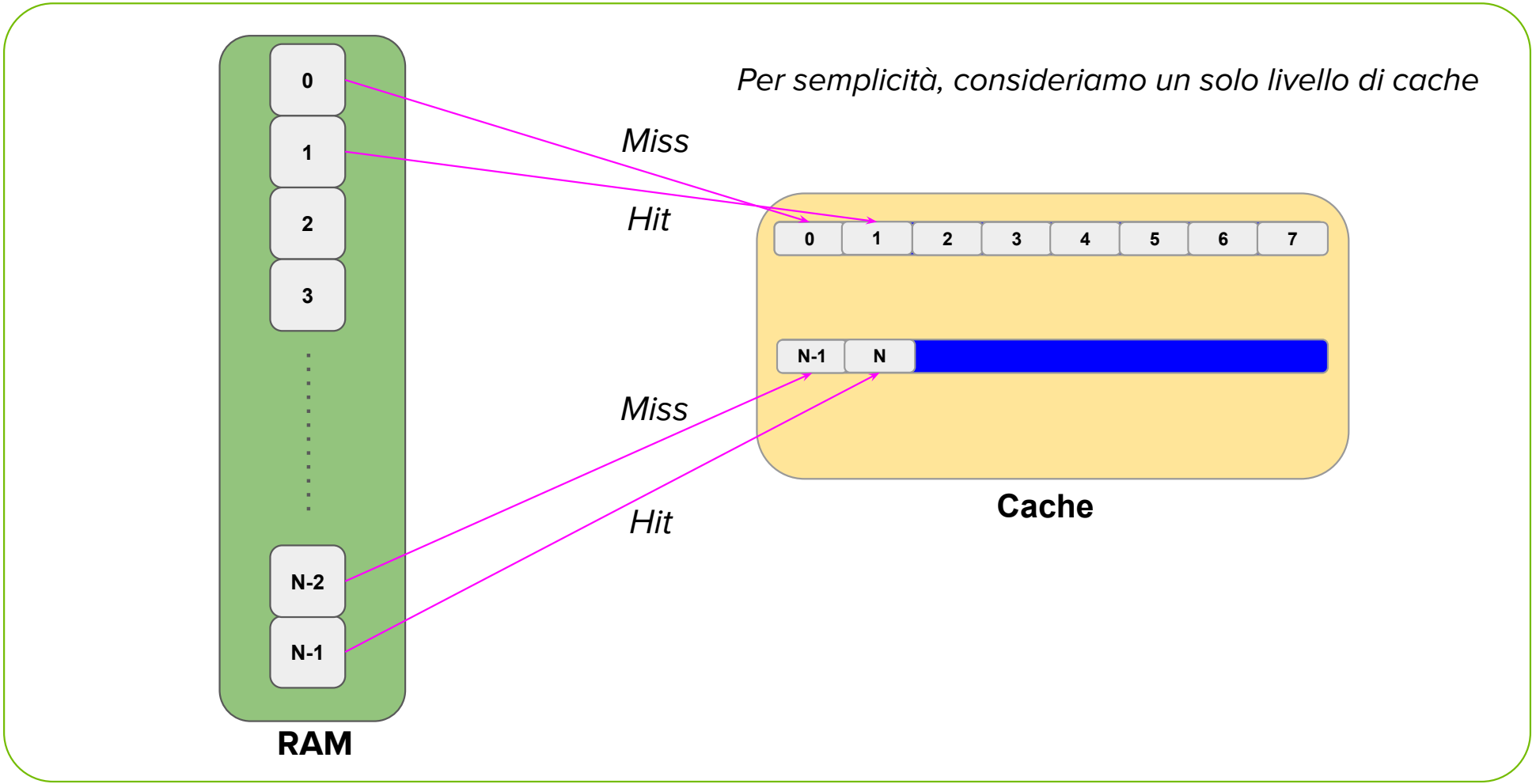
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32															
48															



