1

Prestazioni (performance) del software

La Top500 list classifica i supercalcolatori sulla base delle prestazioni ottenute con il **LINPACK benchmark**, un software per la risoluzione di sistemi di equazioni lineari

Le prestazioni sono misurate contando il **numero di operazioni f.p. eseguite per unita' di tempo,** cioe'

$$Perf = \frac{N_{flop}}{T_{tot}}$$

Numero complessivo di operazioni

Tempo complessivo di esecuzione

2

unita' di misura

prestazioni

- 1 Gflop/sec = 10^9 op.f.p./sec 1 Gigaflop/sec (1985)
- 1 Tflop/sec = 10^{12} op.f.p./sec 1 Teraflop/sec (1997)
- 1 Pflop/sec = 10^{15} op.f.p./sec 1 Petaflop/sec (2008)
- 1 Eflop/sec = 10^{18} op.f.p./sec 1 Exaflop/sec (2022)

Dim. memoria

- 1 GB = 10^9 Bytes 1 Gigabytes
- 1 TB = 10^{12} Bytes 1 Terabytes
- 1 PB = 10^{15} Bytes 1 Petabytes
- 1 EB = 10^{18} Bytes 1 Exabytes

3

complessita di tempo / prestazioni

Tipicamente la bonta' di un algoritmo si misura mediante la

- Complessita' di tempo (numero di operazioni): N_{flop}
- Complessita' di spazio (numero di locazioni id memoria)

T_{tot} dipende anche da altri fattori

$$T_{tot} = N_{flop}\ t_{flop}$$

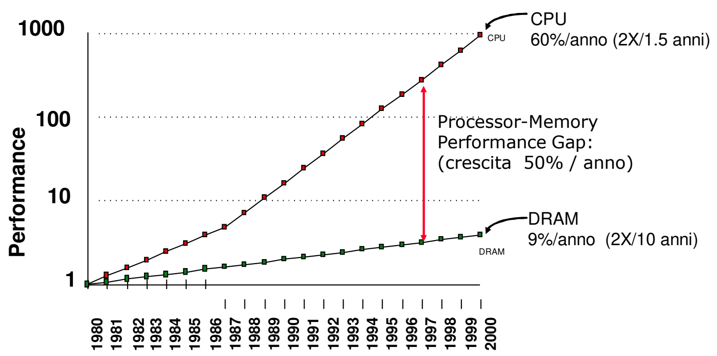
Algoritmo Hardware

Tempo per eseguire solo le operazioni f.p.

Cosi' pero' si trascura il tempo di accesso alla memoria T_{mem}

4

E' ragionevole trascurare T_{mem} ?



Nel valutare il tempo di esecuzione non e' realistico trascurare

il tempo di accesso alla memoria T_{mem}

5

Quindi…

$$T_{tot} = \underbrace{N_{mem}}_{\text{Algoritmo}}\ \underbrace{t_{mem}}_{\text{Hardware}} + \underbrace{N_{flop}}_{\text{Algoritmo}}\ \underbrace{t_{flop}}_{\text{Hardware}}$$

T_{mem}: Tempo per accessi alla memoria

T_{flop}: Tempo per op. floating point

N_{mem} e' un fattore di amplificazione per t_{mem}

Valutiamo l'impatto di N_{mem} su $Perf$

6

Calcoliamo $Perf = N_{flop} / T_{tot}$

- caso ideale (t_{mem} = 0)

$$Perf^* = \frac{N_{flop}}{T_{tot}} = \frac{N_{flop}}{N_{flop} t_{flop}} = \frac{1}{t_{flop}}$$

$Perf^*$ = **Peak performance**

Esempio: CPU a 3 Ghz e 4 op f.p./ secondo ogni ciclo di clock

- 1 ciclo di clock = 1/frequenza = $0.33\ 10^{-9}$ sec
- $t_{flop} = 0.33\ 10^{-9} / 4 \text{ sec} = 0.0825\ 10^{-9} \text{ sec}$

$$Perf^* = {}^{1}/_{t_{flop}} \sim 12 \cdot 10^{9} \text{ op f.p./sec} = 12\ Gflops$$

