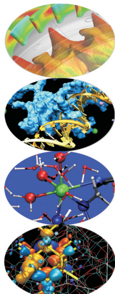
CINECA SCAI SuperComputing Applications and Innovation

SAPIENZA UNIVERSITÀ DI ROMA

Programmazione Avanzata / 2

Francesco Salvadore

CINECA Roma - SCAI Department



Roma, 2014

CINECA

Outline

Compilatori e ottimizzazione

Linguaggi

- Esiste un'infinità di linguaggi differenti
- `http://foldoc.org/contents/language.html`

```
20-GATE; 2.PAK; 473L Query; 51forth; A#; A-0; a1; a56;
Abbreviated Test Language for Avionics Systems; ABC;
ABC ALGOL; ABCL/1; ABCL/c+; ABCL/R; ABCL/R2; ABLE;
ABSET; abstract machine; Abstract Machine Notation;
abstract syntax; Abstract Syntax Notation 1;
Abstract-Type and Scheme-Definition Language; ABSYS;
Accent; Acceptance, Test Or Launch Language; Access;
ACOM; ACOS; ACT++; Act1; Act2; Act3; Actalk; ACT ONE;
Actor; Actra; Actus; Ada; Ada++; Ada 83; Ada 95; Ada 9X;
Ada/Ed; Ada-O; Adaplan; Adaplex; ADAPT; Adaptive Simulated
Annealing; Ada Semantic Interface Specification;
Ada Software Repository; ADD 1 TO COBOL GIVING COBOL;
ADELE; ADES; ADL; AdLog; ADM; Advanced Function Presentation;
Advantage Gen; Adventure Definition Language; ADVSYS; Aeolus;
AFAC; AFP; AGORA; A Hardware Programming Language; AIDA;
AIr MAterial COmmand compiler; ALADIN; ALAM; A-language;
A Language Encouraging Program Hierarchy; A Language for Attributed ...
```

Linguaggi

- Linguaggi interpretati
  - il linguaggio viene "tradotto" statement per statement dall'interprete durante l'esecuzione
  - impossibili ottimizzazioni tra differenti statement
  - permette di verificare la correttezza statement dopo statement seguendo il flusso d'istruzioni (migliore gestione degli errori semantici)
  - Esempi: Python, PHP, Java, MATLAB,...
- Linguaggi compilati
  - il programma viene "tradotto" dal compilatore prima dell'esecuzione
  - possibili ottimizzazioni tra differenti statement
  - gestione minimale degli errori semantici
  - Fortran, C, C++

Le 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 (e.g. IBM Power)
  - istruzioni semplici
  - formato regolare delle istruzioni
  - decodifica ed esecuzione delle istruzioni semplificata
  - codice macchina molto "verboso"
- CISC: Complex Instruction Set CPU (e. g. Intel x86)
  - 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

Statement & Istruzione

- Lo statement è una riga di un codice
  - `write(*,*)`
  - `printf();`
  - `do i = 1,n`
  - `x(i) = y(i) + 1`
- L'istruzione è l'operazione che la CPU "realmente" fa, nelle macchine RISC sono essenzialmente:
  - `load/store`
  - `somma/prodotto`
  - `operazioni tra bit` ...
- Uno statement può tradursi in una sola, poche o tante istruzioni

Architettura vs. Implementazione

- Architettura:
  - set di istruzioni
  - registri architetturali interi, floating point e di stato
- Implementazione
  - registri fisici ($2.5 \div 20 \times$ 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: Power4, Power5, Power6, . . .
  - x86: Pentium III, Pentium 4, Xeon, Pentium M, Pentium D, Core, Core2, Athlon, Opteron, . . .
  - prestazioni differenti
  - "regole" diverse per ottenere alte 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
  - ottimizzazioni dipendenti dalla CPU
  - ottimizzazioni dell'uso della memoria e della cache
- 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

HPC e Linguaggi

- Linguaggi ideali per l'High Performance Computing?
  - adatti all'implementazione di algoritmi "scientifici"
  - devono permettere elevati livelli di ottimizzazione
  - integrandosi nelle recenti architetture di supercalcolo
- Quali sono?
  - Fortran/C/C++, ci limitiamo a Fortran e C con qualche cenno specifico a C++
  - il Python è in ascesa, ma per le parti computazionalmente intensive si usa appoggiarsi a C o Fortran
- I compilatori più usati per l'HPC (non sono molti)
  - GNU (gfortran, gcc, g++): "libero"
  - Intel (ifort, icc, icpc)
  - IBM (xlf, xlc, xlC)
  - Portland Group (pgf90, pgcc, pgCC)
  - PathScale, Oracle/Solaris, Fujitsu, Nag, Microsoft,...
- Linux, Windows or Mac OS X?
  - discorso complesso, ma la grande maggioranza delle architetture di supercalcolo ad oggi gira su piattaforma Linux

Il compilatore: schema

- Composto da differenti blocchi
- Front-end
  - Controllo sintassi
  - Analisi semantica
  - Traduzione rappresentazione intermedia
- Back-end
  - Ottimizzazione
  - Generazione codice eseguibile
- In genere
  - Front-end: dipende dal linguaggio
  - Back-end: indipendente dal linguaggio

Rappresentazione Intermedia

- Rappresenta gli stessi calcoli del codice di alto livello
  - in una forma più adatta per l'analisi
  - include calcoli non presenti nel sorgente, come il calcolo degli indirizzi
- C

```
while  (j<n)  {
k  =  k+j*2;
m  =  j*2;
j++
}
```

- Rappresentazione Intermedia

```
A:  t1  =  j  t2  =  n  t3  =  (t1<t2)
    jmp  (B)  t3
    jmp  (C)  TRUE
B:  t4  =  k  t5  =  j  t6  =  t5*2
    t7  t4+t6
    k=t7
    t8=j  t9=t8*2
    m=t9
    t10=j  t11=t10+1
    j=t11
    jmp(A)  TRUE
C:
```

Assembler

- La rappresentazione intermedia, dopo l'ottimizzazione, è tradotta in assembly, e eventualmente ulteriormente ottimizzata