Caso di studio II: accesso a un vettore 1/2

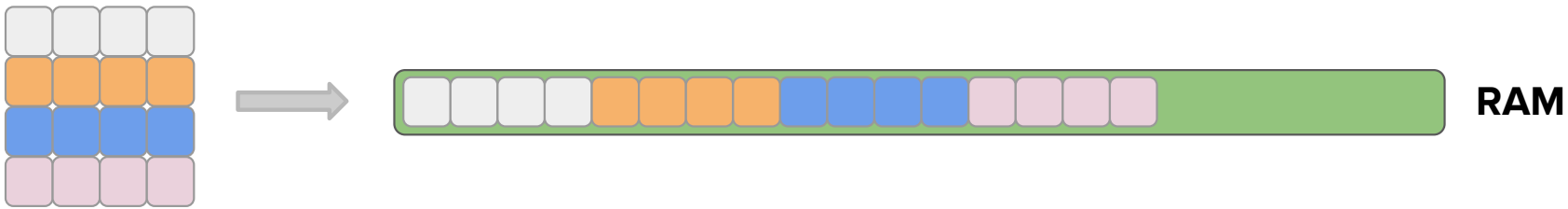
- Spesso, l'accesso a un vettore avviene facendo accessi a elementi consecutivi
- Se questo è il caso, le cache sono molto efficaci
- Si verifica un *miss*, con conseguente line-fill, solo quando si accede a un elemento non presente nella cache
- Nel caso di vettori sarebbe facile scrivere del codice che rende meno efficace lo sfruttamento della cache. Come?

Caso di studio II: accesso a un vettore 2/2



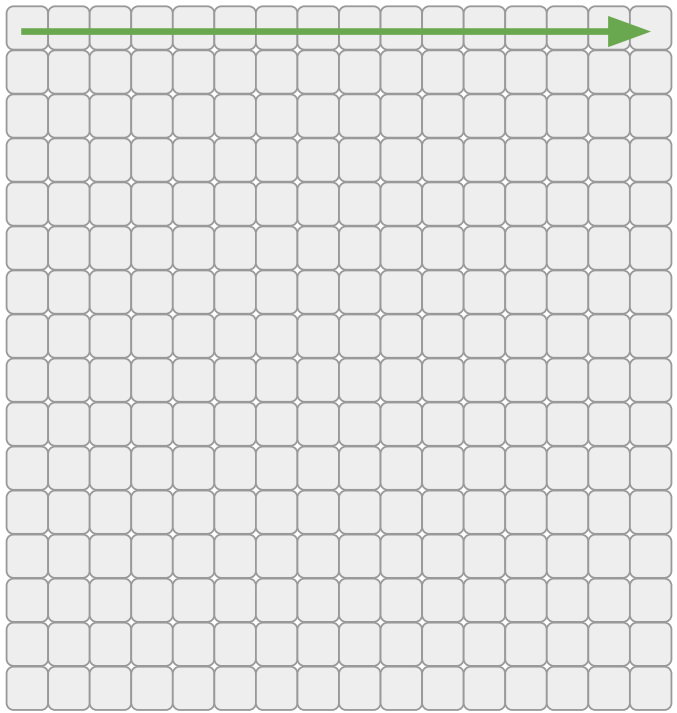
Caso di studio III: accesso a una matrice 1/3

- L'accesso a una matrice è ancora più critico e potrebbe facilmente portare a un utilizzo non efficace della cache
- Una matrice è organizzata in memoria come un vettore composto dalla concatenazione di righe consecutive

