Programmazione di Processori *MultiCore* – IV lezione

Federico Massaioli (**federico.massaioli@caspur.it**)

CASPUR e Università degli Studi di Roma "La Sapienza"
Laurea Magistrale in Informatica
Anno accademico 2008-2009

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*, chiamata 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 aspetto
- Può essere necessario inquadrare il problema in un contesto più ampio della singola *routine*

I registri

- I registri sono locazioni di memoria interne alla CPU
- Poche (tipicamente 40÷80), 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 ↔ registri sono fasi separate

- Il compilatore utilizza i registri:
  - per i valori intermedi durante il calcolo delle espressioni
    ma 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 elementari, non per array o struct!

Elementi di vettori…

```
for(x = 0; x < nx/2; ++x)
      for(y = 1; y < ny; ++y) {
           k2=eta[y];
           for(z = 0; z < nz; ++z) {
                k3=beta[z];
                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;
           }
      }
```

…e scalari di appoggio

```
for(x = 0; x < nx/2; ++x)
      for(y = 1; y < ny; ++y) {
           k2=eta[y];
           for(z = 0; z < nz; ++z) {
                k3=beta[z];
                hr1=hr[x][y][z][1]*norm;
                hi1=hi[x][y][z][1]*norm;
                hr2=hr[x][y][z][2]*norm;
                hi2=hi[x][y][z][2]*norm;
                hr3=hr[x][y][z][3]*norm;
                hi3=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*hr1+k2*hr2+k3*hr3;
                si=k1*hi1+k2*hi2+k3*hi3;
                hr[x][y][z][1]=hr1-sr*k1*k_quad;
                hr[x][y][z][2]=hr2-sr*k2*k_quad;
                hr[x][y][z][3]=hr3-sr*k3*k_quad;
                hi[x][y][z][1]=hi1-si*k1*k_quad;
                hi[x][y][z][2]=hi2-si*k2*k_quad;
                hi[x][y][z][3]=hi3-si*k3*k_quad;
                k_quad_cfr=0;
           }
      }
```

Tempo di esecuzione:
- 25 % ☺

CPU: parallelismo interno

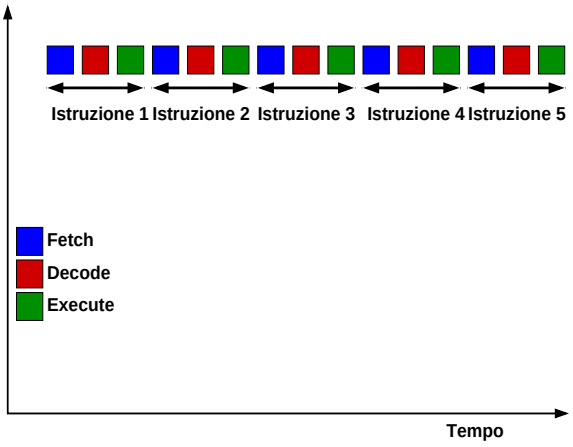
- Le CPU sono internamente parallele
  - *pipelining*
  - esecuzione superscalare
  - unità SIMD MMX, SSE, SSE2
- Per ottenere *performance* paragonabili con quelle “sbandierate” dal produttore:
  - fornire istruzioni in quantità
  - fornire gli operandi delle istruzioni

La *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 e per poter aumentare la frequenza di *clock* (limiti fisici di propagazione dei segnali nel *chip*)

