Programmazione di Processori *MultiCore* – II 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 CPU

- È composta di:
  - registri (operandi delle istruzioni)
  - unità funzionali (eseguono le istruzioni)
- Unità funzionali:
  - aritmetica intera
  - operazioni logiche *bitwise*
  - aritmetica *floating-point*
  - calcolo di indirizzi
  - lettura e scrittura in memoria (*load & store*)
  - previsione ed esecuzione di "salti" (*branch*) nel flusso di esecuzione

Le CPU

- RISC: *Reduced Instruction Set CPU*
  - istruzioni semplici
  - formato regolare delle istruzioni
  - decodifica ed esecuzione delle istruzioni semplificata
  - codice macchina molto “verboso”
- CISC: *Complex Instruction Set CPU*
  - istruzioni di semantica “ricca”
  - formato irregolare delle istruzioni
  - decodifica ed esecuzione delle istruzioni complicata
  - codice macchina molto “compatto”
- Differenza non più rilevante quanto a prestazioni: le CPU CISC di oggi convertono le istruzioni in micro operazioni RISC-*like*

3

Architettura *vs.* implementazione

- Architettura:
  - set di istruzioni
  - registri architetturali interi, *floating point* e di stato
- Implementazione
  - registri fisici (2.5÷20 × registri architetturali)
  - frequenza di *clock* e tempo di esecuzione delle istruzioni
  - numero di unità funzionali
  - dimensione, numero, caratteristiche delle *cache*
  - *Out Of Order execution, Simultaneous Multi-Threading*
- Una architettura, più implementazioni:
  - Power: Power3, Power4, Power5, …
  - x86: Pentium III, Pentium 4, Xeon, Pentium M, Pentium D, Athlon, Opteron, …

4

/proc/cpuinfo

```
<amatig@egina ~>cat /proc/cpuinfo
processor       : 1
vendor_id       : GenuineIntel
cpu family      : 15
model           : 2
model name      : Intel(R) Xeon(TM) CPU 2.80GHz
stepping        : 7
cpu MHz         : 2799.284
cache size      : 512 KB
physical id     : 3
siblings        : 1
runqueue        : 1
fdiv_bug        : no
hlt_bug         : no
f00f_bug        : no
coma_bug        : no
fpu             : yes
fpu_exception   : yes
cpuid level     : 2
wp              : yes
flags           : fpu vme de pse tsc msr …
bogomips        : 5596.77
```

Implementazioni di una CPU: impatto

- Prestazioni differenti
- “Regole” diverse per ottenere alte prestazioni
- Stato esplicito e stato “nascosto”
  - Io stato dei registri architetturali è esplicito
  - Io stato dei registri fisici e delle *cache* è nascosto
- Lo stato nascosto:
  - è per lo più “trasparente” quanto alla semantica
  - può essere visibilissimo quanto alle prestazioni

Il compilatore

- Traduce il codice sorgente in codice macchina
- Rifiuta codici sintatticamente errati
- Segnala (alcuni) potenziali problemi semantici
- Può tentare di "ottimizzare" il codice
  - ottimizzazioni indipendenti dal linguaggio
  - ottimizzazioni dipendenti dal linguaggio
  - otttimizzazioni dipendenti dalla CPU
  - ottimizzazioni dipendenti dall'implementazione della CPU
  - ottimizzazioni dell'uso della memoria e della *cache*
  - suggerimenti al processore su cosa probabilmente farà il codice
- È uno strumento
  - potente: può risparmiare lavoro al programmatore
  - complesso: a volte può fare cose sorprendenti o controproducenti
  - limitato: è un "sistema esperto", ma non ha l'intelligenza di un essere umano, non può capire pienamente il codice

Livelli di ottimizzazione

Il compilatore Fortran IBM presenta questi livelli incrementali di ottimizzazione:

- **`-O0:`** nessuna ottimizzazione, il codice è tradotto letteralmente
- **`-O2,-O:`** ottimizzazioni locali, compromesso tra velocità di compilazione, ottimizzazione e dimensioni dell'eseguibile (livello di *default*)
- **`-O3:`**ottimizzazioni *memory-intensive*, può alterare la semantica del programma
- **`-O4:`** ottimizzazione aggressive **`(-qarch=auto, -qtune=auto, -qipa, -qhot`**)
- **`-O5:`** come **`-O4`** ma con –**`qipa=level2`** (molto aggressiva e lenta)

Algoritmi e compilatore (TSP)

IBM Power4@1100 Mhz, ottimizzazione `-O2`

	9	10	11	12	13	14	15	16
1	0.92	10.28	123.48	1583	-	-	-	-
2	0.10	1.03	11.22	134.07	-	-	-	-
3	-	0.46	4.59	50.50	605.99	-	-	-
4	-	-	0.29	1.50	11.29	98.73	-	-
5	-	-	0.11	0.57	4.29	37.56	287.83	-

IBM Power4@1100 Mhz, ottimizzazione `-O5`

	9	10	11	12	13	14	15	16
1	0.11	1.22	14.58	192.96	-	-	-	-
2	0.01	0.13	1.33	15.98	185.44	-	-	-
3	-	0.05	0.57	6.20	74.42	972.33	-	-
4	-	-	0.06	0.29	2.25	19.15	143.44	995.04
5	-	-	-	0.15	1.15	9.95	72.93	501.01

C'è compilatore e compilatore…

- Su alcune architetture esiste un solo compilatore
- Su altre c'è possibilità di scelta

- Quale è il migliore compilatore?
- Ha senso questa domanda?
  - contano solo le prestazioni?
  - o ci sono altre misure di "qualità"?