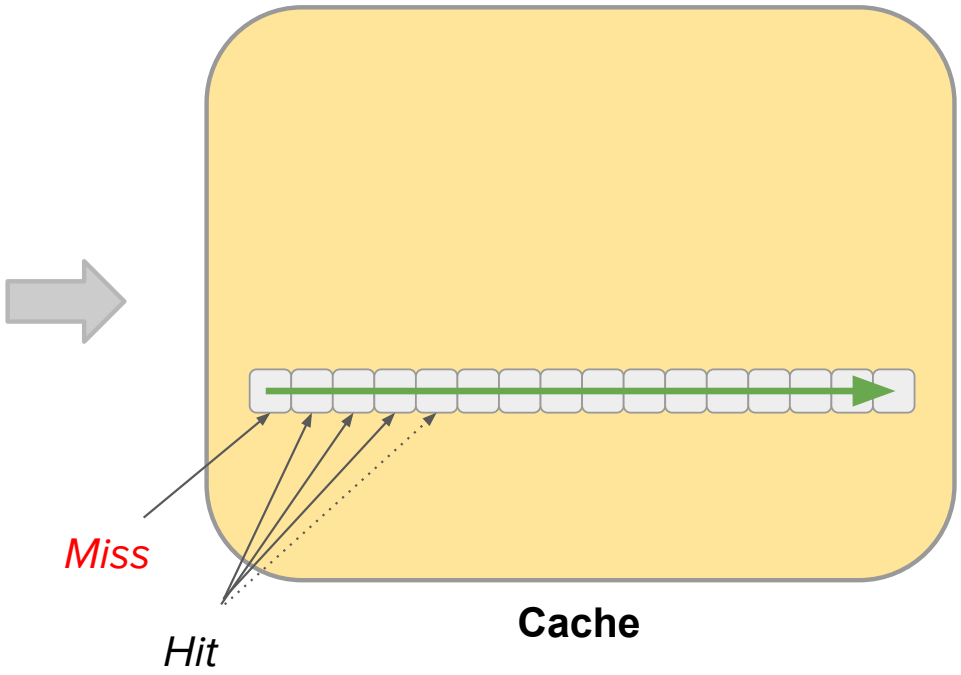


- Gli accessi a elementi nella stesso riga potrebbero* essere compatibili con il principio di località
- Al contrario, accessi a elementi di una stessa colonna, potrebbero* risultare non ottimali per una cache

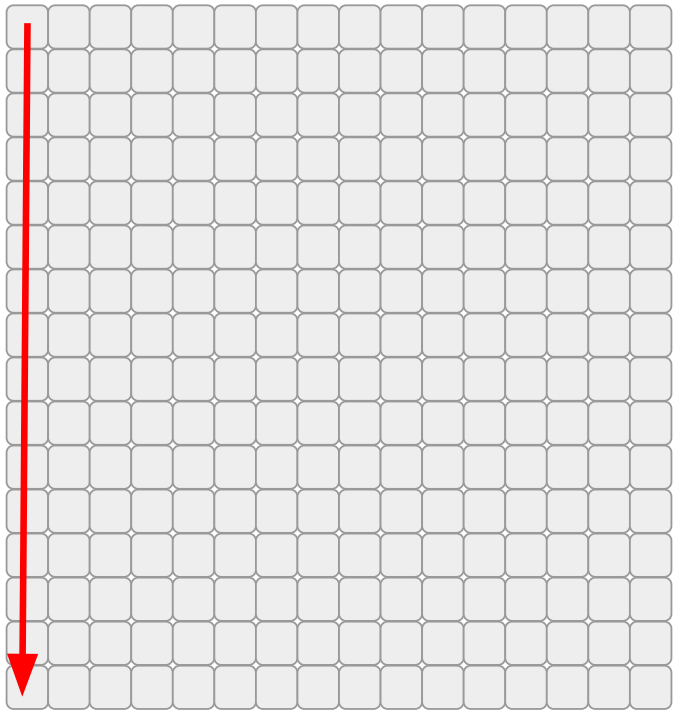
Caso di studio III: accesso a una matrice (per righe) 2/3