Unità di calcolo NON *pipelined*

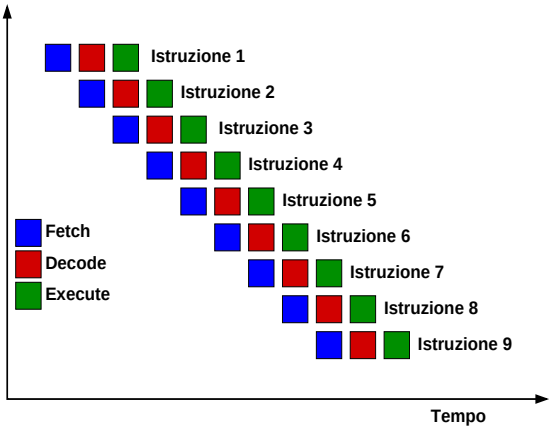
- Una istruzione completata ogni tre cicli
- Esempio:
  clock = 100 MHz
  MFlops = 33



Unità di calcolo *pipelined*

- Un risultato per ciclo a *pipeline* piena
- Per riempirla servono 3 istruzioni indipendenti, con gli operandi
- All'estremo opposto, un risultato ogni 3 cicli a *pipeline* "vuota"

- Esempio:
  clock = 100 MHz
  MFlops = 100

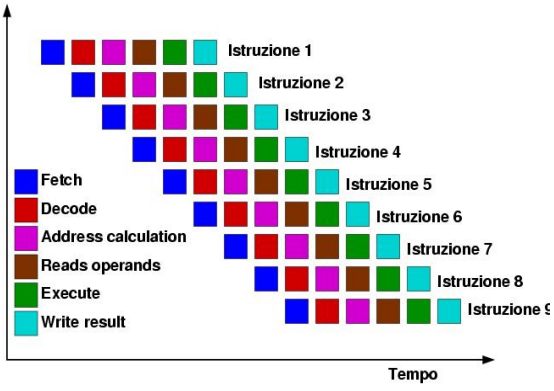


9

Unità di calcolo *superpipelined*

- Un risultato per ciclo a *pipeline* piena
- Per riempirla servono 6 istruzioni indipendenti, con gli operandi
- All'estremo opposto, un risultato ogni 6 cicli a *pipeline* "vuota"
- È possibile dimezzare il ciclo di *clock* (ossia raddoppiare la frequenza!!!)

- Esempio:
  clock = 200 MHz
  MFlops = 200



10

Pipeline e frequenza di *clock*

Profondità di *pipeline* e *clock* di alcuni processori (non recentissimi…)

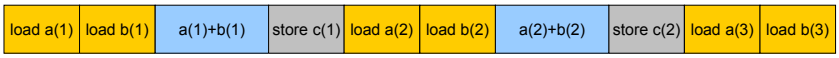
Motorola G4	867 Mhz	7 stage
Intel P4	2000 Mhz	20 stage
Intel P3	1000 Mhz	10 stage
AMD XP	1600 Mhz	14 stage
Power3	375 Mhz	10 stage
UltraSparcIII	900 Mhz	17 stage

L'esecuzione superscalare

- Le CPU contengono più unità indipendenti
  - differenziazione funzionale
  - replicazione funzionale
- Operazioni indipendenti sono eseguite in contemporanea
  - operazioni su interi
  - operazioni su *floating point*
  - calcolo di salti
  - accessi in memoria
- Parallelismo sulle istruzioni
- Mascheramento delle latenze
- I processori sfruttano intensivamente la superscalarità per aumentare la capacità di elaborazione a parità di *clock*

Perché tanta complicazione?

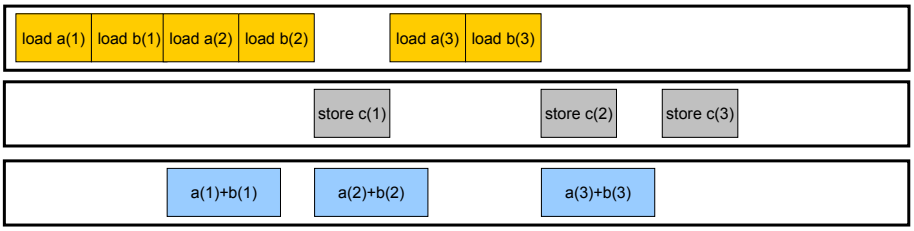
- Esempio c(i) = a(i) + b(i):
  - Load 4 cicli di clock
  - Store 4 cicli di clock
  - Somma: 6 cicli di clock



  - Per sommare 3 elementi del vettore sono necessari 50 cicli di clock

Perché conviene!