7

in generale

- Caso **reale**: $t_{mem} > 0$

$$Perf = \frac{N_{flop}}{T_{tot}} = \frac{N_{flop}}{N_{flop} t_{flop} + N_{mem} t_{mem}}$$

Moltiplicando num. e den. per $1/(N_{flop}\ t_{flop})$

$$Perf = \frac{1/t_{flop}}{1 + \left(\frac{N_{mem}}{N_{flop}} \frac{t_{mem}}{t_{flop}}\right)} = \frac{Perf^*}{1 + \left(\frac{N_{mem}}{N_{flop}} \frac{t_{mem}}{t_{flop}}\right)}$$

Fattore di decadimento della peak performance

8

impatto di N_{mem} su $Perf$

$$\frac{N_{mem}}{N_{flop}} \frac{t_{mem}}{t_{flop}}$$

dipende dall'algoritmo
(e' il numero di accessi alla memoria per ogni operazione eseguita)

dipende dall'hardware
(c'e' poco da fare)

posto $q = \frac{N_{mem}}{N_{flop}}$

Ridurre q significa avvicinare il caso reale al caso ideale

(e quindi le prestazioni $Perf$ alla Peak Performance $Perf^*$)

9

osservazione

posto $q = \frac{N_{mem}}{N_{flop}}$ si ha $Perf = \frac{Perf^*}{1 + \left(q \frac{t_{mem}}{t_{flop}}\right)}$

Se $q \sim \frac{t_{flop}}{t_{mem}}$ si ottiene $Perf \sim \frac{Perf^*}{2}$ (cioe' circa meta della peak perf.)

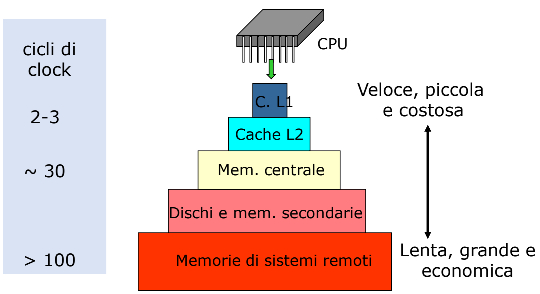
Esempio:
Con una CPU a 3 GHz e una memoria DDR4 si ha: $20 \leq \frac{t_{mem}}{t_{flop}} \leq 100$

quindi deve essere $\frac{1}{100} \leq q \leq \frac{1}{20}$ per avere circa la meta' della peak perf.

decine di operazioni floating point per ogni accesso alla memoria

10

la memoria gerarchica



Utilizzare i dati nei livelli alti della memoria (cache L1 e L2) significa **ridurre gli accessi** alla memoria e sostenere piu' facilmente la velocita' operativa della CPU

11

esempio:

CPU capace di eseguire 1 operazione f.p. in un ciclo di 10^{-9} sec

Dati nella cache L2 (~ 5 cicli per l'accesso)

- Se $q = N_{mem}/N_{flop} = 0.2$ (cioe' 5 operazioni f.p. per dato trasferito) si ha circa il 50% di Perf*

Dati in memoria centrale (~30 cicli per l'accesso)

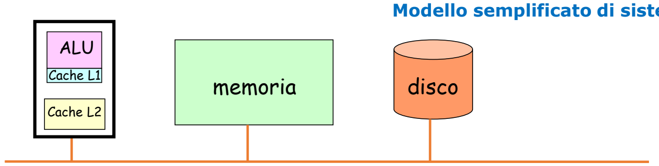
- Se $q = N_{mem}/N_{flop} = 0.03$ (cioe' 30 operazioni f.p. per dato trasferito) si ha circa il 50% di Perf*

Il gap tra t_{mem} e t_{flop} puo' essere colmato dall'uso delle cache

12

come funzionano le cache

Modello semplificato di sistema



i dati passano dalla memoria alle cache L1 e L2

- le cache sono sempre interne alla CPU

tali dati permangono nelle cache per i riferimenti successivi

- quando la CPU richiede un dato viene interrogata la cache L1
- se il dato e' presente in cache L1 viene utilizzato, se e' assente (L1-miss) si interroga la cache L2
- se il dato e' presente in cache L2 viene utilizzato, se e' assente (L2-miss) si interroga la memoria centrale