- Prodotto matrice-matrice 512×512 su Intel P4 @2800 Mhz
  - Intel `ifc` rel. 7.1
  - PGI `pgf77` rel. 4.1-2
  - GNU `f77` rel. 3-2

Compilatori a confronto: MFlops

Ottimizz.	Padding	unroll	Blocking	Ifc	g77	pgf77
Default	0	0/0	0	459	239	324
-O3	0	0/0	0	460	459	419
Default	9	0/0	128	1112	200	320
-O3	9	0/0	128	1101	996	704
Default	9	4/4/4	128	1431	747	1206
-O1	9	4/4/4	128	1432	1582	427
-O3	9	4/4/4	128	1431	1141	1234
-O1 –unroll	9	4/4/4	128	-	1712	-

Dal sorgente all’eseguibile

- La macchina astratta “vista” a livello sorgente è molto diversa da quella reale
- Esempio: prodotto di matrici

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

- Il nocciolo: carica dalla memoria tre valori, fa una moltiplicazione ed una somma, scrive il risultato

Prodotto di matrici: *performance*

- Tempo totale in secondi per prodotto matrice-matrice
  - Macchina IBM Power4@1100 Mhz
  - Matrici 1024×1024, doppia precisione

Opzione	secondi	MFlops
`-O0`	24''	89
`-O2`	6.35''	338
`-O3`	4.87''	487
`-O4`	2.14''	1003
`-O5`	1.93''	1111

- Perché tanta varietà di risultati?
- Basta passare da **`-On`** a **`-On+1`**?

Prodotto di matrici: codice macchina

Matrix Multiply inner loop code with -qnoopt

38 instructions, 31.4 cycles per iteration

```
__L1:
   lwz    r3,160(SP)
   lwz    r9,STATIC_BSS
   lwz    r4,24(r9)
   subfi  r5,r4,-8
   lwz    r11,40(r9)
   mullw  r6,r4,r11
   lwz    r4,36(r9)
   rlwinm r4,r4,3,0,28
   add    r7,r5,r6
   add    r7,r4,r7
   lfdx   fp1,r3,r7
   lwz    r7,152(SP)
   lwz    r12,0(r9)
   subfi  r10,r12,-8
   lwz    r8,44(r9)
   mullw  r12,r12,r8
   add    r10,r10,r12
   add    r10,r4,r10
   lfdx   fp2,r7,r10
   lwz    r7,156(SP)
   lwz    r10,12(r9)
   subfi  r9,r10,-8
   mullw  r10,r10,r11
   rlwinm r8,r8,3,0,28
   add    r9,r9,r10
   add    r8,r8,r9
   lfdx   fp3,r7,r8
   fmadd  fp1,fp2,fp3,fp1
   add    r5,r5,r6
   add    r4,r4,r5
   stfdx  fp1,r3,r4
   lwz    r4,STATIC_BSS
   lwz    r3,44(r4)
   addi   r3,1(r3)
   stw    r3,44(r4)
   lwz    r3,112(SP)
   addic. r3,r3,-1
   stw    r3,112(SP)
   bgt    __L1
```

Courtesy of IBM ©

Prodotto di matrici: l’essenziale

Matrix Multiply inner loop code with -qnoopt

necessary instructions

```
__L1:
    lwz     r3,160(SP)
    lwz     r9,STATIC_BSS
    lwz     r4,24(r9)
    subfi   r5,r4,-8
    lwz     r11,40(r9)
    mullw   r6,r4,r11
    lwz     r4,36(r9)
    rlwinm  r4,r4,3,0,28
    add     r7,r5,r6
    add     r7,r4,r7
    lfdx    fp1,r3,r7
    lwz     r7,152(SP)
    lwz     r12,0(r9)
    subfi   r10,r12,-8
    lwz     r8,44(r9)
    mullw   r12,r12,r8
    add     r10,r10,r12
    add     r10,r4,r10
    lfdx    fp2,r7,r10
    lwz     r7,156(SP)
    lwz     r10,12(r9)
    subfi   r9,r10,-8
    mullw   r10,r10,r11
    rlwinm  r8,r8,3,0,28
    add     r9,r9,r10
    add     r8,r8,r9
    lfdx    fp3,r7,r8
    fmadd   fp1,fp2,fp3,fp1
    add     r5,r5,r6
    add     r4,r4,r5
    stfdx   fp1,r3,r4
    lwz     r4,STATIC_BSS
    lwz     r3,44(r4)
    addi    r3,1(r3)
    stw     r3,44(r4)
    lwz     r3,112(SP)
    addic.  r3,r3,-1
    stw     r3,112(SP)
    bgt     __L1
```