- Esempio c(i) = a(i) + b(i):
  - Load 4 cicli di clock
  - Store 4 cicli di clock
  - Somma: 6 cicli di clock



  - Per sommare 3 elementi del vettore sono necessari 38 cicli di clock

Come sfruttare il parallelismo interno?

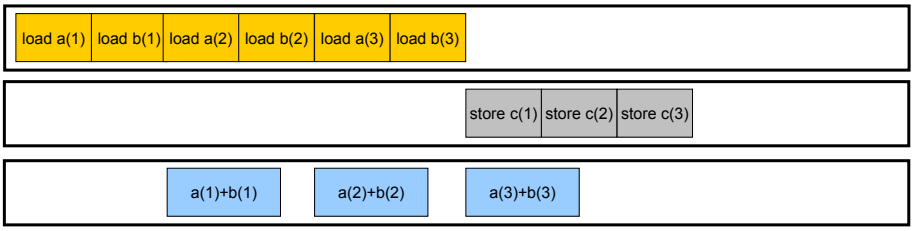
- I problemi principali sono:
  - come rimuovere le dipendenze tra le istruzioni?
  - come fornire abbastanza istruzioni indipendenti?
  - cosa fare in presenza di salti condizionali (*if* e *loop*)?
  - come fornire tutti i dati necessari?
- Chi deve modificare il codice?
  - la CPU? → sì, per quel che può, *OOO* & *branch prediction*
  - il compilatore? → sì, per quel che può, se lo evince dal codice
  - l'utente? → sì, nei casi più complessi
- Tecniche
  - *loop unrolling* → "srotolo" un *loop*
  - *loop merging* → fondo più *loop* insieme
  - *loop splitting* → decompongo *loop* complessi
  - *inlining* di funzioni → evito interruzioni del flusso di istruzioni

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
  - il tutto dipendentemente dalle unità funzionali "libere"
- Si appoggia intensivamente su:
  - *renaming* dei registri (registri fisici vs. architetturali)
  - *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

Complesso ma può convenire

- Esempio c(i) = a(i) + b(i):
  - Load 4 cicli di clock
  - Store 4 cicli di clock
  - Somma: 6 cicli di clock



  - Per sommare 3 elementi del vettore sono necessari 36 cicli di clock
  - In realtà si applica su centinaia di istruzioni

Un semplice *loop*

- Prodotto matrice-matrice:

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

- 2 istruzioni “utili” per iterazione
- Le 2 istruzioni sono dipendenti tra loro
- Dominato da:
  - incremento degli indici di *loop*
  - salti condizionali impliciti alla fine del *loop*
  - calcolo degli indirizzi

Loop unrolling

- Prodotto matrice-matrice con *unrolling* del *loop* esterno:

```
for(i =0; i< n; i+=2)
   for(k = 0; k < n; ++k)
     for(j = 0; j < n; ++j) {
          c[i+0][j] = c[i+0][j] + a[i+0][k]*b[k][j];
          c[i+1][j] = c[i+1][j] + a[i+1][k]*b[k][j];
     }
```

- "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 **`a[i+1][k]`** si calcola da quello di **`a[i+0][k]`**
  - idem per **`c`**
  - il "peso" di indici e salti dei *loop* è dimezzato

Effetto dell'*unrolling*

- Prodotto matrice-matrice
- Sistema IBM Power4@1100 Mhz
- Matrici 1024×1024, in doppia precisione

unrolling	MFlops
1	382
2	663
4	295
8	186

Prodotto di matrici: *unrolling&padding*

- Matrici 1024×1024, doppia precisione
- Macchina IBM Power4@1100 Mhz
- Prestazioni misurate in MFlops

unrolling	*padding* = 16	*padding* = 0
1	395	382
2	692	663
4	857	295
8	771	186
16	838	140
32	842	142

