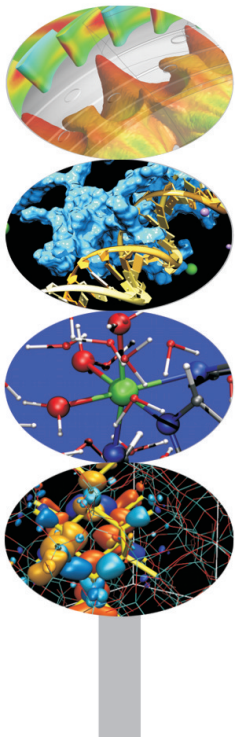
CINECA SCAI
SuperComputing Applications and Innovation

SAPIENZA
UNIVERSITÀ DI ROMA

Programmazione Avanzata

Vittorio Ruggiero

(v.ruggiero@cineca.it)



Roma, 26 Marzo 2014

CINECA

Outline

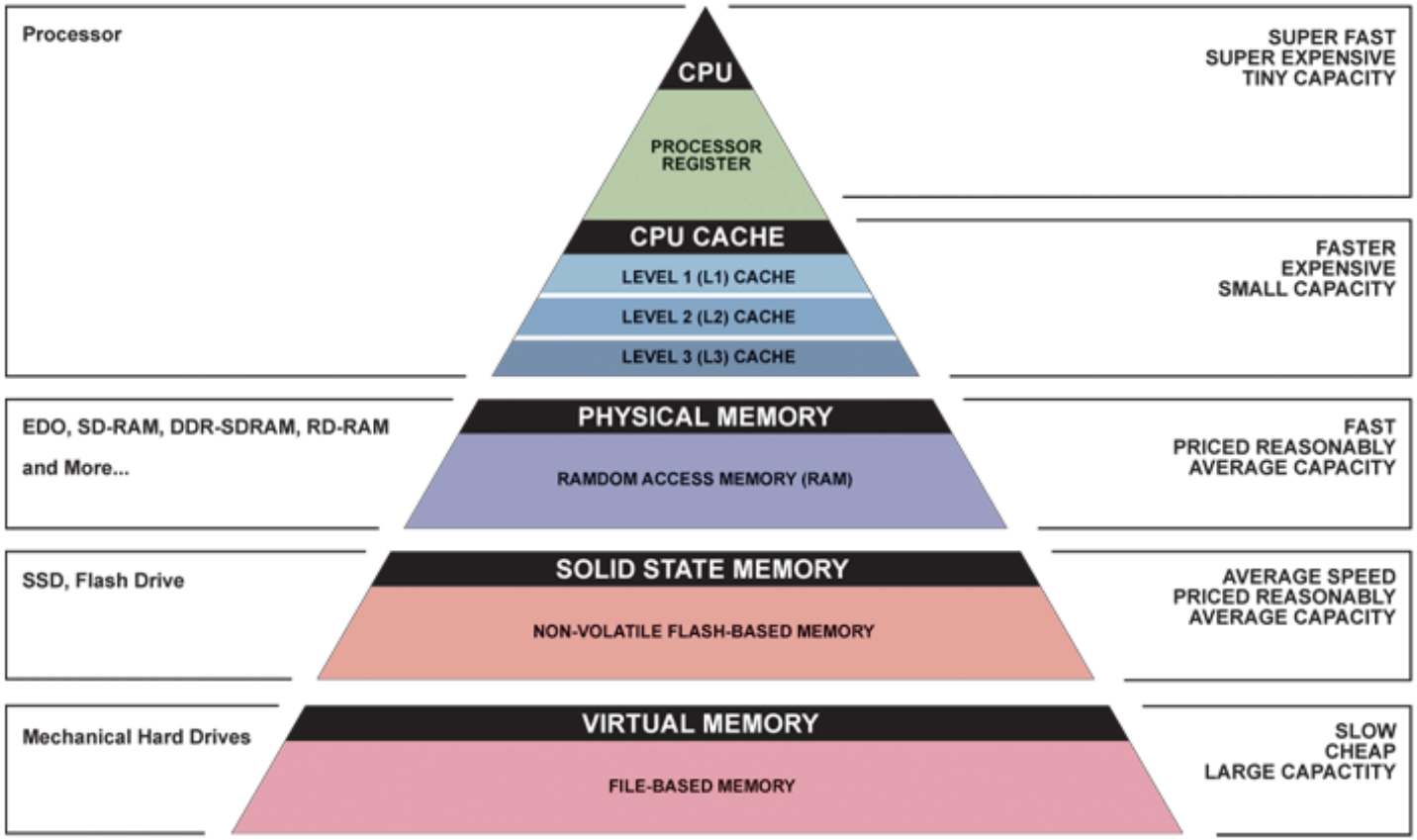
La cache e il sistema di memoria

Il sistema di memoria

- Capacità di calcolo delle CPU 2× ogni 18 mesi
- Velocità di accesso alla RAM 2× ogni 120 mesi
- Inutile ridurre numero e costo delle operazioni se i dati non arrivano dalla memoria

- Soluzione: memorie intermedie veloci
- Il sistema di memoria è una struttura profondamente gerarchica
- La gerarchia è trasparente all'applicazione, i suoi effetti no

La gerachia di memoria



▲ Simplified Computer Memory Hierarchy
Illustration: Ryan J. Leng

La Cache

Perché questa gerarchia?

La Cache

Perché questa gerarchia?

Non servono tutti i dati disponibili subito

La Cache

Perché questa gerarchia?

Non servono tutti i dati disponibili subito

La soluzione?

La Cache

Perché questa gerarchia?
Non servono tutti i dati disponibili subito
La soluzione?

- La cache è composta di uno (o più livelli) di memoria intermedia, abbastanza veloce ma piccola (kB $\div$ MB)
- Principio fondamentale: si lavora sempre su un sottoinsieme ristretto dei dati
  - dati che servono $\rightarrow$ nella memoria ad accesso veloce
  - dati che (per ora) non servono $\rightarrow$ nei livelli più lenti
- Regola del pollice:
  - il 10% del codice impiega il 90% del tempo
- Limitazioni
  - accesso casuale senza riutilizzo
  - non è mai abbastanza grande . . .
  - più è veloce, più scalda e . . . costa $\rightarrow$ gerachia di livelli intermedi.

La Cache

- La CPU accede al (più alto) livello di cache:
- Il controllore della cache determina se l'elemento richiesto è effettivamente presente in cache:
  - Si: trasferimento fra cache e CPU
  - No: Carica il nuovo dato in cache; se la cache è piena, applica la politica di rimpiazzamento per caricare il nuovo dato al posto di uno di quelli esistenti
- Lo spostamento di dati tra memoria principale e cache non avviene per parole singole ma per blocchi denominati linee di cache
- blocco = minima quantità d'informazione trasferibile fra due livelli di memoria (fra due livelli di cache, o fra RAM e cache)

Rimpiazzamento: principi di località

- Località spaziale dei dati
  - vi è alta la probabilità di accedere, entro un breve intervallo di tempo, a celle di memoria con indirizzo vicino (es.: istruzioni in sequenza; dati organizzati in vettori o matrici a cui si accede sequenzialmente, etc.).

Rimpiazzamento: principi di località

- Località spaziale dei dati
  - vi è alta la probabilità di accedere, entro un breve intervallo di tempo, a celle di memoria con indirizzo vicino (es.: istruzioni in sequenza; dati organizzati in vettori o matrici a cui si accede sequenzialmente, etc.).
  - Viene sfruttata leggendo normalmente più dati di quelli necessari (un intero blocco) con la speranza che questi vengano in seguito richiesti.

