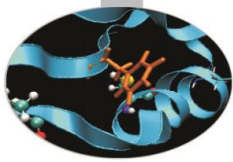
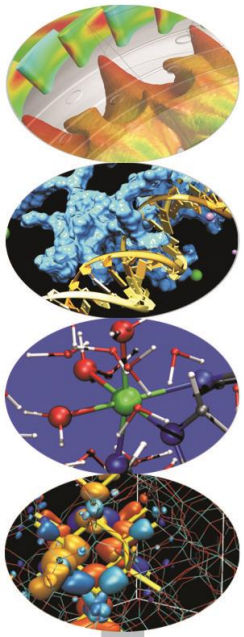
CINECA SCAI
SuperComputing Applications and Innovation



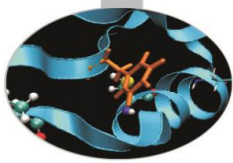
Tecniche di ottimizzazione del codice



Paolo Ramieri, *CINECA*

Febbraio 2014

CINECA



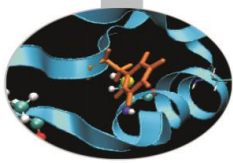
Ottimizzare un codice

Perché ottimizzare?

- Per sfruttare al massimo le risorse hardware a nostra disposizione
- Per ottenere risultati con tempi di attesa inferiori

Come ottimizzare?

- Utilizzando le potenzialità dei compilatori
- Scrivendo il codice con tecniche adeguate ai compilatori e all'hardware che si utilizzerà



Il prodotto matrice-matrice

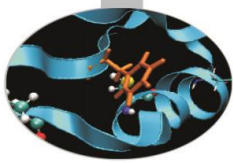
Prodotto matrice-matrice:

```
Fortran: c(i,j) = c(i,j)+a(i,k)*b(k,j)
```

```
C: c[i][j]= c[i][j]+a[i][k]*b[k][j];
```

Esiste un ordine ideale per l'accesso ai dati?

Se sì, qual é?



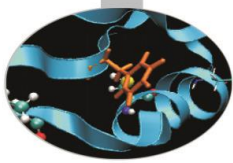
Per il Fortran…

Ordine j,k,i:

```
do j=1,n
      do k=1,n
            do i=1,n
                  c(i,j)=c(i,j)+a(i,k)*b(k,j)
            enddo
      enddo
enddo
```

Ordine i,k,j:

```
do i=1,n
   do k=1,n
     do j=1,n
            c(i,j)=c(i,j)+a(i,k)*b(k,j)
     enddo
   enddo
enddo
```



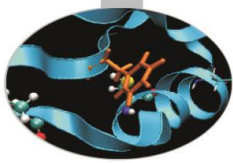
… e per il C

Ordine j,k,i:

```
for(j=0;j<nn;j++){
        for(k=0;k<nn;k++){
                for(i=0;i<nn;i++){
                        c[i][j]=c[i][j]+a[i][k]*b[k][j];
                }
        }
}
```

Ordine i,k,j:

```
for(i=0;i<nn;i++){
        for(k=0;k<nn;k++){
                for(j=0;j<nn;j++){
                        c[i][j]=c[i][j]+a[i][k]*b[k][j];
                }
        }
}
```



Cosa è il calcolo HPC?

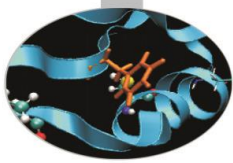
Nel calcolo HPC bisogna:

eseguire il calcolo nel minore tempo possibile

sfruttare al massimo le risorse disponibili

rispondere (almeno parzialmente) a vincoli esterni di:

- memoria
- tempo
- qualità



Evoluzione delle CPU

CPU vettoriali

CPU pipelined

CPU superscalari

CPU out of order

CPU superpipelined

CPU con unità SIMD (SSE, SSE1,…)

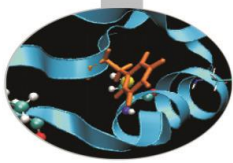
CPU SMT (Simultaneous Multi Threading)

CPU multicore

GPU massively parallel

…

CPU manycore (core molto semplificati)



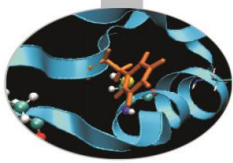
Legge di Moore (1965)

- Il numero di transistor sul chip raddoppia ogni 18 mesi

- Raddoppio delle unità funzionali
  unità di calcolo: 2× capacità di calcolo
  logica di controllo: ~4× complessità
  2× potenza assorbita

- Raddoppio del clock
  unità di calcolo: 2×velocità di picco
  8×potenza assorbita

- La reale efficienza dei codici:
  dipende dall'efficienza dell'applicazione
  è limitata dalle dipendenze tra operazioni
  l'ottimizzazione del codice è importante



Ancora Prodotto Matrice-Matrice

Versione base:

1 istruzione, 3 loop, 7 righe di codice

prestazioni: 80 MFlops

Versione ottimizzata a mano:

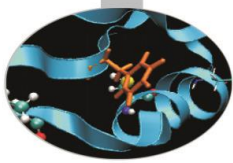
16 istruzioni, 32 scalari di appoggio

6 loop, unrolling 4/4/4, 64 righe di codice

blocking di 128 elementi

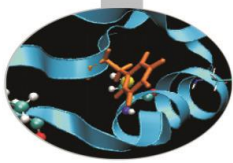
padding di 16

prestazioni: 391 MFlops



Sfruttando il compilatore…

OPZIONE	SECONDI	Mflops
-O0	24	89
-O1	6.35	338
-O2	4.87	487
-O3	2.14	1003
-O4	1.93	1111



Input/Output

MAI mescolare calcolo intensivo ed I/O!

