Calcolo Scientifico: II lezione

sviluppo di software

efficiente

per le operazioni di base

del calcolo matriciale

1

Efficienza del software…

$$T_s \cong k \cdot \mu \cdot T(N)$$

Tecnologia

Hardware

Algoritmo

Software

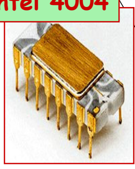
Evoluzione della tecnologia hardware

	1970	1995	2010
Velocità del proc.	20 M flops	6 G flops	10 Tera flops
Memoria RAM	200 K words	50 M words	100 G words
Hard disk	200 M words	500 G words	1 Peta words

IBM RS/6000 SP

IBM® Power 3®

Intel 4004



ASCII Red

CRAY 1

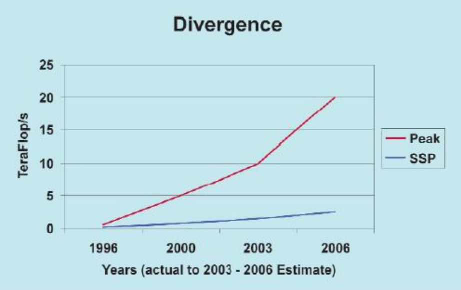
Evoluzione della tecnologia software

	1945	1955	1965	1975	1985
Tempo	2×10^6 anni	20 anni	1 giorno	12 ore	0.2 sec
Memoria	800 M words	5 M words	300 K words	170 K words	50 K words

costo computazionale necessario alla risoluzione di un problema differenziale 3D <u>su uno stesso calcolatore</u>

MA .....

Sono sempre maggiori le difficoltà nell'ottenere le massime prestazioni



Obiettivo: ridurre il gap tra la *peak* performance (la massima prestazione del calcolatore) e la *sustained* performance (prestazione massima del software su quel calcolatore) (SSP)

Nel dettaglio…

tempo di esecuzione di un sw

$$T_S(N) = \underbrace{\underbrace{N_{mem}}_{\text{Algoritmo}} \times \underbrace{t_{mem}}_{\text{Hardware}}}_{T_{mem}(N)} + \underbrace{\underbrace{N_{flop}}_{\text{Algoritmo}} \times \underbrace{t_{flop}}_{\text{Hardware}}}_{T_A(N)}$$

T_{mem}(N)
Tempo per accessi alla memoria

T_A(N)
Tempo per op. floating point

Quando si raggiunge la massima prestazione ?

$$T_S(N) = T_A(N) \Rightarrow \frac{T_S(N)}{T_A(N)} = 1$$

OVVERO quando il tempo di esecuzione del software $T_s(N)$

è uguale al tempo $T_A(N)$ dell'algoritmo

MA in generale…..

$$\frac{T_S(N)}{T_A(N)} = \frac{N_{mem} x t_{mem} + N_{flop} x t_{flop}}{N_{flop} x t_{flop}} = 1 + \frac{N_{mem} x t_{mem}}{N_{flop} x t_{flop}}$$

avviene che

$$T_S(N) > T_A(N)$$

per migliorare l'efficienza dobbiamo

rendere "piccolo" il rapporto

Per migliorare l'efficienza…

$$\frac{T_S(N)}{T_A(N)} = 1 + \frac{N_{mem} \times t_{mem}}{N_{flop} \times t_{flop}}$$

Problema 1

Ridurre il numero di accessi alla memoria

Problema 2

Ridurre il tempo di accesso alla memoria

Problema 1

$$\frac{T_S(N)}{T_A(N)} = 1 + \frac{N_{mem} \times t_{mem}}{N_{flop} \times t_{flop}}$$

$$q = \frac{N_{mem}}{N_{flop}} = \frac{\#\,\text{accessi in memoria}}{\#\,\text{operazioni floating-point}}$$

Parametro di valutazione del traffico "parassita"

E' possibile ridurre il traffico parassita q ?

Quanto vale il parametro q

(misura del traffico parassita)

per le operazioni di base di algebra lineare

?

Aggiornamento di un vettore

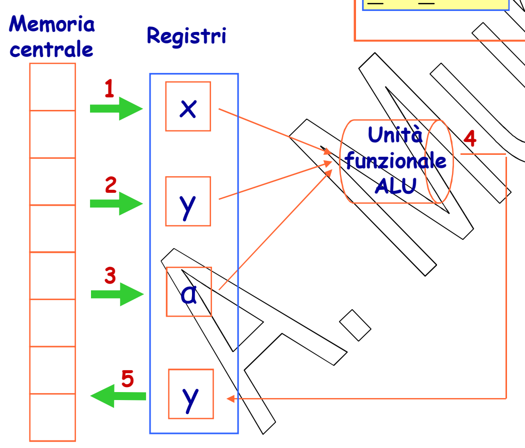
Prodotto scalare

Prodotto Matrice vettore

Prodotto Matrice -Matice

Esempio1: esecuzione di una aggiornamento di un vettore (saxpy)

$\underline{y} = \underline{y} + \alpha \underline{x}$ — $\underline{x}, \underline{y}$ Vettori di dim. n; α scalare



1 - preleva x → n accessi

2 - preleva y → n accessi

3 - preleva α → 1 accesso

4 - calcolo → 2n op. fl.p.

5 - riponi y → n accessi

Calcoliamo q….

SAXPY ➡ $q = \frac{m}{f} = \frac{3n+1}{2n} \approx \frac{3}{2}$

1 - preleva x → n accessi

2 - preleva y → n accessi

3 - preleva α → 1 accesso

4 - calcolo → 2n op. fl.p.

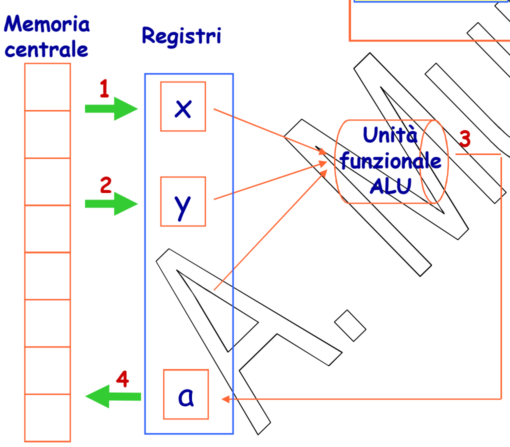
5 - riponi y → n accessi

$\boxed{\text{\# accessi > \# operazioni f.p.}}$

Esempio2: esecuzione di un prodotto scalare (dot)

$a = \underline{x}^T \underline{y}$ $\quad \underline{x}, \underline{y}$ Vettori di dim. n

α scalare



1 - preleva x → n accessi

2 - preleva y → n accessi

3 - calcolo → 2n op. fl.p.

4 riponi α → 1 accesso

Calcoliamo q….