Rimpiazzamento: principi di località

- Località spaziale dei dati
  - vi è alta la probabilità di accedere, entro un breve intervallo di tempo, a celle di memoria con indirizzo vicino (es.: istruzioni in sequenza; dati organizzati in vettori o matrici a cui si accede sequenzialmente, etc.).
  - Viene sfruttata leggendo normalmente più dati di quelli necessari (un intero blocco) con la speranza che questi vengano in seguito richiesti.
- Località temporale dei dati
  - vi è alta probabilità di accedere nuovamente, entro un breve intervallo di tempo, ad una cella di memoria alla quale si è appena avuto accesso (es: ripetuto accesso alle istruzioni del corpo di un ciclo sequenzialmente, etc.)

Rimpiazzamento: principi di località

- Località spaziale dei dati
  - vi è alta la probabilità di accedere, entro un breve intervallo di tempo, a celle di memoria con indirizzo vicino (es.: istruzioni in sequenza; dati organizzati in vettori o matrici a cui si accede sequenzialmente, etc.).
  - Viene sfruttata leggendo normalmente più dati di quelli necessari (un intero blocco) con la speranza che questi vengano in seguito richiesti.

- Località temporale dei dati
  - vi è alta probabilità di accedere nuovamente, entro un breve intervallo di tempo, ad una cella di memoria alla quale si è appena avuto accesso (es: ripetuto accesso alle istruzioni del corpo di un ciclo sequenzialmente, etc.)
  - viene sfruttata decidendo di rimpiazzare il blocco utilizzato meno recentemente.

Rimpiazzamento: principi di località

- Località spaziale dei dati
  - vi è alta la probabilità di accedere, entro un breve intervallo di tempo, a celle di memoria con indirizzo vicino (es.: istruzioni in sequenza; dati organizzati in vettori o matrici a cui si accede sequenzialmente, etc.).
  - Viene sfruttata leggendo normalmente più dati di quelli necessari (un intero blocco) con la speranza che questi vengano in seguito richiesti.

- Località temporale dei dati
  - vi è alta probabilità di accedere nuovamente, entro un breve intervallo di tempo, ad una cella di memoria alla quale si è appena avuto accesso (es: ripetuto accesso alle istruzioni del corpo di un ciclo sequenzialmente, etc.)
  - viene sfruttata decidendo di rimpiazzare il blocco utilizzato meno recentemente.

Dato richiesto dalla CPU viene mantenuto in cache insieme a celle di memoria contigue il più a lungo possibile.

Cache: qualche definizione

- Hit: l'elemento richiesto dalla CPU è presente in cache
- Miss: l'elemento richiesto dalla CPU non è presente in cache
- Hit rate: frazione degli accessi a memoria ricompensati da uno hit (cifra di merito per le prestazioni della cache)
- Miss rate: frazione degli accessi a memoria cui risponde un miss (miss rate = 1-hit rate)
- Hit time: tempo di accesso alla cache in caso di successo (include il tempo per determinare se l'accesso si conclude con hit o miss)
- Miss penalty: tempo necessario per sostituire un blocco in cache con un altro blocco dalla memoria di livello inferiore (si usa un valore medio)
- Miss time: = miss penalty + hit time, tempo necessario per ottenere l'elemento richiesto in caso di miss.

Cache: qualche stima quantitativa

Livello	costo di accesso
L1	1 ciclo di clock
L2	7 cicli di clock
RAM	36 cicli di clock

- 100 accessi con 100% cache hit: $\rightarrow$ t=100
- 100 accessi con 5% cache miss in L1: $\rightarrow$ t=130
- 100 accessi con 10% cache miss:in L1 $\rightarrow$ t=160
- 100 accessi con 10% cache miss:in L2 $\rightarrow$ t=450
- 100 accessi con 100% cache miss:in L2 $\rightarrow$ t=3600

Cache miss in tutti i livelli

1. cerco due dati, A e B
2. cerco A nella cache di primo livello (L1) O(1) cicli
3. cerco A nella cache di secondo livello (L2) O(10) cicli
4. copio A dalla RAM alla L2 alla L1 ai registri O(10) cicli
5. cerco B nella cache di primo livello (L1) O(1) cicli
6. cerco B nella cache di secondo livello (L2) O(10) cicli
7. copio B dalla RAM alla L2 alla L1 ai registri O(10) cicli
8. eseguo l'operazione richiesta

O(100) cicli di overhead!!!

Cache hit in tutti i livelli

- cerco i due dati, A e B
- cerco A nella cache di primo livello(L1) O(1) cicli

- cerco B nella cache di primo livello(L1) O(1) cicli

- eseguo l'operazione richiesta

O(1) cicli di overhead

SRAM vs. DRAM

- Dynamic RAM (DRAM) memoria centrale
  - Una cella di memoria è composta da 1 transistor
  - Economica
  - Ha bisogno di essere "ricaricata"
    - I dati non sono accessibili nella fase di ricarica
- Static RAM (SRAM) memoria cache
  - Una cella di memoria è composta da 6-7 transistor
  - Costosa
  - Non ha bisogno di "ricarica"
    - I dati sono sempre accessibili

Misura prestazioni:esempio


```
float sum = 0.0f;
  for (i = 0; i < n; i++)
    sum = sum + x[i]*y[i];
```

- Ad ogni iterazione viene eseguita una somma ed una moltiplicazione floating-point
- Il numero di operazioni è $2 \times n$

Tempo di esecuzione T_{es}

- $T_{es} = N_{flop} * t_{flop}$
- $N_{flop} \rightarrow$ Algoritmo

Tempo di esecuzione T_{es}

- $T_{es} = N_{flop} * t_{flop}$
- $N_{flop} \rightarrow$ Algoritmo
- $t_{flop} \rightarrow$ Hardware

Tempo di esecuzione T_{es}

- $T_{es} = N_{flop} * t_{flop}$
- $N_{flop} \rightarrow$ Algoritmo
- $t_{flop} \rightarrow$ Hardware
- Considera solamente il tempo di esecuzione in memoria
- Trascuro qualcosa?

Tempo di esecuzione T_{es}

- $T_{es} = N_{flop} * t_{flop}$
- $N_{flop} \rightarrow$ Algoritmo
- $t_{flop} \rightarrow$ Hardware
- Considera solamente il tempo di esecuzione in memoria
- Trascuro qualcosa?
- t_{mem} il tempo di accesso in memoria.

Per cui . . .

- $T_{es} = N_{flop} * t_{flop} + N_{mem} * t_{mem}$
- $t_{mem} \rightarrow$ Hardware
- Qual è l'impatto di N_{mem} sulle performance?

Impatto di N_{mem}

- $Perf = \frac{N_{Flop}}{T_{es}}$
- per $N_{mem} = 0 \rightarrow Perf^* = \frac{1}{t_{flop}}$
- per $N_{mem} > 0 \rightarrow Perf = \frac{Perf^*}{1 + \frac{N_{mem} * t_{mem}}{N_{flop} * t_{flop}}}$
- Fattore di decadimento delle performance
- $\frac{N_{mem}}{N_{flop}} * \frac{t_{mem}}{t_{flop}}$
- Come posso avvicinarmi alle prestazioni di picco della macchina?

Impatto di N_{mem}

- $Perf = \frac{N_{Flop}}{T_{es}}$
- per $N_{mem} = 0 \rightarrow Perf^* = \frac{1}{t_{flop}}$
- per $N_{mem} > 0 \rightarrow Perf = \frac{Perf^*}{1 + \frac{N_{mem} * t_{mem}}{N_{flop} * t_{flop}}}$
- Fattore di decadimento delle performance
- $\frac{N_{mem}}{N_{flop}} * \frac{t_{mem}}{t_{flop}}$
- Come posso avvicinarmi alle prestazioni di picco della macchina?
- Riducendo gli accesssi alla memoria.