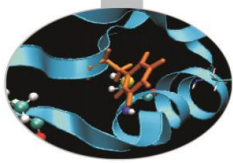
leggere/scrivere i dati in un blocco e non pochi per volta

È sempre mediato dal sistema operativo:

- causa chiamate di sistema
- comporta lo svuotamento delle pipeline di calcolo
- distrugge la coerenza dei dati in cache
- può alterare la priorità di scheduling dell'applicazione
- è lento

Attenzione ad I/O nascosti: swapping

- avviene quando la RAM è insufficiente
- usa il disco come surrogato



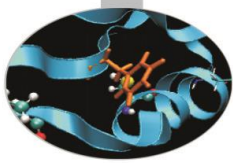
Performance

Avere **risultati corretti** nel minor tempo possibile

Se i risultati non sono corretti il lavoro fatto è nullo!!!!

Tutti i fattori contano:

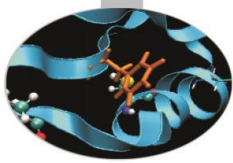
- algoritmo
- implementazione (codice)
- uso del compilatore
- I/O ed interazioni con l'OS



Come migliorare le performance

Possibili approcci:

- Ridurre numero di operazioni
- Velocizzare scambio dati CPU/memoria
- Usare operazioni meno onerose
- Massimizzare attività della CPU
- Scrivere il codice in modo da agevolare il compilatore
- Ottimizzazioni a mano
- Usare librerie ottimizzate



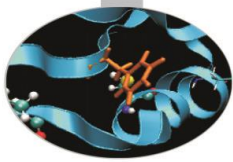
Un comando utile: `time`

Comando **unix** che fornisce indicazioni sull'esecuzione del processo, in particolare sul tempo impiegato:

> time ./a.out

> 19.450u 0.010s 0:19.45 100.0% 20+151k 0+0io 0pf+0w

19.450u	**User time**
0.010s	**System time**
0:19.45	**Elapsed time**
100.0%	**CPU = (User+System)/Elapsed**
20+151k	**memory segment**
0+0io	**numero operazioni I/O**
0pf+0w	**numero di page fault**



Problemi identificabili con `time`

- Problemi di page fault

.....

2998.43u 421.01s 55:02.58 97.3% 278+447k 0+0io 3170pf+0w

- Problemi di I/0

....

57.37u 220.91s 5:06.98 90.6% 0+126k 9225+4217514io 2pf+0w

- Problemi di system time

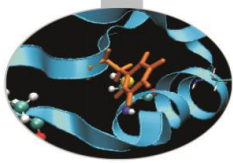
.....

1.74u 1.39s 0:03.15 99.6% 0+540k 0+0io 0pf+0w

- Problemi di multitasking

.....

12.81u 0.33s 0:20.94 62.7% 0+687k 1+9io 0pf+0w



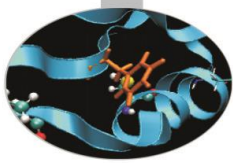
Come misurare le performance

- Occorre instrumentare il codice con chiamate a funzioni che restituiscono informazioni sul tempo impiegato

- Permette di misurare:
  - Sezioni di codice
  - Evoluzione temporale

- Attenzione però a:
  - intrusività
  - affidabilità
  - overhead
  - multithreading

- cosa si misura:
  - CPU time
  - Elapsed time
  - I/O time



Come misurare le performance

fortran77:

`etime()`,`dtime()` **(obsoleta)**

fortran90:

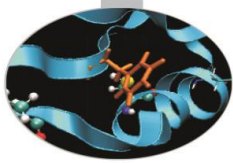
`cputime()`

`system_clock()`

`date_and_time()`

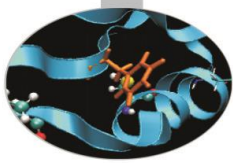
C/C++

`clock()`



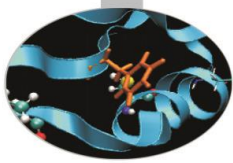
Esempio – system_clock

```
integer :: count_rate, count_max
integer :: count1, count0
…
call system_clock(count0, count_rate, count_max)
do j = 1,n
   do k = 1,n
        do i = 1,n
                c(i,j) = c(i,j) + a(i,k)*b(k,j)
        enddo
   enddo
enddo
call system_clock(count1, count_rate, count_max)
write(6,*) real(count1-count0)/(count_rate)
```



