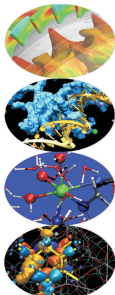


# HPC Scientific programming: tools and techniques

P. Lanucara V. Ruggiero  
F. Salvatore

CINECA Roma - SCAI Department

Roma, 9-11 April 2013





## 9 aprile 2013

9.30-10.30 Architetture

10.30-13.00 La cache ed il sistema di memoria + esercitazioni

14.00-15.00 Pipeline + esercitazioni

15.00-17.00 Profilers + esercitazioni

## 10 aprile 2013

9.30-13.00 Compilatori+esercitazioni

14.00-15.30 Librerie + esercitazioni

15.00-17.00 Floating-point +esercitazioni

## 11 aprile 2013

9.30-11.00 Makefile + esercitazioni

11.00-13.00 Debugging+esercitazioni

14.00-17.00 Debugging+esercitazioni



Compilatori e ottimizzazione

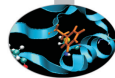
Librerie scientifiche

Floating Point Computing

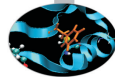


- ▶ Esiste un'infinità di linguaggi differenti
- ▶ <http://foldoc.org/contents/language.html>

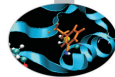
20-GATE; 2.PAK; 473L Query; 51forth; A#; A-0; a1; a56;  
 Abbreviated Test Language for Avionics Systems; ABC;  
 ABC ALGOL; ABCL/1; ABCL/c++; ABCL/R; ABCL/R2; ABLE;  
 ABSET; abstract machine; Abstract Machine Notation;  
 abstract syntax; Abstract Syntax Notation 1;  
 Abstract-Type and Scheme-Definition 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-0; 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 interpretati
  - ▶ il linguaggio viene "tradotto" statement per statement dall'interprete durante l'esecuzione
  - ▶ impossibili ottimizzazioni tra differenti statement
  - ▶ migliore gestione degli errori semantici
  - ▶ linguaggi di scripting, Java (bytecode), . . .
  
- ▶ Linguaggi compilati
  - ▶ il programma viene "tradotto" dal compilatore prima dell'esecuzione
  - ▶ possibili ottimizzazioni tra differenti statement
  - ▶ gestione minimale degli errori semantici
  - ▶ Fortran, C, C++



- ▶ **É 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



- ▶ **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

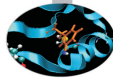


- ▶ **Architettura:**
  - ▶ set di istruzioni
  - ▶ registri architetturali interi, floating point e di stato
- ▶ **Implementazione**
  - ▶ registri fisici ( $2.5 \div 20 \times$  registri architetturali)
  - ▶ frequenza di clock e tempo di esecuzione delle istruzioni
  - ▶ numero di unità funzionali
  - ▶ 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





- ▶ 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 dipendenti dall'implementazione della CPU
  - ▶ ottimizzazioni dell'uso della memoria e della cache
  - ▶ suggerimenti al processore su cosa probabilmente farà il codice
- ▶ É uno strumento potente
  - ▶ 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

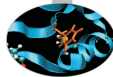


- ▶ Linguaggi ideali per l'High Performance Computing?
  - ▶ adatti all'implementazione di algoritmi "scientifici"
  - ▶ devono permettere elevati livelli di ottimizzazione
  - ▶ integrandosi nelle recenti architetture di supercalcolo
- ▶ Quali sono?
  - ▶ Fortran/C/C++, ci limitiamo a Fortran e C con qualche cenno specifico a C++
  - ▶ il Python é in ascesa, ma per le parti computazionalmente intensive si usa appoggiarsi a C o Fortran
- ▶ I compilatori piú 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



- ▶ Creare un eseguibile dai sorgenti é in generale un processo a tre fasi
- ▶ Pre-processing:
  - ▶ ogni sorgente é letto dal pre-processore
    - ▶ sostituire (**#define**) MACROs
    - ▶ inserire codice per gli statement **#include**
    - ▶ inserire o cancellare codice valutando **#ifdef**, **#if** ...
- ▶ Compilazione:
  - ▶ ogni sorgente é tradotto in un codice oggetto
    - ▶ un file oggetto é 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 dá il comando:

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

vengono eseguiti automaticamente i tre passi

- ▶ Pre-processing

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

- ▶ l'opzione `-E -cpp` dice a `gfortran` di fermarsi after pre-process
- ▶ semplicemente chiama `cpp` (automaticamente chiamata se l'estensione é `f90`)

- ▶ Compilazione dei sorgenti

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

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



- ▶ Linkare oggetti tra di loro

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

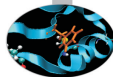
- ▶ Per risolvere i simboli definiti in librerie esterne specificare:
  - ▶ le librerie da usare (opzione `-l`)
  - ▶ le directory in cui stanno (opzione `-L`)
- ▶ Come linkare `libblas.a` nella cartella `/opt/lib`

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

- ▶ Come creare e linkare una libreria statica

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

- ▶ `ar` create the archive `libdsp.a` containing `dsp.o`
- ▶ `ranlib` generare un indice per l'archiviazione



- Il manuale in linea, e.g.

**man gcc**

riporta le opzioni del compilatore C di GNU e il loro significato

- Attenzione: **man gfortran** dá solo le opzioni ulteriori rispetto a gcc, mentre per gli altri compilatori solitamente i man sono replicati nelle parti comuni
- Il numero di opzioni é 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 piú importante, **-g**, crea i simboli di debugging necessari per l'uso di debugger
  - warning: per avere informazioni sulla compilazione, utile per debugging e/o ottimizzazione



- ▶ 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



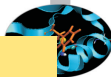
- ▶ Analizzare ed ottimizzare globalmente codici molto grossi (a meno di abilitare l'IPO, molto costoso in tempo e risorse)
- ▶ Capire dipendenze tra dati con indirizzamenti indiretti
- ▶ Strenght reduction di potenze non intere, o maggiori di  $2 \div 4$
- ▶ Common subexpression elimination attraverso chiamate a funzione
- ▶ Unrolling, Merging, Blocking con:
  - ▶ chiamate a funzioni e(o) subroutine
  - ▶ 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 di `-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 `O3` e altro
- ▶ Per GNU una scelta comune insieme a `-O3` è `-funroll-loops`
- ▶ Attenzione all'ottimizzazione di default: per GNU è `-O0` mentre per gli altri di solito è `-O2`



## `icc (or ifort) -O3`

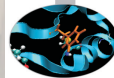
- ▶ Automatic vectorization (use of packed SIMD instructions)
- ▶ Loop interchange (for more efficient memory access)
- ▶ Loop unrolling (more instruction level parallelism)
- ▶ Prefetching (for patterns not recognized by h/w prefetcher)
- ▶ Cache blocking (for more reuse of data in cache)
- ▶ Loop peeling (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)



- ▶ 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



- ▶ 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)



- ▶ 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
```

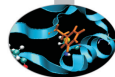
- ▶ Il file di script viene sottomesso in coda con il comando:  
`qsub qsub.script`
- ▶ La coda private permette di accedere ai nodi `fat`



- ▶ 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: Livelli di Ottimizzazione



- ▶ Prodotto matrice-matrice,  $1024 \times 1024$ , 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	—			—		

# Prodotto di matrici: performance



- ▶ Prodotto matrice-matrice,  $1024 \times 1024$ , 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

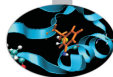
- ▶ Perché tanta varietà di risultati?
- ▶ Basta passare da -On a -On+1?



# Cosa fa il compilatore: report



- ▶ Cosa accade ai diversi livelli di ottimizzazione?
  - ▶ Perché il compilatore IBM su Fermi al livello **-O4** degrada così 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 **\_\_x1\_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 é bene testare questo miglioramento per il proprio codice



## ► Esempio datato, utile però per capire

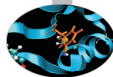
### Matrix Multiply inner loop code with -qnoot

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
  