Località spaziale: ordine di accesso

- Prodotto di matrici in doppia precisione 1024X1024
- MFlops misurati su eurora (Intel(R) Xeon(R) CPU E5-2658 0 @ 2.10GHz)
- compilatore gfortran (4.4.6) con ottimizzazione -O0

Ordine indici	Fortran	C
i,j,k	234	262
i,k,j	186	357
j,k,i	234	195
j,i,k	347	260
k,j,i	340	195
k,i,j	186	357

L'ordine di accesso più efficiente dipende dalla disposizione dei dati in memoria e non dall'astrazione operata dal linguaggio.

Layout degli array in memoria

- Memoria $\rightarrow$ sequenza lineare di locazioni elementari
- Matrice A, elemento a_{ij}: i indice di riga, j indice di colonna
- Le matrici sono rappresentate con array

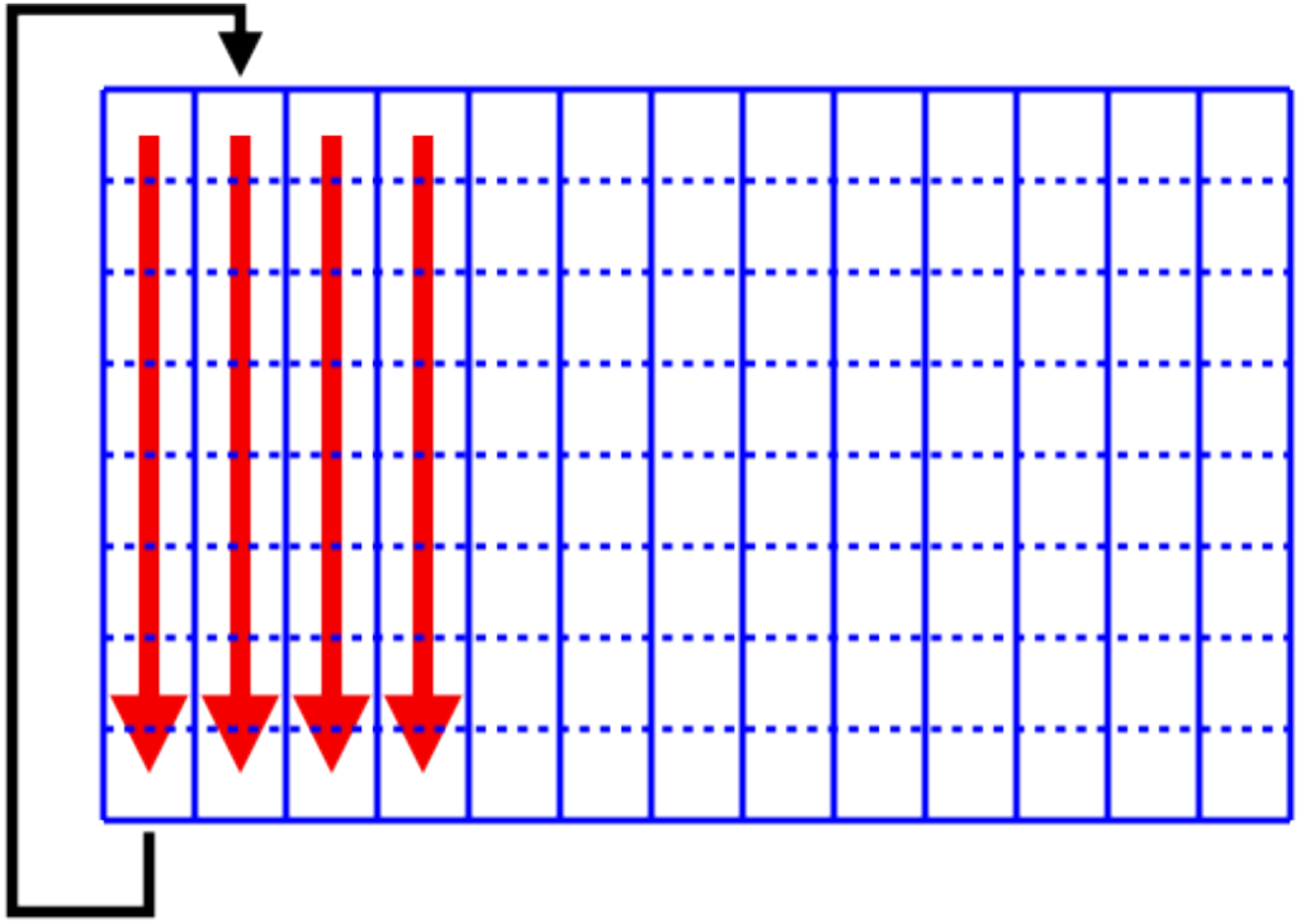
- Come sono memorizzati gli elementi di un array?

- C: in successione seguendo l'ultimo indice, poi il precedente . . .
  a[1][1] a[1][2] a[1][3] a[1][4] . . .
  a[1][n] a[2][1] . . . a[n][n]
- Fortran: in successione seguendo il primo indice, poi il secondo . . .
  a(1,1) a(2,1) a(3,1) a(4,1) . . .
  a(n,1) a(1,2) . . . a(n,n)