DOT ➡ $q = \frac{m}{f} = \frac{2n+1}{2n} \approx 1$

1 - preleva x → n accessi

2 - preleva y → n accessi

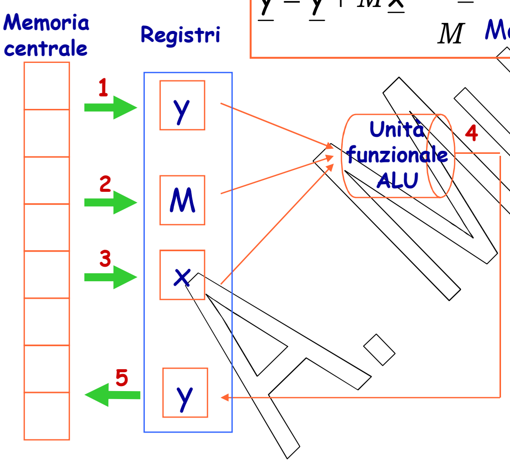
3 - calcolo → 2n op. fl.p.

4 riponi α→ 1 accesso

$\boxed{\text{\# accessi = \# operazioni f.p.}}$

Esempio3: esecuzione di un prodotto matrice vettore (gemmv)

$\underline{y} = \underline{y} + M\underline{x}$ — $\underline{x}, \underline{y}$ Vettori di dim. n; M Matrice di dim. n



- Preleva dalla memoria ⟶ n accessi
- Preleva dalla memoria ⟶ n^2 accessi
- Preleva dalla memoria ⟶ n accessi
- Esegui ⟶ $2n^2$ flop
- Riponi in memoria ⟶ n accessi

Calcoliamo q….

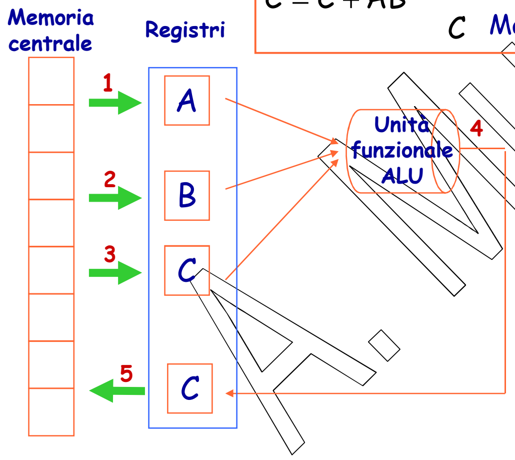
GEMMV ⟹ $q_{gemv} = \dfrac{3n+n^2}{2n^2} = \dfrac{1}{2} + \dfrac{3}{2n}$

- Preleva dalla memoria ⟶ n accessi
- Preleva dalla memoria ⟶ n^2 accessi
- Preleva dalla memoria ⟶ n accessi
- Esegui ⟶ $2n^2$ flop
- Riponi in memoria ⟶ n accessi

$\boxed{\#\text{ accessi} > \#\text{ operazioni f.p.}}$

Esempio 4: esecuzione di un prodotto matrice marice (gemm)

$C = C + AB$ — A, B Matrici di dim. n; C Matrice di dim. n



- Preleva dalla memoria ⟶ n^2 accessi
- Preleva dalla memoria ⟶ n^2 accessi
- Preleva dalla memoria ⟶ n^2 accessi
- Esegui ⟶ $2n^3$ flop
- Riponi in memoria ⟶ n^2 accessi

Calcoliamo q….

GEMM → $q_{gemm} = \frac{3n^2 + (n)^2}{2(n)^3} = \frac{2}{n}$

- Preleva dalla memoria → n^2 accessi
- Preleva dalla memoria → n^2 accessi
- Preleva dalla memoria → n^2 accessi
- Esegui → $2n^3$ flop
- Riponi in memoria → n^2 accessi

\# accessi < # operazioni f.p.

In conclusione….

$q_{saxpy} \approx \frac{3}{2}$

\# accessi > # operazioni f.p.

$q_{dot} \approx 1$

\# accessi = # operazioni f.p.

$q_{gemv} \approx \frac{1}{2}$

\# accessi < # operazioni f.p.

$q_{gemm} \approx \frac{2}{n}$

\# accessi << # operazioni f.p.

Il traffico parassita q è una costante che DIPENDE dal nucleo computazionale di base

(un prodotto tra matrici e'piu' conveniente in termini di traffico parassita)!!

In sintesi per migliorare l'efficienza…

$$\frac{T_S(N)}{T_A(N)} = 1 + \frac{N_{mem} x t_{mem}}{N_{flop} x t_{flop}} = 1 + q \frac{t_{mem}}{t_{flop}}$$

Il numero di accessi alla memoria q è costante e dipende dal problema

BISOGNA
Ridurre il tempo di accesso alla memoria

E' possibile ridurre il tempo di accesso alla "memoria" ?

Risposta

$$\frac{T_S(N)}{T_A(N)} = 1 + q \frac{t_{mem}}{t_{flop}}$$

numero di accessi alla memoria q costante

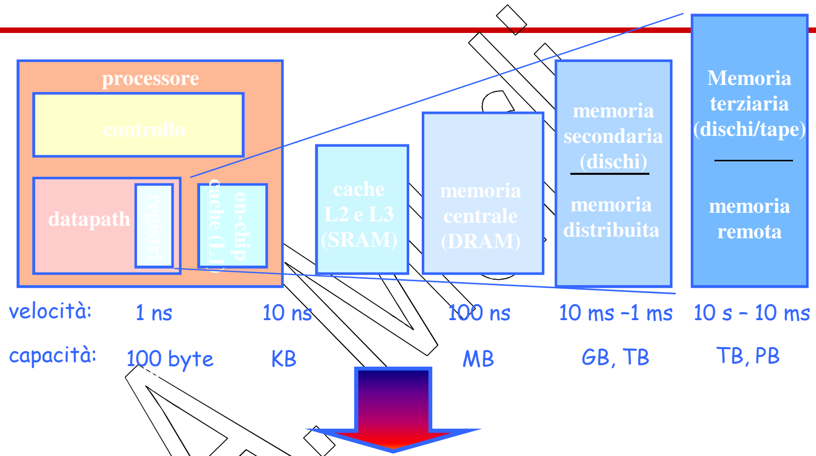
Ridurre il tempo di accesso alla memoria

Principio di Località dei dati

Utlizzando la memoria "gerarchica"

Rivisitazione degli algoritmi al fine di…

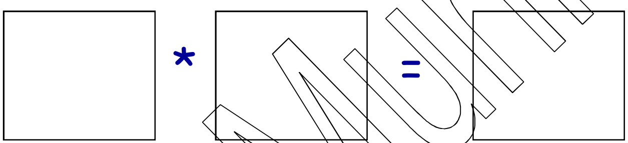
…Utilizzare la localita' dei dati …



OVVERO… proggettare sw che riusano i dati nei
"Livelli alti della memoria"

Un caso studio il prodotto di due matrici:

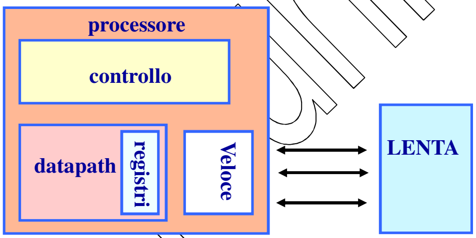
$$A * B = C$$



```
for …
    for …
        for …
            c(i,j)=c(i,j)+a(i,k)*b(k,j)
        endfor
    endfor
endfor
```

Alcune Ipotesi:

1. Il Calcolatore in considerazione ha soli 2 livelli nella gerarchia di memoria (“veloce” e “lenta”)



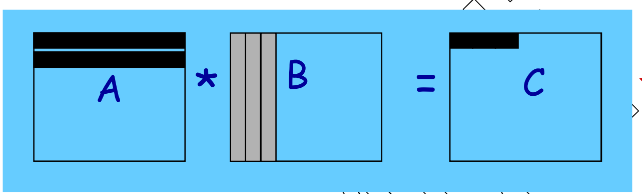
2. Tutti i dati all'inizio sono presenti nella memoria lenta

3. Nel modello di prestazione si assume che:

$$\frac{T_S(N)}{T_A(N)} = 1 + q\frac{t_{mem}}{t_{flop}} \rightarrow 10^{-1}\ \text{sec}$$

Esempio: A,B,C matrici quadrate di dimensione 10

Strategia I

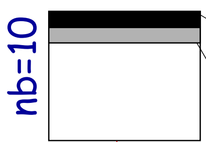


Memoria lenta

- Il primo elemento di C e' ottento da un prodotto scalare (saxpy)

  (trasferimento memoria Lenta-Veloce 1 riga di A ed 1 di B)

nb=10



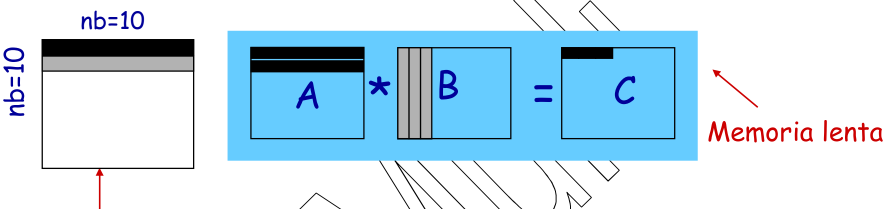
- Il secondo elemento di C e' ottenuto da un prodotto scalare (saxpy)

  (trasferimento memoria Lenta-Veloce 1 riga di A ed 1 di B)

- …

Memoria veloce

Strategia I (cont)



La prima riga di C e' ottenuta mediante 10 saxpy

(trasferimento memoria Lenta-Veloce 10 riga di A ed 10 di B)

Gli elementi di C sono ottenuti mediante 100 saxpy

(trasferimento memoria Lenta-Veloce 10 riga di A ed 10 di B 10 volte!)

Domanda

Strategia I:

Ogni elemento di C ottenuto facendo un prodotto scalare (saxpy)…

… quanto costa tale strategia in termini di ACCESSI alla memoria ?

Risposta

Consideriamo gli accessi alla memoria

$$\frac{T_S(N)}{T_A(N)} = 1 + q\frac{t_{mem}}{t_{flop}}$$

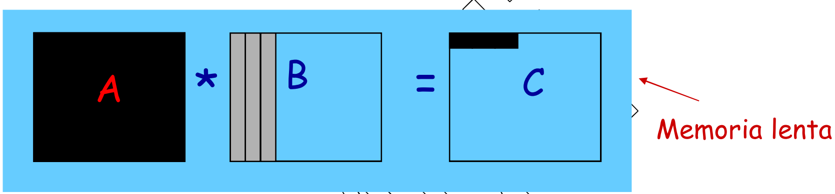
Sono necessarie 100 saxpy per ottenere C dunque…

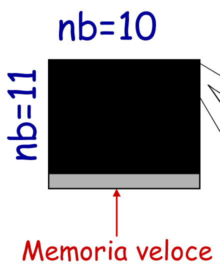
$$100 \times \underbrace{q_{saxpy}}_{\approx 1} \times \underbrace{\frac{t_{mem}}{t_{flop}}}_{10^{-1}} = 100 \times 1 \times 10^{-1} = 10\,\text{sec}$$

La strategia I richiede 10 sec solo per gli accessi alla memoria!

Esempio: A,B,C matrici quadrate di dimensione 10

Strategia II





- La prima riga di C e' ottento mediante un prodotto matrice vettore (GEMMV).

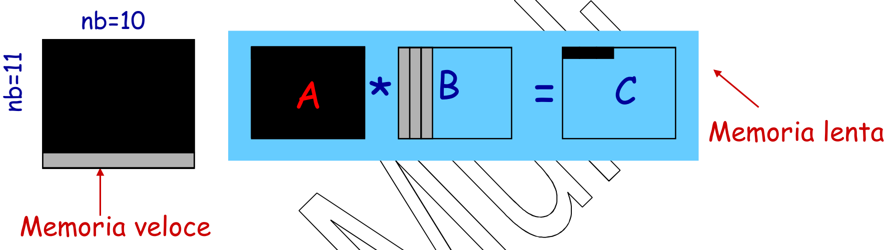
(trasferimento memoria Lenta-Veloce di 1 sola riga di B e tutta A)

- La seconda riga di C e' ottenta mediante un prodotto matrice vettore (GEMMV)

(trasferimento memoria Lenta-Veloce di 1 sola riga di B)

- …

Strategia II (cont)



Le righe di C sono ottenute mediante 1 GEMMV

(trasferimento memoria Lenta-Veloce 1 elemento di B ad ogni passo e di tutta A una volta sola)

Gli elementi di C sono ottenuti mediante 10 GEMMV

Domanda

Strategia II:

Ogni riga di C è ottenuta mediante un prodotto matrice-vettore (GEMMV)…

… Ma quanto costa tale strategia in termini di

ACCESSI alla memoria ?

Risposta

Consideriamo gli accessi alla memoria

$$\frac{T_S(N)}{T_A(N)} = 1 + q\frac{t_{mem}}{t_{flop}}$$

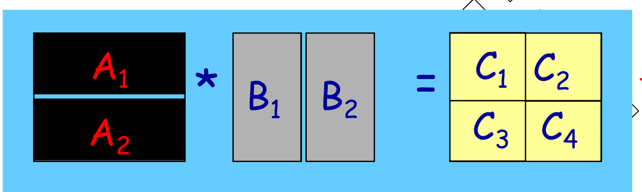
Sono necessarie 10 GEMMV per ottenere C dunque…

$$10 \times \underbrace{q_{gemmv}}_{\approx 1/2} \times \underbrace{\frac{t_{mem}}{t_{flop}}}_{10^{-1}} = 10 \times \frac{1}{2} \times 10^{-1} = 0.5\,\text{sec}$$

La strategia II richiede 0.5 sec solo per gli accessi alla memoria!

Esempio: A,B,C matrici quadrate di dimensione 10

Strategia III



Memoria lenta

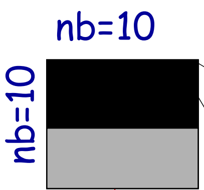
- C_1 il primo blocco di C e' ottento mediante un prodotto matrice matrice (GEMM)

(trasferimento memoria Lenta-Veloce di 1 blocco di A e di B)

- C_2 il secondo blocco di C e' ottento mediante un prodotto matrice matrice (GEMM)

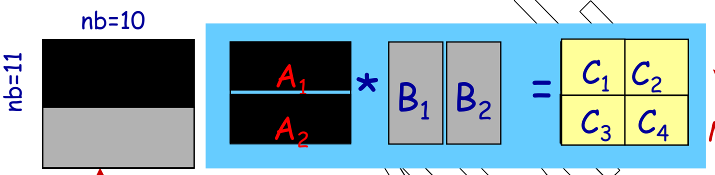
(trasferimento memoria Lenta-Veloce di 1 solo blocco di B)

- …



Memoria veloce

Strategia III (cont)



Memoria lenta

Memoria veloce

Ogni blocco di C è ottenuto mediante 1 GEMM

Gli elementi di C sono ottenuti calcolando 4 blocchi ovvero mediante 4 GEMM

Domanda

Strategia III:

Ogni blocco di C è ottenuto mediante un prodotto matrice-matrice (GEMM)…

… Ma quanto costa tale strategia in termini di

ACCESSI alla memoria ?