Esempio – `date_and_time`

```
real(my_kind), intent(out) :: t
integer :: time_array(8)
…
call date_and_time(values=time_array)
t1 = 3600.*time_array(5)+60.*time_array(6)+time_array(7)+time_array(8)/1000.
do j = 1,n
    do k = 1,n
          do i = 1,n
                    c(i,j) = c(i,j) + a(i,k)*b(k,j)
          enddo
    enddo
enddo
call date_and_time(values=time_array)
t2 = 3600.*time_array(5)+60.*time_array(6)+time_array(7)+time_array(8)/1000.
write(6,*) t2-t1
```



Ordine dei loop

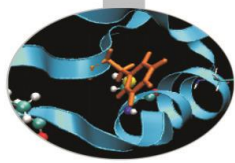
Prodotto matrice matrice 1024*1024

fortran: **gfortran 4.3.0**

c: **gcc 4.0.1**

L'importanza della cache

Ordine	Fortran	C
i,k, j	32.2”	2.29”
j, k, i	2.27”	32.2”



Gerarchia di memoria

Maggior velocità di accesso ai dati

Minori dimensioni

Registri

Cache 1

Cache 2

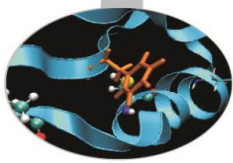
(Cache 3)

Memoria RAM

Disco

Minor velocità di accesso ai dati

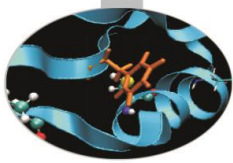
Maggiori dimensioni



Perché questa gerarchia

– Non servono tutti i dati disponibili subito
  - Si lavora quasi sempre su un piccolo sotto-insieme: mettere i dati che servono nella memoria rapida
  - mettere i dati che non servono (per ora) nei livelli più lenti

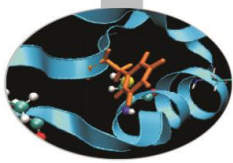
– Soluzione: “caching”
  - sono necessari livelli intermedi di “memoria”



Il flusso dei dati in lettura

Cerco un dato

- Lo si cerca in cache di primo livello (L1)≈1-5 cicli
- Lo si cerca in cache di secondo livello (L2) ≈20 cicli
- Lo si recupera dalla memoria RAM
- Lo si copia dalla RAM alla L2 > 100 cicli
- Lo si copia dalla L2 alla L1
- Lo si copia nei registri



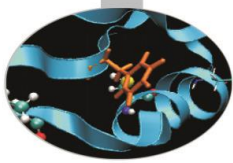
Il flusso dei dati in lettura

Un programma tende a riutilizzare dati ed istruzioni usati in precedenza:

- stesse operazioni su differenti set di dati
- differenti operazioni sullo stesso set di dati

Cerco un dato

- Lo si cerca in cache di primo livello (L1)≈1-5 cicli
- Lo si copia nei registri



Allocazione dei dati

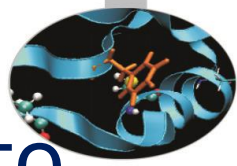
Come viene allocata una matrice?

Fortran: data una matrice questa verrà allocata in memoria ordinata rispetto l'indice più interno:

```
A(1,1),A(2,1),A(3,1),A(4,1)...
A(n,1),A(1,2), ... A(n,n)
```

C: data una matrice questa verrà allocata in memoria ordinata rispetto l'indice più esterno:

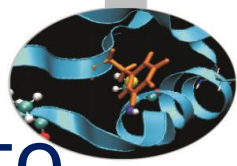
```
A[1][1],A[1][2],A[1][3],A[1][4]...
A[1][n],A[2][1], ... A[n][n]
```



Prodotto matrice-matrice ottimizzato

Fortran (indice più interno)

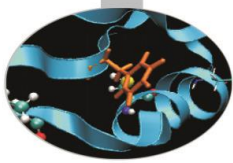
```
do j=1,n
  do k=1,n
    do i=1,n
      c(i,j)=c(i,j)+a(i,k)*b(k,j)
    enddo
  enddo
enddo
```



Prodotto matrice-matrice ottimizzato

C (indice più esterno)

```
for(i=0;i<nn;i++){
  for(k=0;k<nn;k++){
    for(j=0;j<nn;j++){
      c[i][j]=c[i][j]+a[i][k]*b[k][j];
    }
  }
}
```



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

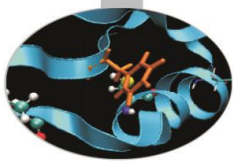
Iterazione i=1

1. CercoA(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. CercoB(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

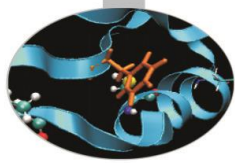
1. CercoA(2) nella cache di primo livello (L1) -> cache hit
2. Copio A(2) in un registro
3. CercoB(2) nella cache di primo livello (L1) -> cache hit
4. Copio B(2) in un registro
5. Eseguo somma

Iterazione i=3



Capacity miss & 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 thrashing:
  - per caricare nuovi dati si getta via una riga prima che sia stata completamente utilizzata
  - è più lento che non avere cache