Come “frenare” il compilatore

- Il compilatore sa fare automaticamente l’*unrolling*
- Ma non lo applica se lo ritiene “pericoloso”!
- Esempio: *aliasing*

```
void accumulate(int n, double *a, double *s) {
  int i;

  for(i = 0; i < n; ++i)
    a[i] += s[i];
}
```

- Il compilatore non fa *unrolling*, nel timore di un possibile *aliasing* di **a** ed **s** in chiamate tipo:

```
accumulate(10, b+1, b); /*che succede con unrolling?*/
```

Come “convincere” il compilatore

- Dichiarando che non vi sarà *aliasing*:

```
void accumulate(int n,
      double * restrict a, double * restrict s) {
  int i;

  for(i = 0; i < n; ++i)
    a[i] += s[i];
}
```

- Il compilatore ora potrà fare *unrolling*, fiducioso che il programmatore non verrà meno alla parola data
- Ma chi lo facesse, verrà punito con risultati errati!!

Più informazioni ha il compilatore…

- Estremi noti a *compile time* e non:
  - `i1=j1=k1=1`
  - `i2=j2=k2=1024`

Opzione	Estremi da input	Estremi noti
-O3	3.8’’	3.8’’
-O4	2.1’’	1.3’’
-O5	1.9’’	1.3’’

- Attenzione: molto dipendente dal compilatore!!!

Loop fusion

- Dipendenze tra iterazioni successive impediscono l'*unrolling*:

```
for(i = 1; i < n; ++i)
   a[i] = a[i-1] + 1.0;
for(i = 1; i < n; ++i)
   b[i] = b[i-1]*2.0;
```

- Un unico *loop* ha più istruzioni indipendenti per iterazione:

```
for(i = 1; i < n; ++i) {
  a[i] = a[i-1] + 1.0;
  b[i] = b[i-1]*2.0:
}
```

Spesso il compilatore riesce a farlo automaticamente

Loop splitting

- Il "corpo" del *loop* può essere troppo grande
  - molte istruzioni e variabili di appoggio temporaneo
  - poche istruzioni ma espressioni lunghe e complicate
- In questi casi il compilatore può:
  - rinunciare
  - tentare l'*unrolling* e peggiorare le cose
- *Register spilling*
  - le variabili temporanee vengono allocate in registri della CPU
  - se sono troppe, si rende necessario memorizzarne il contenuto
  - quando il valore serve, va ricaricato dalla memoria (lento)
  - in attesa del valore, si crea una "bolla" nella *pipeline*
- Dividendo il *loop* in due o più *loop* separati si può guadagnare velocità

Salti condizionali (1)

- I salti condizionali (*loop*, *if*) interrompono il flusso regolare delle *pipeline* finché il valore della condizione non è nota
- Esecuzione speculativa
  - la *Branch Prediction Unit* accumula statistica sulle più recenti istruzioni di salto condizionato, e "predice" cosa avverrà la prossima volta
  - predizione sbagliata → ripristinare lo stato precedente al salto
  - ☺ vantaggiosa per pattern ben definiti (*loop*, *if* di gestione errori, …)
  - ☹ inutile quando la condizione è casuale
- Esecuzione predicativa degli *if-then-else* (GPU, Itanium)
  - sia il *then* che l'*else* vengono eseguiti
  - quando la condizione è nota, si scelgono gli effetti dell'uno o dell'altro
  - ☺ il tempo di esecuzione è costante, le *pipeline* meglio utilizzate
  - ☹ onerosa per *then* e *else* "corposi"