Risposta

Consideriamo gli accessi alla memoria

$$\frac{T_S(N)}{T_A(N)} = 1 + q\frac{t_{mem}}{t_{flop}}$$

Sono necessarie 4 GEMM per ottenere C dunque...

$$4 \times q_{gemm} \times \frac{t_{mem}}{t_{flop}} = 4 \times \frac{1}{10} \times 10^{-1} = 4 \times 10^{-2}\ \text{sec}$$

$\approx 1/10$ $\qquad 10^{-1}$

La strategia III richiede 4×10^{-2} sec per gli accessi alla memoria!

In conclusione

Strategia I (saxpy) — richiede 10 sec per gli accessi alla memoria!

Strategia II (GEMMV) — richiede 1 sec per gli accessi alla memoria!

Strategia III (GEMM) — richiede 4×10^{-2} sec per gli accessi alla memoria!

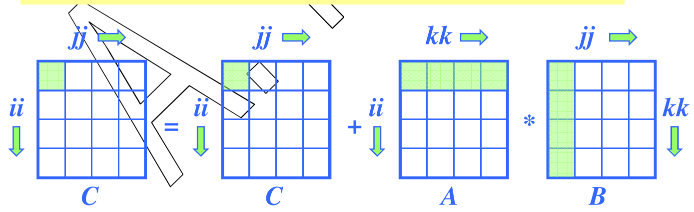
È Conveniente organizzare gli algoritmi con operazioni a blocchi (nulcei computazionali Matrice -Matrice)

In conclusione

Algoritmi con operazioni a blocchi

Riuso ottimale dei dati nei

"Livelli alti della memoria"



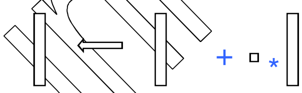
Quanto descritto è alla base di…

BLAS:
Basic Linear Algebra Subroutines:

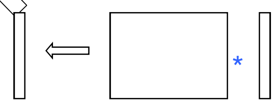
Libreria di software matematico per l'esecuzione di operazioni di base del calcolo matriciale che
ottimizza gli accessi alla memoria

BLAS

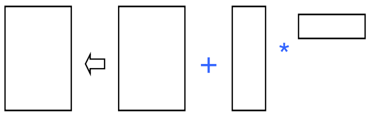
- **BLAS 1:**
  - Operazioni di base tra vettori
    - Somma
    - Aggiornamento
    - Norma
    - Prodotto scalare
    - ……



- **BLAS 2**
  - Operazioni di base tra matrice e vettori
  - Prodotto matrice -vettore
  - Aggiornamento
  - …..



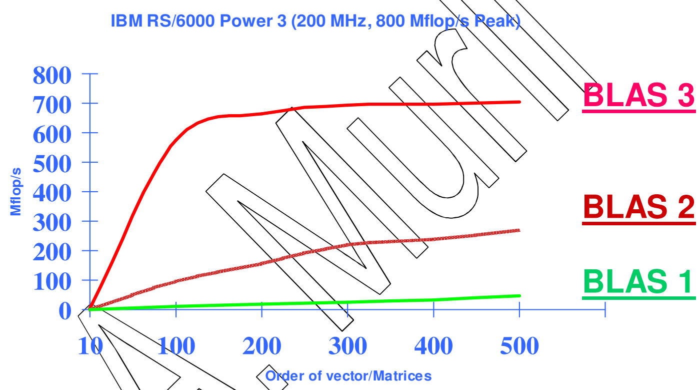
- **BLAS 3**
  - Operazioni di base tra matrici
    - Prodotto tra matrici
    - Aggiornamento
    - Norma
    - Somma
    - …..



Confronto BLAS1, BLAS2, BLAS3

- **BLAS 1:**
  - Ottimizza le operazioni tra vettori (loop unrolling)
- **BLAS 2**
  - Ottimizza il riuso dei dati che risiedono nei registri (riduce lo spostamento dei dati dalla cache ai registri)
- **BLAS 3**
  - Ottimizza il riuso dei dati che risiedono nella cache (riduce lo spostamento dei dati dalla memoria centrale alla cache)

Un confronto tra BLAS1, BLAS2, BLAS3



Solo BLAS 3 riesce a raggiungere

la peak performance del processore IBM /RS 6000

perché ottimizza l'uso di tutti i livelli della memoria gerarchica

"As machines become more powerful, the efficiency of algorithms grows more important, not less."

Nick Trefethen, 1997

Quanto più potente è l'ambiente di calcolo tanto più diventa complesso e difficile lo sviluppo di software in grado di sfruttarne appieno le capacità.

FINE LEZIONE

Esercitazione

Come sviluppare software efficiente per il calcolo matriciale ?

Case study n. 1: Prodotto matrice x matrice

Calcolo di

$A \cdot B = C$

$A, B, C \in \Re^{n\times n}$

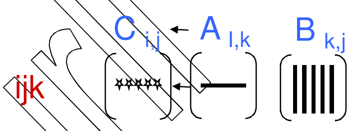
```
for _ = 1:n;
  for _ = 1:n;
    for _ = 1:n;
      C_{i,j} ← C_{i,j} + A_{i,k} B_{k,j}
    end
  end
end
```

Indipendentemente dall' ordine dei cicli abbiamo sempre $O(n^3)$ operazioni f.p.

6 Varianti della moltiplicazione di matrici

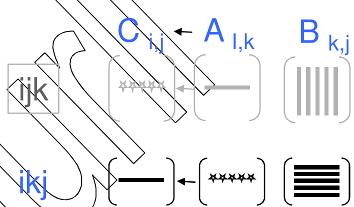
```
for _ = 1:n;
  for _ = 1:n;
    for _ = 1:n;
      C_{i,j} ← C_{i,j} + A_{i,k} B_{k,j}
    end
  end
end
```

6 Varianti della moltiplicazione di matrici



for i = 1:n;
  for j = 1:n;
    for k = 1:n;
      $C_{i,j} \leftarrow C_{i,j} + A_{i,k} B_{k,j}$
    end
  end
end

6 Varianti della moltiplicazione di matrici



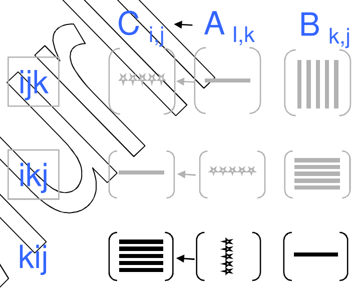
for i = 1:n;
  for k = 1:n;
    for j = 1:n;
      $C_{i,j} \leftarrow C_{i,j} + A_{i,k} B_{k,j}$
    end
  end
end