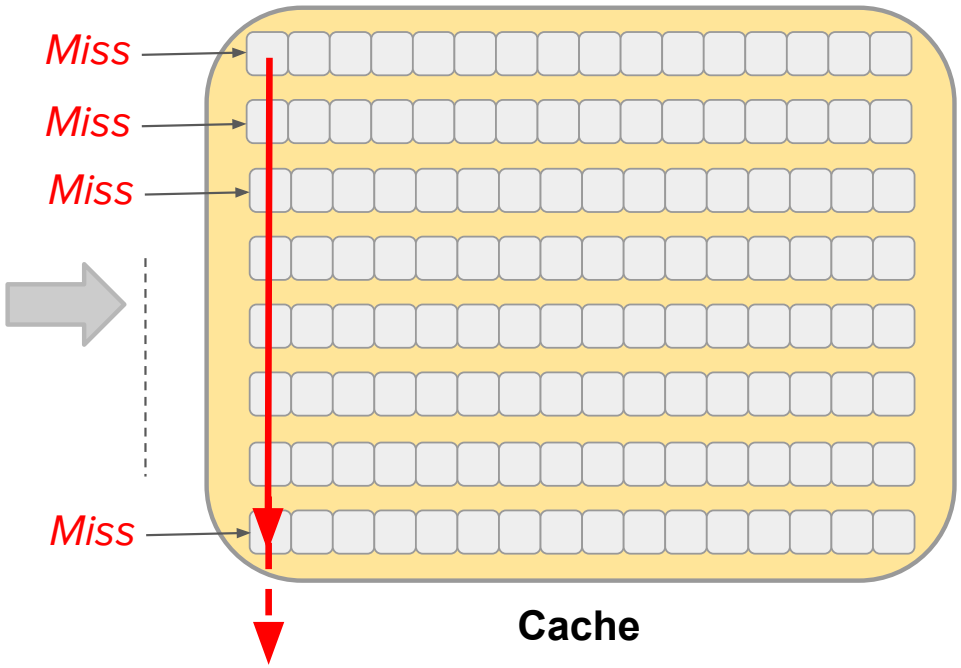
Per semplicità, nell'esempio, si assume che una riga sia della stessa dimensione di una linea di cache



Caso di studio III: accesso a una matrice (per colonne) 3/3



Per semplicità, nell'esempio, si assume che una riga sia della stessa dimensione di una linea di cache



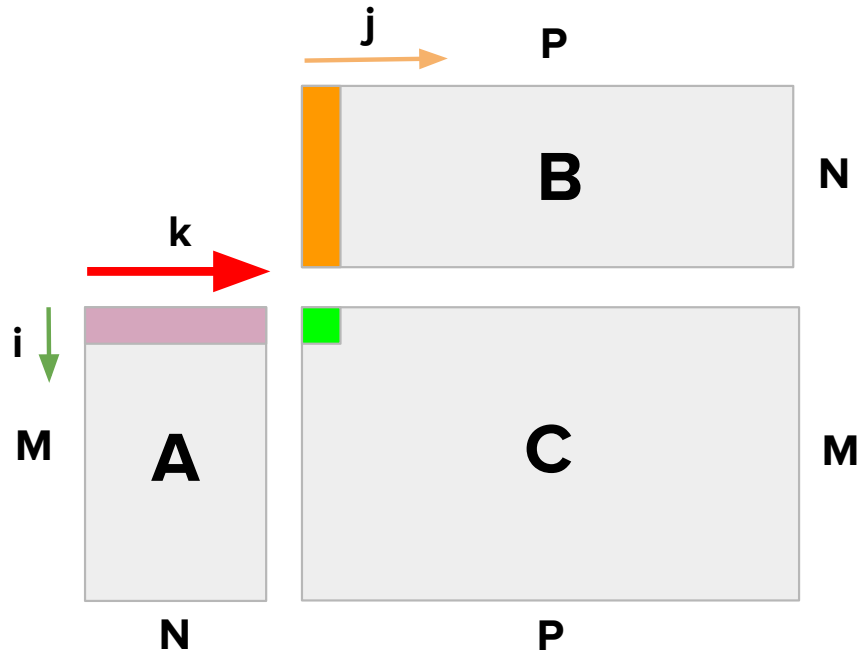
Linee finite: cosa accade?