- C

```
int add (int a, int b)
{
   return a+b;
}
```

- Assembly

```
int add (int a , int b) {
  0:   55          push    %ebp
  1:   8b ec       mov     %esp,%ebp
/home/marco/Master10/prog/add.c:2
  return a+b;
   3:   8b 45 0c mov     0xc(%ebp),%eax
   6:   03 45 08 add     0x8(%ebp),%eax
   9:   c9         leave
   a:   c3         ret
   b:   90         nop
```

Building di un Programma

- Creare un eseguibile dai sorgenti è in generale un processo a tre fasi
- Pre-processing:
  - ogni sorgente è letto dal pre-processore
    - sostituire (**`#define`**) MACROs
    - inserire codice per gli statement **`#include`**
    - inserire o cancellare codice valutando **`#ifdef`**, **`#if`** ...
- Compilazione:
  - ogni sorgente è tradotto in un codice oggetto
    - un file oggetto è una collezione organizzata di simboli che si riferiscono a variabili e funzioni definite o usate nel sorgente
- Linking:
  - file oggetti sono combinati per costruire il singolo eseguibile finale
  - ogni simbolo deve essere risolto
    - i simboli possono essere definiti nei file oggetto
    - o disponibili in altri codici oggetti (librerie esterne)

Compilare con GNU gfortran

- Quando si dà il comando:

```
user@$> gfortran dsp.f90 dsp_test.f90
```

vengono eseguiti automaticamente i tre passi

- Pre-processing

```
user@$> gfortran -E  dsp.f90
user@$> gfortran -E  dsp_test.f90
```

  - l'opzione `-E` – permette di eseguire solo il pre-processing e avere a stdout il file pre-procssato

- Compilazione dei sorgenti

```
user@$> gfortran -c dsp.f90
user@$> gfortran -c dsp_test.f90
```

  - l'opzione `-c` dice `gfortran` di compilare solo i sorgenti
  - da ogni sorgente viene prodotto un file oggetto `.o`

Linking con GNU gfortran

- Linkare oggetti tra di loro

```
user@$> gfortran dsp.o dsp_test.o
```

- Per risolvere i simboli definiti in librerie esterne specificare:
  - le librerie da usare (opzione `-l`)
  - le directory in cui stanno (opzione `-L`)
- Come linkare `libblas.a` nella cartella `/opt/lib`

```
user@$> gfortran file1.o file2.o -L/opt/lib -ldsp
```

- Come creare e linkare una libreria statica

```
user@$> gfortran -c dsp.f90
ar curv libdsp.a dsp.o
ranlib libdsp.a
gfortran test_dsp.f90 -L. -ldsp
```

  - `ar` archivia in `libdsp.a dsp.o`
  - `ranlib` genera l'indice dell'archivio

Prendere confidenza

- Per il manuale in linea

```
man gcc
```

  riporta le opzioni del compilatore C di GNU e il loro significato
- Attenzione: **`man gfortran`** dà solo le opzioni ulteriori rispetto a gcc, mentre per gli altri compilatori solitamente i man sono replicati nelle parti comuni
- Il numero di opzioni è notevole e purtroppo differisce da compilatore a compilatore
- Tipologia di opzioni
  - linguaggio: sullo standard (o sul dialetto) da seguire
  - ottimizzazione: argomento delle prossime slide...
  - target: per l'integrazione con l'architettura di calcolo
  - debugging: la più importante, **`-g`**, crea i simboli di debugging necessari per l'uso di debugger
  - warning: per avere informazioni sulla compilazione

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 unrolling/merging
  - Loop 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
- Strenght reduction di potenze non intere, o maggiori di $2 \div 4$
- Common subexpression elimination attraverso chiamate a funzione
- Unrolling, Merging, Blocking con:
  - chiamate a procedure
  - chiamate o statement di Input-Output 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

Livelli di ottimizzazione

- I compilatori forniscono dei livelli di ottimizzazione "predefiniti" utilizzabili con la semplice opzione `-O<n>`
  - `n` accresce il livello di ottimizzazione, da 0 a 3 (a volte fino a 5)
- Fortran IBM:
  - `-O0`: nessuna ottimizzazione (utile insieme a `-g` in debugging)
  - `-O2, -O` : ottimizzazioni locali, compromesso tra velocità di compilazione, ottimizzazione e dimensioni dell'eseguibile
  - `-O3`: ottimizzazioni memory-intensive, può alterare la semantica del programma (da qui in poi da considerare l'uso si `-qstrict` per evitare risultati errati)
  - `-O4`: ottimizzazioni aggressive (`-qarch=auto`, `-qhot`, `-qipa`, `-qtune=auto`, `-qcache=auto`, `-qsimd=auto`)
  - `-O5`: come `-O4` con `-qipa=level=2` aggressiva e lenta
- Alcuni compilatori hanno `-fast`, che include `On` e altro
- Per GNU una scelta comune insieme a `-O3` é `-funroll-loops`
- Attenzione all'ottimizzazione di default: per GNU é `-O0` mentre per gli altri di solito é `-O2`

Loop con -O3 (Compilatore Intel)


```
icc (or ifort) -O3
```

- Automatic vectorization (use of packed SIMD instructions)
- Loop interchange (for more efficient memory access)
- Loop unrolling (more instruction level parallelism)
- Prefetching (for patterns not recognized by h/w prefetcher)
- Cache blocking (for more reuse of data in cache)
- Loop peeling (allow for misalignment)
- Loop versioning (for loop count; data alignment; runtime dependency tests)
- Memcpy recognition (call Intel's fast memcpy, memset)
- Loop splitting (facilitate vectorization)
- Loop fusion (more efficient vectorization)
- Scalar replacement (reduce array accesses by scalar temps)
- Loop rerolling (enable vectorization)
- Loop reversal (handle dependencies)

Dal sorgente all'eseguibile

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

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

- Il “nocciolo”
  - carica dalla memoria tre valori
  - fa una moltiplicazione ed una somma
  - immagazzina il risultato

Hands-on: cluster PLX

- Accedere al cluster PLX con le proprie credenziali

```
ssh -X <user_name>@login.plx.cineca.it
```