```



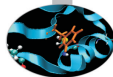
## Matrix Multiply inner loop code with -qnoot

### 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
  
```



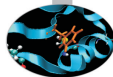
## Matrix Multiply inner loop code with -qnoot

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
    
```

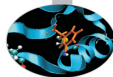


## Matrix Multiply inner loop code with -qnoot

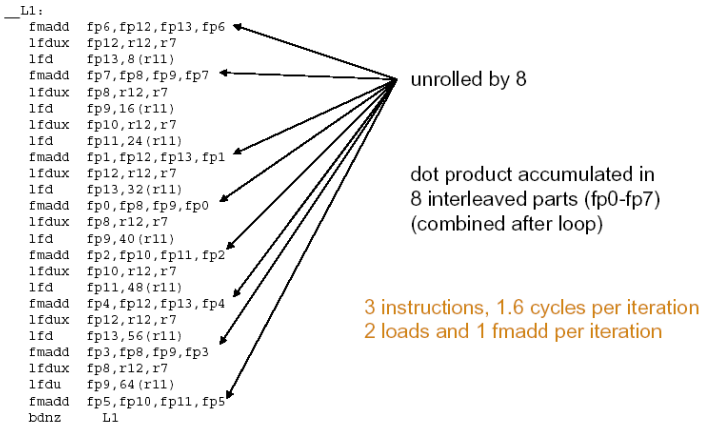
necessary instructions	loop control	addressing code
<u>__L1:</u>		lwz r7,156(SP)
lwz r3,160(SP)		lwz r10,12(r9)
lwz r9,STATIC_BSS		subfi r9,r10,-8
lwz r4,24(r9)		mullw r10,r10,r11
subfi r5,r4,-8		rlwinm r8,r8,3,0,28
lwz r11,40(r9)		add r9,r9,r10
mullw r6,r4,r11		add r8,r8,r9
lwz r4,36(r9)		lfdx fp3,r7,r8
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



- ▶ 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?



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





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

```

_L1:
fmadd    fp1, fp4, fp2, fp1
fmadd    fp0, fp3, fp5, fp0
lfd      fp2, r29, r9
lfd      fp4, 32(r30)
fmadd    fp10, fp7, fp28, fp10
fmadd    fp7, fp9, fp7, fp8
lfd      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
lfd      fp9, r29, r9
lfd      fp7, 32(r28)
fmadd    fp31, fp11, fp5, fp31
fmadd    fp6, fp26, fp11, fp6
lfd      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 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)) {
    .....
}
  
```

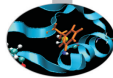
- ▶ Redundant code removal

```

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 → 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

# Il compilatore: 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

# Il compilatore: index reordering / 2



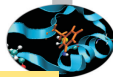
- ▶ Per loop nesting piú complicati il compilatore a volte no...
  - ▶ anche al piú alto livello di ottimizzazione
  - ▶ conoscere il meccanismo di cache é quindi utile!

```
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

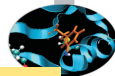
# Modularizzazione e cache



```
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



```
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 é richiesto l'"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 capacità 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):

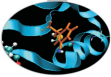
```
-finline-functions  
-finline-limit=n
```

- ▶ In alcuni compilatori automaticamente attivate ad alti livelli di ottimizzazione



- ▶ 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 é necessariamente identico
- ▶ Se la locazione di un array é acceduta piú 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, però non é così
  - ▶ 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  $f(x) + f(x) + f(x)$  può essere valutato come  $3 * f(x)$

- ▶ Se:

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

allora la precedente valutazione non é più corretta



- ▶ 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);
```

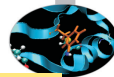
- ▶ 4 chiamate a funzioni, 4 prodotti (1 variabile temporanea):

```
temp=r*sin(a)  
x=temp*cos(b);  
y=temp*sin(b);  
z=r*cos(a);
```

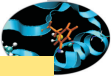


- ▶ Core loop troppo grossi:
  - ▶ il compilatore lavora su finestre di dimensioni finite: potrebbe non accorgersi di una grandezza da riutilizzare
- ▶ Funzioni:
  - ▶ se altero l'ordine delle chiamate ottengo lo stesso risultato?
- ▶ Ordine e valutazione:
  - ▶ solo ad alti livelli di ottimizzazione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

# Cosa può fare il compilatore?



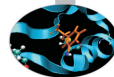
```
do k=1,n3m
  do j=n2i,n2do
    jj=my_node*n2do+j
    do i=1,n1m
      acc =1. / (1.-coe*aciv(i) * (1.-int (forclo (nve,i,j,k))) )
      aci (jj,i)= 1.
      api (jj,i)=-coe*apiv(i) *acc* (1.-int (forclo (nve,i,j,k)))
      ami (jj,i)=-coe*amiv(i) *acc* (1.-int (forclo (nve,i,j,k)))
      fi (jj,i)=qcap(i,j,k)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



```

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

```

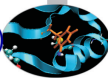


```

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)

```



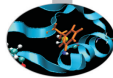
- ▶ Per capirne di piú in questo come in altri casi é bene attivare le flag di report di ottimizzazione
- ▶ Con Intel ifort attivare

```
-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 modalità 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



- ▶ 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:
  - può 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 può dare al compilatore più informazioni per ottimizzare
  - ▶ a prezzo di un codice più rigido
  - ▶ l'elasticità 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 però 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 **-mcmmodel=medium**
  - ▶ a run-time assicurarsi che sul nodo la stack non sia limitata (se si usa bash)

```
ulimit -a
```

ed eventualmente

```
ulimit -s unlimited
```



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

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

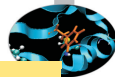
- Con l'allocazione dinamica occorre fare attenzione
  - "the wrong way" (= non efficiente)

```

/* 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 può 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;
}
```





- ▶ In C, se due puntatori puntano ad una stessa area di memoria, si parla di “aliasing”
- ▶ Il rischio di aliasing può **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)



- ▶ 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 però 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**



- ▶ 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



- ▶ 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

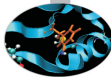
do k=1,n ; do j=1,n ; do i=1,n
write(69) a(i,j,k)                                    ! binario
enddo      ; enddo      ; enddo

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
  
```



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



- ▶ 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,...