Salti condizionali (2)

```
// 0.0 <= soglia <= 1.0,  a[i] random
....
     for(j = 0; j < repliche; ++j)
        for(i = 0; i < nd; ++i)
           if(a[i] >= soglia)
              somma1 = somma1 + a[i]*a[i];
           else
              somma2 = somma2 – b[i]*b[i];

...
```

Esecuzione speculativa e predicativa

- Esecuzione speculativa: Intel Pentium D @ 3.2 GHz
- Esecuzione predicativa: Intel Itanium2 @ 1.5 GHz

Soglia	Pentium D	Itanium2
0.0	0.6''	0.5''
0.1	0.9''	0.5''
0.25	1.2''	0.5''
0.5	1.4''	0.5''
0.75	1.1''	0.5''
0.9	0.8''	0.5''
1.0	0.6''	0.5''

Branch prediction

- Ordinando i dati l'esecuzione speculativa non presenta più la dipendenza dal valore di soglia

***Branch prediction* inefficiente**

Soglia	Senza *sorting*	Con *sorting*
0	0.6''	0.6''
0.1	0.9''	0.6''
0.25	1.2''	0.6''
0.5	1.4''	0.6''
0.75	1.1''	0.6''
0.9	0.8''	0.6''
1	0.6''	0.6''

Inlining di funzioni

- Le chiamate a funzione “costano” perché la CPU deve saltare ad eseguire un’altra porzione del codice, e bisogna passare i parametri alla *subroutine*
- Si crea una “bolla” nella *pipeline*
- *Inlining* in un codice cinetico
  - tramite opzione di compilatore (**`-inline…`**)
  - esplicito (“a mano”)
- Effetti, a livello di ottimizzazione medio/alto:

Macchina	chiamate a funzione	*inlining* dal compilatore	*inlining* “a mano”
Power 4	2911	2620	2032”
Alpha EV68	1838	-	1869”
Itanium2	3456	904	831”

“Giocando” col compilatore…

- Prodotto matrice/matrice, Intel Pentium D930 3.2 GHz, **`ifort`** 9.0
- Prestazioni in MFlops al variare delle opzioni di compilazione

	Singola prec.	Doppia Prec.	Rapporto
`-O0`	292	269	1.08
`-O1`	799	736	1.08
`-O2`	796	737	1.08
`-xP`	2115	1072	1.96

- Non male! Ma perche????

Indaghiamo…

- Dal manuale di **`ifort`**:
  - **`-xP`**: *Can generate SSE3, SSE2, and SSE instructions for Intel processors, and it can optimize for processors based on Intel Core microarchitecture and Intel NetBurst microarchitecture, like Intel, Core Duo processors, Pentium 4 processors with SSE3, and Intel Xeon processors with SSE3*.
- Compilando con l'opzione **`-xP`** appare il seguente *warning*:
  - … **`ifort -r8 -i4 -xP -c mm.f`**
    `mm.f(68):(col. 18) remark: LOOP WAS VECTORIZED....`
  - non appare compilando con l'opzione `-O3`

Approfondiamo…

Guardando l'assembler:

- Livello **`-O0`**: 26 istruzioni
  - ✓ 1 istruzione di somma **`faddp`**
  - ✓ 1 istruzione di prodotto **`fmulp`**
- Livello **`-O1`**: 5 istruzioni
  - ✓ 1 istruzione di somma **`addss`**
  - ✓ 1 istruzione di prodotto **`mulss`**
- Livello **`-xP`**: 24 istruzioni
  - ✓ 4 istruzioni di somma **`addps`**
  - ✓ 4 istruzioni di prodotto **`mulps`**

Che istruzioni sono?