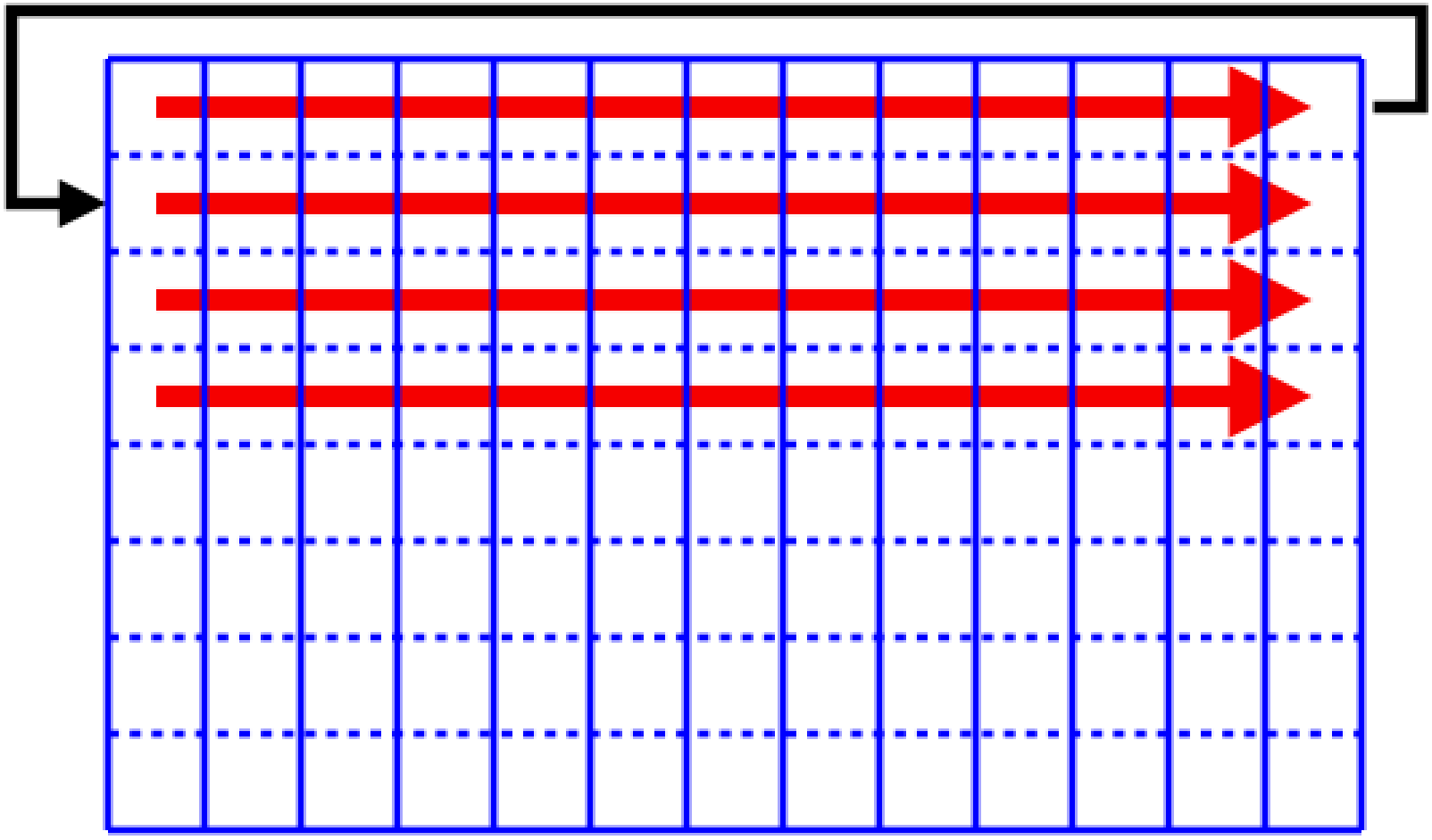
Lo stride

- È la distanza tra due dati successivamente acceduti
  - stride=1 $\rightarrow$ sfrutto la località spaziale
  - stride » 1 $\rightarrow$ non sfrutto la località spaziale
- Regola d'oro
  - Accedere sempre, se possibile, a stride unitario

Ordine di memorizzazione: Fortran



Ordine di memorizzazione: C



Ordine ottimale di accesso

- Calcolare il prodotto matrice-vettore:
  - Fortran: d(i) = a(i) + b(i,j)*c(j)
  - C: d[i] = a[i] + b [i][j]*c[j];
- Fortran
  - do j=1,n
    do i=1,n
    d(i) = a(i) + b(i,j)*c(j)
    end do
    end do
- C
  - for(i=0;i<n,i++1)
    for(j=0;i<n,j++1)
    d[i] = a[i] + b [i][j]*c[j];

Località spaziale: sistema lineare

Soluzione sistema triangolare

- $Lx = b$
- Dove:
  - L è una matrice $n \times n$ triangolare inferiore
  - x è un vettore di n incognite
  - b è un vettore di n termini noti
- Questo sistema è risolvibile tramite:
  - forward substitution
  - partizionamento della matrice

Località spaziale: sistema lineare

Soluzione sistema triangolare

- $Lx = b$
- Dove:
  - L è una matrice $n \times n$ triangolare inferiore
  - x è un vettore di n incognite
  - b è un vettore di n termini noti
- Questo sistema è risolvibile tramite:
  - forward substitution
  - partizionamento della matrice

Qual è più veloce?
Perché?

Forward substitution

Soluzione:

```
. . .
do i = 1, n
     do j = 1, i-1
        b(i) = b(i) - L(i,j) b(j)
    enddo
    b(i) = b(i)/L(i,i)
enddo
. . .
```

Forward substitution

Soluzione:

```
...
do i = 1, n
     do j = 1, i-1
        b(i) = b(i) - L(i,j) b(j)
    enddo
    b(i) = b(i)/L(i,i)
enddo
...
```

```
[~@fen07 TRI]$ ./a.out
```

```
Calcolo sistema L * y = b
time for solution   8.0586
```

Partizionamento della matrice

Soluzione:

```
. . .
do j = 1, n
    b(j) = b(j)/L(j,j)
     do i = j+1,n
        b(i) = b(i) - L(i,j)*b(j)
    enddo
enddo
. . .
```

Partizionamento della matrice

Soluzione:

```
...
do j = 1, n
    b(j) = b(j)/L(j,j)
     do i = j+1,n
        b(i) = b(i) - L(i,j)*b(j)
    enddo
enddo
...
```

```
[~@fen07 TRI]$ ./a.out
```

```
Calcolo sistema L * y = b
time for solution   2.5586
```

Cosa è cambiato

- Forward substitution

```
do i = 1, n
     do j = 1, i-1
        b(i) = b(i) - L(i,j) b(j)
     enddo
     b(i) = b(i)/L(i,i)
enddo
```

- Partizionamento della matrice

```
do j = 1, n
     b(j) = b(j)/L(j,j)
     do i = j+1,n
        b(i) = b(i) - L(i,j)*b(j)
     enddo
enddo
```

Cosa è cambiato

- Forward substitution

```
do i = 1, n
     do j = 1, i-1
        b(i) = b(i) - L(i,j) b(j)
    enddo
    b(i) = b(i)/L(i,i)
enddo
```

- Partizionamento della matrice

```
do j = 1, n
    b(j) = b(j)/L(j,j)
     do i = j+1,n
        b(i) = b(i) - L(i,j)*b(j)
    enddo
enddo
```

- Stesso numero di operazioni, ma tempi molto differenti (più di un fattore 3). Perché?

Vediamo se il concetto è chiaro . . .

Questa matrice:

A	D	G	L
B	E	H	M
C	F	I	N

In C è memorizzata:

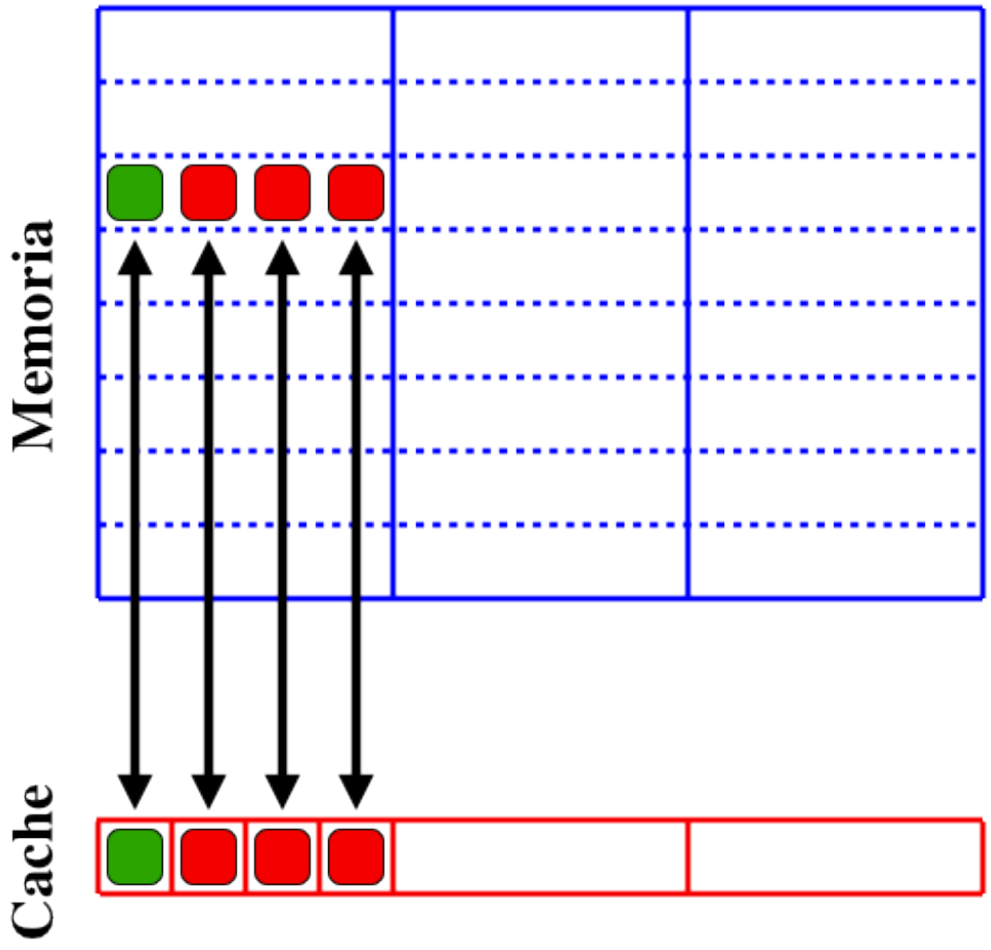
A	D	G	L	B	E	H	M	C	F	I	N

In Fortran è memorizzata:

A	B	C	D	E	F	G	H	I	L	M	N

Località spaziale: righe di cache

- La cache è organizzata in blocchi (righe)
- La memoria è suddivsa in blocchi grandi quanto una riga
- Richiedendo un dato si copia in cache il blocco che lo contiene



Dimensione e riutilizzo dei dati

- Prodotto matrice-matrice in doppia precisione
- Versioni alternative, differenti chiamate della libreria BLAS
- Prestazioni in MFlops su Intel(R) Xeon(R) CPU X5660 2.80GHz

Dimensioni	1 DGEMM	N DGEMV	N^2 DDOT
500	5820	3400	217
1000	8420	5330	227
2000	12150	2960	136
3000	12160	2930	186

Stesso numero di operazioni, l'uso della cache cambia!!!