13

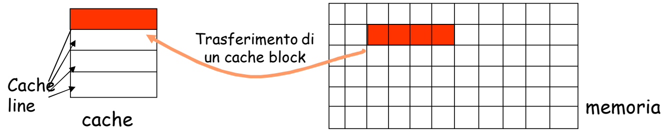
come funzionano le cache

Le cache sono di **piccole dimensioni**

- Esempio Intel Core i7-4720 4 core (2015)
  - L1 di 4x64 KB, L2 di 4x256 KB, L3 di 6 MB
- quando la CPU richiede nuovi dati e la cache e' piena, **i vecchi dati sono sostituiti dai nuovi nelle cache (con algoritmi LRU-like)**

Le cache sono organizzate in "**cache line**" e il trasferimento dei dati dalla memoria avviene in blocchi di **dati contigui** (cache block)

- quando un dato e' richiesto viene trasferito un **blocco** di elementi vicini
- se tutti i dati del blocco vengono utilizzati si **ammortizza il costo** del cache miss



14

come funzionano le cache

Ogni livello di cache ha differenti

- latenza (tempo di accesso alla cache)
- bandwidth (numero di bytes trasferiti nell'unita' di tempo)

Numerosi livelli di cache

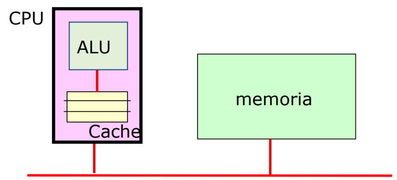
Una gestione efficiente della cache e' possibile solo attraverso linguaggi a **basso livello** e dipende fortemente dal **sistema operativo**, ma qualcosa e' possibile fare anche a livello di applicazione

Principale metodologia
Ristrutturare gli algoritmi in maniera da
riutilizzare piu' volte tutti i dati del cache block

15

alcune semplificazioni in seguito

- Sistema con **2 livelli** di memoria (Memoria cache e Memoria centrale)
- Si **ignora** un eventuale **parallelismo** tra accesso in memoria e ALU
- Memoria cache di dimensioni tali da contenere **almeno 3 righe**
- **Tempo** di accesso alla memoria cache **trascurabile**



16

caso di studio:

$$C = C + A * B$$

A, B e C matrici di ordine N

```
for __ = 1 to N
    for __ = 1 to N
        for __ = 1 to N
            C (i,j) = C(i,j)+ A(i,k)*B(k,j)
        endfor
    endfor
endfor
```

6 combinazioni di indici

- i j k
- j i k
- i k j
- j k i
- k i j
- k j i

Per ogni combinazione di indici sono sempre $2N^3$ operazioni
(ma il numero di accessi?)

Osservazione: C richiede 2 accessi; A e B un solo

prestazioni delle 6 versioni

- Esecuzione su Intel core i5 4460S, 2.9 GHz, 4 core (Perf* ~ 40 Gflops /core)
- Compilatore cc con opzione –O3, S.O. Ubuntu 16
- Matrici double di ordine N=50 (LD = 1500)

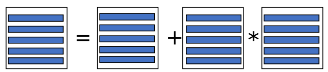
	Gflops
i j k	1.45
j i k	1.13
i k j	5.57
j k i	1.83
k i j	5.43
k j i	1.90

Max perf

Max perf

forma ikj (caso N=L intera riga in cache)

```
for i = 1 to N
    for k = 1 to N
        for j = 1 to N
            C (i,j) = C(i,j)+ A(i,k)*B(k,j)
        endfor
    endfor
endfor
```



i=1
(ripetere per i=2,..,N)

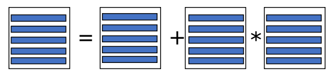
		Accessi su		
		C	A	B
k=1	j=1,..,n	1	1	1
k=2	j=1,..,n	0	0	1
…	j=1,..,n	…	…	…
k=N	j=1,..,n	1	0	1
tot		2	1	N