  - in questo modo si accede al cosiddetto nodo di front-end
  - il front-end guida l'utente per operare sui nodi di calcolo attraverso un sistema di code

```
Model: IBM iDataPlex DX360M3
Architecture: Linux Infiniband Cluster
Processors Type:
-Intel Xeon (Esa-Core Westmere) E5645 2.4 GHz (Compute)
-Intel Xeon (Quad-Core Nehalem) E5530 2.66 GHz (Service & Login)
Number of nodes: 274 Compute + 1 Login + 1 Service + 8 Fat +
                 6 RVN + 8 Storage  + 2 Management
Number of cores: 3288 (Compute)
Number of GPUs:528 nVIDIA Tesla M2070 + 20 nVIDIA Tesla M2070Q
RAM: 14 TB (48 GB/Compute node + 128GB/Fat node)
```

Hands-on: PLX e code

- Per testare le performance nei casi reali occorre usare i nodi calcolo (diversi da quelli di front-end!)
  - occorre preparare un file di script con in comandi, e.g. **qsub.script**

```
#!/bin/bash
#PBS -A train_cspR2013
#PBS -W group_list="train_cspR2013"
#PBS -l walltime=00:10:00 # or other needed time
#PBS -l select=1:ncpus=1  # or ncpus=N, N is the number of CORES
#PBS -q private
cd $PBS_O_WORKDIR
# (a) Load Modules, e.g.
module load profile/advanced
module load gnu/4.7.2
# (b) Run executables
./matmul >& matmul.out
```

- Il file di script viene sottomesso in coda con il comando:

```
qsub qsub.script
```

- La coda private permette di accedere ai nodi `fat`

Hands-on: PLX e moduli

- Il software installato é organizzato in moduli
  - per poter usare i moduli della lista completa advanced (farlo come prima cosa!): **`module load profile/advanced`**
  - per avere la lista dei moduli disponibili: **`module av`**
  - per avere la lista dei moduli caricati: **`module li`**
  - per caricare un modulo, e.g: **`module load gnu/4.7.2`**
  - per scaricare un modulo, e.g: **`module unload gnu/4.7.2`**
  - per scaricare tutti i moduli caricati, e.g: **`module purge`**

- I codici per le esercitazioni si trovano in

```
/gpfs/scratch/userinternal/fsalvado/OPT_2013/Exercises
```

Hands-on: OPT levels

- Prodotto matrice-matrice, 1024×1024, doppia precisione
- Ordine dei loop ottimale per la cache
- Architettura considerata per la prove, PLX:
  - 2 esa-core XEON 5650 Westmere CPUs 2.40 GHz per nodo
- Per caricare i compilatori: (**`module load profile/advanced`**):
  - GNU: **`module load gnu/4.7.2`**
  - Intel: **`module load intel/cs-xe-2013--binary`**
  - PGI: **`module load pgi/12.10`**
  - Fare **`unload`** di un compilatore prima di caricarne un altro

	GNU	Intel	PGI	GNU	Intel	PGI
Opzione	secondi	secondi	secondi	GFlops	GFlops	GFlops
-O0						
-O1						
-O2						
-O3						
-O3 -funroll-loops		—	—		—	—
-fast	—			—		

Hands-on: Soluzione

- Prodotto matrice-matrice, 1024×1024, doppia precisione
- Risultati con compilatori Fortran

	GNU	Intel	PGI	GNU	Intel	PGI
Opzione	secondi	secondi	secondi	GFlops	GFlops	GFlops
-O0	7.3	8.94	3.4	0.3	0.2	0.6
-O1	1.8	1.41	3.4	1.2	1.5	0.6
-O2	1.4	0.72	1.4	1.5	2.9	1.5
-O3	0.75	0.33	1.4	2.8	6.4	1.5
-O3 -funroll-loops	0.67	——	——	3.2	——	——
-fast	——	0.32	0.7	——	6.5	3.1

- I tempi sono stabili?
- Perché tanta varietà di risultati?
- Basta passare da -On a -On+1?
- Il compilatore Intel é davvero il migliore?

MatrixMul: performance

- Prodotto matrice-matrice, 1024×1024, doppia precisione
- Ordine dei loop ottimale per la cache
- Scritto in Fortran
- Architetture considerate per le prove
  - FERMI: IBM Blue Gene/Q system, nodi da 16 core single-socket PowerA2 a 1.6 GHz di frequenza
  - PLX: 2 esa-core XEON 5650 Westmere CPUs 2.40 GHz per nodo

FERMI - xlf

Opzione	secondi	Mflops
-O0	65.78	32.6
-O2	7.13	301
-O3	0.78	2735
-O4	55.52	38.7
-O5	0.65	3311

PLX - ifort

Opzione	secondi	MFlops
-O0	8.94	240
-O1	1.41	1514
-O2	0.72	2955
-O3	0.33	6392
-fast	0.32	6623

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

Cosa fa il compilatore: report

- Cosa accade ai diversi livelli di ottimizzazione?
  - Perché il compilatore IBM su Fermi al livello **`-O4`** degrada cosí le performance?
- Utilizzare le opzioni di report é un buon modo di capire cosa sta facendo il compilatore
- Su IBM **`-qreport`** mostra che per **`-O4`** l'ottimizzazione prende un percorso completamente diverso dagli altri casi
  - il compilatore riconosce il pattern del prodotto matrice-matrice e sostituisce le righe di codice con la chiamata a una funzione di libreria BLAS **`__xl_dgemm`**
  - che però si rivela molto lenta perché non fa parte delle librerie matematiche ottimizzate da IBM (ESSL)
  - anche il compilatore Intel fa questo per dgemm, ma invoca le efficienti MKL
- Aumentando il livello di ottimizzazione, solitamente le performance migliorano
  - ma è bene testare questo miglioramento per il proprio codice

Uno sguardo all'assembler

- Esempio datato, utile peró per capire

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
```

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
```