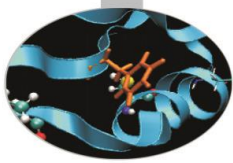


Trashing: C(i) = A(i) + B(i)

Iterazione **i=1**

Cerco**A(1)**nella cache di primo livello (L1) -> **cache miss**

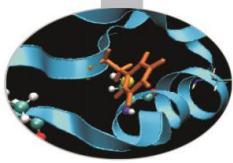
1. Recupero **A(1)** nella memoria RAM
2. Copio da **A(1)** a **A(8)**nella L1
3. Copio **A(1)** in un registro
4. Cerco**B(1)** nella cache di primo livello (L1) -> **cache miss**
5. Recupero **B(1)** nella memoria RAM
6. Scarico la riga di cache che contiene **A(1)-A(8)**
7. Copio da **B(1)** a **B(8)**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



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

Iterazione **i=2**

1. Cerco**A(2)**nella cache di primo livello (L1) -> **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) -> **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



Cache blocking

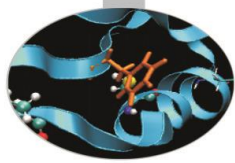
I dati elaborati in blocchi di dimensione adeguata alla cache

All'interno di ogni blocco c'è riuso delle righe caricate

Lo può fare il compilatore, se il loop è semplice, ma a livelli di ottimizzazione elevati

Esempio della tecnica: trasposizione di 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)
                enddo
        enddo
   enddo
enddo
```



Accessi disallineati

Raddoppiano le transazioni sul bus

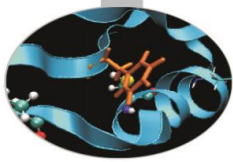
- su alcune architetture: causano errore a runtime

Sono un problema:con tipi dati strutturati (**TYPE** e **struct**)

- con le variabili locali alle 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**

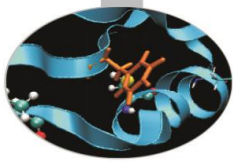


Accessi allineati e disallineati

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

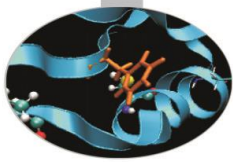
- /data1/ a,c,b t = 2.74''
- /data1/ c,a,b t = 3.75''
- /data1/ a,b,c t = 1.41''

Elementi di vettori...

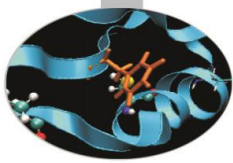


```
do 3000 z=1,nz
k3=beta(z)
do 3000 y=1,ny
k2=eta(y)
      do 3000 x=1,nx/2
      hr(x,y,z,1)=hr(x,y,z,1)*norm
      hi(x,y,z,1)=hi(x,y,z,1)*norm
      hr(x,y,z,2)=hr(x,y,z,2)*norm
      hi(x,y,z,2)=hi(x,y,z,2)*norm
      hr(x,y,z,3)=hr(x,y,z,3)*norm
      hi(x,y,z,3)=hi(x,y,z,3)*norm
...
      k1=alfa(x,1)
      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
```

… e 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
```



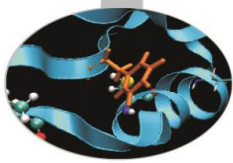
Pipeline

Pipeline = tubazione, catena di montaggio

Un'operazione è divisa in più passi indipendenti (stage) e differenti passi di differenti operazioni vengono eseguiti contemporaneamente

Parallelismo sulle fasi diverse delle istruzioni

I processori sfruttano intensivamente il pipelining per aumentare la capacità di elaborazione