Courtesy of IBM ©

Prodotto di matrici: *loop control*

Matrix Multiply inner loop code with -qnoopt

necessary instructions   loop control

```
__L1:
    lwz     r3,160(SP)
    lwz     r9,STATIC_BSS
    lwz     r4,24(r9)
    subfi   r5,r4,-8
    lwz     r11,40(r9)
    mullw   r6,r4,r11
    lwz     r4,36(r9)
    rlwinm  r4,r4,3,0,28
    add     r7,r5,r6
    add     r7,r4,r7
    lfdx    fp1,r3,r7
    lwz     r7,152(SP)
    lwz     r12,0(r9)
    subfi   r10,r12,-8
    lwz     r8,44(r9)
    mullw   r12,r12,r8
    add     r10,r10,r12
    add     r10,r4,r10
    lfdx    fp2,r7,r10
    lwz     r7,156(SP)
    lwz     r10,12(r9)
    subfi   r9,r10,-8
    mullw   r10,r10,r11
    rlwinm  r8,r8,3,0,28
    add     r9,r9,r10
    add     r8,r8,r9
    lfdx    fp3,r7,r8
    fmadd   fp1,fp2,fp3,fp1
    add     r5,r5,r6
    add     r4,r4,r5
    stfdx   fp1,r3,r4
    lwz     r4,STATIC_BSS
    lwz     r3,44(r4)
    addi    r3,1(r3)
    stw     r3,44(r4)
    lwz     r3,112(SP)
    addic.  r3,r3,-1
    stw     r3,112(SP)
    bgt     __L1
```

Courtesy of IBM ©

Prodotto di matrici: l'indirizzamento!!

Matrix Multiply inner loop code with -qnoopt

necessary instructions loop control addressing code

```
__L1:                          lwz    r7,156(SP)
  lwz    r3,160(SP)            lwz    r10,12(r9)
  lwz    r9,STATIC_BSS         subfi  r9,r10,-8
  lwz    r4,24(r9)             mullw  r10,r10,r11
  subfi  r5,r4,-8              rlwinm r8,r8,3,0,28
  lwz    r11,40(r9)            add    r9,r9,r10
  mullw  r6,r4,r11             add    r8,r8,r9
  lwz    r4,36(r9)             lfdx   fp3,r7,r8
  rlwinm r4,r4,3,0,28          fmadd  fp1,fp2,fp3,fp1
  add    r7,r5,r6              add    r5,r5,r6
  add    r7,r4,r7              add    r4,r4,r5
  lfdx   fp1,r3,r7             stfdx  fp1,r3,r4
  lwz    r7,152(SP)            lwz    r4,STATIC_BSS
  lwz    r12,0(r9)             lwz    r3,44(r4)
  subfi  r10,r12,-8            addi   r3,1(r3)
  lwz    r8,44(r9)             stw    r3,44(r4)
  mullw  r12,r12,r8            lwz    r3,112(SP)
  add    r10,r10,r12           addic. r3,r3,-1
  add    r10,r4,r10            stw    r3,112(SP)
  lfdx   fp2,r7,r10            bgt    __L1
```

Courtesy of IBM ©

Prodotto di matrici: cosa fare?

- Le operazioni dominanti sono quelle di conversione indici → indirizzo di memoria
- Osservazioni:
  - il *loop* "percorre" la memoria sequenzialmente
  - gli indirizzi degli elementi successivi sono calcolabili facilmente sommando una costante
  - sfruttare una conversione indice → indirizzo per più elementi successivi
- Può essere fatto automaticamente?

Prodotto di matrici: sì, ottimizziamo

Matrix Multiply inner loop code with -O3 -qtune=pwr4

```
__L1:
    fmadd   fp6,fp12,fp13,fp6
    lfdux   fp12,r12,r7
    lfd     fp13,8(r11)
    fmadd   fp7,fp8,fp9,fp7
    lfdux   fp8,r12,r7
    lfd     fp9,16(r11)
    lfdux   fp10,r12,r7
    lfd     fp11,24(r11)
    fmadd   fp1,fp12,fp13,fp1
    lfdux   fp12,r12,r7
    lfd     fp13,32(r11)
    fmadd   fp0,fp8,fp9,fp0
    lfdux   fp8,r12,r7
    lfd     fp9,40(r11)
    fmadd   fp2,fp10,fp11,fp2
    lfdux   fp10,r12,r7
    lfd     fp11,48(r11)
    fmadd   fp4,fp12,fp13,fp4
    lfdux   fp12,r12,r7
    lfd     fp13,56(r11)
    fmadd   fp3,fp8,fp9,fp3
    lfdux   fp8,r12,r7
    lfdu    fp9,64(r11)
    fmadd   fp5,fp10,fp11,fp5
    bdnz    __L1
```

unrolled by 8

dot product accumulated in
8 interleaved parts (fp0-fp7)
(combined after loop)

3 instructions, 1.6 cycles per iteration
2 loads and 1 fmadd per iteration

Courtesy of IBM ©

Ottimizziamo di più!

Matrix multiply inner loop code with -O3 -qhot -qtune=pwr4

```
__L1:
    fmadd   fp1,fp4,fp2,fp1
    fmadd   fp0,fp3,fp5,fp0
    lfdux   fp2,r29,r9
    lfdu    fp4,32(r30)
    fmadd   fp10,fp7,fp28,fp10
    fmadd   fp7,fp9,fp7,fp8
    lfdux   fp26,r27,r9
    lfd     fp25,8(r29)
    fmadd   fp31,fp30,fp27,fp31
    fmadd   fp6,fp11,fp30,fp6
    lfd     fp5,8(r27)
    lfd     fp8,16(r28)
    fmadd   fp30,fp4,fp28,fp29
    fmadd   fp12,fp13,fp11,fp12
    lfd     fp3,8(r30)
    lfd     fp11,8(r28)
    fmadd   fp1,fp4,fp9,fp1
    fmadd   fp0,fp13,fp27,fp0
    lfd     fp4,16(r30)
    lfd     fp13,24(r30)
    fmadd   fp10,fp8,fp25,fp10
    fmadd   fp8,fp2,fp8,fp7
    lfdux   fp9,r29,r9
    lfdu    fp7,32(r28)
    fmadd   fp31,fp11,fp5,fp31
    fmadd   fp6,fp26,fp11,fp6
    lfdux   fp11,r27,r9
    lfd     fp28,8(r29)
    fmadd   fp12,fp3,fp26,fp12
    fmadd   fp29,fp4,fp25,fp30
    lfd     fp30,-8(r28)
    lfd     fp27,8(r27)
    bdnz      L1
```