6 Varianti della moltiplicazione di matrici

for $\underline{k}$ = 1:n;
    for $\underline{i}$ = 1:n;
        for $\underline{j}$ = 1:n;

$$C_{i,j} \leftarrow C_{i,j} + A_{i,k} B_{k,j}$$

        end
    end
end

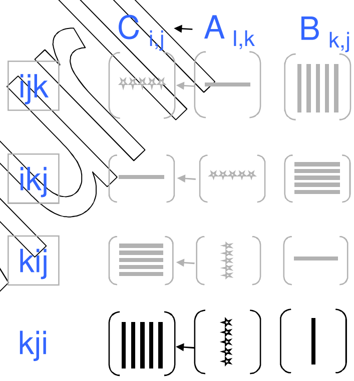


6 Varianti della moltiplicazione di matrici

for $\underline{k}$ = 1:n;
    for $\underline{j}$ = 1:n;
        for $\underline{i}$ = 1:n;

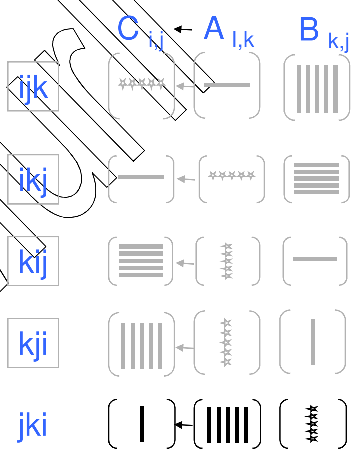
$$C_{i,j} \leftarrow C_{i,j} + A_{i,k} B_{k,j}$$

        end
    end
end



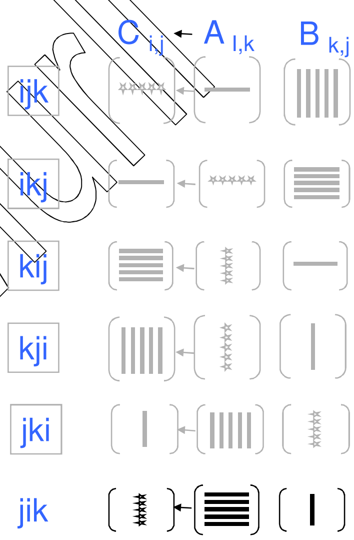
6 Varianti della moltiplicazione di matrici

for j = 1:n;
  for k = 1:n;
    for i = 1:n;
      $C_{i,j} \leftarrow C_{i,j} + A_{i,k} B_{k,j}$
    end
  end
end



6 Varianti della moltiplicazione di matrici

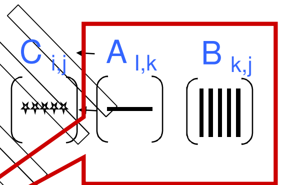
for j = 1:n;
  for i = 1:n;
    for k = 1:n;
      $C_{i,j} \leftarrow C_{i,j} + A_{i,k} B_{k,j}$
    end
  end
end



Quale versione implementare?

6 Varianti della moltiplicazione di matrici

ijk



for i = 1:n;
  for j = 1:n;
    for k = 1:n;
      $C_{i,j} \leftarrow C_{i,j} + A_{i,k} B_{k,j}$
    end
  end
end

Operazione di base:

prodotto scalare

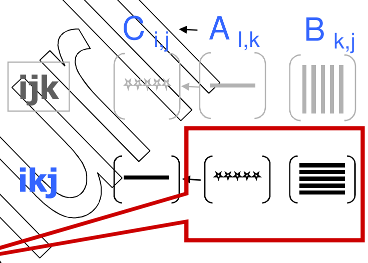
(BLAS1)

della riga i-ma di A e

della colonna j -ma di B

6 Varianti della moltiplicazione di matrici

for i = 1:n;
  for k = 1:n;
    for j = 1:n;
      $C_{i,j} \leftarrow C_{i,j} + A_{i,k} B_{k,j}$
    end
  end
end



Operazione di base:

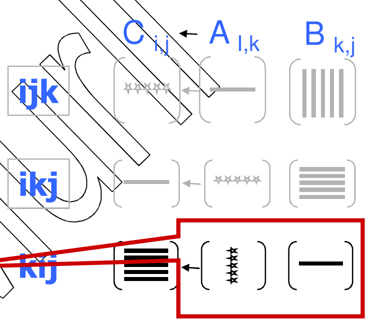
Matrice B x vettore(riga) A_{ik}

(aggiornamento per riga)

(BLAS 2)

6 Varianti della moltiplicazione di matrici

for k = 1:n;
  for i = 1:n;
    for j = 1:n;
      $C_{i,j} \leftarrow C_{i,j} + A_{i,k} B_{k,j}$
    end
  end
end



Operazione di base:saxpy

della riga k -ma di B

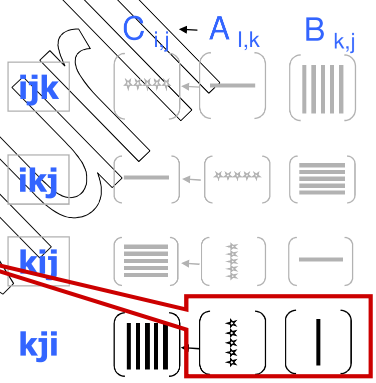
con lo scalare A_{ik}

(BLAS 1)

6 Varianti della moltiplicazione di matrici

```
for k = 1:n;
  for j = 1:n;
    for i = 1:n;
      C(i,j) <- C(i,j) + A(i,k) B(k,j)
    end
  end
end
```

$$C_{i,j} \leftarrow C_{i,j} + A_{i,k} B_{k,j}$$



Operazione di base: saxpy

della colonna k-ma di A

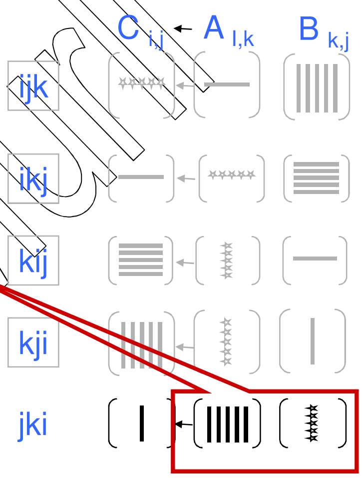
con lo scalare B_{kj}

(BLAS 1)

6 Varianti della moltiplicazione di matrici

```
for j = 1:n;
  for k = 1:n;
    for i = 1:n;
      C(i,j) <- C(i,j) + A(i,k) B(k,j)
    end
  end
end
```

$$C_{i,j} \leftarrow C_{i,j} + A_{i,k} B_{k,j}$$



Operazione di base:

Matrice x vettore

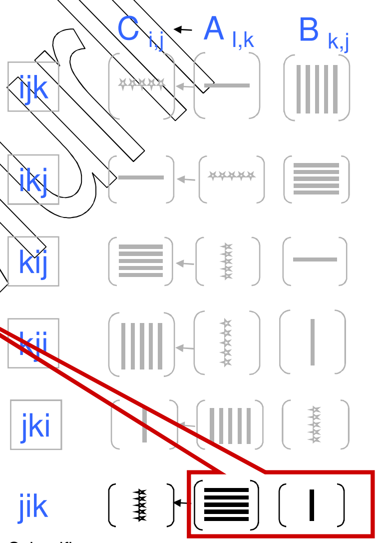
(aggiornamento per colonna)$_j$

(BLAS 2)