- ▶ 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 può diventare una singola istruzione

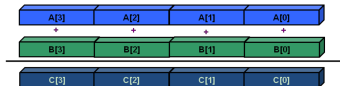
$$\begin{aligned}
 c(0) &= a(0) + b(0) \\
 c(1) &= a(1) + b(1) \\
 c(2) &= a(2) + b(2) \\
 c(3) &= a(3) + b(3)
 \end{aligned}$$

non vettorizzato

e.g. 3 x 32-bit unused integers



vettorizzato



- ▶ L'esecuzione SSE é sincrona

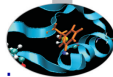




- ▶ **SSE:** registri a 128 bit (intel Core - AMD Opteron)
  - ▶ 4 operazioni floating/integer in singola precisione
  - ▶ 2 operazioni floating/integer in doppia precisione
  
- ▶ **AVX:** registri a 256 bit (intel Sandy Bridge - AMD Bulldozer)
  - ▶ 8 operazioni floating/integer in singola precisione
  - ▶ 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



- ▶ 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 a meno di implementazioni come assegnazione di mask)
  - ▶ 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 Addition)



- ▶ 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

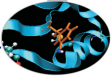
```
for( i = 0; i < n-1; ++i ){  
    b[i] = a[i] + a[i+1];  
}
```

- ▶ **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 può provare ad aiutare il compilatore
  - ▶ modificando il codice per renderlo vettorizzabile
  - ▶ inserendo direttive per forzare la vettorizzazione

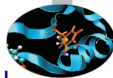
# Direttive di vettorizzazione



- ▶ 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**
- ▶ Poiché **inow** é diverso da **inew**, la dipendenza é 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)
  
```



- ▶ Confrontare le performance con e senza vettorizzazione del loop presente nel programma `simple_loop.f90` con i compilatori PGI e Intel
  - ▶ usando **-fast**, per disabilitare la vettorizzazione specificare le opzioni **-Mnovect** or **-no-vec**, rispettivamente
- ▶ Il programma `vectorization_test.f90` contiene 18 loops con differenti condizioni paradigmatiche
  - ▶ provare a predire quale loop é vettorizzabile, quale no e le relative cause
  - ▶ verificare le proprie assunzioni con il compilatore PGI, attivando le ottimizzazioni e il reporting **-fast -Minfo**
  - ▶ ripetere con il compilatore Intel e le opzioni **-fast -opt-report3 -vec-report3**
  - ▶ qualche idea per rendere qualche loop vettorizzabile?
  - ▶ per inciso, usando GNU l'opzione é **-ftree-vectorizer-verbose=n**

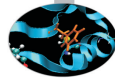
# Hands-on: Vettorizzazione / 2



	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			

# 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”

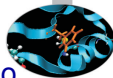
```
void scalar(float* restrict result,
           const float* restrict v,
           unsigned length)
{
    for (unsigned i = 0; i < length; ++i)
    {
        float val = v[i];
        if (val >= 0.f)
            result[i] = sqrt(val);
        else
            result[i] = val;
    }
}
```

```
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 più limitate
- ▶ Intel:

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

- ▶ IBM:

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

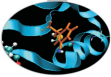


Compilatori e ottimizzazione

Librerie scientifiche

Floating Point Computing

# Librerie statiche e dinamiche



- ▶ Una libreria può essere statica o dinamica
  - ▶ entrambe sono richieste in fase di compilazione aggiungendo  
**-L<directory\_libreria> -l<nome\_libreria>**
- ▶ Libreria statica:
  - ▶ estensione **.a**
  - ▶ tutti i simboli oggetti della libreria vengono inclusi nell'eseguibile al momento del linking
  - ▶ se si crea una libreria che si appoggia ad un'altra non include i suoi simboli: l'eseguibile dovrà linkare tutte le librerie in cascata
  - ▶ eseguibile più efficiente
- ▶ Libreria dinamica:
  - ▶ estensione **.so**
  - ▶ richiede di specificare la directory in cui cercare la librerie in fase di esecuzione (per esempio settando la variabile di ambiente **LD\_LIBRARY\_PATH**)
  - ▶ **ldd <nome\_eseguibile>** dice le librerie dinamiche richieste dall'eseguibile
  - ▶ eseguibile più snello



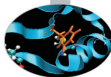
- ▶ Le librerie sono insiemi di funzioni che implementano una varietà di algoritmi, spesso numerici.
- ▶ Operazioni aritmetiche di basso livello (e.g. prodotto scalare o numeri casuali), ma anche algoritmi più complicati (trasformata di Fourier o diagonalizzazione di matrici)
- ▶ Le performance delle migliori librerie sono difficilmente superabili da un utente ordinario
- ▶ A volte ottimizzate in assembler
- ▶ Libere o proprietarie
- ▶ Occorre fare attenzione ad utilizzare la migliore libreria possibile per una certa coppia compilatore-libreria



- ▶ **Vantaggi:**
  - ▶ migliorano la modularità
  - ▶ standardizzazione
  - ▶ portabilità
  - ▶ efficienza
  - ▶ pronte all'uso
  
- ▶ **Svantaggi:**
  - ▶ dettagli nascosti
  - ▶ spesso non si sa cosa si usa
  - ▶ troppa fiducia nell'implementazione



- ▶ Difficile avere una panoramica completa
  - ▶ tante tipologie
  - ▶ e uno scenario in continua evoluzione
  - ▶ anche per le nuove architetture in arrivo (GPU)
  
- ▶ Tipologie di uso comune
  - ▶ algebra lineare
  - ▶ fft
  - ▶ input-output
  - ▶ calcolo parallelo
  - ▶ mesh decomposition
  - ▶ suite



- ▶ Per applicazioni massive é cruciale il tipo di parallelizzazione
  - ▶ alcune sono già fornite multi-threaded o anche parallele a memoria distribuita
- ▶ Memoria condivisa
  - ▶ BLAS
  - ▶ GOTOBLAS
  - ▶ LAPACK/CLAPACK/LAPACK++
  - ▶ ATLAS
  - ▶ PLASMA
  - ▶ SuiteSparse
- ▶ Memoria distribuita
  - ▶ Blacs (solo suddivisione)
  - ▶ ScaLAPACK
  - ▶ PSBLAS
  - ▶ Elemental



## ► BLAS: Basic Linear Algebra Subprograms

- la Basic Linear Algebra Subprograms é tra le prime librerie scritte (1979), in origine per calcolatori con architettura vettoriale
- comprende operazioni elementari tra vettori e matrici come il prodotto scalare e la moltiplicazione tra scalari, vettori, matrici, anche in forma trasposta
- é utilizzata da numerose librerie di livello piú alto, perciò ne sono state prodotte diverse versioni, ottimizzate per varie piattaforme di calcolo

## ► 3 livelli

- BLAS liv. 1 subroutine Fortran per il calcolo di operazioni di base scalare-vettore. Sono la conclusione di un progetto terminato nel 1977
- BLAS liv. 2 operazioni vettore-matrice. Scritte tra il 1984 e 1986
- BLAS liv. 3 subroutine Fortran per operazioni matrice-matrice, disponibili dal 1988





- ▶ Le subroutine BLAS si applicano a dati reali e complessi, in semplice o doppia precisione
- ▶ Operazioni scalare-vettore (  $O(n)$  )
  - ▶ SWAP scambio vettori
  - ▶ COPY copia vettori
  - ▶ SCAL cambio fattore di scala
  - ▶ NRM2 norma L2
  - ▶ AXPY somma:  $Y + A * X$
- ▶ Operazioni vettore-matrice (  $O(n^2)$  )
  - ▶ GEMV prodotto vettore/matrice generica
  - ▶ HEMV prodotto vettore/matrice hermitiana
  - ▶ SYMV prodotto vettore/matrice simmetrica



- ▶ Operazioni matrice-matrice (  $O(n^3)$  )
  - ▶ GEMM prodotto matrice/matrice generica
  - ▶ HEMM prodotto matrice/matrice hermitiana
  - ▶ SYMM prodotto matrice/matrice simmetrica
  