Totale accessi $N_{mem} = N(3 + N) = 3N + N^2 = O(N^2)$

analogamente per la forma kij

forma jki (caso N=L intera riga in cache)

```
for j = 1 to N
    for k = 1 to N
        for i = 1 to N
            C (i,j) = C(i,j)+ A(i,k)*B(k,j)
        endfor
    endfor
endfor
```



j=1
(ripetere per j=2,..,N)

		Accessi su		
		C	A	B
k=1	i=1,..,n	2N	N	1
k=2	i=1,..,n	2N	N	1
…	i=1,..,n	…	…	…
k=N	i=1,..,n	2N	N	1
tot		$2N^2$	N^2	N

Totale accessi $N_{mem} = N(3N^2 + N) = 3N^3 + N^2 = O(N^3)$

analogamente per la forma kji

forma ijk (caso N=L intera riga in cache)

```
for i = 1 to N
    for j = 1 to N
        for k = 1 to N
            C (i,j) = C(i,j)+ A(i,k)*B(k,j)
        endfor
    endfor
endfor
```

= + *

i=1
(ripetere per i=2,..,N)

		Accessi su		
		C	A	B
j=1	k=1,..,n	1	1	N
j=2	k=1,..,n	0	0	N
…	k=1,..,n	…	…	…
j=N	k=1,..,n	1	0	N
tot		2	1	N^2

Totale accessi $N_{mem} = N(3 + N^2) = 3N + N^3 = O(N^3)$

analogamente per la forma jik

21

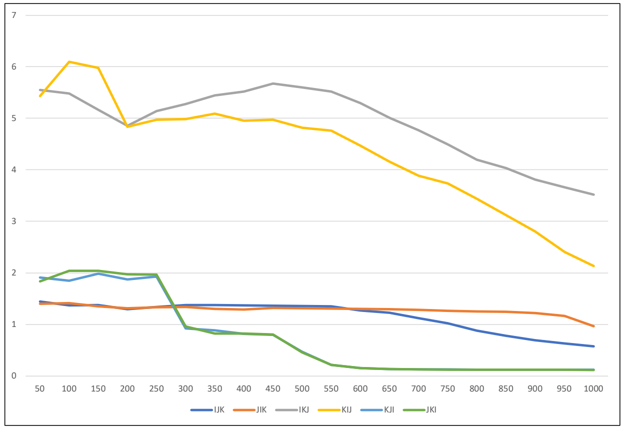
prestazioni delle 6 versioni

- Esecuzione su Intel core i5 4460S, 2.9 GHz, 4 core (Perf* ~ 40 Gflops /core)
- Compilatore cc con opzione –O3, S.O. Ubuntu 16
- Matrici double di ordine N=50 (LD = 1500)

	Gflops	N_{mem}	
i j k	1.45	$O(N^3)$	
j i k	1.13	$O(N^3)$	
i k j	5.57	$O(N^2)$	Max perf
j k i	1.83	$O(N^3)$	
k i j	5.43	$O(N^2)$	Max perf
k j i	1.90	$O(N^3)$	

22

prodotto $C = C + A*B$ al variare di N



Al crescere di N le prestazioni diminuiscono per tutte le versioni

23

calcolo dei cache miss (N/L intero)

```
for i = 1 to N
    for k = 1 to N
        for j = 1 to N
            C (i,j) = C(i,j)+ A(i,k)*B(k,j)
        endfor
    endfor
endfor
```

= + *

i=1
(ripetere per i=2,..,N)

		Accessi su		
		C	A	B
k=1	j=1,..,n	2N/L	1	N/L
k=2	j=1,..,n	2N/L	0	N/L
…	j=1,..,n	…	1	…
k=N	j=1,..,n	2N/L	0	N/L
tot		$2N^2/L$	N/L	N^2/L

Totale accessi $N_{mem} = N\left(\frac{3N^2}{L} + \frac{N}{L}\right) = \frac{3N^3 + N^2}{L} = O(N^3)$