Riuso della cache


```
...
d=0.0
do I=1,n
j=index(I)
d = d + sqrt(x(j)*x(j) + y(j)*y(j) + z(j)*z(j))
...
```

Riuso della cache


```
...
d=0.0
do I=1,n
j=index(I)
d = d + sqrt(x(j)*x(j) + y(j)*y(j) + z(j)*z(j))
...
```

Cosa succede alla cache ad ogni passo del ciclo?

Riuso della cache


```
...
d=0.0
do I=1,n
j=index(I)
d = d + sqrt(x(j)*x(j) + y(j)*y(j) + z(j)*z(j))
...
```

Cosa succede alla cache ad ogni passo del ciclo?
Posso modificare il codice per ottenere migliori prestazioni?

Riuso della cache


```
...
d=0.0
do I=1,n
j=index(I)
d = d + sqrt(x(j)*x(j) + y(j)*y(j) + z(j)*z(j))
...
```

Cosa succede alla cache ad ogni passo del ciclo?
Posso modificare il codice per ottenere migliori prestazioni?

```
...
d=0.0
do I=1,n
j=index(I)
d = d + sqrt(r(1,j)*r(1,j) + r(2,j)*r(2,j) + r(3,j)*r(3,j))
...
```

I registri

- I registri sono locazioni di memoria interne alla CPU
- poche (tipicamente < 128), ma con latenza nulla
- Tutte le operazioni delle unità di calcolo:
  - prendono i loro operandi dai registri
  - riportano i risultati in registri
- i trasferimenti memoria $\leftrightarrow$ registri sono fasi separate
- il compilatore utilizza i registri:
  - per valori intermedi durante il calcolo delle espressioni
  - espressioni troppo complesse o corpi di loop troppo lunghi forzano lo "spilling" di registri in memoria.
  - per tenere "a portata di CPU" i valori acceduti di frequente
  - ma solo per variabili scalari, non per elementi di array

Elementi di vettori . . .


```
do 3000 z=1,nz
         k3=beta(z)
         do 3000 y=1,ny
              k2=eta(y)
              do 3000 x=1,nx/2
                   hr(x,y,z,1)=hr(x,y,z,1)*norm
                   hi(x,y,z,1)=hi(x,y,z,1)*norm
                   hr(x,y,z,2)=hr(x,y,z,2)*norm
                   hi(x,y,z,2)=hi(x,y,z,2)*norm
                   hr(x,y,z,3)=hr(x,y,z,3)*norm
                   hi(x,y,z,3)=hi(x,y,z,3)*norm
.....................
                   k1=alfa(x,1)
                   k_quad=k1*k1+k2*k2+k3*k3+k_quad_cfr
                   k_quad=1./k_quad
                   sr=k1*hr(x,y,z,1)+k2*hr(x,y,z,2)+k3*hr(x,y,z,3)
                   si=k1*hi(x,y,z,1)+k2*hi(x,y,z,2)+k3*hi(x,y,z,3)
                   hr(x,y,z,1)=hr(x,y,z,1)-sr*k1*k_quad
                   hr(x,y,z,2)=hr(x,y,z,2)-sr*k2*k_quad
                   hr(x,y,z,3)=hr(x,y,z,3)-sr*k3*k_quad
                   hi(x,y,z,1)=hi(x,y,z,1)-si*k1*k_quad
                   hi(x,y,z,2)=hi(x,y,z,2)-si*k2*k_quad
                   hi(x,y,z,3)=hi(x,y,z,3)-si*k3*k_quad
                   k_quad_cfr=0.
3000  continue
```

scalari di appoggio


```
do 3000 z=1,nz
         k3=beta(z)
         do 3000 y=1,ny
              k2=eta(y)
              do 3000 x=1,nx/2
                   br1=hr(x,y,z,1)*norm
                   bi1=hi(x,y,z,1)*norm
                   br2=hr(x,y,z,2)*norm
                   bi2=hi(x,y,z,2)*norm
                   br3=hr(x,y,z,3)*norm
                   bi3=hi(x,y,z,3)*norm
.................
                   k1=alfa(x,1)
                   k_quad=k1*k1+k2*k2+k3*k3+k_quad_cfr
                   k_quad=1./k_quad
                   sr=k1*br1+k2*br2+k3*br3
                   si=k1*bi1+k2*bi2+k3*bi3
                   hr(x,y,z,1)=br1-sr*k1*k_quad
                   hr(x,y,z,2)=br2-sr*k2*k_quad
                   hr(x,y,z,3)=br3-sr*k3*k_quad
                   hi(x,y,z,1)=bi1-si*k1*k_quad
                   hi(x,y,z,2)=bi2-si*k2*k_quad
                   hi(x,y,z,3)=bi3-si*k3*k_quad
                   k_quad_cfr=0.
3000  Continue
```

scalari di appoggio
(durata -25%)


```
do 3000 z=1,nz
         k3=beta(z)
         do 3000 y=1,ny
              k2=eta(y)
              do 3000 x=1,nx/2
                   br1=hr(x,y,z,1)*norm
                   bi1=hi(x,y,z,1)*norm
                   br2=hr(x,y,z,2)*norm
                   bi2=hi(x,y,z,2)*norm
                   br3=hr(x,y,z,3)*norm
                   bi3=hi(x,y,z,3)*norm
.................
                   k1=alfa(x,1)
                   k_quad=k1*k1+k2*k2+k3*k3+k_quad_cfr
                   k_quad=1./k_quad
                   sr=k1*br1+k2*br2+k3*br3
                   si=k1*bi1+k2*bi2+k3*bi3
                   hr(x,y,z,1)=br1-sr*k1*k_quad
                   hr(x,y,z,2)=br2-sr*k2*k_quad
                   hr(x,y,z,3)=br3-sr*k3*k_quad
                   hi(x,y,z,1)=bi1-si*k1*k_quad
                   hi(x,y,z,2)=bi2-si*k2*k_quad
                   hi(x,y,z,3)=bi3-si*k3*k_quad
                   k_quad_cfr=0.
3000  Continue
```

Località spaziale e temporale

- Trasposizione di matrice

```
do j = 1, n
   do i = 1, n
      a(i,j) = b(j,i)
   end do
end do
```

- Qual è l'ordine del loop con stride minimo?
- Per dati all'interno della cache non c'è dipendenza dallo stride
  - se dividessi l'operazione in blocchi abbastanza piccoli da entrare in cache?
  - posso bilanciare tra località spaziale e temporale.