unroll-and-jam 2x2
inner unroll by 4
interchange "i" and "j" loops

2 instructions, 1.0 cycles per
iteration
balanced: 1 load and 1 fmadd
per iteration

Courtesy of IBM ©

Ottimizzazione e istruzioni macchina

- Istruzioni per `c[i][j] = c[i][j] + a[i][k]*b[k][j];`

- -O0: 24 istruzioni
  - 3 load/1 store
  - 1 floating point multiply+add → Flop/istruzione 1/12
- -O2: 9 istruzioni (riuso calcolo indirizzi)
  - 4 load/1 store
  - 2 floating point multiply+add → Flop/istruzione 4/9
- -O3: 150 istruzioni (*unrolling*)
  - 68 load/ 34 store
  - 48 floating point multiply+add → Flop/istruzione 48/75
- -O4: 344 istruzioni (*unrolling&blocking*)
  - 139 load / 74 store
  - 100 floating point multiply+add → Flop/istruzione 100/172

Quanto "costa" ottimizzare a mano?

- È molto faticoso?
- Quanto conta il *coding* nella velocità di esecuzione?
- Confrontiamo la versione più naturale:

```
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];
```

con …

Ottimizzazione manuale (1)

```
for (ii = 0; ii < nn; ii= ii+step)
  for (kk = 0; kk < nn; kk = kk+step)
    for (jj = 0; jj < nn; jj = jj+step)
      for ( i = ii; i < ii+step; i+=4 )
        for ( k = kk; k < kk+step; k+=4 ) {
          double b00 = b[k+0][j+0]; // 16 scalari di appoggio
          double b01 = b[k+0][j+1];
          double b02 = b[k+0][j+2];
          double b03 = b[k+0][j+3];
          double b10 = b[k+1][j+0];
          double b11 = b[k+1][j+1];
          double b12 = b[k+1][j+2];
          double b13 = b[k+1][j+3];
          double b20 = b[k+2][j+0];
          double b21 = b[k+2][j+1];
          double b22 = b[k+2][j+2];
          double b23 = b[k+2][j+3];
          double b30 = b[k+3][j+0];
          double b31 = b[k+3][j+1];
          double b32 = b[k+3][j+2];
          double b33 = b[k+3][j+3];
```

(continua ...)

Ottimizzazione manuale (2)

(... segue ...)

```
for(i = ii;i <ii+step; i+=4) {
  double a00 = a[i+0][k+0]; // altri 16 scalari di appoggio
  double a01 = a[i+0][k+1];
  double a02 = a[i+0][k+2];
  double a03 = a[i+0][k+3];
  double a10 = a[i+1][k+0];
  double a11 = a[i+1][k+1];
  double a12 = a[i+1][k+2];
  double a13 = a[i+1][k+3];
  double a20 = a[i+2][k+0];
  double a21 = a[i+2][k+1];
  double a22 = a[i+2][k+2];
  double a23 = a[i+2][k+3];
  double a30 = a[i+3][k+0];
  double a31 = a[i+3][k+1];
  double a32 = a[i+3][k+2];
  double a33 = a[i+3][k+3];
```

(continua ...)

Ottimizzazione manuale (3)

(… segue …)

```
      c[i+0][j+0] = (a00*b00+a01*b10)+(a02*b20+a03*b30)+c[i+0][j+0];
      c[i+0][j+1] = (a00*b01+a01*b11)+(a02*b21+a03*b31)+c[i+0][j+1];
      c[i+0][j+2] = (a00*b02+a01*b12)+(a02*b22+a03*b32)+c[i+0][j+2];
      c[i+0][j+3] = (a00*b03+a01*b13)+(a02*b23+a03*b33)+c[i+0][j+3];
      c[i+1][j+0] = (a10*b00+a11*b10)+(a12*b20+a13*b30)+c[i+1][j+0];
      c[i+1][j+1] = (a10*b01+a01*b11)+(a12*b21+a13*b31)+c[i+1][j+1];
      c[i+1][j+2] = (a10*b02+a01*b12)+(a12*b22+a13*b32)+c[i+1][j+2];
      c[i+1][j+3] = (a10*b03+a01*b13)+(a12*b23+a13*b33)+c[i+1][j+3];
      c[i+2][j+0] = (a20*b00+a21*b10)+(a22*b20+a23*b30)+c[i+2][j+0];
      c[i+2][j+1] = (a20*b01+a21*b11)+(a22*b21+a23*b31)+c[i+2][j+1];
      c[i+2][j+2] = (a20*b02+a21*b12)+(a22*b22+a23*b32)+c[i+2][j+2];
      c[i+2][j+3] = (a20*b03+a21*b13)+(a22*b23+a23*b33)+c[i+2][j+3];
      c[i+3][j+0] = (a30*b00+a31*b10)+(a32*b20+a33*b30)+c[i+3][j+0];
      c[i+3][j+1] = (a30*b01+a31*b11)+(a32*b21+a33*b31)+c[i+3][j+1];
      c[i+3][j+2] = (a30*b02+a31*b12)+(a32*b22+a33*b32)+c[i+3][j+2];
      c[i+3][j+3] = (a30*b03+a31*b13)+(a32*b23+a33*b33)+c[i+3][j+3];
    }
  }
```

(è finita!)