- Le istruzioni **`faddp`** e **`fmulp`** sono istruzioni di somma e prodotto
- Le istruzioni **`addss`** e **`mulss`** sono istruzioni SSE3 di somma e prodotto:
  - **istruzioni seriali**: vengono usati due registri a 128 bit per eseguire la somma o il prodotto di una coppia di valori (i restanti 64/96 bit dei registri non vengono sfruttati).
- Le istruzioni **`addps`** e **`mulps`** sono istruzioni SSE3 di somma e prodotto:
  - **istruzioni parallele**: vengono usati due registri a 128 bit per eseguire in parallelo le somme o i prodotti di 2 coppie di valori in doppia precisione (**`addpd/mulpd`**) o di 4 coppie in singola precisione (**`addps/mulps`**).

Efficienza

- Livello **`-O0`**: 26 istruzioni
  - ✓ 1 **`faddp`** + 1 **`fmulp`**
  - ✓ rapporto operazioni/istruzioni: 2/26
- Livello **`-O1`**: 5 istruzioni
  - ✓ 1 **`addss`** + 1 **`mulss`**
  - ✓ rapporto operazioni/istruzioni: 2/5
- Livello **`-xP`**: 24 istruzioni
  - ✓ 4 **`addps`** + 4 **`mulps`**
  - ✓ rapporto operazioni/istruzioni in doppia: 16/24
  - ✓ rapporto operzioni/istruzioni in singola: 32/24

Vettorizzazione

- Operazioni tra vettori:

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

- Le istruzioni “utili” in questo *loop* sono:
  - 2*n* istruzioni di load
  - *n* istruzioni di store
  - *n* operazioni di somma
- Però:
  - ✓ i dati sono contigui
  - ✓ le istruzioni sono le stesse su dati differenti
- Vettorizzare corrisponde a fare *unrolling* di più iterazioni in *hardware*

Istruzioni vettoriali

- In una architettura vettoriale si agisce a livello di vettori e non più a livello di singole grandezze scalari.
- Il corpo del *loop* in esame si traduce in solo **quattro** istruzioni.
  - 2 istruzioni di *load* (**`vload`**) in cui si dice di caricare dalla memoria *n* elementi contigui di due vettori (**a** e **b**)
  - 1 istruzione di somma (**`vadd`**) tra coppie di elementi dei due vettori
  - 1 istruzione di *store* (**`vstore`**) per scrivere in memoria i risultati nel vettore **c**.
- Le istruzioni macchina da eseguire si riducono notevolmente (di un fattore *n*), ma non le operazioni matematiche
  - ✓ posso utilizzare efficientemente le *pipeline*
  - ✓ posso aumentare le unità funzionali ed usarle efficientemente

Registri vettoriali

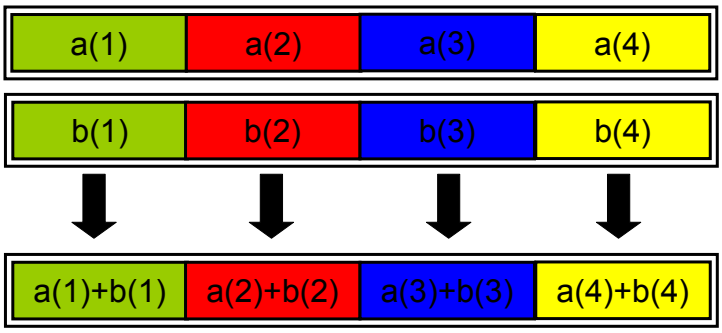
- Ovviamente il vantaggio si ha se i registri associati alle unità di calcolo sono “vettoriali”
  - ogni registro contiene *n* valori scalari
  - le operazioni coinvolgono tutti gli elementi contenuti nei registri
  - ci sono modi “veloci” per trasferimenti registri↔RAM, anche saltando la *cache*
- NEC SX-6:
  - registri vettoriali di 256 elementi (doppia o singola precisione)
- Intel Core:
  - registri SSE3: registri “larghi” 128 bit
  - si possono “impacchettare” 4 numeri in singola precisione
  - si possono “impacchettare” 2 numeri in doppia precisione