Cache blocking

- I dati elaborati in blocchi di dimensione adeguata alla cache
- All'interno di ogni blocco c'è il riuso delle righe caricate
- Lo può fare il compilatore, se il loop è semplice, ma a livelli di ottimizzazione elevati
- Esempio della tecnica: trasposizione della matrice

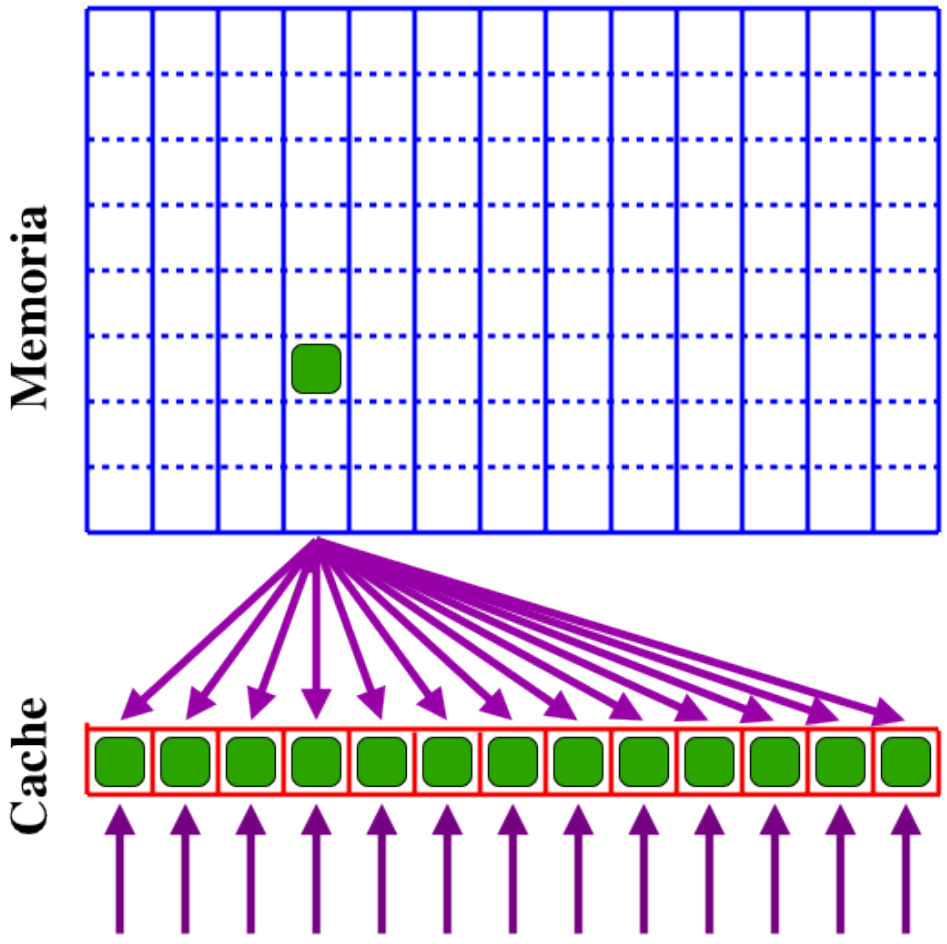
```
do jj = 1, n , step
  do ii = 1, n, step
    do j= jj,jj+step-1,1
      do i=ii,ii+step-1,1
        a(i,j)=b(j,i)
      end do
    end do
  end do
end do
```

Cache: capacity miss e trashing

- La cache può soffrire di capacity miss:
  - si utilizza un insieme ristretto di righe (reduced effective cache size)
  - si riduce la velocità di elaborazione
- La cache può soffrire di trashing:
  - per caricare nuovi dati si getta via una riga prima che sia stata completamente utilizzata
  - è più lento di non avere cache
- Capita quando più flussi di dati/istruzioni insistono sulle stesse righe di cache
- Dipende dal mapping memoria $\leftrightarrow$ cache
  - fully associative cache
  - direct mapped cache
  - N-way set associative cache

Fully associative cache

- Ogni blocco di memoria può essere mappato in una riga qualsiasi di cache

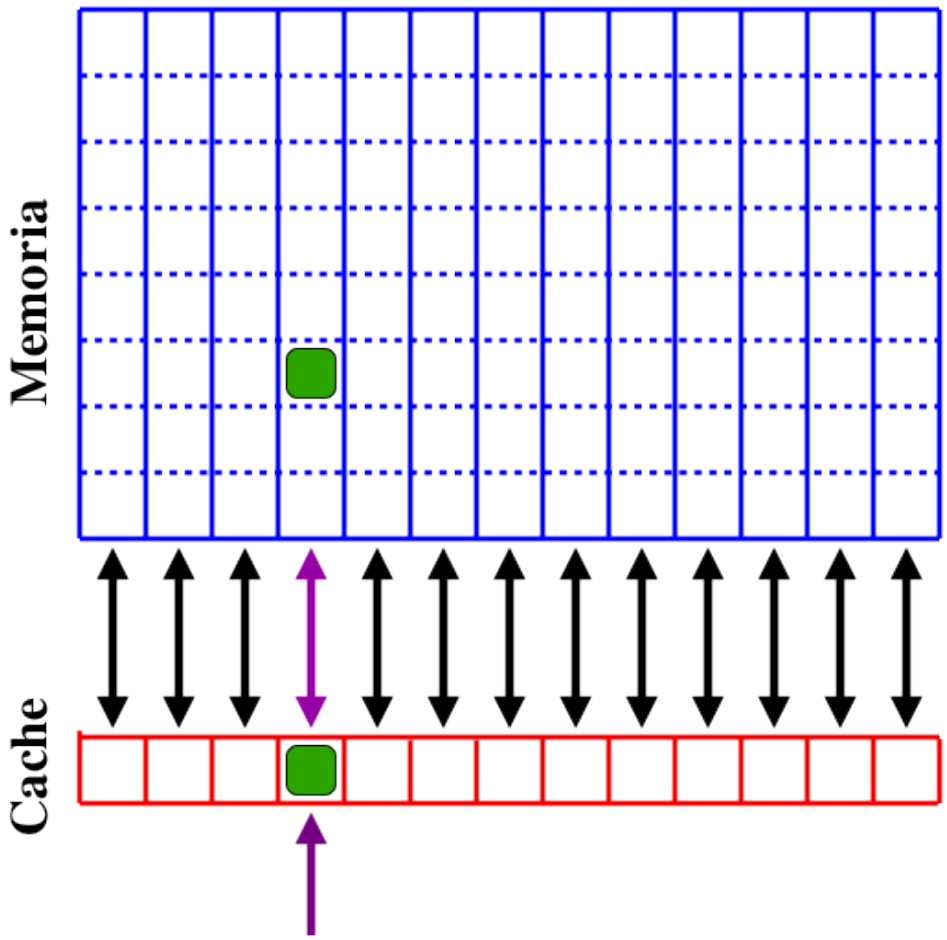


Fully associative cache

- Pro:
  - sfruttamento completo della cache
  - relativamente "insensibile" ai pattern di accesso alla memoria
- Contro:
  - strutture circuitali molto complesse per identificare molto rapidamente un hit
  - algoritmo di sostituzione oneroso Least Recently Used (LRU) o limitatamente efficiente First In First Out (FIFO)
  - costosa e di dimensioni limitate

Direct mapped cache

- Ogni blocco di memoria può essere mappato in una sola riga di cache (congruenza lineare)