Caso di studio IV: moltiplicazione tra matrici 1/9

- La moltiplicazione tra matrici è essenziale in molteplici contesti (eg, uno tra tutti AI)
- Tuttavia, può risultare critica,per le cache quando le matrici hanno grandi dimensioni
- Date due matrici **A**[MxN] e **B** [§NxP], il prodotto **C**[MxN] risulta:

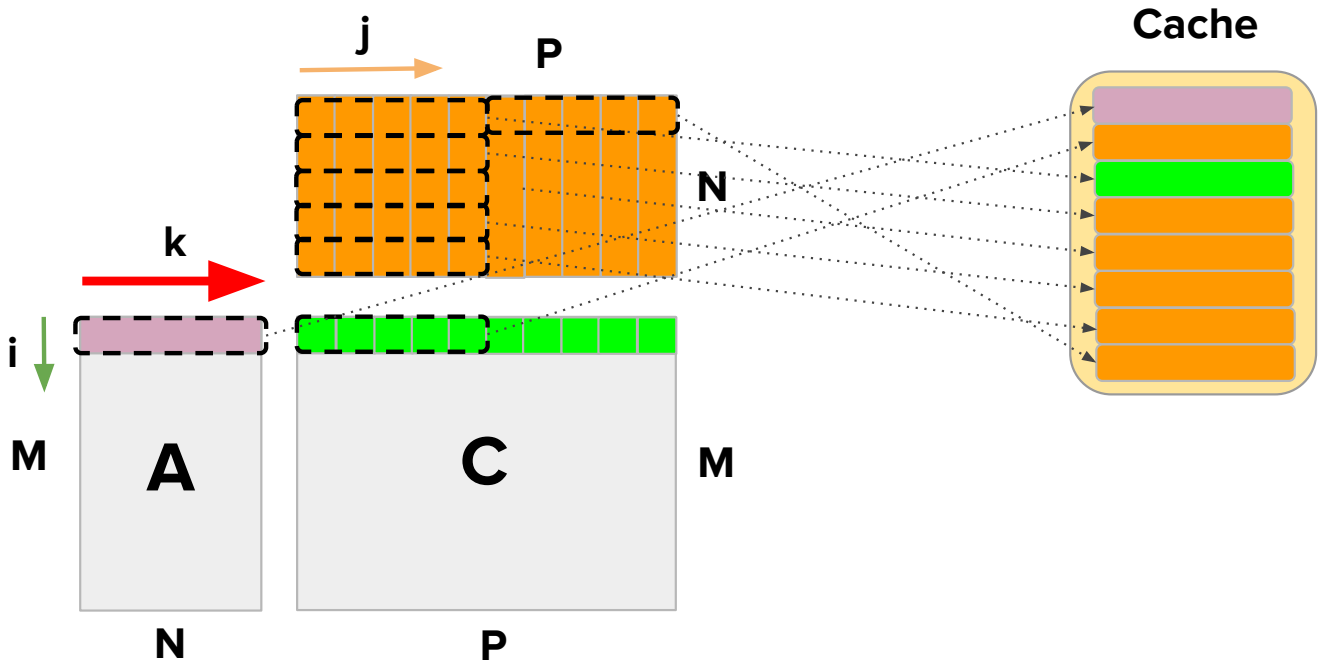
$$\mathbf{C}[i][j] = \sum \mathbf{A}[i][k] \cdot \mathbf{B}[k][j]$$

```
for (int i=0;i<M;i++)
 for (int j=0;j<P;j++)
  for (int k=0;k<N;k++)
    C[i][j] += A[i][k]*B[k][j];
```



Caso di studio IV: moltiplicazione tra matrici 2/9

- Con riferimento al codice precedente, si verificano continue miss (*capacity*) nel caso non si riesca a memorizzare interamente la matrice B nelle linee di cache