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
```

L'indirizzamento!

Matrix Multiply inner loop code with -qnoopt

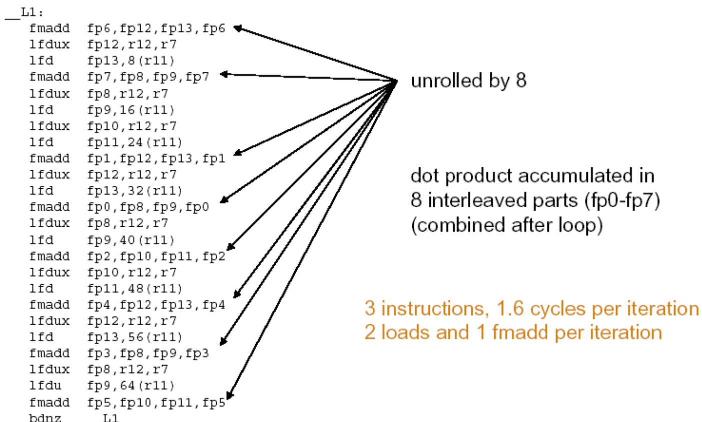
necessary instructions   loop control   addressing code

```
__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
```

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 piú elementi successivi
- Puó essere fatto automaticamente?

Ottimizziamo



Ottimizziamo di piú

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

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 2/24
- -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 96/150
- -O4: 344 istruzioni (unrolling&blocking)
  - 139 load / 74 store
  - 100 floating point multiply+add Flop/istruzione 200/344

Fa tutto il compilatore?

- **`-fast`** realizza uno speed-up di 30 volte rispetto a **`-O0`** per il caso matrice-matrice (ifort su PLX)
  - mette in atto una vasta gamma di ottimizzazioni più o meno complicate
- **Ha senso ottimizzare il codice anche manualmente?**
- Il compilatore sa fare automaticamente
  - Dead code removal: per esempio rimuovere un if

```
b = a + 5.0;
if ((a>0.0) && (b<0.0)) {
    ......
}
```

  - Redudant code removal

```
integer, parameter :: c=1.0
f=c*f
```

- Ma la vita non è sempre così facile

Loop counters: attenzione ai tipi

- Usare sempre i tipi corretti
- Usare un real per l'indice dei loop implica una trasformazione implicita reale $\rightarrow$ intero . . .
- Secondo i recenti standard Fortran si tratta di un vero e proprio errore, ma i compilatori tendono a tollerarlo

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

Risultati in secondi

Compilazione	integer	real
(PLX) gfortran -O0	9.96	8.37
(PLX) gfortran -O3	0.75	2.63
(PLX) ifort -O0	6.72	8.28
(PLX) ifort -fast	0.33	1.74
(PLX) pgif90 -O0	4.73	4.85
(PLX) pgif90 -fast	0.68	2.30
(FERMI) bgxlf -O0	64.78	104.10
(FERMI) bgxlf -O5	0.64	12.38

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

Index reordering

- Per un loop nest semplice ci pensa il compilatore
  - a patto di usare un opportuno livello di ottimizzazione

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

- Tempi in secondi

Compilazione	j-k-i	i-k-j
(PLX) ifort -O0	6.72	21.8
(PLX) ifort -fast	0.34	0.33

Index reordering / 2

- Per loop nesting più complicati il compilatore a volte no...
  - anche al più alto livello di ottimizzazione
  - conoscere il meccanismo di cache è quindi utile!

```
do jj = 1, n, step
   do kk = 1, n, step
      do ii = 1, n, step
         do j = jj, jj+step-1
            do k = kk, kk+step-1
               do i = ii, ii+step-1
                  c(i,j) = c(i,j) + a(i,k)*b(k,j)
               enddo
            enddo
         enddo
      enddo
   enddo
enddo
```

- Tempi in secondi

Compilazione	j-k-i	i-k-j
(PLX) ifort -O0	10	11.5
(PLX) ifort -fast	1.	2.4

Modularizzazione e cache


```
do i=1,nwax+1
    do k=1,2*nwaz+1
       call diffus (u_1,invRe,qv,rv,sv,K2,i,k,Lu_1)
       call diffus (u_2,invRe,qv,rv,sv,K2,i,k,Lu_2)
....
    end do
 end do


 subroutine diffus (u_n,invRe,qv,rv,sv,K2,i,k,Lu_n)
   do j=2,Ny-1
     Lu_n(i,j,k)=invRe*(2.d0*qv(j-1)*u_n(i,j-1,k)-(2.d0*rv(j-1)
               +K2(i,k))*u_n(i,j,k)+2.d0*sv(j-1)*u_n(i,j+1,k))
   end do
 end subroutine
```

- 5 accessi in memoria contigui in i
- 3 accessi in memoria contigui in j

Modularizzazione e cache / 2


```
call diffus (u_1,invRe,qv,rv,sv,K2,Lu_1)
call diffus (u_2,invRe,qv,rv,sv,K2,Lu_2)
....

subroutine diffus (u_n,invRe,qv,rv,sv,K2,i,k,Lu_n)
 do k=1,2*nwaz+1
  do j=2,Ny-1
   do i=1,nwax+1
    Lu_n(i,j,k)=invRe*(2.d0*qv(j-1)*u_n(i,j-1,k)-(2.d0*rv(j-1)
              +K2(i,k))*u_n(i,j,k)+2.d0*sv(j-1)*u_n(i,j+1,k))
   end do
  end do
 end do
end subroutine
```

- Modularizzare così permette di avere l'ordine ottimale dei loop
- A volte il compilatore trasforma da solo: in questo caso è richiesto l'"inlining"

Inlining

- Ottimizzazione manuale o effettuata dal compilatore che sostituisce una funzione col suo corpo
  - elimina il costo della chiamata e potenzialmente l'instruction cache
  - rende più facile l'ottimizzazione interprocedurale
- In C e C++ la keyword **`inline`** è un "suggerimento"
- Non ogni funzione è "inlinable" e in ogni caso dipende dalle capacitá del compilatore
  - oltre che dalle capacità del programmatore
- Intel (n: 0=disable, 1=secondo la keyword, 2=se opportuno)

```
-inline-level=n
```

- GNU (n: size, default is 600):