Unità Intel SSE*n*

- Le architetture Intel presentano
  - 16 registri a 128 bit
  - Unita logiche a 128 bit (SSE3)
- Le unità logiche possono fare:
  - 4 operazioni “impacchettate” floating point singola precisione
  - 2 operazioni “impacchettate” floating point doppia precisione
  - 16 operazioni “impacchettate” tra interi a 8 bit
  - 8 operazioni “impacchettate” tra interi a 16 bit
  - 4 operazioni “impacchettate” tra interi a 32 bit
  - 2 operazioni “impacchettate” tra interi a 64 bit
  - 1 operazioni “impacchettate” tra interi a 128 bit
- “*The software optimization cookbook*”, Intel Press 2006

SIMD

- Single Istruction Multiple Data



41

Loop vettorizzabili e non

Come per l'*unrolling*, non tutti i *loop* sono vettorizzabili

- Loop vettorizzabile:

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

- Loop non vettorizzabile: dipendenza in avanti

```
do i = 1, n
     c(i) = a(i) + c(i+1)
enddo
```

42

Possibili limitazioni

La vettorizzazione permette un aumento delle prestazioni su Intel

- x2 in singola precisione
- vantaggi minimi in doppia precisione
- attenzione all'accesso ai dati
  - ✓ allineamento
- possibili contrasti con
  - ✓ *padding*
  - ✓ *unrolling*
  - ✓ *blocking*
- attenzione al supporto disponibile nel processore ed a conflitti tra ottimizzazioni

Vettorizzazione & compilatore

- **`intel icc`**
- Prodotto matrice matrice, 2048*2048, singola precisione
- Intel Pentium D, 3.2 Ghz
- Prestazioni in Mflops
  - ✓ **`-O3`** 856
  - ✓ **`-xP`** 2144
  - ✓ **`-O3 -xP`** 2618
  - ✓ **`-01 -xP`** 867
  - ✓ **`-fast`** errore!!!!
  - ✓ **`-O3 -xP -ipo -static`** 2872

Vettorizzazione & *padding*

Il *padding* ha effetti molto rilevanti (dati in MFlops)

- 0 → 2872
- 1 → 2834
- 2 → 2659
- 3 → 1692
- 4 → 1711
- 5 → 1771
- 6 → 3019
- 7 → 1776
- 8 → 3851
- 9 → 2045
- 10 → 3008
- 11 → 1742
- 12 → 3878
- 13 → 1742
- 14 → 3008
- 15 → 1751
- **16** → **4004**

Vettorizzazione & unrolling

Interazione tra vettorizzazione e *unrolling* manuale

	Opz.	Unroll	Pad.	Mflops	
✓	**-O3**	8	0	532	
✓	**-O3**	8	8	1081	
✓	**-O3**	4	8	1142	
✓	**-O3**	2	8	932	
✓	**-xP**	4	8	3422	
✓	**-xP**	4	0	3491	
✓	**-xP**, …	4	8	3542	
✓	**-xP**, …	4	16	3931	(un loop vettorizzati)
✓	**-xP**, …	**2**	**16**	**4231**	**(tre loop vettorizzati)**
✓	**-xP**, …	2	17	1580	(tre loop vettorizzati)

Mettendo tutto insieme…

- Punto di partenza (codice naif): MFlops
  - ✓ Senza vettorizzazione 800
  - ✓ Con vettorizzazione 2800

- Unrolling loop esterno di 2
- Blocking di 64
- Padding 16
  - – Opzioni MFlops
  - ✓ **-xP** … 4655
  - ✓ **-xP** … (J block*2) 4824
  - ✓ **-xP** … (J block*4) 5436
  - ✓ **-O3** 1540