6 Varianti della moltiplicazione di matrici

for j = 1:n;
  for i = 1:n;
    for k = 1:n;
      $C_{i,j} \leftarrow C_{i,j} + A_{i,k} B_{k,j}$
    end
  end
end

Operazione di base:
prodotto scalare
(BLAS1)
della riga i-ma di *A* e
della colonna j -ma di *B*



Ciascuna permutazione degli indici implica
l'esecuzione di una diversa operazione di base
tra vettori (BLAS 1) o
tra vettori e matrici (BLAS 2)

Case study 2

Risoluzione di

$$Ax = b$$
$$A \in \Re^{n\times n}, \quad x, b \in \Re^n$$

Mediante
L' algoritmo di
eliminazione di Gauss

```
for k = 1 to n-1
   for i = k+1 to n
         a(i,k) = a(i,k)/a(k,k)
         for j = k+1 to n
         a(i,j) = a(i,j) – a(i,k) * a(k,j)
         endfor
     endfor
endfor
```

Indipendentemente dall' ordine dei cicli abbiamo sempre $O(n^3)$ operazioni f.p.

Versione k i j :

Operazione di base?

$$\boxed{\underline{a}^i = \underline{a}^i - a_{ik} \cdot \underline{a}^k}$$

kij

```
for k = 1 to n-1
   for i = k+1 to n
         a(i,k) = a(i,k)/a(k,k)
         for j = k+1 to n
           a(i,j) = a(i,j) – a(i,k) * a(k,j)
         endfor
     endfor
endfor
```

Aggiornamento di un vettore (**riga**) mediante il prodotto di uno scalare per un vettore (**O(n)** op. f.p.)

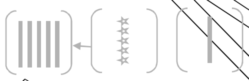
Invertendo i cicli ……Versione (k j i):

```
for k = 1 to n-1
   for j = k+1 to n
      for i = k+1 to n
         a(i,k) = a(i,k)/a(k,k)
         a(i,j) = a(i,j) – a(i,k) * a(k,j)
      endfor
   endfor
endfor
```

Operazione di base?

$$\boxed{\underline{a}^{j} = \underline{a}^{j} - a_{kj} \cdot \underline{a}^{k}}$$

kji



Aggiornamento di un vettore (**colonna**) mediante il prodotto di uno scalare per un vettore (**O(n)** op. f.p.)

Invertendo i cicli ….Versione (j k i):

```
for j = … to …
   for k = … to …
      for i = … to …
         ………
         a(i,j) = a(i,j) – a(i,k) * a(k,j)
      endfor
   endfor
endfor
```

Operazioni di base?

$$\boxed{\underline{a}^{j} = \underline{a}^{j} - M \cdot \underline{a}^{j}}$$

jki

Aggiornamento di un vettore mediante il prodotto di una matrice per un vettore (**O(n²)** op. f.p.)

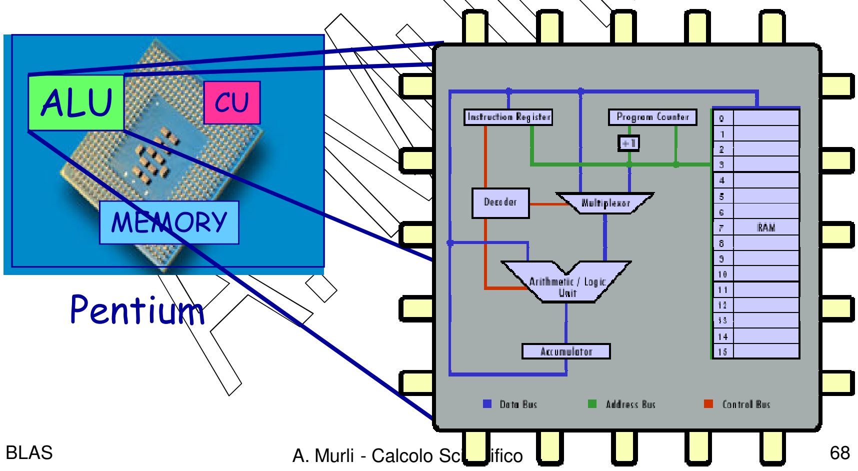
Quale versione dello stesso algoritmo conviene implementare?

In generale,

la scelta dipende dall'ambiente di calcolo …

Ambiente computazionale

- Architetture monoprocessore
- Linguaggi ad alto livello (ad esempio Fortran, C)



In che modo questo ambiente computazionale influenza l'implementazione di questi algoritmi ?

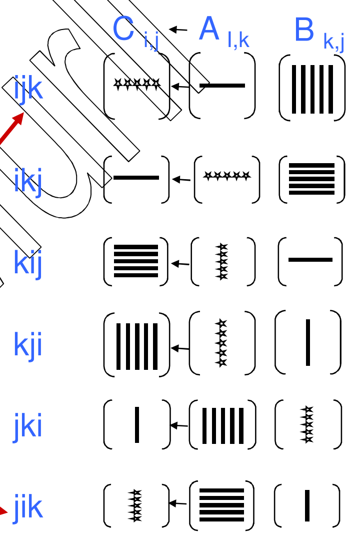
Individuiamo le varianti le cui operazioni di base sono

operazioni tra vettori

(ad es saxpy, prodotto scalare)

moltiplicazione di matrici con BLAS 1

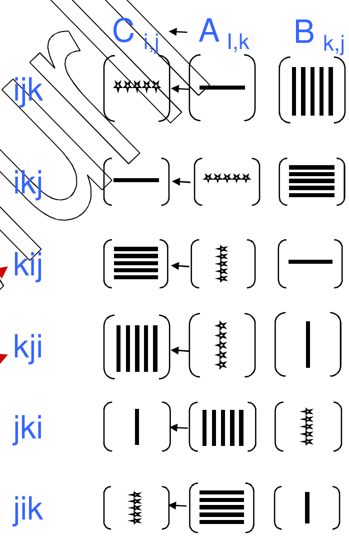
```
for _ = 1:n;
   for _ = 1:n;
      for _ = 1:n;
```
$$C_{i,j} \leftarrow C_{i,j} + A_{i,k} B_{k,j}$$
```
      end
   end
end
```



(DOT=prodotto scalare)

moltiplicazione di matrici

```
for _ = 1:n;
   for _ = 1:n;
      for _ = 1:n;
```
$$C_{i,j} \leftarrow C_{i,j} + A_{i,k} B_{k,j}$$
```
      end
   end
end
```



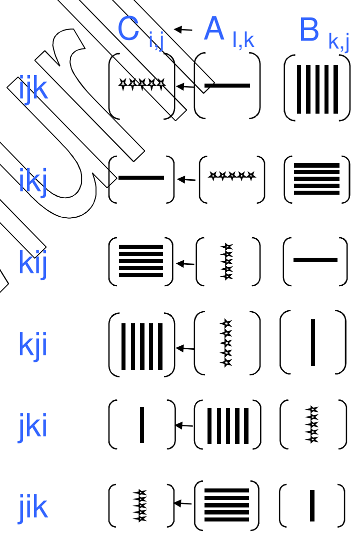
(saxpy=aggiornamento vettore)