```
-finline-functions
-finline-limit=n
```

- In alcuni compilatori automaticamente attivate ad alti livelli di ottimizzazione

Common Subexpression Elimination

- Per i calcoli intermedi si riusano spesso alcune espressioni: può essere vantaggioso riciclare quantità già calcolate:
  A= B+C+D
  E= B+F+C
- Richiede: 4 load,2 store, 4 somme
  A=(B+C) + D
  E=(B+C) + F
- Richiede: 4 load, 2 store, 3 somme
- Attenzione: dal punto di vista numerico il risultato non è necessariamente identico
- Se la locazione di un array è acceduta più di una volta può convenire effettuare lo "Scalar replacement"
  - ad opportune ottimizzazioni il compilatore può farlo

Funzioni e Side Effects

- Lo scopo “primo” di una funzione é in genere dare un valore in ritorno
  - a volte, per vari motivi, peró non é cosí
  - la modifica di varaibili passate, o globali o anche l’I/O si chiamano comunque side effects (effetti collaterali)
- La presenza di funzioni con side effects può inibire il compilatore dal fare ottimizzazioni
- Se:

```
function f(x)
   f=x+dx
end
```

allora `f(x)+f(x)+f(x)` puó essere valutato come `3*f(x)`

- Se:

```
function f(x)
   x=x+dx
   f=x
end
```

allora la precedente valutazione non é piú corretta

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);
```

CSE: limitazioni

- Core loop troppo grossi:
  - il compilatore lavora su finestre di dimensioni finite: potrebbe non accorgersi di una grandezza da riutilizzare
- Funzioni:
  - se altero l'ordine delle chiamate ottengo lo stesso risultato?
- Ordine e valutazione:
  - solo ad alti livelli di ottimizzazione il compilatore altera l'ordine delle operazioni (**`-qnostrict`** per IBM)
  - per inibirla in certe espressioni: mettere le parantesi (il programmatore ha sempre ragione)
- Aumenta l'uso di registri per l'appoggio dei valori intermedi ("register spilling")

Cosa può fare il compilatore?


```
do k=1,n3m
   do j=n2i,n2do
      jj=my_node*n2do+j
      do i=1,n1m
         acc =1./(1.-coe*aciv(i)*(1.-int(forclo(nve,i,j,k))))
         aci(jj,i)= 1.
         api(jj,i)=-coe*apiv(i)*acc*(1.-int(forclo(nve,i,j,k)))
         ami(jj,i)=-coe*amiv(i)*acc*(1.-int(forclo(nve,i,j,k)))
         fi(jj,i)=qcap(i,j,k)*acc
     enddo
   enddo
enddo
...
...
do i=1,n1m
   do j=n2i,n2do
      jj=my_node*n2do+j
      do k=1,n3m
         acc =1./(1.-coe*ackv(k)*(1.-int(forclo(nve,i,j,k))))
         ack(jj,k)= 1.
         apk(jj,k)=-coe*apkv(k)*acc*(1.-int(forclo(nve,i,j,k)))
         amk(jj,k)=-coe*amkv(k)*acc*(1.-int(forclo(nve,i,j,k)))
         fk(jj,k)=qcap(i,j,k)*acc
      enddo
   enddo
enddo
```

Probabilmente questo


```
do k=1,n3m
   do j=n2i,n2do
      jj=my_node*n2do+j
      do i=1,n1m
         temp = 1.-int(forclo(nve,i,j,k))
         acc =1./(1.-coe*aciv(i)*temp)
         aci(jj,i)= 1.
         api(jj,i)=-coe*apiv(i)*acc*temp
         ami(jj,i)=-coe*amiv(i)*acc*temp
         fi(jj,i)=qcap(i,j,k)*acc
      enddo
   enddo
enddo
...
...
do i=1,n1m
   do j=n2i,n2do
      jj=my_node*n2do+j
      do k=1,n3m
         temp = 1.-int(forclo(nve,i,j,k))
         acc =1./(1.-coe*ackv(k)*temp)
         ack(jj,k)= 1.
         apk(jj,k)=-coe*apkv(k)*acc*temp
         amk(jj,k)=-coe*amkv(k)*acc*temp
         fk(jj,k)=qcap(i,j,k)*acc
      enddo
   enddo
enddo
```

Ma non questo

```
do k=1,n3m
   do j=n2i,n2do
      do i=1,n1m
         temp_fact(i,j,k) = 1.-int(forclo(nve,i,j,k))
      enddo
   enddo
enddo
...
...
do i=1,n1m
   do j=n2i,n2do
      jj=my_node*n2do+j
      do k=1,n3m
         temp = temp_fact(i,j,k)
         acc =1./(1.-coe*ackv(k)*temp)
         ack(jj,k)= 1.
         apk(jj,k)=-coe*apkv(k)*acc*temp
         amk(jj,k)=-coe*amkv(k)*acc*temp
         fk(jj,k)=qcap(i,j,k)*acc
      enddo
   enddo
enddo
...
...
!  idem per l'altro loop
```

Array syntax

- Traslazione dei primi due indici di un array a tre indici (512^3)
- Il “caro vecchio” loop (stile Fortran 77): 0.19 secondi
  - l’ordine dei cicli negli indici che traslano é inverso alla traslazione per evitare di “sporcare” i dati della matrice

```
do k = nd, 1, -1
  do j = nd, 1, -1
    do i = nd, 1, -1
      a03(i,j,k) = a03(i-1,j-1,k)
    enddo
  enddo
enddo
```

- Array syntax (stile Fortran 90): 0.75 secondi
  - secondo lo standard, le cose vanno “come se il membro a destra fosse tutto valutato prima di effettuare le operazioni richieste”

```
a03(1:nd, 1:nd, 1:nd) = a03(0:nd-1, 0:nd-1, 1:nd)
```

- Array syntax con un hint al compilatore: 0.19 secondi

```
a03(nd:1:-1,nd:1:-1,nd:1:-1) = a03(nd-1:0:-1, nd-1:0:-1, nd:1:-1)
```

Report di ottimizzazione

- Per capirne di piú in questo come in altri casi é bene attivare le flag di report di ottimizzazione
- Con Intel ifort attivare

```
-opt-report[n]     n=0(none),1(min),2(med),3(max)
-opt-report-file<file>
-opt-report-phase<phase>
-opt-report-routine<routine>
```