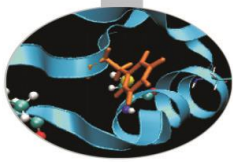


Fetching, decoding, executing

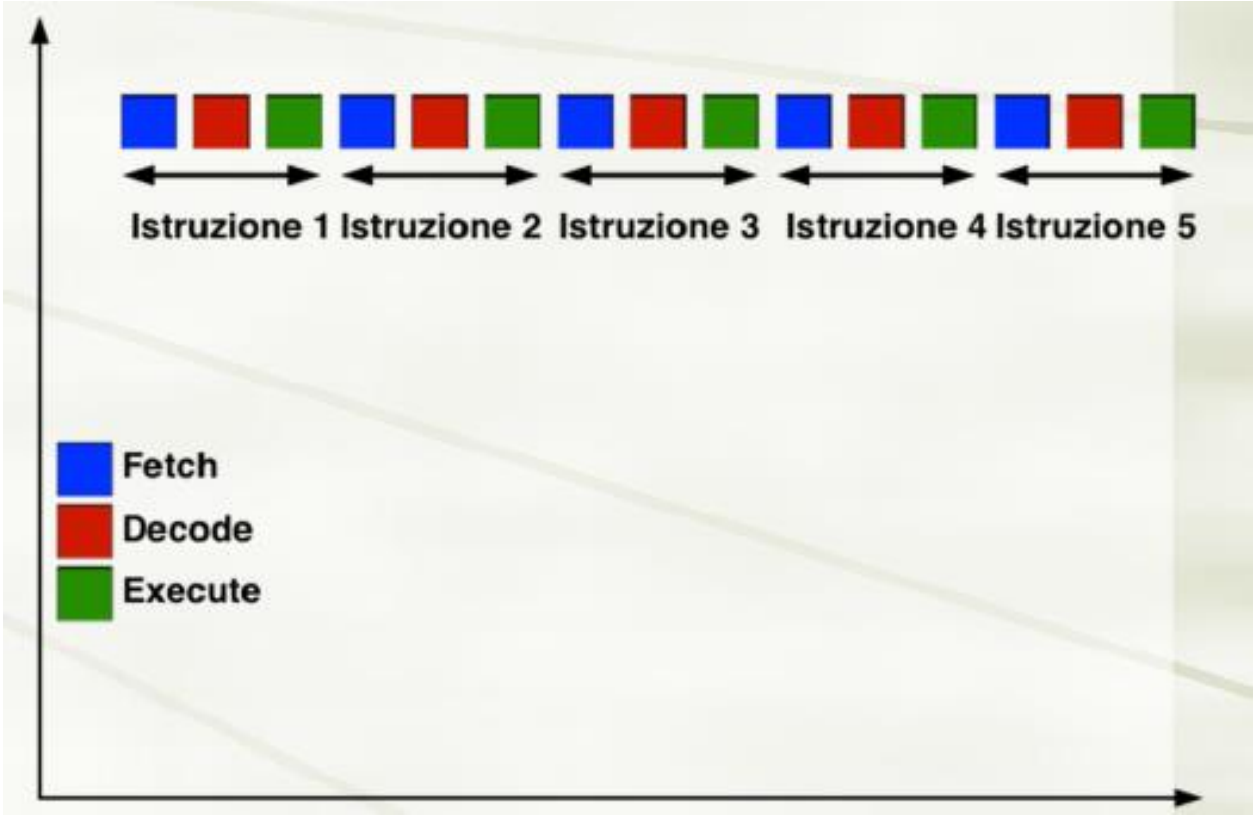
- Il ***fetch*** (prendere, prelevare) è la prima delle tre fasi fondamentali dell'elaborazione sequenziale di un programma. Si tratta dell'astrazione procedurale di tutte quelle operazioni che portano al caricamento dell'istruzione da parte del processore.

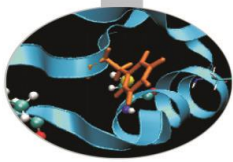
- Il ***decoding*** (o decodifica) è una fase dell'esecuzione di un'istruzione, in cui al codice macchina vengono associate una o più operazioni mappate direttamente nelle unità funzionali del processore stesso. Poiché questa operazione è direttamente riconducibile alla fase di caricamento dell'istruzione nei registri d'esecuzione, molto spesso viene considerata una sotto-fase del fetch vero e proprio.

- L'***esecuzione*** è il processo tramite il quale il processore esegue le istruzioni di un codice.