24

calcolo di $q = N_{mem} / N_{flop}$

Operazione tra matrici C =C+A*B

- N_{flop} = $2N^3$

$$q = \frac{3N^3/L + N^2/L}{2N^3} = \frac{3}{2L} + \frac{1}{2NL}$$

Tale valore e' maggiore del corrispondente q del caso N=L

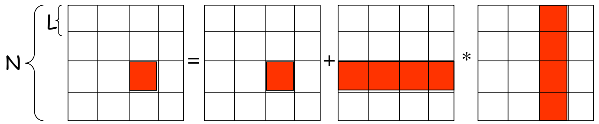
$$q = \frac{3}{2L^2} + \frac{1}{2L}$$

25

possible soluzione (N/L intero)

Suddividere le matrici in **blocchi di ordine L e applicare il caso N=L**

Ogni blocco di C e' il prodotto di un blocco di righe di A e un blocco di colonne di B



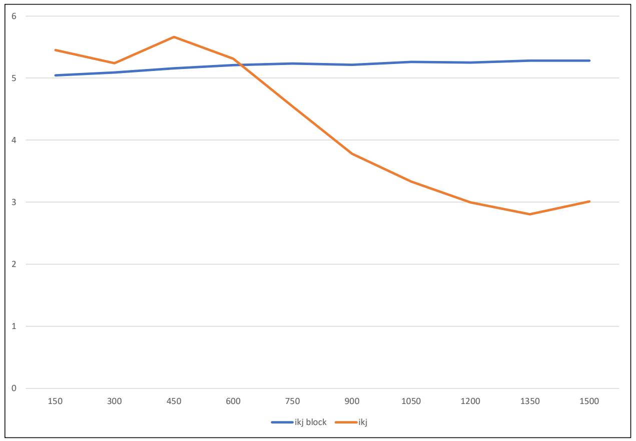

```
for ii = 1 to N/L
    for jj = 1 to N/L
        for kk = 1 to N/L
            C (ii,jj) = C(ii,jj) + A(ii,kk) * B(kk,jj)
        endfor
    endfor
endfor
```

Blocchi di ordine L !!

6 cicli innestati !!!

26

prodotto matrici a blocchi



Al crescere di N le prestazioni della versione a blocchi rimane sostenuta

27

calcolo di q per la versione a blocchi

```
for ii = 1 to N/L
    for jj = 1 to N/L
        for kk = 1 to N/L
            C (ii,jj) = C(ii,jj) + A(ii,kk) * B(kk,jj)
        endfor
    endfor
endfor
```

N^3/L^3 prodotti a blocchi di ordine L $\Rightarrow$ $N_{mem} = \frac{N^3}{L^3}(3L + L^2)$

$$q = \frac{\frac{N^3}{L^3}(3L + L^2)}{2N^3} = \frac{3L + L^2}{2L^3} = \boxed{\frac{3}{2L^2} + \frac{1}{2L}} < \boxed{\frac{3}{2L} + \frac{1}{2NL}}$$

versione a blocchi

versione non a blocchi

28

che accade con 2 o piu' livelli di cache?

- E' necessario
  - Minimizzare la comunicazione tra tutti i livelli
  - trovare le giuste dimensioni dei blocchi
- Fortemente dipendente dall'architettura
  - Solo memoria centrale ⇒ 3 cicli innestati
  - 1 livello di cache ⇒ 6 cicli innestati
  - 2 livelli di cache ⇒ 9 cicli innestati
- Necessita' di strumenti di piu' "basso livello"

29

elaborati proposti (1)