Naturale *vs.* ottimizzata manualmente

Macchina & Compilatore:
- Intel PIII@700 Mhz

Versione naturale:
- 1 istruzione, 3 loop, 7 righe di codice
- prestazioni: 80 MFlops (11% del picco, ☹)

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 (56% del picco, ☺)

Non solo compilatore…

- Il compilatore ha associata una *runtime library*
- Contiene funzioni chiamate esplicitamente
  - funzioni trigonometriche e trascendenti
  - manipolazioni di bit
  - I/O (C)
- Contiene funzioni chiamate implicitamente
  - funzioni di I/O (Fortran)
  - operatori “complessi” del linguaggio
  - *routine* di “utilità” generiche, gestione eccezioni, …
  - *routine* di supporto ad un particolare modello di calcolo (OpenMP, UPC, GAF)
- Può essere fondamentale per le prestazioni
  - qualità dell’implementazione
  - funzioni matematiche accurate *vs.* veloci

Primo: rimuovere operazioni inutili

- Il compilatore sa fare automaticamente:
- *Dead code removal*: per esempio eliminare un *if*

```
b = a + 5.0;
if ((a>0.0) && (b<0.0)) {
      /* some statements… */
}
```

- *Redundant code removal*:

```
#define C 1.0
…
 f = C*f;
```

- Ma la vita non è sempre così semplice…

Il compilatore può fare tutto?

- Il compilatore può fare molto… ma non è un essere umano
- È piuttosto facile intralciare il suo lavoro
  - "corpo" del *loop* troppo lungo
  - *loop* con i due estremi di iterazione variabili
  - uso eccessivo di costrutti condizionali (*if*)
  - uso eccessivo di puntatori ed indici
  - uso improprio di variabili intermedie
- Importante:
  - due codici semanticamente uguali possono avere prestazioni ben diverse!
  - il compilatore può fare assunzioni erronee ed alterare la semantica!

Bit counting

- 1011010000010101 → 7
- Approccio semplice, *bit-shift*:

```
unsigned int s, bc;

…

bc = 0;
for(i = 0; i < 32; ++i) {      /* 32 bits! */
  bc += s & 1;
  s >>= 1;
}
```

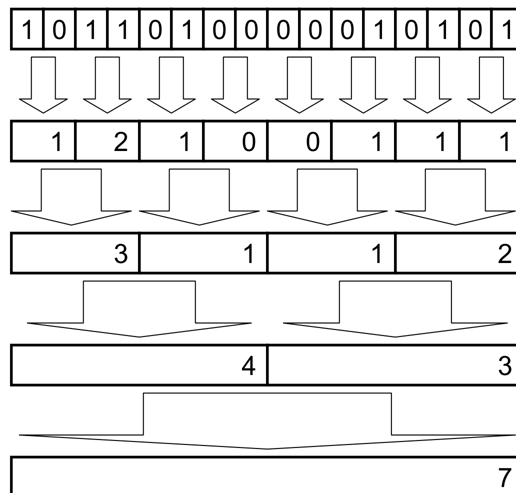
Bit counting: metodo *lookup table*

```
unsigned char bctable[256] = { 0, 1, 1, 2, 1, 2, 2, 3,
  1, /* … e così via fino a */ 6, 7, 7, 8};

unsigned int s, bc;

…

bc = 0;
for(i = 0; i < 4; ++i) {       /* 32 bits/8 bits! */
  bc += bctable[s & 0xFFu];
  s >>= 8;
}
```

Bit counting: metodo *divide et impera*

- Il totale è uguale alla somma dei *bit count* delle due metà del valore iniziale
- In ogni metà, il totale parziale è uguale alla somma dei *bit count* dei due quarti del numero originale che la compongono
- E così via fino alle coppie di bit adiacenti
- Il risultato ad ogni passo è rappresentabile con non più bit di quelli esaminati
- Attenzione ai riporti!!
  - è necessario usare maschere anche per evitarli
  - non sono un problema da un certo punto in poi

Bit counting: metodo *divide et impera*



Bit counting: metodo *divide et impera*

```
unsigned int s, bc;
…
bc = 0;
bc = (s & 0x55555555u) + (s >> 1 & 0x55555555u);
bc = (bc & 0x33333333u) + (bc >> 2 & 0x33333333u);

/* ora bc è zero a nibble (4 bit) alterni */
bc = bc + (bc >> 4 & 0x0F0F0F0Fu);

/* ora in ogni byte si fa lo stesso calcolo */
bc = bc + (bc >> 8)

/* alla fine bisogna eliminare le parti inutili */
bc = (bc + (bc >> 16)) & 0x7Fu;
```

Uguali o diversi?