- Le tre modalitá sono alle righe 55,64,69 rispettivamente

```
Loop at line:55 simple MEMOP Intrinsic disabled-->SIMPLE reroll
Loop at line:64 memcopy generated
Loop at line:69 simple MEMOP Intrinsic disabled-->SIMPLE reroll
```

- Tutto questo é ovviamente molto dipendente dal compilatore

Report di ottimizzazione / 2

- Purtroppo non é presente un opzione equivalente con compilatori GNU
  - La migliore alternativa é specificare

    ```
    -fdump-tree-all
    ```

    in modo che vengano stampate tutte le fase intermedie di compilazione
  - ma la lettura non é decisamente agevole
- Con il compilatore PGI

```
-Minfo=accel,inline,ipa,loop,lre,mp,opt,par,unified,vect
```

oppure senza opzioni per averle tutte

Dare informazioni al compilatore

- Estremi del loop noti a compile time o solo a run-time:
  - puó inibire alcune ottimizzazioni, tra cui l'unrolling

```
real a(1:1024,1:1024)
real b(1:1024,1:1024)
real c(1:1024,1:1024)
...
read(*,*) i1,i2
read(*,*) j1,j2
read(*,*) k1,k2
...
do j = j1, j2
do k = k1, k2
do i = i1, i2
c(i,j)=c(i,j)+a(i,k)*b(k,j)
enddo
enddo
enddo
```

- Tempi in secondi (Loop Bounds Compile-Time o Run-Time)

Compilazione	LB-CT	LB-RT
(PLX) ifort -O0	6.72	9
(PLX) ifort -fast	0.34	0.75

- Molto dipendente dal tipo di loop, dal compilatore, etc.

Allocazione statica o dinamica ?

- Potenzialmente l'allocazione statica puó dare al compilatore piú informazioni per ottimizzare
  - a prezzo di un codice piú rigido
  - l'elasticitá permessa dall'allocazione dinamica é particolarmente utile nel calcolo parallelo

```
integer :: n
parameter(n=1024)
real a(1:n,1:n)
real b(1:n,1:n)
real c(1:n,1:n)
```

```
real, allocatable, dimension(:,:) :: a
real, allocatable, dimension(:,:) :: b
real, allocatable, dimension(:,:) :: c
print*,'Enter matrix size'
read(*,*) n
allocate(a(n,n),b(n,n),c(n,n))
```

Allocazione statica o dinamica ? / 2

- Per i compilatori recenti peró spesso le prestazioni statica vs dinamica si equivalgono
  - per il semplice matrice-matrice l'allocazione dinamica gestisce meglio i loop bounds letti da input

Compilazione	statica	dinamica	dinamica-LBRT
(PLX) ifort -O0	6.72	18.26	18.26
(PLX) ifort -fast	0.34	0.35	0.36

- L'allocazione statica viene fatta nella memoria cosiddetta "stack"
  - in compilazione possono esserci dei limiti di utilizzo per cui occorre specificare l'opzione **-mcmodel=medium**
  - a run-time assicurarsi che sul nodo la stack non sia limitata (se si usa bash)

```
ulimit -a
```

ed eventualmente

```
ulimit -s unlimited
```

Allocazione dinamica in C

- C non conosce matrici ma array di array
  - l'allocazione statica garantisce allocazione contigua di tutti i valori

```
   double A[nrows][ncols];
```

- Con l'allocazione dinamica occorre fare attenzione
  - "the wrong way" (= non efficiente)

```
/* Allocate a double matrix with many malloc */
double** allocate_matrix(int nrows, int ncols) {
   double **A;
   /* Allocate space for row pointers */
   A = (double**) malloc(nrows*sizeof(double*) );
   /* Allocate space for each row */
   for (int ii=1; ii<nrows; ++ii) {
      A[ii] = (double*) malloc(ncols*sizeof(double));
   }
   return A;
}
```

Allocazione dinamica in C / 2

- Si puó allocare un array lineare

```
/* Allocate a double matrix with one malloc */
double* allocate_matrix_as_array(int nrows, int ncols) {
   double *arr_A;
   /* Allocate enough raw space */
   arr_A = (double*) malloc(nrows*ncols*sizeof(double));
   return arr_A;
}
```

- e usarlo come una matrice (linearizzazione dell'indice)

```
arr_A[i*ncols+j]
```

- le MACROs possono aiutare
- e, eventualmente aggiungere una matrice di puntatori che puntano all'array allocato

```
/* Allocate a double matrix with one malloc */
double** allocate_matrix(int nrows, int ncols, double* arr_A) {
   double **A;
   /* Prepare pointers for each matrix row */
   A = new double*[nrows];
   /* Initialize the pointers */
   for (int ii=0; ii<nrows; ++ii) {
      A[ii] = &(arr_A[ii*ncols]);
   }
   return A;
}
```

Aliasing e Restrict

- In C, se due puntatori puntano ad una stessa area di memoria, si parla di “aliasing”
- Il rischio di aliasing puó **molto** limitare l’ottimizzazione del compilatore
  - difficile invertire l’ordine delle operazioni
  - particolarmente per gli argomenti passati a una funzione
- Lo standard C99 introduce la keyword **`restrict`** per indicare che l’aliasing non é possibile

```
void saxpy(int n, float a, float *x, float* restrict y)
```

- In C++, si assume che l’aliasing non possa avvenire tra puntatori a tipi diversi (strict aliasing)

Aliasing e Restrict / 2

- Il Fortran assume che gli argomenti di procedure non possano puntare a identiche aree di memoria
  - tranne che per gli array per i quali gli indici permettono comunque un'analisi corretta
  - o per i **`pointer`** che peró vengono usati ove necessario
  - un motivo per cui il Fortran spesso ottimizza meglio del C!
- I compilatori permettono di configurare le assunzioni dell'aliasing (vedere il man)
  - GNU (solo strict-aliasing): **`-fstrict-aliasing`**
  - Intel (eliminazione completa): **`-fno-alias`**
  - IBM (no overlap per array): **`-qalias=noaryovrlp`**