- ▶ GOTOBLAS
  - ▶ Kazushige Goto, un ricercatore del Texas Advanced Computing Center (University of Texas at Austin), ha ottimizzato manualmente in assembler le subroutine BLAS per diversi supercomputer



- ▶ **LAPACK: Linear Algebra PACKage**
  - ▶ evoluzione di LINPACK e EISPACK
  - ▶ soluzione di problemi di algebra lineare, tra cui sistemi di equazioni lineari, problemi di minimi quadrati, autovalori
- ▶ **ATLAS: Automatically Tuned Linear Algebra Software**
  - ▶ implementazione BLAS e di alcune routine di LAPACK efficiente grazie alla procedura di autotuning che avviene durante l'installazione
- ▶ **PLASMA: Parallel Linear Algebra Software for Multi-core Architectures**
  - ▶ soluzione di sistemi lineari, progettate per essere efficienti su processori multi-core, funzionalità simili a LAPACK ma più limitate
- ▶ **SuiteSparse**
  - ▶ collezione di pacchetti per matrici sparse



- ▶ **Calcolo di autovalori/autovettori**
  - ▶ EISPACK: calcolo di autovalori e autovettori, con versioni specializzate per matrici di diversi tipi, reali e complesse, hermitiane, simmetriche, tridiagonali
  - ▶ ARPACK: problemi agli autovalori di grandi dimensioni. La versione parallela é un'estensione della libreria classica e usa le librerie BLACS e MPI
- ▶ **Algebra lineare a memoria distribuita**
  - ▶ BLACS: linear algebra oriented message passing interface
  - ▶ ScaLAPACK: Scalable Linear Algebra PACKage
  - ▶ Elemental: framework per algebra lineare densa
  - ▶ PSBLAS: Parallel Sparse Basic Linear Algebra Subroutines
  - ▶ SLEPc: problemi agli autovalori



- ▶ Le librerie di I/O risultano particolarmente utili per
  - ▶ interoperabilità: C/Fortran, Little Endian/Big Endian,...
  - ▶ visualizzazione
  - ▶ analisi di sub-set
  - ▶ metadati
  - ▶ I/O parallelo
- ▶ HDF5: “is a data model, library, and file format for storing and managing data”
- ▶ NetCDF: “NetCDF is a set of software libraries and self-describing, machine-independent data formats that support the creation, access, and sharing of array-oriented scientific data”
- ▶ VTK: “open-source, freely available software system for 3D computer graphics, image processing and visualization”



- ▶ **MPI: Message Passing Interface**
  - ▶ standard piú diffuso per la parallelizzazione in memoria distribuita
  - ▶ implementazioni piú importanti come librerie: MPICH e OpenMPI
  
- ▶ **Decomposizione di mesh**
  - ▶ METIS e ParMETIS: “can partition a graph, partition a finite element mesh, or reorder a sparse matrix”
  - ▶ Scotch e PT-Scotch: “sequential and parallel graph partitioning, static mapping and clustering, sequential mesh and hypergraph partitioning, and sequential and parallel sparse matrix block ordering”



## ► Trilinos

- collezione di algoritmi in stile object-oriented per la soluzione di problemi scientifici e ingegneristici multi-fisica
- struttura a due livelli concepita attorno a collezione di “package” sviluppati da diversi esperti in materia (precondizionatori, solutori non lineari,...)

## ► PETSc

- strutture dati e funzioni per la soluzione su calcolatori paralleli di equazioni alle derivate parziali
- include solutori per equazioni lineari, non lineari e integratori ODE
- utilizza MPI per realizzare il parallelismo e permette di sviluppare programmi che richiedono ingenti risorse computazionali



- ▶ **MKL: Intel Math Kernel Library**
  - ▶ Major functional categories include Linear Algebra, Fast Fourier Transforms (FFT), Vector Math and Statistics. Cluster-based versions of LAPACK and FFT are also included to support MPI-based distributed memory computing.
- ▶ **ACML: AMD Core Math Library**
  - ▶ Libreria di funzioni altamente ottimizzate per processori AMD. Include tra l'altro BLAS, LAPACK, FFT, Random Generators
- ▶ **GSL: GNU Scientific Library**
  - ▶ The library provides a wide range of mathematical routines such as random number generators, special functions and least-squares fitting. There are over 1000 functions in total with an extensive test suite.
- ▶ **ESSL (IBM): Engineering and Scientific Subroutine library**
  - ▶ BLAS, LAPACK, ScaLAPACK, solutori sparsi, FFT e altro. La versione parallela utilizza MPI.

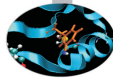




- ▶ Per utilizzare le librerie nei programmi é necessario innanzitutto che la sintassi di chiamata delle funzioni sia corretta
- ▶ Inoltre in fase di generazione dell'eseguibile é indispensabile fornire tutte le indicazioni necessarie per l'individuazione della versione corretta della libreria
- ▶ Spesso nel caso di librerie proprietarie esistono modalità di compilazione e linking specifiche
- ▶ Può non essere banale, e.g. Intel ScaLAPACK

```
mpif77 <programma> -L$MKLROOT/lib/intel64 \  
-lmkl_scalapack_lp64 -lmkl_blacs_openmpi \  
-lmkl_intel_lp64 -lmkl_intel_thread -lmkl_core \  
-liomp5 -lpthread
```

# Librerie: Interoperabilità



- ▶ Molte librerie scientifiche sono scritte in C, molte in Fortran
- ▶ Chiamare le funzioni di libreria da un linguaggio diverso dall'originario può dare qualche difficoltà
  - ▶ matching dei tipi: l'**int** del C non é garantito corrispondere all'**integer** del Fortran
  - ▶ matching dei simboli: Fortran e C++ deformano i nomi dei simboli sorgenti nel produrre gli oggetti
- ▶ Problema affrontato in modo un po' "rozzo" fino a non molto tempo fa
  - ▶ tentando di matchare i tipi e aggiungendo gli `_` necessari per matchare con gli oggetti della libreria
  - ▶ il comando **nm** `<file_oggetto>` elenca i simboli ivi contenuti
  - ▶ alcune librerie fornivano a questo scopo wrapper già pronti
- ▶ Problema affrontato in modo efficace nello standard Fortran 2003 (modulo **iso\_c\_binding**)
  - ▶ librerie più importanti forniscono le interfacce Fortran 2003
- ▶ In C++ vedere il comando **extern "C"**

# Librerie: Interoperabilità / 2



- ▶ Per chiamare librerie scritte in C dal Fortran o viceversa
- ▶ **mpi** scritta in C/C++:
  - ▶ vecchia modalità : `include "mpif.h"`
  - ▶ nuova modalità: `use mpi`
  - ▶ non sono del tutto equivalenti: usare il modulo vuol dire avere il check dei tipi a compile-time
- ▶ **fftw** scritta in C
  - ▶ legacy : `include "fftw3.f"`
  - ▶ modern:

```
use iso_c_binding
include 'fftw3.f03'
```

- ▶ di nuovo: la versione moderna offre maggiori potenzialità
- ▶ **BLAS** scritte in Fortran
  - ▶ legacy : chiamare `dgemm_`
  - ▶ modern: chiamare `cblas_dgemm`
- ▶ Purtroppo ancora poca standardizzazione
  - ▶ studiare il manuale e tentare di essere standard o almeno portabile



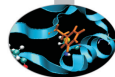
- ▶ Argomenti delle funzioni sul sito “netlib”