47

Quando il compilatore non vettorizza

- Compilando il codice abbiamo i seguenti *warning*:

```
federico@xios:~/make "DBL = " "OPT  = -O3 -xP -vec-report5"
col.f(60): (col. 9) remark: loop was not vectorized: not
    inner loop.
col.f(61): (col. 12) remark: loop was not vectorized:
    existence of vector dependence.
col.f(122): (col. 15) remark: vector dependence: proven FLOW
    dependence between a05 line 122, and a05 line 64.
```

- "Giochi complicati" di indici inibiscono la vettorizzazione
- Si può forzare la vettorizzazione con una direttiva di compilazione
  - **fortran:!DEC$ IVDEP**
  - **C: #pragma IVDEP**

48

Direttive di compilazione (1)

```
60          do k = 1,n
61     !DEC$ IVDEP
61             do i = 1,l
62                x02 = a02(i-1,k+1,inow)
63                x04 = a04(i-1,k-1,inow)
64                x05 = a05(i-1,k  ,inow)
65                x06 = a06(i  ,k-1,inow)
66                x11 = a11(i+1,k+1,inow)
67                x13 = a13(i+1,k-1,inow)
68                x14 = a14(i+1,k  ,inow)
69                x15 = a15(i  ,k+1,inow)
70                x19 = a19(i  ,k  ,inow)
71
72                rho =+x02+x04+x05+x06+x11+x13+x14+x15+x19
...
122               a05(i,k,inew) = x05 - omega*(x05-e05) + force
123               a06(i,k,inew) = x06 - omega*(x06-e06)
```

inow ≠ inew, la dipendenza è solo apparente

Direttive di compilazione (2)

- Caso test 1000*200
- Tempo routine di collisione
- Singola precisione
  - ✓ **-03** 21''
  - ✓ **-xP** 19'' (x1.1)
  - ✓ **-xP** + direttive 8.4'' (x2.5)

Riassumendo…

- È possibile ottenere *performance* paragonabili a quelle “di targa” del sistema
- Ed è possibile far fare buona parte del lavoro al compilatore…
- … se si scrive il codice in maniera che il compilatore capisca cosa fare
- E soprattutto, se il *layout* dei dati in memoria e l’ordine di accesso agli stessi non è in contrasto con le assunzioni fatte da chi ha progettato il tutto

Il compilatore: cosa sa fare

- Esegue trasformazioni del codice come:
  - *Register allocation*
  - *Register spilling*
  - *Copy propagation*
  - *Code motion*
  - *Dead and redundant code removal*
  - *Common subexpression elimination*
  - *Strength reduction*
  - *Inlining*
  - *Index reordering*
  - *Loop pipelining/unrolling, merging*
  - *Cache blocking*
  - …..
- Lo scopo è massimizzare le prestazioni

Il compilatore: cosa non sa fare

- Analizzare ed ottimizzare globalmente codici molto grossi (a meno di abilitare l'IPO, molto costoso in tempo e risorse)

- Capire dipendenze tra dati con indirizzamenti indiretti
- *Strength reduction* di potenze non intere, o maggiori di 2÷4
- *Common subexpression elimination* attraverso chiamate a funzione

- *Unrolling/Merging/Blocking* con:
  - chiamate a funzioni/*subroutines*
  - chiamate o *statement* di *I/O* in mezzo al codice
- Fare *inlining* di funzioni se non viene detto esplicitamente
- Sapere a *run-time* i valori delle variabili per i quali alcune ottimizzazioni sono inibite

Evoluzione dei compilatori

- I compilatori evolvono, ogni *release*:
  - cura vecchi *bug*
  - introduce nuovi *bug*
  - migliora/peggiora le prestazioni
  - cambia comportamento…
- Compilatore Intel per Itanium2@1.5 Ghz, tempi in secondi

Codice	Rel. 7.1	Rel 9.0
Bgk – *original*	1052	850
Bgk - *oneMatrix*	7812	2000
Bgk - *fused*	717	367
Bgk - *bundle*	844	497
Bgk - *vect*	504	445

- Verificare sempre correttezza dei risultati e prestazioni!!!!!