Non solo compilatore

- Il compilatore ha associata una runtime library
- Contiene funzioni chiamate esplicitamente
  - funzioni trigonometriche e trascendenti
  - manipolazioni di bit
  - funzioni Input Output (C)
- Contiene funzioni chiamate implicitamente
  - funzioni Input Output (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

Input/Output

- È sempre mediato dal sistema operativo
  - causa chiamate di sistema
  - comporta lo svuotamento della pipeline
  - distrugge la coerenza dei dati in cache
  - può alterare la priorità di scheduling
  - è lento
- Regolo d'oro n.1: MAI mescolare calcolo intensivo con I/O
- Regolo d'oro n.2: leggere/scrivere i dati in blocco, non pochi per volta
- Attenzione ad I/O nascosti: swapping
  - avviene quando la RAM è insufficiente
  - usa il disco come surrogato
  - unica soluzione: fuggirlo come la peste

Ci sono più modi di fare I/O


```
do k=1,n   ;  do j=1,n   ;  do i=1,n
write(69,*) a(i,j,k)                                       ! formattato
enddo      ;  enddo      ;  enddo

do k=1,n   ;  do j=1,n   ;  do i=1,n
write(69) a(i,j,k)                                         ! binario
enddo      ;  enddo      ;  enddo

do k=1,n   ;  do j=1,n
write(69) (a(i,j,k),i=1,n)                                 ! colonne
enddo      ;  enddo

do k=1,n
write(69) ((a(i,j,k),i=1),n,j=1,n)                         ! matrice
enddo

write(69) (((a(i,j,k),i=1,n),j=1,n),k=1,n)                 ! blocco

write(69) a                                                ! dump
```

Con prestazioni differenti

Opzione	secondi	Kbyte
formattato	81.6	419430
binario	81.1	419430
colonne	60.1	268435
matrice	0.66	134742
blocco	0.94	134219
dump	0.66	134217

- Il file-system e anche il suo utilizzo hanno un notevole impatto sui tempi

I/O riassumendo

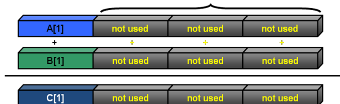
- La lettura/scrittura dei dati formattati è lenta
- Leggere/scrivere i dati in formato binario
- Leggere/scrivere in un blocco e non uno per volta
- Scegliere il file system più efficiente a disposizione
- I buffer di scrittura possono nascondere latenze
- Ma l'impatto sul calcolo sarà comunque devastante
- Attenzione al dump di array in caso di padding
- Soprattutto per il calcolo parallelo:
  - usare librerie di I/O: MPI-I/O, HDF5, NetCDF,...

Unità vettoriali

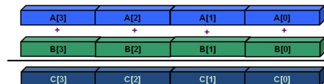
- Da non confondere con le macchine vettoriali!
- Le unità vettoriali lavorano con set di istruzioni SIMD e circuiti dedicati a operazioni floating-point simultanee
  - Intel MMX (1996), AMD 3DNow! (1998), Intel SSE (1999) che aggiungono nuovi registri e possibilità floating point
  - Nuove istruzioni (packet) SSE2, SSE3, SSE4, AVX
- Esempio di vettorizzazione: addizione di due array a 4 componenti puó diventare una singola istruzione

```
c(0) = a(0) + b(0)
c(1) = a(1) + b(1)
c(2) = a(2) + b(2)
c(3) = a(3) + b(3)
```

non vettorizzato



vettorizzato



- L'esecuzione SSE é sincrona

Evoluzione unità SIMD

- SSE: registri a 128 bit (Intel Core - AMD Opteron)
  - 4 operazioni floating/integer in singola precisione
  - 2 operazioni floating/integer in doppia precisione
- AVX: registri a 256 bit (Intel Sandy Bridge - AMD Bulldozer)
  - 8 operazioni floating/integer in singola precisione
  - 4 operazioni floating/integer in doppia precisione
- MIC: registri a 512 bit (Intel Knights Corner - 2013)
  - 16 operazioni floating/integer in singola precisione
  - 8 operazioni floating/integer in doppia precisione

CINECA SCAI SuperComputing Applications and Innovation — SAPIENZA Università di Roma

Vettorizzazione

- La vettorizzazione dei loop può incrementare drammaticamente le performance
- Ma per essere vettorizzabili, i loop devono obbedire a certi criteri
- E il programmatore deve aiutare il compilatore a verificarli
- Anzitutto, l'assenza di dipendenza tra i dati di diverse iterazioni
  - circostanza frequente ma non troppo in ambito HPC
- Altri criteri
  - Countable (numero delle iterate costante)
  - Single entry-single exit (nessun break or exit)
  - Straight-line code (nessun branch)
  - Deve essere il loop interno di nest
  - Nessuna chiamata a funzione (eccetto quelle matematiche o quelle inlined)
- AVX può essere una sorgente di risultati diversi in calcolo numerico (e.g., Fused Multiply Addiction)

Algoritmi e vettorizzazione

- Differenti algoritmi per lo stesso scopo possono comportarsi diversamente rispetto alla vettorizzazione
  - Gauss-Seidel: dipendenza tra le iterazioni, non vettorizzabile

```
for( i = 1; i < n-1; ++i )
  for( j = 1; j < m-1; ++j )
    a[i][j] = w0 * a[i][j] +
      w1*(a[i-1][j] + a[i+1][j] + a[i][j-1] + a[i][j+1]);
```

- Jacobi: nessuna dipendenza tra le iterazioni, vettorizzabile

```
for( i = 1; i < n-1; ++i )
  for( j = 1; j < m-1; ++j )
     b[i][j] = w0*a[i][j] +
       w1*(a[i-1][j] + a[i][j-1] + a[i+1][j] + a[i][j+1]);
for( i = 1; i < n-1; ++i )
  for( j = 1; j < m-1; ++j )
     a[i][j] = b[i][j];
```

- Alcuni comuni “coding tricks” possono impedire la vettorizzazione
  - vettorizzabile

```
for( i = 0; i < n-1; ++i ){
   b[i] = a[i] + a[i+1];
}
```