<http://www.netlib.org/blas/>

- ▶ Poiché le BLAS sono scritte in Fortran 77, anche dal Fortran alcuni compilatori rendono disponibili interfacce per chiamarle in sicurezza (check dei tipi) e con le features del Fortran 95 (assumed shape arrays e argomenti opzionali)
  - ▶ Con Intel e MKL

```
use mkl95_blas
```

# BLAS: Interoperabilità / 2



- ▶ C (modalità legacy):
  - ▶ aggiungere l'underscore ai nomi delle funzioni
  - ▶ poiché il Fortran passa tutti gli argomenti per reference, é necessario sempre passare i puntatori
  - ▶ assumere matching dei tipi (compiler dependent): probabilmente `double`, `int`, `char` per `double precision`, `integer`, `character`
  - ▶ ma l'ordinamento di array multidimensionali verrà trasposto!
- ▶ C (modalità moderna)
  - ▶ usare le interafcce `cbblas`: le GSL di GNU o le MKL di Intel le forniscono
  - ▶ in effetti le GSL includono anche l'implementazione delle BLAS (non solo wrapper)
  - ▶ includere l'header `#include <gsl.h>` O `#include <mk1.h>`
  - ▶ rispettare la sintassi (si può specificare l'ordinamento delle matrici)

# Hands-on: BLAS



- ▶ Sostituire il codice matrice-matrice di `matrixmul` con una chiamata a `DGEMM`, la routine BLAS che esegue il prodotto matrice-matrice in doppia precisione
- ▶ `DGEMM` esegue (l'operatore `op` consente trasposizioni)

```
C := alpha*op( A )*op( B ) + beta*C,
```

- ▶ Argomenti della `DGEMM` : <http://www.netlib.org/blas/dgemm.f>
- ▶ Fortran: GNU, BLAS efficienti sono le `acml`, nelle versioni:
  - ▶ `gfortran64` (seriali)
  - ▶ `gfortran64_mp` (multi-thread)

```
module load profile/advanced
module load gnu/4.7.2 acml/5.3.0--gnu--4.7.2
gfortran -O3 -L$ACML_HOME/gfortran64/lib/ -lacml matrixmulblas.F90
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$ACML_HOME/gfortran64/lib/
```

- ▶ Fortran: Intel, BLAS efficienti sono le proprietarie MKL
  - ▶ `sequential` (seriali)
  - ▶ `parallel` (multi-thread)

```
module load profile/advanced
module load intel/cs-xe-2013--binary
ifort -O3 -mkl=sequential matrixmulblas.F90
```



- ▶ C: compilatore Intel (MKL con cblas)
  - ▶ includere l'header file `#include<mk1.h>` nel sorgente
  - ▶ provare `-mk1=sequential` e `-mk1=parallel`

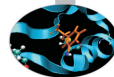
```
module load profile/advanced
module load intel/cs-xe-2013--binary
icc -O3 -mk1=sequential matrixmulblas.c
```

- ▶ C: compilatore GNU (GSL con cblas)
  - ▶ includere l'header file `#include <gsl/gsl_cblas.h>` nel sorgente

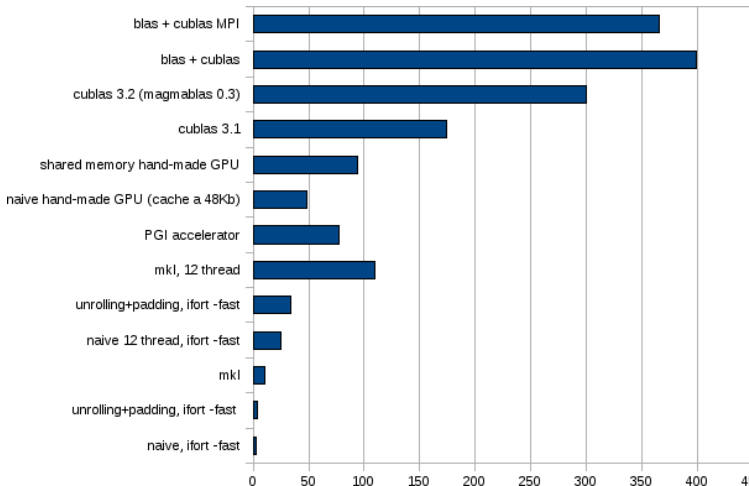
```
module load profile/advanced
module load gnu/4.7.2 gsl/1.15--gnu--4.7.2
gcc -O3 -L$GSL_HOME/lib -lgslcblas matrixmulblas.c -I$GSL_INCLUDE
```

- ▶ Confrontare le performance con quelle ottenibili con `-o3/-fast`
- ▶ Provare anche le versioni multithreaded: osservazioni?
- ▶ Riportare i **GFlop** e considerare matrici 4096x4096 (risultati più stabili)

GNU -O3	Intel -fast	GNU-ACML/GSL seq	Intel-MKL seq
—	Intel -fast -parallel	GNU-ACML par	Intel-MKL par
—			



## ► GPU...







Compilatori e ottimizzazione

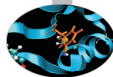
Librerie scientifiche

Floating Point Computing

# Why talking about data formats?



- ▶ The “numbers” used in computers are different from the “usual” numbers
- ▶ Some differences have known consequences
  - ▶ size limits
  - ▶ numerical stability
  - ▶ algorithm robustness
- ▶ Other differences are often misunderstood
  - ▶ portability
  - ▶ exceptions
  - ▶ surprising behaviours with arithmetic

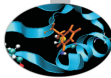


- ▶ Computers usually handle bits
- ▶ An integer number  $n$  may be stored as a sequence of bits
- ▶ Of course, you have a range

$$-2^{r-1} \leq n \leq 2^{r-1} - 1$$

- ▶ Two common sizes
  - ▶ 32 bit: range  $-2^{31} \leq n \leq 2^{31} - 1$
  - ▶ 64 bit: range  $-2^{63} \leq n \leq 2^{63} - 1$
- ▶ Languages allow for declaring different flavours of integers
  - ▶ select the type you need compromising on avoiding overflow and saving memory
- ▶ Is it difficult to have an integer overflow?
  - ▶ consider a cartesian discretization mesh ( $1536 \times 1536 \times 1536$ ) and a linearized index  $i$

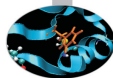
$$0 \leq i \leq 3623878656 > 2^{31} = 2147483648$$



- ▶ Fortran “officially” does not let you specify the size of declared data
  - ▶ you request **kind** and the language do it for you
  - ▶ in principle very good, but interoperability must be considered with attention
  - ▶ and the underlying types are usually just a few of “well known” types
- ▶ C standard types do not match exact sizes, too
  - ▶ look for **int**, **long int**, **unsigned int**, ...
  - ▶ **char** is an 8 bit integer
  - ▶ unsigned integers available, doubling the maximum value  
 $0 \leq n \leq 2^r - 1$



- ▶ **Note:** From now on, some examples will consider base 10 numbers just for readability
- ▶ Representing reals using bits is not natural
- ▶ Fixed size approach
  - ▶ select a fixed point corresponding to comma
  - ▶ e.g., with 8 digits and 5 decimal places 36126234 gets interpreted as 361.26234
- ▶ **Cons:**
  - ▶ limited range: from 0.00001 to 999.99999, spanning  $10^8$
  - ▶ only numbers having at most 5 decimal places can be exactly represented
- ▶ **Pros:**
  - ▶ constant resolution, i.e. the distance from one point to the closest one (0.00001)