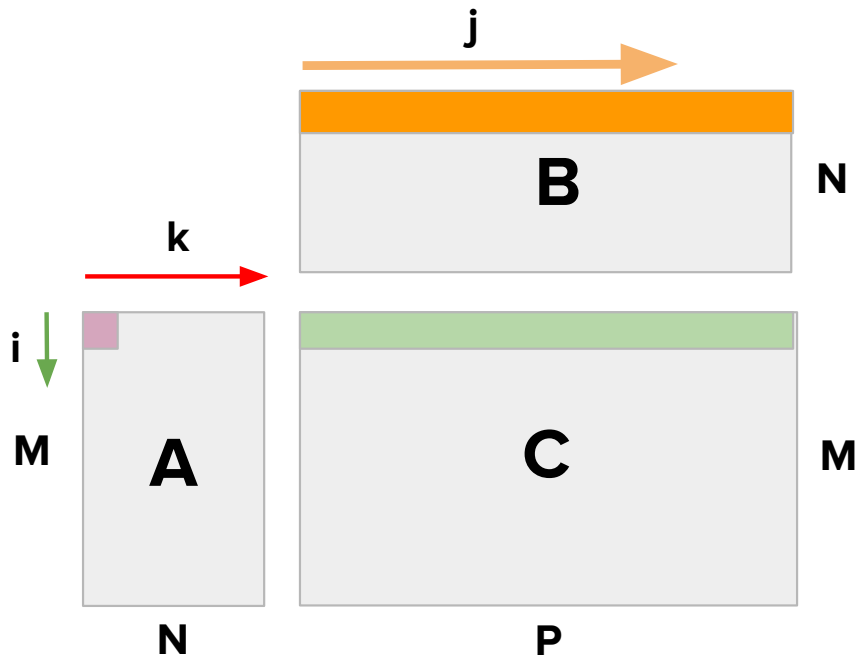


Caso di studio IV: moltiplicazione tra matrici 3/9

- Invertendo i due loop più interni (k e j) si calcolano i valori di C progressivamente con una maggiore efficacia nell'utilizzo della cache

```
for (int i=0;i<M;i++)
for (int k=0;k<N;k++)
for (int j=0;j<P;j++)
    C[i][j] += A[i][k]*B[k][j];
```

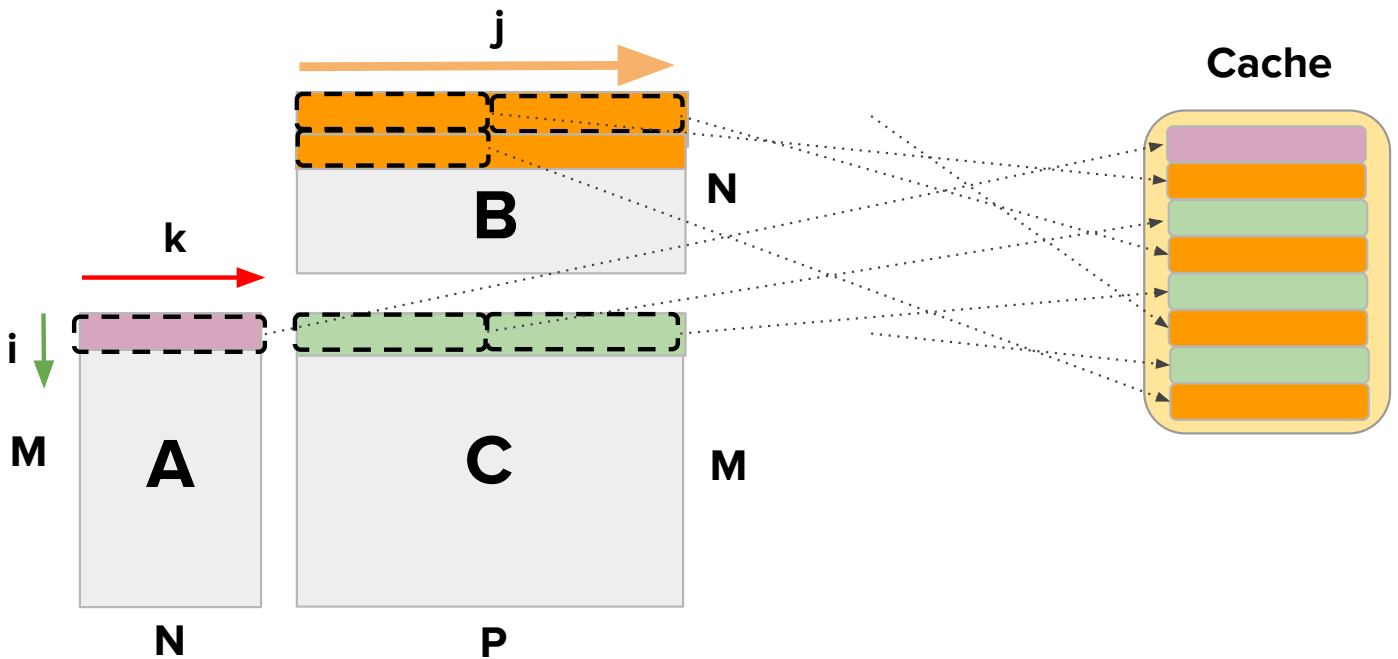
Perché si fanno più iterazioni hit leggendo gli elementi consecutivi delle righe e una volta per ciclo esterno gli elementi di colonna => più efficiente