  - `x` a una certa iterazione é necessaria per lo step successivo

```
x = a[0];
for( i = 0; i < n-1; ++i ){
  y = a[i+1];
  b[i] = x + y;
  x = y;
}
```

- Quando si compila é bene controllare se la vettorizzazione é stata attivata
- In caso contrario, si puó provare ad aiutare il compilatore
  - modificando il codice per renderlo vettorizzabile
  - inserendo direttive per forzare la vettorizzazione

Direttive di vettorizzazione

- Se il programmatore ha certezza che una certa dipendenza riscontrata dal compilatore sia in realtà solo apparente può forzare la vettorizzazione con direttive “compiler dependent”
  - Intel Fortran: **`!DIR$ simd`**
  - Intel C: **`#pragma simd`**
- Poichè **`inow`** è diverso da **`inew`**, la dipendenza è solo apparente

```
62      do k = 1,n
63!DIR$ simd
         do i = 1,l
...
66            x02 = a02(i-1,k+1,inow)
67            x04 = a04(i-1,k-1,inow)
68            x05 = a05(i-1,k  ,inow)
              x06 = a06(i  ,k-1,inow)
              x11 = a11(i+1,k+1,inow)
              x13 = a13(i+1,k-1,inow)
72            x14 = a14(i+1,k  ,inow)
73            x15 = a15(i  ,k+1,inow)
74            x19 = a19(i  ,k  ,inow)
75
76            rho =+x02+x04+x05+x06+x11+x13+x14+x15+x19
...
126           a05(i,k,inew) = x05 - omega*(x05-e05) + force
127           a06(i,k,inew) = x06 - omega*(x06-e06)
```

Hands-on: Vettorizzazione

- Confrontare le performance con o senza vettorizzazione del loop presente nel programma `simple_loop.c` o `simple_loop.f90`
  - usare `-O3` per abilitare la vettorizzazione
  - trovare nel man l'opzione per disabilitare la vettorizzazione
- Il programma `vectorization_test.c/f90` contiene 18 loops con differenti condizioni paradigmatiche
  - provare a predire quale loop è vettorizzabile, quale no e le relative cause
  - verificare le proprie assunzioni con il compilatore GNU, attivando le ottimizzazioni e il reporting `-O3 -ftree-vectorizer-verbose=2`
  - ripetere con il compilatore Intel e le opzioni `-fast -opt-report3 -vec-report3`, e PGI?
  - qualche idea per rendere qualche loop vettorizzabile?
  - riprovare aggiungendo l'opzione per GNU `-unsafe-math-optimizations` o `-ffast-math`

Hands-on: Vettorizzazione / 2

	PGI	Intel
Vectorized time		
Non-Vectorized time		

# Loop	# Description	Vect/Not	PGI	Intel
1	Simple			
2	Short			
3	Previous			
4	Next			
5	Double write			
6	Reduction			
7	Function bound			
8	Mixed			
9	Branching			
10	Branching-II			
11	Modulus			
12	Index			
13	Exit			
14	Cycle			
15	Nested-I			
16	Nested-II			
17	Function			
18	Math-Function			

Hands-on: Risultati

	PGI	Intel
Vectorized time	6.32	4.05
Non-Vectorized time	12.16	5.58

# Loop	Description	PGI	Intel
1	Simple	yes	yes
2	Short	no: unrolled	yes
3	Previous	no: data dep.	no: data dep.
4	Next	yes: how?	yes: how?
5	Double write	no: data dep.	no: data dep.
6	Reduction	yes	? ignored
7	Function bound	yes	yes
8	Mixed	no: mixed type	yes
9	Branching	? ignored	yes
10	Branching-II	? ignored	yes
11	Modulus	no: mixed type	no: inefficient
12	Index	no: mixed type	yes
13	Exit	no: exits	no: exits
14	Cycle	? ignored	yes
15	Nested-I	yes	yes
16	Nested-II	yes	yes
17	Function	no: function call	yes
18	Math-Function	yes	yes

Vettorizzazione manuale

- É possibile istruire direttamente il codice delle funzioni vettoriali da utilizzare
- In pratica, si tratta di scrivere un loop che fa quattro iterazioni alla volta usando registri e operazioni vettoriali, ed essere pratici con le “mask”

```
void scalar(float* restrict result,
            const float* restrict v,
            unsigned length)
{
  for (unsigned i = 0; i < length; ++i)
  {
    float val = v[i];
    if (val >= 0.f)
      result[i] = sqrt(val);
    else
      result[i] = val;
  }
}
```

```
void sse(float* restrict result,
         const float* restrict v,
         unsigned length)
{
  __m128 zero = _mm_set1_ps(0.f);

  for (unsigned i = 0; i <= length - 4; i += 4)
  {
    __m128 vec  = _mm_load_ps(v + i);
    __m128 mask = _mm_cmpge_ps(vec, zero);
    __m128 sqrt = _mm_sqrt_ps(vec);
    __m128 res  =
         _mm_or_ps(_mm_and_ps(mask, sqrt),
         _mm_andnot_ps(mask, vec));
    _mm_store_ps(result + i, res);
  }
}
```

Parallelizzazione automatica

- Alcuni compilatori offrono opzioni per sfruttare il parallelismo architetturale delle macchina (e.g., i cores) senza modificare il codice sorgente
- Shared Memory Parallelism (solo intra-nodo)
- Simile a OpenMP ma non richiede direttive
  - performance attese piú limitate
- GNU:

```
-floop-parallelize-all
-ftree-parallelize-loops=n - n= number of threads
```

- Intel:

```
-parallel
-par-threshold[n] - set loop count threshold
-par-report{0|1|2|3}
```

- IBM:

```
-qsmp                   la abilita automaticamente
-qsmp=openmp:noauto     per disabilitare la
                        parallelizzazione automatica
```

Rights & Credits

These slides are ©CINECA 2013 and are released under the Attribution-NonCommercial-NoDerivs (CC BY-NC-ND) Creative Commons license, version 3.0.

Uses not allowed by the above license need explicit, written permission from the copyright owner. For more information see:

`http://creativecommons.org/licenses/by-nc-nd/3.0/`