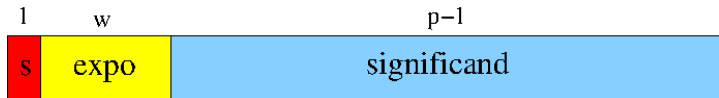


- Consider scientific notation

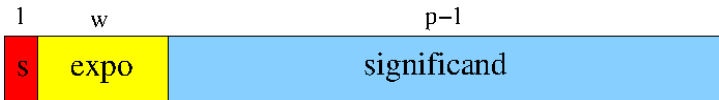
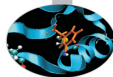
$$n = (-1)^s \cdot m \cdot \beta^e$$

$$0.0046367 = (-1)^0 \cdot 4.6367 \cdot 10^{-3}$$

- Represent it using bits reserving
  - one digit for sign  $s$
  - “p-1” digits for significand (mantissa)  $m$
  - “w” digits for exponent  $e$



# Exponent and Significand



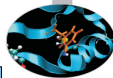
## ► Exponent

- unsigned biased exponent
- $e_{min} \leq e \leq e_{max}$
- $e_{min}$  must be equal to  $(1 - e_{max})$

## ► Mantissa

- precision  $p$ , the digits  $x_i$  are  $0 \leq x_i < \beta$
- “hidden bit” format used for normal values:  $1.xx...x$

IEEE Name	Format	Storage Size	w	p	$e_{min}$	$e_{max}$
Binary32	Single	32	8	24	-126	+127
Binary64	Double	64	11	53	-1022	+1023
Binary128	Quad	128	15	113	-16382	+16383



- ▶ Cons:
  - ▶ only “some” real numbers are floating point numbers (see later)
- ▶ Pros:
  - ▶ constant relative resolution (relative precision), each number is represented with the same *relative error* which is the distance from one point to the closest one divided by the number (see later)
  - ▶ wide range: “normal” positive numbers from  $10^{e_{min}}$  to  $9,999..9 \cdot 10^{e_{max}}$
- ▶ The representation is unique assuming the mantissa is

$$1 \leq m < \beta$$

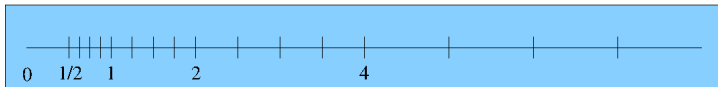
i.e. using “normal” floating-point numbers





- ▶ The distance among “normal” numbers is not constant
- ▶ E.g.,  $\beta = 2$ ,  $p = 3$ ,  $e_{min} = -1$  and  $e_{max} = 2$ :
  - ▶ 16 positive “normalized” floating-point numbers

$e = -1$  ;  $m = 1 + [0:1/4:2/4:3/4] \implies [4/8:5/8:6/8:7/8]$   
 $e = 0$  ;  $m = 1 + [0:1/4:2/4:3/4] \implies [4/4:5/4:6/4:7/4]$   
 $e = +1$  ;  $m = 1 + [0:1/4:2/4:3/4] \implies [4/2:5/2:6/2:7/2]$   
 $e = +2$  ;  $m = 1 + [0:1/4:2/4:3/4] \implies [4/1:5/1:6/1:7/1]$





- ▶ What does it mean “constant relative resolution”?
- ▶ Given a number  $N = m \cdot \beta^e$  the nearest number has distance

$$R = \beta^{-(p-1)} \beta^e$$

- ▶ E.g., given  $3.536 \cdot 10^{-6}$ , the nearest (larger) number is  $3.537 \cdot 10^{-6}$  having distance  $0.001 \cdot 10^{-6}$
- ▶ The relative resolution is (nearly) constant (considering  $m \simeq \beta/2$ )

$$\frac{R}{N} = \frac{\beta^{-(p-1)}}{m} \simeq 1/2\beta^{-p}$$



- ▶ Not any real number can be expressed as a floating point number
  - ▶ because you would need a larger exponent
  - ▶ or because you would need a larger precision
- ▶ The resolution is directly related to the intrinsic error
  - ▶ if  $p = 4$ , 3.472 may approximate numbers between 3.4715 and 3.4725, its intrinsic error is 0.0005
  - ▶ the instrinsic error is (less than)  $(\beta/2)\beta^{-p}\beta^e$
  - ▶ the relative intrinsic error is

$$\frac{(\beta/2)\beta^{-p}}{m} \leq (\beta/2)\beta^{-p} = \varepsilon$$

- ▶ The intrinsic error  $\varepsilon$  is also called “machine epsilon” or “relative precision”



- ▶ When performing calculations, floating-point error may propagate and exceed the intrinsic error

```

real value           = 3.14145
correctly rounded value = 3.14
current value        = 3.17
  
```

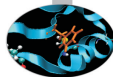
- ▶ The most natural way to measure rounding error is in “ulps”, i.e. units in the last place
  - ▶ e.g., the error is 3 ulps
- ▶ Another interesting possibility is using “machine epsilon”, which is the relative error corresponding to 0.5 ulps

```

relative error  = 3.17-3.14145 = 0.02855
machine epsilon = 10/2*0.001   = 0.005
relative error  = 5.71 ε
  
```



- ▶ Featuring a constant relative precision is very useful when dealing with rescaled equations
- ▶ Beware:
  - ▶ 0.2 has just one decimal digit using radix 10, but is periodic using radix 2
  - ▶ periodicity arises when the fractional part has prime factors not belonging to the radix
  - ▶ by the way, in Fortran if **a** is double precision, **a=0 . 2** is badly approximated (use **a=0 . 2d0** instead)
- ▶ Beware overflow!
  - ▶ you think it will not happen with your code but it may happen
  - ▶ exponent range is symmetric: if possible, perform calculations around 1 is a good idea



IEEE Name	min	max	$\epsilon$	C	Fortran
Binary32	1.2E-38	3.4E38	5.96E-8	float	real
Binary64	2.2E-308	1.8E308	1.11E-16	double	real(kind(1.d0))
Binary128	3.4E-4932	1.2E4932	9.63E-35	long double	real(kind=...)

- ▶ There are also “double extended” type and parametrized types
- ▶ Extended and quadruple precision devised to limit the round-off during the double calculation of transcendental functions and increase overflow
- ▶ Extended and quad support depends on architecture and compiler: often emulated and, hence, slow!
- ▶ Decimal with 32, 64 and 128 bits are defined by standards, too
- ▶ FPU are usually “conformant” but not “compliant”
- ▶ To be safe when converting binary to text specify 9 decimals for single precision and 17 decimal for double



- ▶ Assume  $p = 3$  and you have to compute the difference  $1.01 \cdot 10^1 - 9.93 \cdot 10^0$
- ▶ To perform the subtraction, usually a shift of the smallest number is performed to have the same exponent
- ▶ First idea: compute the difference exactly and then round it to the nearest floating-point number

$$x = 1.01 \cdot 10^1 \quad ; \quad y = 0.993 \cdot 10^1$$

$$x - y = 0.017 \cdot 10^1 = 1.70 \cdot 10^{-2}$$

- ▶ Second idea: compute the difference with  $p$  digits

$$x = 1.01 \cdot 10^1 \quad ; \quad y = 0.99 \cdot 10^1$$

$$x - y = 0.02 \cdot 10^1 = 2.00 \cdot 10^{-2}$$

the error is 30 ulps!