- Versione *divide et impera*:

```
bc = (s & 0x55555555u) + (s >> 1 & 0x55555555u);
bc = (bc & 0x33333333u) + (bc >> 2 & 0x33333333u);
  …
```

- Semanticamente equivalente:

```
bc = (s & 0x55555555u) + ((s & 0xAAAAAAAAu) >> 1);
bc = (bc & 0x33333333u) + ((bc & 0xCCCCCCCCu) >> 2);
  …
```

- Ma peggiore perché:
  - il compilatore non può capire l'equivalenza
  - si dovranno caricare due costanti in più

35

Un codice cinetico (1)

- Forma compatta (**`npop = 19`**)

```
for(ip=0; ip<npop; ++ip)
      u[i4] = u[i4]+pop[i4][ip]*cx[ip];
```

- Dove **`cx[ip]`** può valere **`-1`**, **`0`**, o **`+1`**
- Forma esplicita:

```
u[i4] = pop[i4][2]-pop[i4][3]-pop[i4][7]+pop[i4][8]
    +pop[i4][11]-pop[i4][12]+pop[i4][15]-pop[i4][16]
    -pop[i4][17]+pop[i4][18]
```

- Quante operazioni per la forma compatta?
- Quante operazioni per la forma esplicita?

36

Un codice cinetico (2)

- Prodotti = 3 × npop × nfluid = 57 × nfluid
- Somme = 4 × npop × nfluid = 76 × nfluid
- Load = 8 × npop × nfluid + nfluid = 153 × nfluid
- Store = 4 × npop × nfluid + 5 × nfluid = 81 × nfluid
- Strutture di controllo = npop × nfluid + nfluid = 20 × nfluid
- Tempo = 0.70’’

- Prodotti = 0
- Somme = 45 × nfluid
- Load = 50 × nfluid
- Store = 5 × nfluid
- Strutture di controllo = nfluid
- Tempo = 0.26’’

Common Subexpression Elimination

- Per i calcoli intermedi si riusano spesso alcune espressioni: può essere vantaggiosa “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!!

Effetti collaterali

- La presenza di funzioni può inibire il compilatore dal fare ottimizzazioni
- Se:
  int `f(int x) {`
```
  return x + dx;
}
```
→ `f(x)+f(x)+f(x) == 3*f(x)`
- Se:
```
int f(int &x) {
  x = x + dx;
  return x;
}
```
→ `f(x)+f(x)+f(x) != 3*f(x)`

CSE e chiamate a funzione

- Alterando l'ordine delle chiamate il compilatore non sa se si altera il risultato (possibili effetti collaterali)
- 5 chiamate a funzioni, 5 prodotti:
```
x = r*sin(a)*cos(b);
y = r*sin(a)*sin(b);
z = r*cos(a);
```
- 4 chiamate a funzioni, 4 prodotti (1 variabile temporanea):
```
temp = r*sin(a);
x = temp*cos(b);
y = temp*sin(b);
z = r*cos(a);
```

LICM & CSE (1)

```
do k=1,2*nwaz+1
do j=2,Ny-1
do i=1,nwax+1
 der1=(uv(i,j+1,k)-uv(i,j,k))/(yv(j+1)-yv(j))*(yv(j)-yv(j-1))/(yv(j+1)-yv(j-1))
     +(uv(i,j,k)-uv(i,j-1,k))/(yv(j)-yv(j-1))*(yv(j+1)-yv(j))/(yv(j+1)-yv(j-1)
 der2=(wv(i,j+1,k)-wv(i,j,k))/(yv(j+1)-yv(j))*(yv(j)-yv(j-1))/(yv(j+1)-yv(j-1))
     +(wv(i,j,k)-wv(i,j-1,k))/(yv(j)-yv(j-1))*(yv(j+1)-yv(j))/(yv(j+1)-yv(j-1))
 der3(vv(i,j+1,k)-vv(i,j,k))/(yv(j+1)-yv(j))*(yv(j)-yv(j-1))/(yv(j+1)-yv(j-1))
     +(vv(i,j,k)-vv(i,j-1,k))/(yv(j)-yv(j-1))*(yv(j+1)-yv(j))/(yv(j+1)-yv(j-1))
 Nu_n(i,j,k)=mm(i)*uu(i,j,k)+der1+nn(k)*uw(i,j,k)
 Nw_n(i,j,k)=mm(i)*uw(i,j,k)+der2+nn(k)*ww(i,j,k)
 Nv_n(i,j,k)=mm(i)*uv(i,j,k)+der3+nn(k)*wv(i,j,k)
end do
end do
end do
```

LICM & CSE (2)

Loop-Invariant Code Motion

```
do k=1,2*nwaz+1
  do j=2,Ny-1
   dyvp1 = (yv(j+1)-yv(j))/(yv(j+1)-yv(j-1))
   dyvm1 = (yv(j)-yv(j-1))/(yv(j+1)-yv(j-1))
   yvp1 = 1.d0/(yv(j+1)-yv(j))
   yvm1 = 1.d0/(yv(j)-yv(j-1))
   do i=1,nwax+1
    der1=(uv(i,j+1,k)-uv(i,j,k))*yvp1*dyvm1+(uv(i,j,k)-uv(i,j-1,k))*yvm1*dyvp1
    der2=(wv(i,j+1,k)-wv(i,j,k))*yvp1*dyvm1+(wv(i,j,k)-wv(i,j-1,k))*yvm1*dyvp1
    der3=(vv(i,j+1,k)-vv(i,j,k))*yvp1*dyvm1+(vv(i,j,k)-vv(i,j-1,k))*yvm1*dyvp1
    Nu_n(i,j,k)=mm(i)*uu(i,j,k)+der1+nn(k)*uw(i,j,k)
    Nw_n(i,j,k)=mm(i)*uw(i,j,k)+der2+nn(k)*ww(i,j,k)
    Nv_n(i,j,k)=mm(i)*uv(i,j,k)+der3+nn(k)*wv(i,j,k)
   end do
  end do
end do
```