Caso di studio IV: moltiplicazione tra matrici 4/9

la matrice è più piccola

- In questo caso, il *working set* è ridotto e ciò migliora l'impatto delle cache



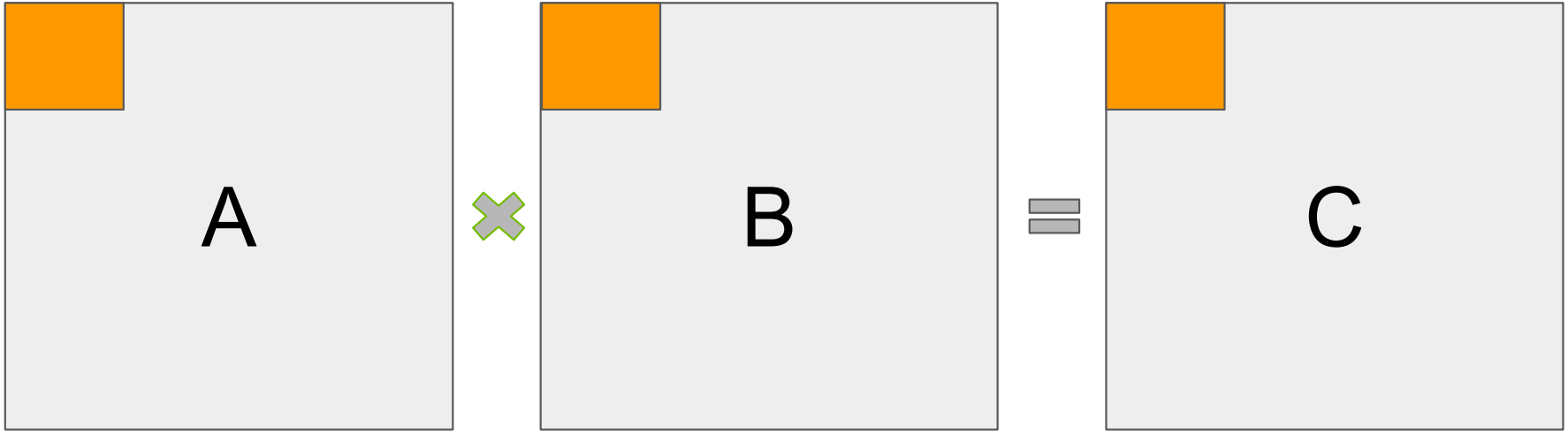
Caso di studio IV: moltiplicazione tra matrici 5/9

- Benchmark con un MacBook Air M2 – gcc optimization -O0 (ie, no optimization)
- Con vari ordini tra i tre loop utilizzati per la moltiplicazione tra matrici risultano i seguenti tempi di elaborazione :

Loop order	time [sec]
IJK	**5.1**
IKJ	**3.6**
KIJ	**3.6**
KJI	**3.7**
JKI	**3.7**
JIK	**5.1**

Caso di studio IV: moltiplicazione tra matrici 6/9

- Una strategia ancora più efficace nella moltiplicazione tra matrici consiste nel ridurre ulteriormente le dimensioni del *working set*
- Mediante una strategia denominata *loop tiling* o *loop blocking* si eseguono calcoli parziali su blocchi di matrici di dimensione limitata che possono sfruttare la meglio la cache