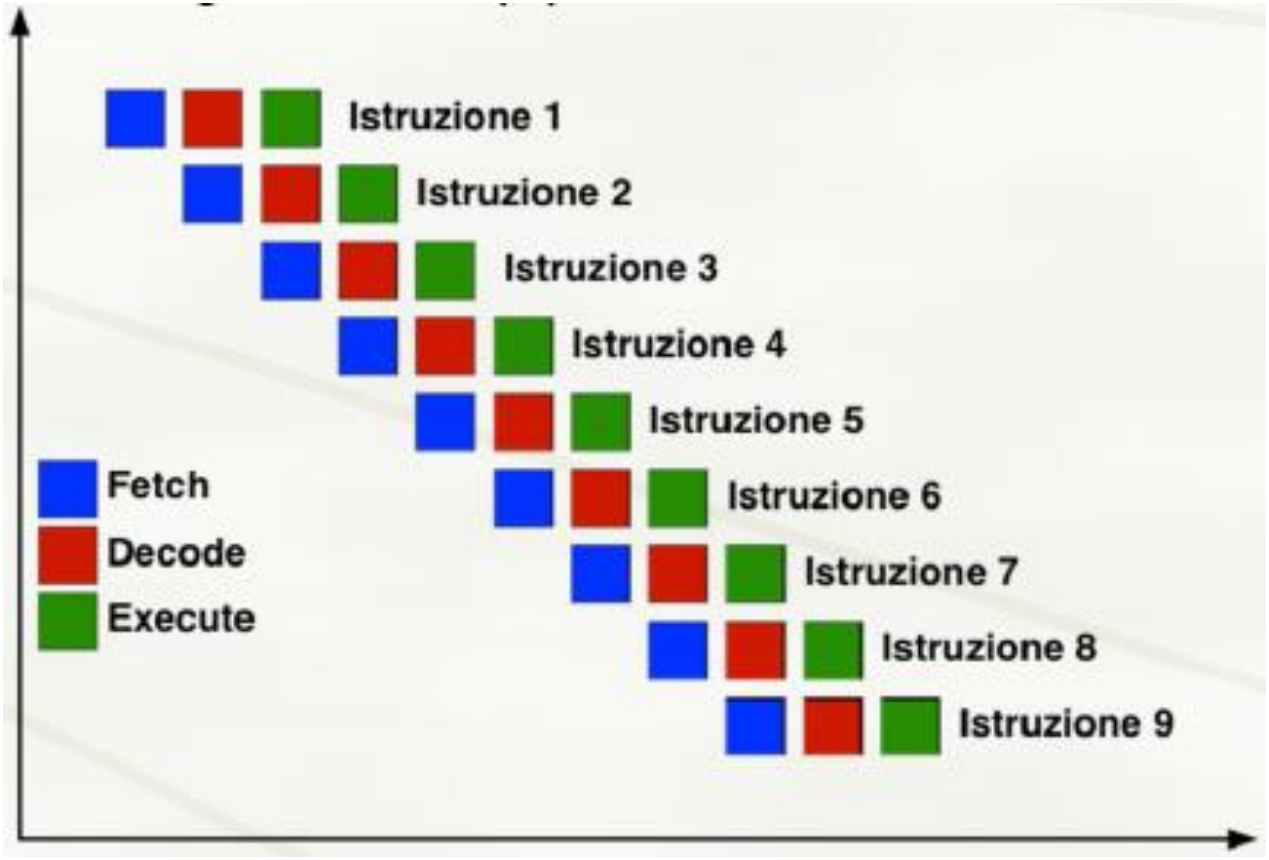


Unità di calcolo non-pipelined

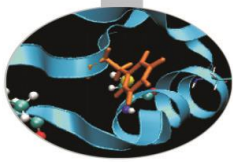




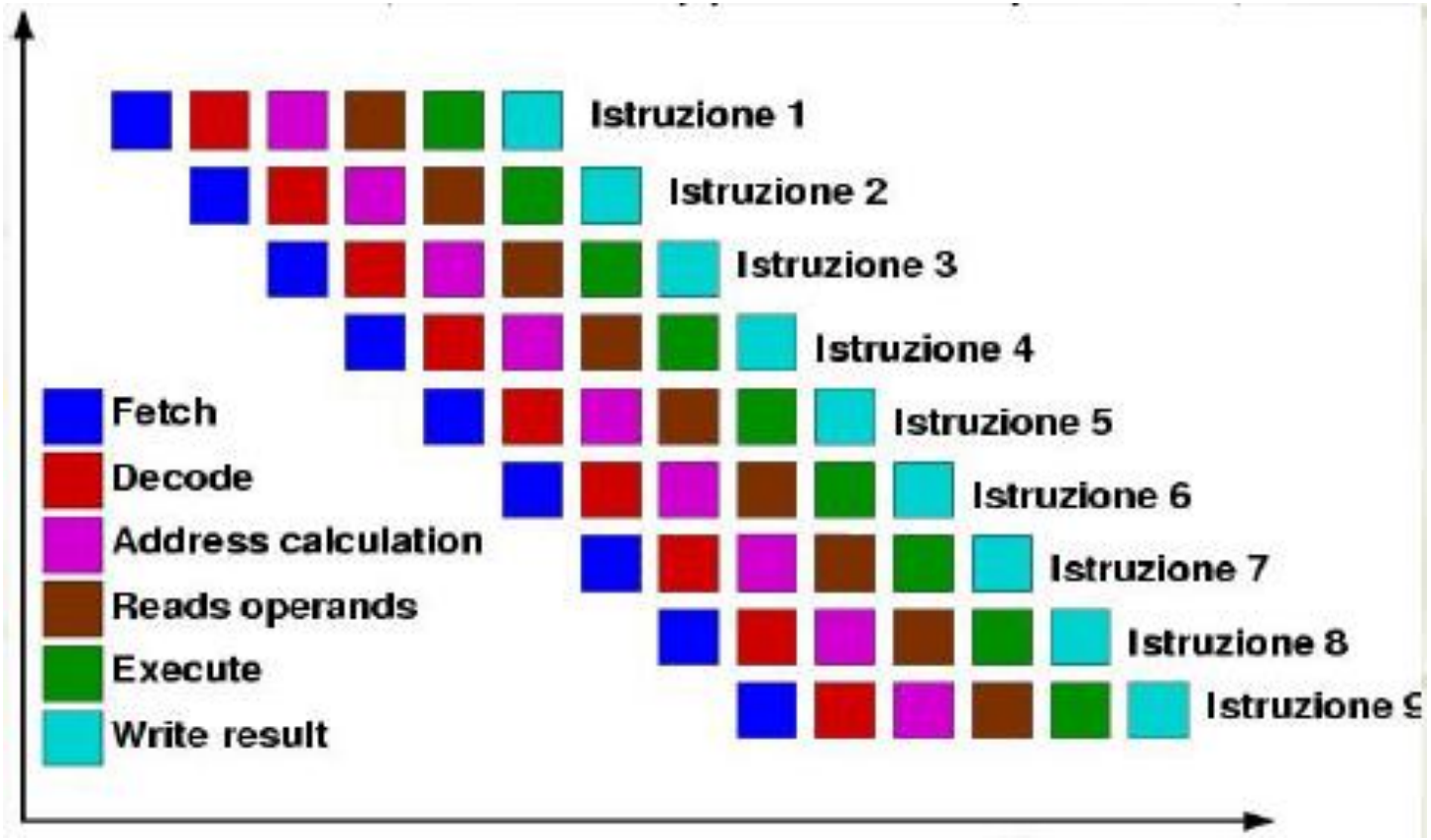
Unità di calcolo pipelined

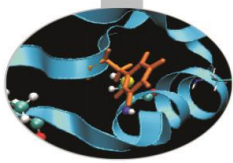


tempo



Unità di calcolo superpipelined





Esecuzione “Out of Order”

Riordina dinamicamente le istruzioni:

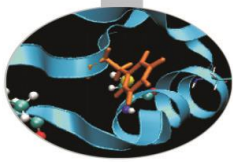