Common Subexpression Elimination

LICM & *CSE* (3)

- Intel Core 2 a 2.0 Ghz, gfortran -O3
  - versione originale (per timestep): 0.98''
  - versione modificata (per timestep): 0.78''
- IBM Power5 a 1.3 Ghz, xlf -O5 (-O2)
  - versione originale (per timestep): 0.42'' (0.60'')
  - versione modificata (per timestep): 0.42'' (0.50'')
- Molto dipendente dal compilatore…

CSE: limitazioni

- "*Core loop*" troppo grossi:
  - il compilatore lavora su "finestre" di dimensioni finite: potrebbe non accorgersi di una grandezza da riutilizzata
- Funzioni:
  - se altero l'ordine delle chiamate ottengo gli stessi risultati?
- Ordine di valutazione
  - solo ad alti livelle di ottimizzazione il compilatore altera l'ordine della operazioni (`-qnostrict` per IBM)
  - per inibirla in certe espressioni: parentesizzare (il programmatore ha sempre ragione)
- Aumenta l'uso di registri per appoggio dei valori intermedi (→ *register spilling*)

Cosa può fare il compilatore?

```
for(k=0; k<n3m; ++k)
  for(j=n2i; j<=n2do; ++j) {
    jj=mynode*n2do+j:
    for(i=0; i<n1m; ++i) {
       acc =1./(1.-coe*aciv[i]*(1.-(int)forclo[k][j][i][nve]));
       aci[jj][i]= 1.;
       api[jj][i]=-coe*apiv[i]*acc*(1.-(int)forclo[k][j][i][nve]);
       ami[jj][i]=-coe*amiv[i]*acc*(1.-(int)forclo[k][j][i][nve]);
       fi[jj][i]=qcap[k][j][i]*acc;
    }
  }
....
....
for(i=0; i<n1m; ++i)
  for(j=n2i; j<=n2do; ++j) {
    jj=mynode*n2do+j;
    for(k=0; k<n3m; ++k) {
         acc =1./(1.-coe*ackv[k]*(1.-(int)forclo[k][j][i][nve]));
         ack[jj][k]= 1.;
         apk[jj][k]=-coe*apkv[k]*acc*(1.-(int)forclo[k][j][i][nve]);
         amk[jj][k]=-coe*amkv[k]*acc*(1.-(int)forclo[k][j][i][nve]);
         fk[jj][k]=qcap[k][j][i]*acc;
    }
  }
```

Probabilmente questo...

```
for(k=0; k<n3m; ++k)
  for(j=n2i; j<=n2do; ++j) {
    jj=mynode*n2do+j:
    for(i=0; i<n1m; ++i) {
       double tmp = 1.-(int)forclo[k][j][i][nve];
       acc =1./(1.-coe*aciv[i]*tmp));
       aci[jj][i]= 1.;
       api[jj][i]=-coe*apiv[i]*acc*tmp;
       ami[jj][i]=-coe*amiv[i]*acc*tmp;
       fi[jj][i]=qcap[k][j][i]*acc;
    }
  }
....
....
for(i=0; i<n1m; ++i)
  for(j=n2i; j<=n2do; ++j) {
    jj=mynode*n2do+j;
    for(k=0; k<n3m; ++k) {
         double tmp = 1.-(int)forclo[k][j][i][nve];
         acc =1./(1.-coe*ackv[k]*tmp);
         ack[jj][k]= 1.;
         apk[jj][k]=-coe*apkv[k]*acc*tmp;
         amk[jj][k]=-coe*amkv[k]*acc*tmp;
         fk[jj][k]=qcap[k][j][i]*acc;
    }
  }
```

Ma non questo...

```
for(k=0; k<n3m; ++k)
  for(j=n2i; j<=n2do; ++j)
    for(i=0; i<n1m; ++i)
       tmpfact[k][j][i] = 1.-(int)forclo[k][j][i][nve];


for(k=0; k<n3m; ++k)
  for(j=n2i; j<=n2do; ++j) {
    jj=mynode*n2do+j:
    for(i=0; i<n1m; ++i) {
       double tmp = tmpfact[k][j][i];
       acc =1./(1.-coe*aciv[i]*tmp));
       aci[jj][i]= 1.;
       api[jj][i]=-coe*apiv[i]*acc*tmp;
       ami[jj][i]=-coe*amiv[i]*acc*tmp;
       fi[jj][i]=qcap[k][j][i]*acc;
    }
  }
....
....
// idem per l'altro loop
```

CSE "a mano"

Codice originale:

[6]	53.7	292.61	73.25	672	.solve13 [6]
		43.60	0.00	172032/172032	.trvpjk [9]
		29.65	0.00	172032/172032	.tripvi [12]

Codice modificato:

[6]	47.8	203.63	68.86	672	.solve13 [6]
		40.85	0.00	172032/172032	.trvpjk [9]
		28.01	0.00	172032/172032	.tripvi [12]

Costo delle operazioni

Operazioni come:

- somma e prodotto tra numeri *floating-point*
- somma e prodotto tra interi
- divisione
- operazioni logiche *bitwise*

sono svolte in *hardware* dalle unità di calcolo

Ma non tutte le operazioni hanno lo stesso costo:

- somma(interi/reali) → ?
- prodotto (interi/reali) → ?
- divisione (interi/reali) → ?