Sviluppare le 6 implementazioni in C del prodotto tra matrici

- `matmatijk`
- `matmatikj`
- `matmatkij`
- `matmatkji`
- `matmatjik`
- `matmatjki`

con il seguente prototipo

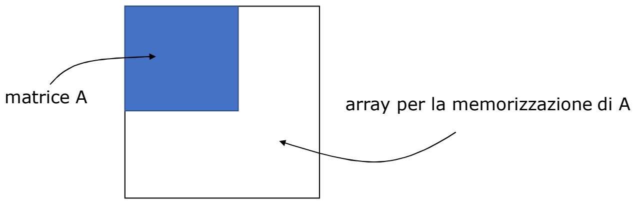
```
void matmatijk (int ldA, int ldB, int ldC,
                       double *A, double *B, double *C,
                       int N, int M, int P) {
…
}
```

analogamente per le altre versioni

30

elaborati proposti (1)

- `ldA, ldB, ldC` — `leading dimension dei 3 array`
- `N, M, P` — `dimensioni delle 3 matrici`
- `*A, *B, *C` — `puntatori al primo element dell'array`

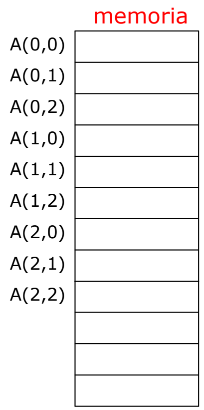


tenere ben distinti le matrice e gli l'array dove esse sono memorizzate
(le dimensioni possono non essere le stesse!)

31

elaborati proposti (1)

$$A = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} \\ A_{1,0} & A_{1,1} & A_{1,2} \\ A_{2,0} & A_{2,1} & A_{2,2} \end{pmatrix}$$



Alcune regole del C

- array memorizzati in locazioni contigue
- array 2-dimensionali memorizzati per righe
- passaggio alla function del solo indirizzo del primo elemento (prog. chiamante e function condividono le locazioni di memoria)
- l'elemento A(i,j) dista in memoria

3*i+ j

locazioni di memoria dall'elemento A[0][0], dove 3 e' il numero di colonne con cui e' stata **dichiarata** la matrice

in questo caso 3 e' la leading dimension

32

elaborati proposti (1)

esempio senza Leading Dimension

programma chiamante

```
N = 2;
A = (double*)malloc(sizeof(double)*3*3);

matmat(A, … N, …);
```

function

```
void matmat (double *A, … int N, …){

}
```

memoria

	memoria	
A(0,0)		A(0,0)
A(0,1)		A(0,1)
A(0,2)		A(1,0)
A(1,0)		A(1,1)
A(1,1)		
A(1,2)		
A(2,0)		
A(2,1)		
A(2,2)		

nella function come determinare la posizione di A(i,j) in memoria?

33

elaborati proposti (1)

esempio con Leading Dimension

programma chiamante

```
N = 2; ldA = 3;
A = (double*)malloc(sizeof(double)*3*3);

matmat(ldA,… A, … N, …);
```

function

```
void matmat (int ldA,… double *A, … int
N, …){

}
```

	memoria	
A(0,0)		A(0,0)
A(0,1)		A(0,1)
A(0,2)		A(0,2)
A(1,0)		A(1,0)
A(1,1)		A(1,1)
A(1,2)		A(1,2)
A(2,0)		A(2,0)
A(2,1)		A(2,1)
A(2,2)		A(2,2)

nella function, la posizione di A(i,j) in memoria e' **A + i*ldA + j**

34

elaborato proposto (1)

- dopo aver sviluppato le 6 function, eseguire il prodotto di matrici per

  N=M=P = 50, 100, 150,…, 1500 (step 50)

  e calcolare le prestazioni in Gflops

  Gflops = Nflops/tempo/ 10^9

  (compilare con l'opzione –O3)

- verificare che la versione migliore e' quella implementata da `matmatikj`
- determinare la dimensione ottimale

35

elaborato proposto (2)

- implementare quindi una versione a blocchi del prodotto tra matrici con il prototipo

```
void matmatblock (int ldA, int ldB, int ldC,
                  double *A, double *B, double *C,
                  int N, int M, int P
                  int dbA, int dbB, int dbC) {
…
}
```

dove dbA, dbB, dbC sono le dimensioni dei blocchi
ricavate dall'esercizio precedente (porre comunque dbA=dbB=dbC)

- richiamare la function `matmatikj`
- verificare che al crescere di N le prestazioni non decadono

36