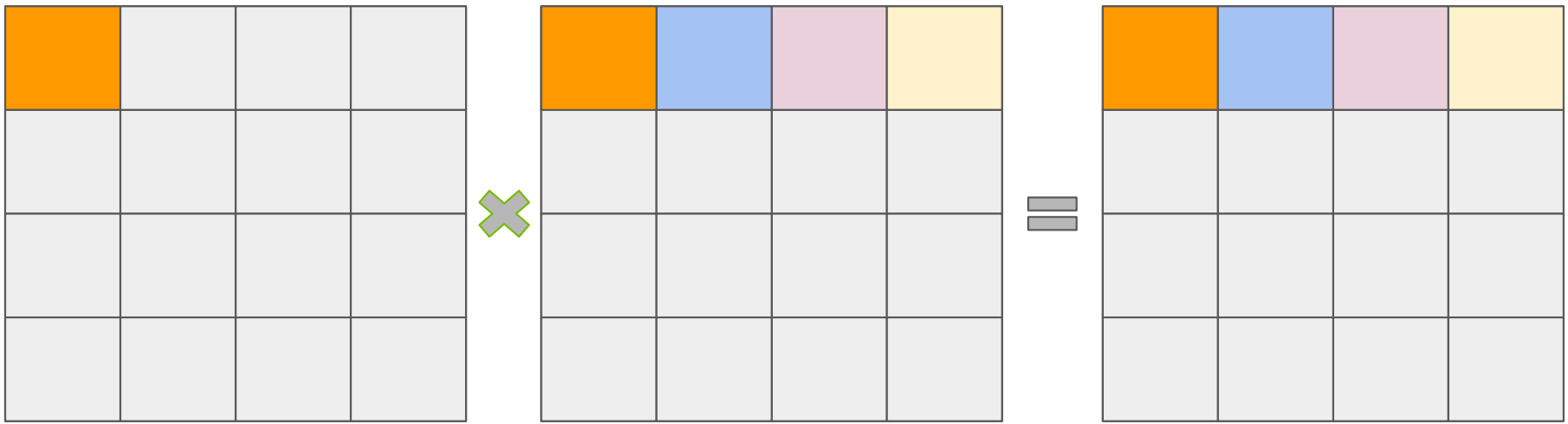
Caso di studio IV: moltiplicazione tra matrici 7/9

- Esempio di codice che esegue l'elaborazione dividendo le matrici in *tile*/*block*

```
for (int iB=0;iB<M;iB=iB+BLOCK_SIZE_i)
    for (int jB=0;jB<P;jB=jB+BLOCK_SIZE_j)
        for (int kB=0;kB<N;kB=kB+BLOCK_SIZE_k)
            for (int i=iB;i<iB+BLOCK_SIZE_i;i++)
                for (int j=jB;j<jB+BLOCK_SIZE_j;j++)
                    for (int k=kB;k<kB+BLOCK_SIZE_k;k++)
                        C[i][j] += A[i][k]*B[k][j];
```

Caso di studio IV: moltiplicazione tra matrici 8/9

- I tre loop sono suddivisi in blocchi di dimensioni tali da rendere più efficace l'utilizzo della/e cache riducendo il working set al solo calcolo all'interno del blocco
- Per esempio, mediante la progressione seguente:



Caso di studio IV: moltiplicazione tra matrici 9/9

- MacBook Air con processore Apple M2 – test eseguiti con gcc (optimization -O0)
  - 16 GB RAM
  - Data Cache* L1: 128 KB, linee da 128 byte, set associativa a 8 vie, L2 16 MB shared
- Test eseguito con matrici di interi: **A**=[512x1024], **B**[1024x512] ➜ **C**[512x512]
  - Peggior risultato: ≃ **1.96 s**, configurazione blocchi [i,j,k] = [1,1,1]
  - Miglior risultato: ≃ **0.54 s**, configurazione blocchi [i,j,k] = [128,512,32]
  - Speed-up ≃ **3.63**
- Il *tiling* consente di accedere ai dati in modo più *cache-friendly* anche se questo è solo un aspetto da considerare in determinati contesti
- Per esempio, con SIMD i dati nella matrice B non sono disposti in modo ottimale per questo paradigma di elaborazione

* Dati non ufficiali, valori stimati e resi disponibili da utenti online