- anticipa istruzioni i cui operandi sono già disponibili
- postpone istruzioni i cui operandi non sono ancora disponibili
- riordina letture e scritture in memoria

Si appoggia intensivamente su:

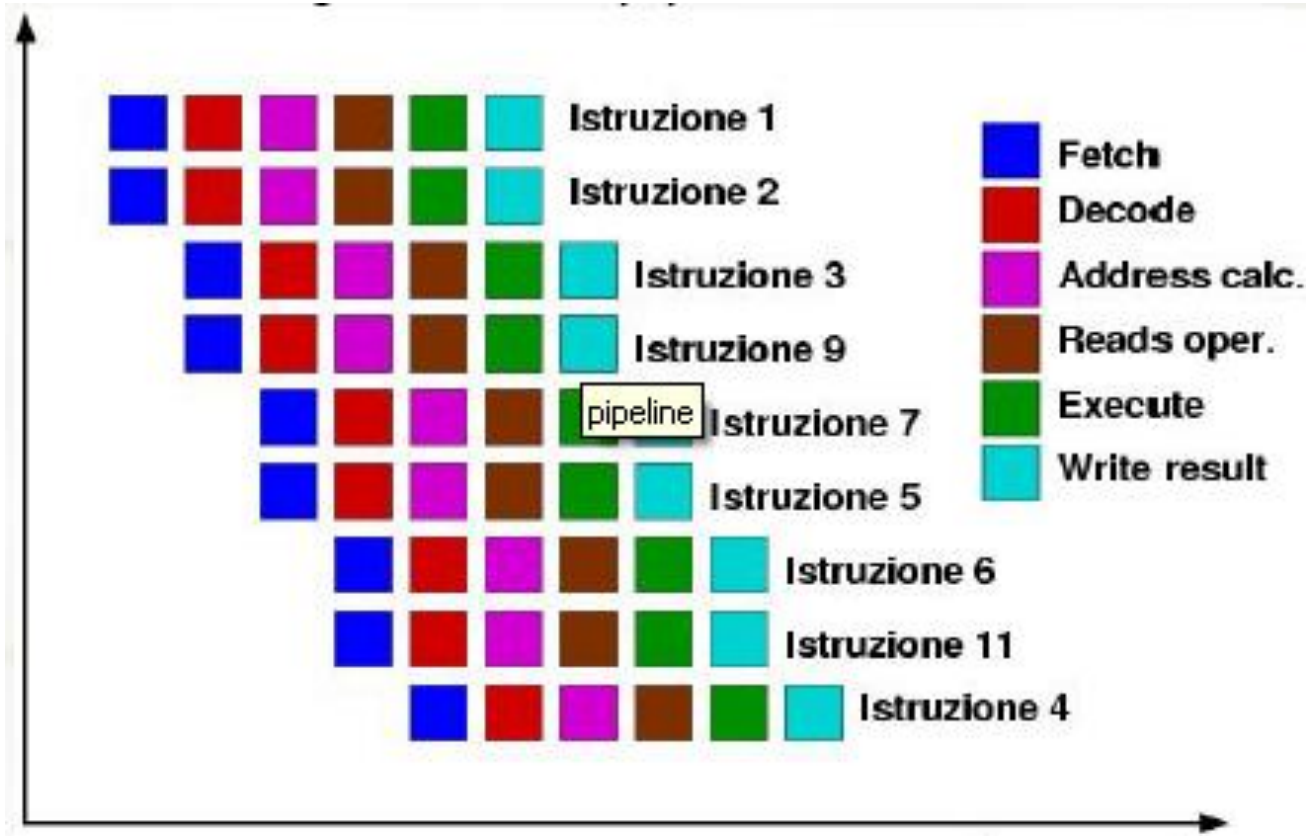
- branch prediction
- combinazione di read e write multiple in memoria

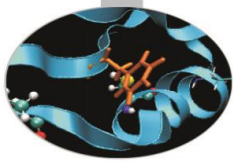
È essenziale per ottenere prestazioni sulle CPU di oggi