- A possible solution: use the guard digit ( $p+1$  digits)

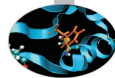
$$x = 1.010 \cdot 10^1$$

$$y = 0.993 \cdot 10^1$$

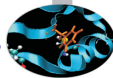
$$x - y = 0.017 \cdot 10^1 = 1.70 \cdot 10^{-2}$$

- Theorem: if  $x$  and  $y$  are floating-point numbers in a format with parameters and  $p$ , and if subtraction is done with  $p + 1$  digits (i.e. one guard digit), then the relative rounding error in the result is less than  $2 \varepsilon$ .





- ▶ When subtracting nearby quantities, the most significant digits in the operands match and cancel each other
- ▶ There are two kinds of cancellation: catastrophic and benign
  - ▶ benign cancellation occurs when subtracting exactly known quantities: according to the previous theorem, if the guard digit is used, a very small error results
  - ▶ catastrophic cancellation occurs when the operands are subject to rounding errors
- ▶ For example, consider  $b = 3.34$ ,  $a = 1.22$ , and  $c = 2.28$ .
  - ▶ the exact value of  $b^2 - 4ac$  is 0.0292
  - ▶ but  $b^2$  rounds to 11.2 and  $4ac$  rounds to 11.1, hence the final answer is 0.1 which is an error by 70ulps
  - ▶ the subtraction did not introduce any error, but rather exposed the error introduced in the earlier multiplications.

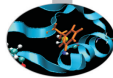


- ▶ The expression  $x^2 - y^2$  is more accurate when rewritten as  $(x - y)(x + y)$  because a catastrophic cancellation is replaced with a benign one
  - ▶ replacing a catastrophic cancellation by a benign one may be not worthwhile if the expense is large, because the input is often an approximation
- ▶ Eliminating a cancellation entirely may be worthwhile even if the data are not exact
- ▶ Consider second-degree equations

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

- ▶ if  $b^2 \gg ac$  then  $b^2 - 4ac$  does not involve a cancellation
- ▶ but, if  $b > 0$  the addition in the formula will have a catastrophic cancellation.
- ▶ to avoid this, multiply the numerator and denominator of  $x_1$  by  $-b - \sqrt{b^2 - 4ac}$  to obtain  $x_1 = (2c)/(-b - \sqrt{b^2 - 4ac})$  where no catastrophic cancellation occurs

# Rounding and IEEE standards



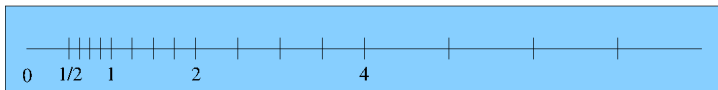
- ▶ The IEEE standards requires correct rounding for:
  - ▶ addition, subtraction, mutiplication, division, remainder, square root
  - ▶ conversions to/from integer
- ▶ The IEEE standards recommends correct rounding for:
  - ▶  $e^x$ ,  $e^x - 1$ ,  $2^x$ ,  $2^x - 1$ ,  $\log_\alpha(\phi)$ ,  $1/\sqrt{x}$ ,  $\sin(x)$ ,  $\cos(x)$ ,  $\tan(x)$ ,....
- ▶ Remember: “No general way exists to predict how many extra digits will have to be carried to compute a transcendental expression and round it correctly to some preassigned number of digits” (W. Kahan)



- ▶ Zero: signed
- ▶ Infinity: signed
  - ▶ overflow, divide by 0
  - ▶  $\text{Inf-Inf}, \text{Inf/Inf}, 0 \cdot \text{Inf} \rightarrow \text{NaN}$  (indeterminate)
  - ▶  $\text{Inf op } a \rightarrow \text{Inf}$  if  $a$  is finite
  - ▶  $a / \text{Inf} \rightarrow 0$  if  $a$  is finite
- ▶ NaN: not a number!
  - ▶ Quiet NaN or Signaling NaN
  - ▶ e.g.  $\sqrt{a}$  with  $a < 0$
  - ▶  $\text{NaN op } a \rightarrow \text{NaN}$  or exception
  - ▶ NaNs do not have a sign: they aren't a number
  - ▶ The sign bit is ignored
  - ▶ NaNS can “carry” information



- ▶ Considering positive numbers, the smallest "normal" floating point number is  $n_{smallest} = 1.0 \cdot \beta^{e_{min}}$
- ▶ In the previous example it is  $1/2$



- ▶ At least we need to add the zero value
  - ▶ there are two zeros:  $+0$  and  $-0$
- ▶ When a computation result is less than the minimum value, it could be rounded to zero or to the minimum value



- ▶ Another possibility is to use denormal (also called subnormal) numbers
  - ▶ decreasing mantissa below 1 allows to decrease the floating point number, e.g.  $0.99 \cdot \beta^{e_{min}}$ ,  $0.98 \cdot \beta^{e_{min}}$ , ...,  $0.01 \cdot \beta^{e_{min}}$
  - ▶ subnormals are linearly spaced and allow for the so called “gradual underflow”
- ▶ Pro:  $k/(a - b)$  may be safe (depending on  $k$ ) even is  $a - b < 1.0 \cdot \beta^{e_{min}}$
- ▶ Con: performance of denormals are significantly reduced (dramatic if handled only by software)
- ▶ Some compilers allow for disabling denormals
  - ▶ Intel compiler has `-ftz`: denormal results are flushed to zero
  - ▶ automatically activated when using any level of optimization!



► Double precision:  $w=11$  ;  $p=53$

```

0x0000000000000000    +zero
0x0000000000000001    smallest subnormal
...
0x000fffffffffffffff    largest subnormal
0x0010000000000000
...
0x001fffffffffffffff    smallest normal
0x0020000000000000    2 X smallest normal
...
0x7fefffffffffffffff    largest normal
0x7ff0000000000000    +infinity
  
```



```

0x7ff0000000000001  NaN
...
0x7fffffffffffffffff  NaN
0x8000000000000000  -zero
0x8000000000000001  negative subnormal
...
0x800fffffffffffffff  'largest' negative subnormal
0x8010000000000000  'smallest' negative normal
...
0xffff000000000000  -infinity
0xffff000000000001  NaN
...
0xffffffffffffffff  NaN
  
```



# Error-Free Transformations



- ▶ An error-free transformation (EFT) is an algorithm which determines the rounding error associated with a floating-point operation
- ▶ E.g., addition/subtraction

$$a + b = (a \oplus b) + t$$

where  $\oplus$  is a symbol for floating-point addition

- ▶ Under most conditions, the rounding error is itself a floating-point number
- ▶ **An EFT can be implemented using only floating-point computations in the working precision**



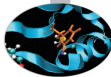
- FastTwoSum: compute  $a + b = s + t$  where

$$|a| \geq |b|$$

$$s = a \oplus b$$

```

void FastTwoSum( const double a, const double b,
                 double* s, double* t ) {
    // No unsafe optimizations !
    *s = a + b;
    *t = b - ( *s - a );
    return;
}
  
```



- ▶ No requirements on  $a$  or  $b$
- ▶ Beware: avoid compiler unsafe optimizations!

```
void TwoSum( const double a, const double b,  
             double* s, double* t ) {  
    // No unsafe optimizations !  
    *s = a + b;  
    double z = *s - b;  
    *t = (a-z)+(b-s-z); return;
```