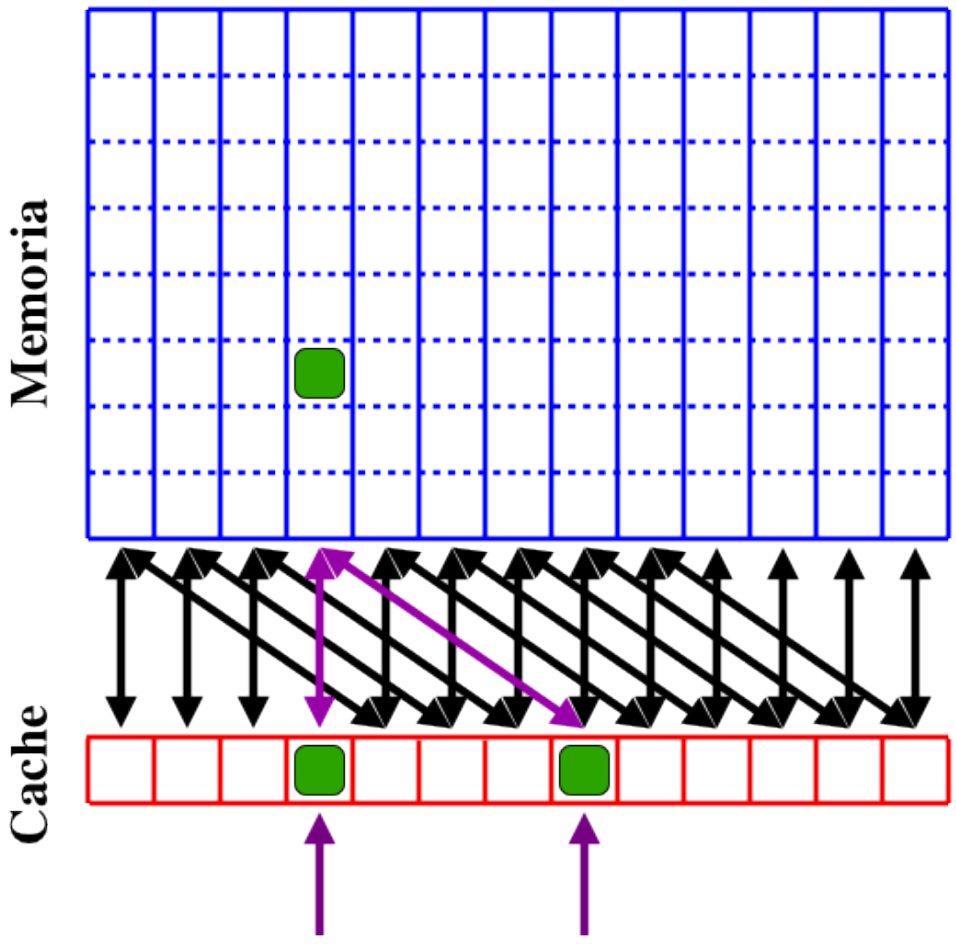


Direct mapped cache

- Pro:
  - identificazione di un hit facilisima (alcuni bit dell'indirizzo identificano la riga da controllare)
  - algoritmo di sostituzione banale
  - cache di dimensione "arbitraria"
- Contro:
  - molto "sensibile" ai pattern di accesso alla memoria
  - soggetta a capacity miss
  - soggetta a cache trashing

N-way set associative cache

- Ogni blocco di memoria può essere mappato in una qualsiasi riga tra N possibili righe di cache



N-way set associative cache

- Pro:
  - è un punto di bilanciamento intermedio
    - N=1 $\rightarrow$ direct mapped
    - N= numero di righe di cache $\rightarrow$ fully associative
  - consente di scegliere il bilanciamento tra complessità circuitale e prestazioni (i.e. costo del sistema e difficoltà di programmazione)
  - consente di realizzare cache di dimensione "soddisfacente"
- Contro:
  - molto "sensibile" ai pattern di accesso alla memoria
  - parzialmente soggetta a capacity miss
  - parzialmente soggetta a cache trashing

Cache: la situazione tipica

- Cache L1: 4÷8 way set associative
- Cache L2÷3: 2÷4 way set associative o direct mapped
- Capacity miss e trashing vanno affrontati
  - le tecniche sono le stesse
  - controllo della disposizione dei dati in memoria
  - controllo delle sequenze di accessi in memoria
- La cache L1 lavora su indirizzi virtuali
  - pieno controllo da parte del programmatore
- le cache L2÷3 lavorano su indirizzi fisici
  - le prestazioni dipendono dalla memoria fisica allocata
  - le prestazioni possono variare da esecuzione a esecuzione
  - si controllano a livello di sistema operativo

Cache Trashing

- Problemi di accesso ai dati in memoria
- Provoca la sostituzione di una riga di cache il cui contenuto verrà richiesto subito dopo
- Si presenta quando due o più flussi di dati insistono su un insieme ristretto di righe di cache
- NON aumenta il numero di load e store
- Aumenta il numero di transazioni sul bus di memoria
- In genere si presenta per flussi il cui stride relativo è una potenza di 2

No trashing: C(i)= A(i)+B(i)

- Iterazione i=1
  1. Cerco A(1) nella cache di primo livello (L1) → cache miss
  2. Recupero A(1) nella memoria RAM
  3. Copio da A(1) a A(8) nella L1
  4. Copio A(1) in un registro
  5. Cerco B(1) nella cache di primo livello (L1) → cache miss
  6. Recupero B(1) nella memoria RAM
  7. Copio da B(1) a B(8) nella L1
  8. Copio B(1) in un registro
  9. Eseguo somma
- Iterazione i=2
  1. Cerco A(2) nella cache di primo livello(L1) → cache hit
  2. Copio A(2) in un registro
  3. Cerco B(2) nella cache di primo livello(L1) → cache hit
  4. Copio B(2) in un registro
  5. Eseguo somma
- Iterazione i=3

Trashing: C(i)= A(i)+B(i)

- Iterazione i=1
  1. Cerco A(1) nella cache di primo livello (L1) $\rightarrow$ cache miss
  2. Recupero A(1) nella memoria RAM
  3. Copio da A(1) a A(8) nella L1
  4. Copio A(1) in un registro
  5. Cerco B(1) nella cache di primo livello (L1) $\rightarrow$ cache miss
  6. Recupero B(1) nella memoria RAM
  7. Scarico la riga di cache che contiene A(1)-A(8)
  8. Copio da B(1) a B(8) nella L1
  9. Copio B(1) in un registro
  10. Eseguo somma

Trashing: C(i)= A(i)+B(i)

- Iterazione i=2
  1. Cerco A(2) nella cache di primo livello(L1) $\rightarrow$ cache miss
  2. Recupero A(2) nella memoria RAM
  3. Scarico la riga di cache che contiene B(1)-B(8)
  4. Copio da A(1) a A(8) nella L1
  5. Copio A(2) in un registro
  6. Cerco B(2) nella cache di primo livello (L1) $\rightarrow$ cache miss
  7. Recupero B(2) nella memoria RAM
  8. Scarico la riga di cache che contiene A(1)-A(8)
  9. Copio da B(1) a B(8) nella L1
  10. Copio B(2) in un registro
  11. Eseguo somma
- Iterazione i=3

Come identificarlo?

- Effetto variabile in funzione della dimensione del data set

```
...
integer,parameter  :: offset=..
integer,parameter  :: N1=6400
integer,parameter  :: N=N1+offset
....
real(8)     :: x(N,N),y(N,N),z(N,N)
...
do j=1,N1
    do i=1,N1
      z(i,j)=x(i,j)+y(i,j)
    end do
end do
...
```

offset	tempo
0	0.361
3	0.250
400	0.252
403	0.253

La soluzione é il padding.

Accessi Disallineati

- Raddoppiano le transazioni sul bus
- Su alcune architetture:
  - causano errori a runtime
  - sono emulati in software
- Sono un problema
  - con tipi dati strutturati(TYPE e struct)
  - con le variabili locali alla routine
  - con i common
- Soluzioni
  - ordinare le variabili per dimensione decrescente
  - opzioni di compilazione (quando disponibili . . .)
  - common diversi/separati
  - inserimento di variabili "dummy" nei common

Accessi disallineati


```
      parameter (nd=1000000)
      real*8 a(1:nd), b(1:nd)
      integer c
      common /data1/ a,c,b
....
      do j = 1, 300
         do i = 1, nd
            somma1 = somma1 + (a(i)-b(i))
         enddo
      enddo
```

Diverse performance per:

```
      common /data1/ a,c,b
      common /data1/ b,c,a
      common /data1/ a,b,c
```

Dipende dall'architettura e dal compilatore che in genere segnala e cerca di sanare con opzione di align common

Come accertare qual è il problema?