Quanto costano le operazioni?

Costo in cicli per differenti operazioni su IBM Power3:

Istruzione	32 bit	64 bit
Prodotto interi	3-4	3-9
Divisione interi	21	34
Somma o prodotto reali:	3-4	3-4
Fused multiply and add:	3-4	3-4
Divisione reali	14-21	18-25
Radice quad. reali	14-23	22-31

Ma non è tutto…

Strength reduction & co.

Sostituzione operazioni lente → sequenze più veloci

a/4 → a>>2

a*32 → a<<5

a%256 → a&255

```
for(i=0; i<N; ++i)
  b[i] = qualcosa();
```

→

```
for(i=N, p=b; i--; )
  *p++= qualcosa();
```

Il compilatore lo sa fare, ma non sempre

Potenze intere

Calcolo di una funzione per punti:

```
#include <math.h>

void x7(int n, double *q) {
  int i;
  double x, step;

  step = 1.0/n;
  for(i=0, x=0.0; i < n; ++i) {
    q[i] = pow(x, 7.0);


    x += step;
  }
}
```

```
void x7(int n, double *q) {
  int i;
  double x, step;

  step = 1.0/n;
  for(i=0, x=0.0; i < n; ++i) {
    double x2 = x*x;
    double x4 = x2*x2;
    q[i] = x4*x2*x;
    x += step;
  }
}
```

Potenze intere: prestazioni

- IBM Power5@1900 MHz
- compilatore xlc 8.0

Opzione	pow()	prodotti
`-O0`	25.62”	2.64”
`-O2`	25.53’’	1.53”
`-O3`	1.41’’	1.36”

Funzioni matematiche

- A volte può essere conveniente “riscrivere” alcune funzioni
- Le funzioni matematiche (e.g. **`sin()`**) sono codificate in modo da minimizzare uniformemente l’errore su tutto il dominio della funzione reale
- Se interessa un insieme ristretto di valori o una precisione minore si possono ridurre i tempi di calcolo usando:
  - sviluppo in serie
  - formule approssimate
  - *fit* polinomiali e razionali
  - formule di prostaferesi
- Attenzione alle proprietà numeriche!!
  - accuratezza
  - accumulazione degli errori
  - distribuzione degli errori sul dominio della funzione

Funzioni trigonometriche (1)

- `sin(x)` è definita sull'intervallo [-π, π]
- In certi casi può essere conveniente usare una serie a pochi termini, se il range di x è limitato
- per **x** compreso tra [-0.1,0.1]:

	Tempo	Errore max	Guadagno
runtime library	0.41''	-	-
III ordine	0.19''	8.3e-7	54%
V ordine	0.24''	1.5e-10	42%
VII ordine	0.28''	5.0e-11	32%

55

Funzioni trigonometriche (2)

```
c1 =  1.0;
c3 = -1.0/(2.0*3.0);
c5 =  1.0/(2.0*3.0*4.0*5.0);
c7 = -1.0/(2.0*3.0*4.0*5.0*6.0*7.0);
c9 =  1.0/(2.0*3.0*4.0*5.0*6.0*7.0*8.0*9.0);

esatto = sin(x);

approx3 = (c3*x*x+c1)*x;

approx5 =((c5*x*x+c3)*x*x+c1)*x;

approx7 =(((c7*x*x+c5)*x*x+c3)*x*x+c1)*x;
```

56

Funzioni trigonometriche (3)

- Il calcolo dell'arcocoseno è computazionalmente oneroso
  - sviluppo in serie di potenze (per $x < 0.35$)
  - approssimazione polinomiale (per $x > 0.35$)
  - singola precisione (errore relativo < 1e-7)
  - tratto da: Abramowitz and Stegun,Handbook of Mathematical Functions,1974

	Tempo	Errore max
runtime library	0.70''	-
approssimato	0.52''	9.9e-8

Funzioni trigonometriche (5)

- Codice "di prova" per analisi dati onde gravitazionali
- Calcolo di funzioni trigonometriche su una griglia 2D uniforme:
  - *n* chiamate a `sin(x)` e `cos(x)`
    - T = 202''
  - uso ricorsivo delle formule di prostaferesi
    - `cos(x+dx) = cos(dx)*cos(x)-sin(dx)*sin(x)`
    - `sin(x+dx) = sin(dx)*cos(x)+cos(dx)*sin(x)`
    - 4 chiamate a funzioni trigonometriche e 2n prodotti e somme
    - T = 170''
- Attenzione all'accumulazione degli errori!!!

Funzioni matematiche: *caveat*

- Considerare molto attentamente:
  - il dominio di valutazione della funzione nel codice
  - l'errore tollerabile
  - l'accumulazione degli errori di calcolo
  - il vantaggio ottenuto (ne vale la pena?)
  - librerie matematiche "*fast*" (opzioni **`-fast`** e simili)

- Esplicitare vincoli ed assunzioni nel codice!!!

- Alternative per funzioni "complicate":
  - *lookup table* ed interpolazioni di ordine "basso" (compromesso calcolo/memoria)
  - interpolazioni con funzioni razionali

59