Non è sufficiente da sola, il codice deve rendere esplicite le possibilità di riordinamento



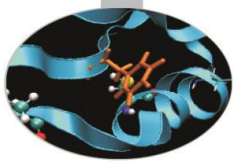
Unità di calcolo "Out of Order"





Il necessario tempo di calcolo

Operazione	Cicli Clock
Somma	12
Prodotto	12
Divisione	20
Sqrt	24
Sin	52
Tan	100
Log	60
Exp(2.7)	130



Strenght reduction

Usare operazioni semplici rispetto a operazioni più complesse:

- `x^3 = x*x*x`
- `x^0.5 = sqrt{x}`
- `x^1.5 = sqrt{x^3}`

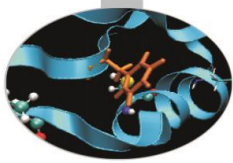
In genere queste modifiche *dovrebbe* farle il compilatore.

Attenzione: in precisione finita si ha, ad esempio, che

`x^1.5` è diverso da `sqrt{x^3}`

Per cui prestare molta cautela a:

- accumulazioni di errori;
- schemi numerici instabili;



Strength reduction: esempio

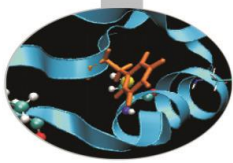
Normalizzazione matrice:

```
do j = 1,n
    do i = 1,n
          c(i,j) = a(i,j)*b(i,j)/(norm)
    enddo
enddo
```

N*N prodotti e N*N divisioni

```
rev_norm = 1.0/(norm)
do j = 1,n
    do i = 1,n
          c(i,j) = a(i,j)*b(i,j)*rev_norm
    enddo
enddo
```

2*N*N prodotti ed 1 divisione



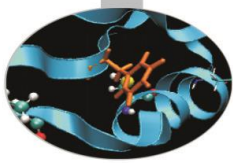
Loop unrolling

Prodotto matrice-matrice con unrolling del loop esterno:

```
do j = 1, n, 2
    do k = 1, n
          do i = 1, n
             c(i,j+0) = a(i,k)*b(k,j+0)+c(i,j+0)
             c(i,j+1) = a(i,k)*b(k,j+1)+c(i,j+1)
          enddo
    enddo
enddo
```

"Srotolando" le iterazioni di un loop:

- si formano più flussi (**stream**) indipendenti di dati
- due coppie di istruzioni indipendenti per iterazione
- un valore è riutilizzato
- l'indirizzo di **b(k,j+1)** si calcola banalmente da quello di **b(k,j+0)**
- La stessa cosa per **c(i,j+1)**
- il "peso" di indici e salti dei loop è dimezzato



Common Subexpression Elimination (CSE)

- Per i calcoli intermedi si riusano spesso alcune espressioni
- Può essere vantaggioso “riciclare” quantità già calcolate:

A = B + C + D

E = B + F + C

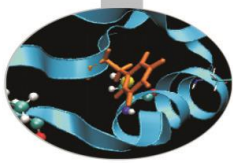
Richiede: 4 load, 2 store, 4 somme

A = (B + C) + D

E = (B + C) + F

Richiede: 4 load, 2 store, 3 somme

- Attenzione: può non essere corretto dal punto di vista numerico!!



CSE e chiamate a funzione

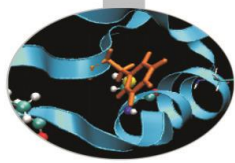
Alterando l'ordine delle chiamate il compilatore non sa se si altera il risultato:

```
x = r*sin(a)*cos(b)
y = r*sin(a)*sin(b)
z = r*cos(a)
```

- 5 chiamate a funzioni, 5 prodotti

```
temp = r*sin(a)
x = temp*cos(b)
y = temp*sin(b)
z = r*cos(a)
```

- 4 chiamate a funzioni, 4 prodotti, 1 variabile temporanea



Loop fusion

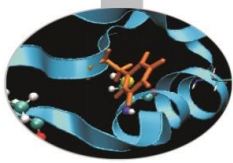
Dipendenze tra iterazioni successive impediscono il loop fusion:

```
do i = 2, n
   a(i) = a(i-1) + 1.0
enddo
do i = 2, n
   b(i) = b(i-1)*2.0
enddo
```

Un unico loop ha più istruzioni per iterazione, e indipendenti:

```
do i = 2, n
   a(i) = a(i-1) + 1.0
   b(i) = b(i-1)*2.0
enddo
```

L'allocazione dinamica inibisce il loop fusion.



Altre operazioni

Loop splitting

- Il “corpo” del loop può essere troppo grande
  - molte istruzioni e variabili di appoggio temporaneo
  - poche istruzioni ma espressioni lunghe e complicate
  - l’unrolling rischi di peggiorare le prestazioni
- Dividendo il loop in due o più loop separati si può guadagnare in prestazione

Inlining di funzioni

- Le chiamate a funzione “costano” perché la CPU deve saltare ad eseguire un’altra porzione del codice, e deve passare i parametri

Index reordering