- Tutti i processori hanno contatori hardware di eventi
- Introdotti dai progettisti per CPU ad alti clock
  - indispensabili per debuggare i processori
  - utili per misurare le prestazioni
  - fondamentali per capire comportamenti anomali
- Ogni architettura misura eventi diversi
- Sono ovviamente proprietari
  - IBM:HPCT
  - INTEL:Vtune
- Esistono strumenti di misura multipiattaforma
  - Valgrind,Oprofile
  - PAPI
  - Likwid
  - . . .

La cache è una memoria

- Mantiene il suo stato finché un cache-miss non ne causa la modifica
- È uno stato nascosto al programmatore:
  - non influenza la semantica del codice (ossia i risultati)
  - influenza le prestazioni
- La stessa routine chiamate in due contesti diversi del codice può avere prestazioni del tutto diverse a seconda dello stato che “trova” nella cache
- La modularizzazione del codice tende a farci ignorare questo problema
- Può essere necessario inquadrare il problema in un contesto più ampio della singola routine

Valgrind

- Software Open Source utile per il Debugging/Profiling di programmi Linux, di cui non richiede i sorgenti (black-box analysis), ed è composto da diversi tool:
  - Memcheck (detect memory leaks, . . .)
  - Cachegrind (cache profiler)
  - Callgrind (callgraph)
  - Massif (heap profiler)
  - Etc.
- http://valgrind.org

Cachegrind


```
valgrind --tool=cachegrind <nome_eseguibile>
```

- Simula come il vostro programma interagisce con la gerarchia di cache
  - due cache indipendenti di primo livello (L1)
    - per istruzioni (I1)
    - per dati (D1)
  - una cache di ultimo livello, L2 o L3(LL)
- Riporta diverse statistiche
  - I cache reads (Ir numero di istruzioni eseguite), I1 cache read misses(I1mr),LL cache instrunction read misses (ILmr)
  - D cache reads, Dr,D1mr,DLlmr
  - D cache writes, Dw,D1mw,DLlmw
- Riporta (opzionale) il numero di branch e quelli mispredicted

Cachegrind:esempio I


```
==14924== I   refs:      7,562,066,817
==14924== I1  misses:            2,288
==14924== LLi misses:            1,913
==14924== I1  miss rate:          0.00%
==14924== LLi miss rate:          0.00%
==14924==
==14924== D   refs:      2,027,086,734  (1,752,826,448 rd   + 274,260,286 wr)
==14924== D1  misses:       16,946,127  (   16,846,652 rd   +      99,475 wr)
==14924== LLd misses:          101,362  (        2,116 rd   +      99,246 wr)
==14924== D1  miss rate:           0.8% (          0.9%     +         0.0%  )
==14924== LLd miss rate:           0.0% (          0.0%     +         0.0%  )
==14924==
==14924== LL refs:          16,948,415  (   16,848,940 rd   +      99,475 wr)
==14924== LL misses:           103,275  (        4,029 rd   +      99,246 wr)
==14924== LL miss rate:            0.0% (          0.0%     +         0.0%  )
```

Cachegrind:esempio II


```
==15572== I   refs:      7,562,066,871
==15572== I1  misses:            2,288
==15572== LLi misses:            1,913
==15572== I1  miss rate:          0.00%
==15572== LLi miss rate:          0.00%
==15572==
==15572== D   refs:      2,027,086,744  (1,752,826,453 rd   + 274,260,291 wr)
==15572== D1  misses:      151,360,463  (  151,260,988 rd   +      99,475 wr)
==15572== LLd misses:          101,362  (        2,116 rd   +      99,246 wr)
==15572== D1  miss rate:           7.4% (          8.6%     +         0.0%  )
==15572== LLd miss rate:           0.0% (          0.0%     +         0.0%  )
==15572==
==15572== LL refs:         151,362,751  (  151,263,276 rd   +      99,475 wr)
==15572== LL misses:           103,275  (        4,029 rd   +      99,246 wr)
==15572== LL miss rate:            0.0% (          0.0%     +         0.0%  )
```

Cachegrind:cg_annotate

- Cachegrind genera automaticamente un file cachegrind.out.<pid>
- Oltre alle precedenti informazioni vengono generate anche statistiche funzione per funzione

```
cg_annotate cachegrind.out.<pid>
```

Cachegrind:opzioni

- —I1=<size>,<associativity>,<line size>
- —D1=<size>,<associativity>,<line size>
- —LL=<size>,<associativity>,<line size>
- —cache-sim=no|yes [yes]
- —branch-sim=no|yes [no]
- —cachegrind-out-file=<file>

Esercitazione: sommario

- Prodotto matrice matrice:ordine dei loop
- Prodotto matrice matrice:blocking
- Prodotto matrice matrice:blocking e padding
- Misura delle prestazioni delle cache

Ordine dei loop

- Andare su *eser_2* (fortran/mm.f90 o c/mm.c)
- Misurare i tempi per N=??? al variare dell'ordine del loop
- Usare fortran e/o c
- Usare i diversi compilatori senza ottimizzazioni(-O0)

Indici	Fortran	C
i,j,k		
i,k,j		
j,k,i		
j,i,k		
k,i,j		
k,j,i		

src/eser_3/fortran/mm.f90


```
1   ...
2       integer, parameter :: n=1024  ! size of the matrix
3       integer, parameter :: step=4
4       integer, parameter :: npad=0
5   ...
6       real(my_kind) a(1:n+npad,1:n) ! matrix
7       real(my_kind) b(1:n+npad,1:n) ! matrix
8       real(my_kind) c(1:n+npad,1:n) ! matrix (destination)
9
10          do jj = 1, n, step
11            do kk = 1, n, step
12               do ii = 1, n, step
13                  do j = jj, jj+step-1
14                     do k = kk, kk+step-1
15                        do i = ii, ii+step-1
16                           c(i,j) = c(i,j) + a(i,k)*b(k,j)
17                        enddo
18  ...
```

src/eser_3/c/mm.c


```
1  #define nn (1024)
2  #define step (4)
3  #define npad (0)
4
5  double a[nn][nn+npad];        /** matrici**/
6  double b[nn][nn+npad];
7  double c[nn][nn+npad];
8  ...
9   for (ii = 0; ii < nn; ii= ii+step)
10    for (kk = 0; kk < nn; kk = kk+step)
11       for (jj = 0; jj < nn; jj = jj+step)
12          for ( i = ii; i < ii+step; i++ )
13             for ( k = kk; k < kk+step; k++ )
14                for ( j = jj; j < jj+step; j++ )
15                   c[i][j] = c[i][j] + a[i][k]*b[k][j];
16 ...
```

Blocking

- Andate su *eser*_3 (fortran/mm.f90 o c/mm.c)
- Misurare i tempi al variare della dimensione del blocking per matrici con N=???
- Usare Fortran e/o c
- Usare i diversi compilatori con livello di ottimizzazione -O3

Step	Fortran	C
4		
8		
16		
32		
64		
128		
256		

Blocking e Padding

- Andate su *eser*_3 (fortran/mm.f90 o c/mm.c)
- Misurare i tempi al variare della dimensione del blocking per matrici con N=??? e npad=???
- Usare Fortran e/o c
- Usare i diversi compilatori con livello di ottimizzazione -O3

Step	Fortran	C
4		
8		
16		
32		
64		
128		
256		

Misura delle prestazioni delle cache

- valgrind
  - Utilizzare il tool cachegrind per "scoprire" come variano le prestazioni misurate modificando l'ordine dei loop