6 Varianti della moltiplicazione di matrici

for _ = 1:n;
  for _ = 1:n;
    for _ = 1:n;

$$C_{i,j} \leftarrow C_{i,j} + A_{i,k} B_{k,j}$$

    end
  end
end



C

Fortran

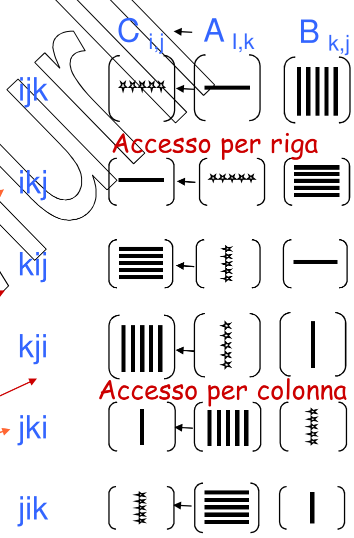
BLAS

6 Varianti della moltiplicazione di matrici

for _ = 1:n;
  for _ = 1:n;
    for _ = 1:n;

$$C_{i,j} \leftarrow C_{i,j} + A_{i,k} B_{k,j}$$

    end
  end
end



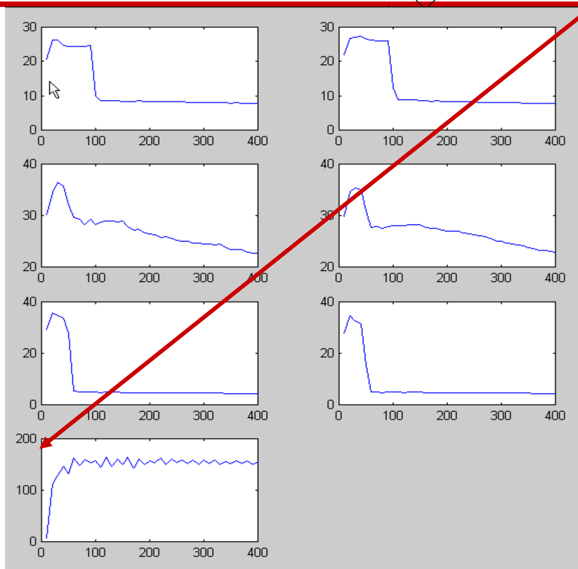
C

Fortran

BLAS

Prestazioni delle 6 varianti di matrice-matrice su un calcolatore SUN Ultra 2 - 200 MHz

- ijk
- jki
- kij
- dgemm
- jik
- kji
- ikj



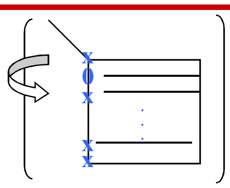
LAPACK

Libreria di software matematico per la risoluzione di problemi di algebra lineare su architetture a memoria gerarchica

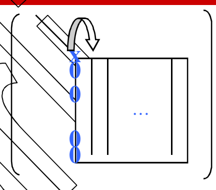
Utilizza BLAS 3 come building block

E. Anderson et al., LAPACK User' Guide, SIAM, 1995

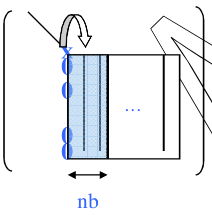
Algoritmo di Gauss con BLAS..



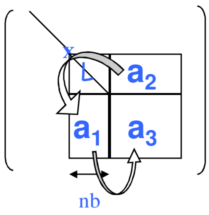
Algoritmo Standard
Saxpy per riga



apply sequence to a column
Saxpy per colonna



BLAS 2



$a_2 = L^{-1} a_2$
$a_3 = a_3 - a_1 * a_2$

BLAS 3

Alcune problematiche ….

- Molti parametri da gestire nell'implementazione degli algoritmi (dimensione dei blocchi, permutazione di indici dei loop, profondità dell'unrolling, numero di processi, topologia, ...)
- L'architettura dei microprocessori diventa sempre più complessa

Necessità di approcci automatici e/o adattativi nello sviluppo di software

Differenti caratteristiche capacità della cache per differenti calcolatori <u>con memoria gerarchica</u>

⇓

Sviluppo di implementazioni specializzate di BLAS 3

ATLAS
Automatically Tuned Linear Algebra Software

- Versione “self-tuned” di BLAS e di alcune routine di LAPACK
- Kernel fondamentale: GEMM

$$C = \alpha\, op(A)op(B) + \beta\, op(C), \qquad op(X) = X,\ X^T,$$

- Routine di BLAS 3 implementate in termini di GEMM

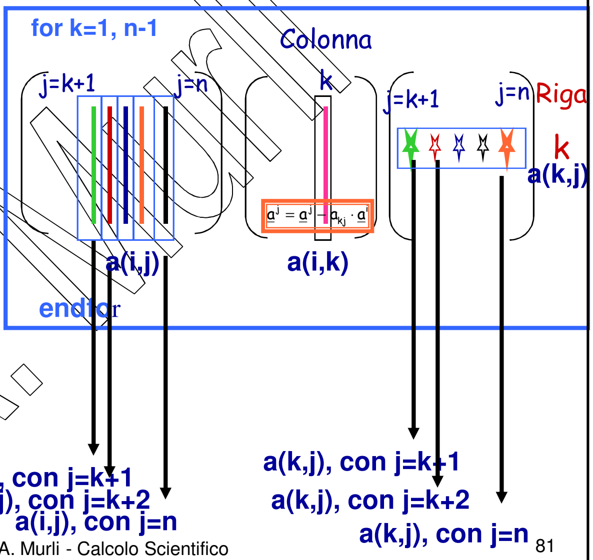
Whaley, J. Dongarra, *Automatically Tuned Linear Algebra Software,* Ninth SIAM Conference on Parallel Processing for Scientific Computing, 1999

Riprendiamo la versione kji dell'algoritmo di Gauss

```
for k = 1 to n-1
   for j = k+1 to n
      ........................
      for i = k+1 to n
         a(i,j) = a(i,j) – a(i,k) * a(k,j)
      endfor
   endfor
endfor
```

$\underline{a}^j = \underline{a}^j - a_{kj} \cdot \underline{a}^i$

Saxpy per colonna



$$\underline{a}^j = \underline{a}^j - a_{kj} \cdot \underline{a}^i$$

Al variare di k, ogni colonna a^j viene aggiornata,

(cioè prelevata e riposta in memoria)

j volte

Uso inefficiente dei registri vettoriali

Mantenere il vettore a^j nei registri finché l'aggiornamento non è completato

OVVERO....

Considerare il ciclo "for … j" come il più esterno

ciò significa far fare all'algoritmo

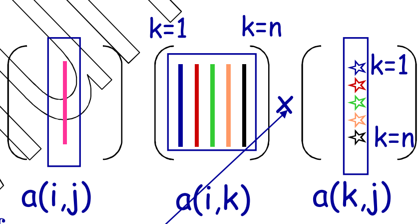
definitivamente

tutti gli aggiornamenti sul vettore j

Versione jki: rappresentazione grafica

```
for j = 1 to n
    for k = 1 to n
        ......................
        for i = k+1 to n
            a(i,j) = a(i,j) – a(i,k) * a(k,j)
        endfor
    endfor
endfor
```

for j=1, n-1



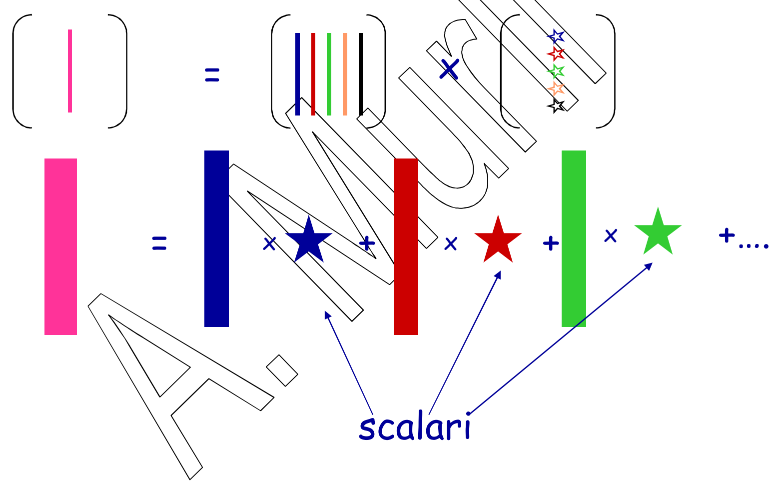
endfor

Prodotto "*colonne x colonne*"

di una matrice per un vettore

GAXPY

Esecuzione del prodotto di una matrice per un vettore secondo l'ordine "colonne x colonne" (ki invece di ik)



Ovvero …Versione jki

```
for j = 1 to n-1
  for k = 1 to n
    ......................
    for i = k+1 to n
      a(i,j) = a(i,j) – a(i,k) * a(k,j)
    endfor
  endfor
endfor
```

Operazione di base

$\underline{y} = \underline{y} + M\underline{x}$

$\underline{x}, \underline{y}$ Vettori di dim. n

M Matrice di dim. n

Aggiornamento di un vettore mediante prodotto matrice-vettore (Gaxpy)

Versioni i k j e j k i con BLAS 2

for i

GEMV

endfor

for j

GEMV

endfor

Operazione di base:

$y = y + A\,x$ $\quad A$ matrice, x, y vettori

GEMV $\Rightarrow$ $q = \dfrac{m}{f} = \dfrac{n^2+3n}{2n^2} \approx \dfrac{1}{2}$

Quindi..

ikj e jki USANO BLAS 2

La GEMV richiede asintoticamente

1 accesso alla memoria ogni 2 operazioni fl. point

Gli algoritmi di BLAS 2 sfruttano

la località dei dati che risiedono nella cache

(riuso dei dati nella cache)

Un PROBLEMA .....

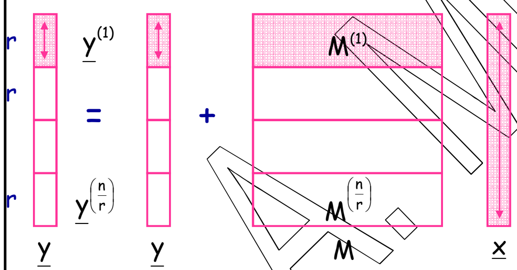
La **capacità** dei registri vettoriali è **limitata**

Nell' esecuzione della GAXPY se la dimensione del vettore y e della matrice è maggiore della capacità dei registri, si **prelevano di fatto solo sottovettori di y e blocchi di M**

"inutile" traffico (parassita) dei dati tra registri e cache

Strategia

scrivere l'algoritmo in maniera da costringere i blocchi di M e di y nei registri finché l'aggiornamento su ciascun blocco non sia stato completato



Ciascun blocco di y è calcolato come prodotto di un blocco di M per il vettore x

Calcolo di q (Gaxpy)

Per ogni i

- Preleva un blocco di $\underline{y}$ → r accessi
- Preleva un blocco di M → nr accessi
- Preleva il vettore $\underline{x}$ → n accessi
- Esegui $\underline{y}^{(j)} = \underline{y}^{(j)} + M^{(j)}\underline{x}$ → 2nr operazioni fl. p.
- Riponi il blocco di $\underline{y}$ → r accessi

$$\tilde{q}_{gaxpy} = \frac{2r+nr+n}{2nr} \approx \frac{1}{2} + \frac{1}{2r} + \frac{1}{n} \Longrightarrow \tilde{q}_{gaxpy} = \frac{1}{2} + \frac{3}{2n} = q_{gaxpy}$$

(n = r caso migliore)

Riprendiamo la versione jki sui blocchi di y

```
for j
    for k
      for i
        a(i,j) = a(i,j) - a(i,k) * a(k,j)
endfor
```

La matrice attiva M viene **prelevata e riposta** nella cache più volte al variare di j e k

IDEA:

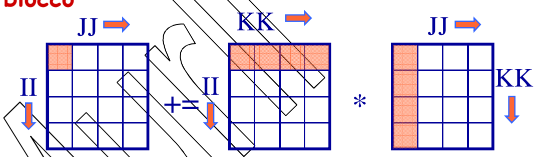
Mantenere la matrice M nella cache finché l'aggiornamento non è completato

Quale formulazione dell' algoritmo di Gauss consente ciò?

SGEMM: matrice -matrice a blocchi (BLAS 3)

Dim. blocco

```
for ii = 1,n,nb
    for jj = 1,n,nb
        for kk = 1,n,nb
            for k = kk,kk+nb-1
                for j = jj,jj+nb-1
                    for i = ii,ii+nb-1
                        c_i,j = c_i,j + a_i,k * b_k,j
                    endfor
        . . .
endfor
```



Il nucleo computazionale è ancora un prodotto matrice-matrice

Calcolo di q

- Preleva C → n^2 accessi
- Preleva A → n^2 accessi
- Preleva B → n^2 accessi
- Esegui C=C+AB → $2n^3$ operazioni fl. p.
- Riponi C → n^2 accessi

$$q_{sgemm} = \frac{4n^2}{2n^3} = \frac{2}{n} < \frac{3}{2n} + \frac{1}{2} \approx q_{gaxpy}$$

Ulteriore riduzione del traffico parassita

Calcolo di q

- Preleva C_{ij} $\Rightarrow$ nb^2 accessi
- Preleva A_{ik} (k = 1,..., p) $\Rightarrow$ nb^2 accessi
- Preleva B_{kj} (k = 1,..., p) $\Rightarrow$ $p(nb)^2$ accessi
- Esegui $C_{ij} = C_{ij} + A_{ik}B_{kj}$ $\Rightarrow$ $2p(nb)^3$ operazioni fl. p.
- Riponi C_{ij} $\Rightarrow$ nb^2 accessi

$$q_{sgemm} = \frac{3nb^2 + p(nb)^2}{2p(nb)^3} = \frac{p+3}{2p(nb)} = \frac{3}{2pnb} + \frac{1}{2nb} = \frac{2}{n} \quad \text{(caso migliore)}$$