- Condition number

$$C_{sum} = \frac{|\sum a_i|}{\sum |a_i|}$$

- If  $C_{sum}$  is “not too large”, the problem is not ill conditioned and traditional methods may suffice
- But if it is “too large”, we want results appropriate to higher precision without actually using a higher precision
- But if higher precision is available, consider to use it!
  - beware: quadruple precision is nowadays only emulated

# Traditional summation



$$S = \sum_{i=0}^n x_i$$

```
double Sum( const double* x, const int n ) {  
    int i;  
    for ( i = 0; i < n; i++ ) {  
        Sum += x[ i ];  
    }  
    return Sum;  
}
```

- ▶ Traditional Summation: what can go wrong?
  - ▶ catastrophic cancellation
  - ▶ magnitude of operands nearly equal but signs differ
  - ▶ loss of significance
  - ▶ small terms encountered when running sum is large
  - ▶ the smaller terms don't affect the result
  - ▶ but later large magnitude terms may reduce the running sum



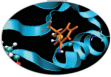
- ▶ Based on FastTwoSum and TwoSum techniques
- ▶ Knowledge of the exact rounding error in a floating-point addition is used to correct the summation
- ▶ Compensated Summation

```

double Kahan( const double* a, const int n ) {
    double s = a[ 0 ];           // sum
    double t = 0.0;              // correction term
    for(int i=1; i<n ; i++) {
        double y = a[ i ] - t;   // next term "plus" correction
        double z = s + y;        // add to accumulated sum
        t = ( z - s ) - y;        // t ← -( low part of y )
        s = z;                   // update sum
    }
    return s;
}
  
```



- ▶ Many variations known (Knutht, Priest,...)
- ▶ Sort the values and sum starting from smallest values (for positive numbers)
- ▶ Other techniques (distillation)
- ▶ Use a greater precision or emulate it (long accumulators)
- ▶ Similar problems for Dot Product, Polynomial evaluation,...



- ▶ Underflow
  - ▶ Absolute value of a non zero result is less than the minimum value (i.e., it is subnormal or zero)
- ▶ Overflow
  - ▶ Magnitude of a result greater than the largest finite value
  - ▶ Result is  $\pm\infty$
- ▶ Division by zero
  - ▶  $a/b$  where  $a$  is finite and non zero and  $b=0$
- ▶ Inexact
  - ▶ Result, after rounding, is not exact
- ▶ Invalid
  - ▶ an operand is sNaN, square root of negative number or combination of infinity

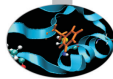




- ▶ Let us say you may produce a NaN
- ▶ What do you want to do in this case?
- ▶ First scenario: go on, there is no error and my algorithm is robust
- ▶ E.g., the function **maxfunc** compute the maximum value of a scalar function  $f(x)$  testing each function value corresponding to the grid points  $g(i)$

```
call maxfunc(f,g)
```

- ▶ to be safe I should pass the domain of  $f$  but the it could be difficult to do
- ▶ I may prefer to check each grid point  $g(i)$
- ▶ if the function is not defined somewhere, I will get a NaN (or other exception) but I do not care: the maximum value will be correct



- ▶ Second scenario: ops, something went wrong during the computation...
- ▶ (Bad) solution: complete your run and check the results and, if you see NaN, throw it away
- ▶ (First) solution: trap exceptions using compiler options (usually systems ignore exception as default)
- ▶ Some compilers allow to enable or disable floating point exceptions
  - ▶ Intel compiler: **-fpe0**: Floating-point invalid, divide-by-zero, and overflow exceptions are enabled. If any such exceptions occur, execution is aborted.
  - ▶ GNU compiler:  
**-ffpe-trap=zero,overflow,invalid,underflow**
- ▶ very useful, but the performance loss may be material!
- ▶ use only in debugging, not in production stage



- ▶ (Second) solution: check selectively
  - ▶ each  $N_{check}$  time-steps
  - ▶ the most dangerous code sections
- ▶ Using language features to check exceptions or directly special values (NaNs,...)
  - ▶ the old print!
  - ▶ Fortran (2003): from module `ieee_arithmetic`,  
`ieee_is_nan(x)`, `ieee_is_finite(x)`
  - ▶ C: from `<math.h>`, `isnan` or `isfinite`, from C99 look for `fenv.h`
  - ▶ do not use old style checks (compiler may remove them):

```

int IsFiniteNumber(double x) {
    return (x <= DBL_MAX && x >= -DBL_MAX);
}
  
```



- ▶ Why doesn't my application always give the same answer?
  - ▶ inherent floating-point uncertainty
  - ▶ we may need reproducibility (porting, optimizing,...)
  - ▶ accuracy, reproducibility and performance usually conflict!
- ▶ Compiler safe mode: transformations that could affect the result are prohibited, e.g.
  - ▶  $x/x = 1.0$ , false if  $x = 0.0, \infty, NaN$
  - ▶  $x - y = -(y - x)$  false if  $x = y$ , zero is signed!
  - ▶  $x - x = 0.0 \dots$
  - ▶  $x * 0.0 = 0.0 \dots$



- ▶ An important case: reassociation is not safe with floating-point numbers

- ▶  $(x + y) + z = x + (y + z)$  : reassociation is not safe
- ▶ compare

$$-1.0 + 1.0e-13 + 1.0 = 1.0 - 1.0 + 1.0e-13 = 1.0e-13 + 1.0 - 1.0$$

- ▶  $a * b / c$  may give overflow while  $a * (b / c)$  does not
- ▶ Best practice:
  - ▶ select the best expression form
  - ▶ promote operands to the higher precision (operands, not results)



- ▶ Compilers allow to choose the safety of floating point semantics
- ▶ GNU options (high-level):

```
-f[no-]fast-math
```

- ▶ It is off by default (different from icc)
  - ▶ Also sets abrupt/gradual underflow (FTZ)
  - ▶ Components control similar features, e.g. value safety (`-funsafe-math-optimizations`)
- ▶ For more detail

```
http://gcc.gnu.org/wiki/FloatingPointMath
```



- ▶ Intel options:

```
-fp-model <type>
```

- ▶ fast=1: allows value-unsafe optimizations (**default**)
- ▶ fast=2: allows additional approximations
- ▶ precise: value-safe optimizations only
- ▶ strict: precise + except + disable fma

- ▶ Also pragmas in C99 standard

```
#pragma STDC FENV_ACCESS etc
```



- ▶ Which is the ordering of bytes in memory? E.g.,

`-1267006353 ==> 10110100011110110000010001101111`

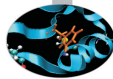
- ▶ Big endian: `10110100 01111011 00000100 01101111`
- ▶ Little endian: `01101111 00000100 01111011 10110100`
- ▶ Other exotic layouts (VAX,...) nowadays unusual
- ▶ Limits portability
- ▶ Possible solutions
  - ▶ conversion binary to text and text to binary
  - ▶ compiler extensions(Fortran):
    - HP Alpha, Intel: `-convert big_endian | little_endian`
    - PGI: `-byteswapio`
    - Intel, NEC: `F_UFMTENDIAN` (variabile di ambiente)
  - ▶ explicit reordering
  - ▶ conversion libraries





- ▶ For C Standard Library a file is written as a stream of byte
- ▶ In Fortran file is a sequence of records:
  - ▶ each read/write refer to a record
  - ▶ there is record marker before and after a record (32 or 64 bit depending on file system)
  - ▶ remember also the different array layout from C and Fortran
- ▶ Possible portability solutions:
  - ▶ read Fortran records from C
  - ▶ perform the whole I/O in the same language (usually C)
  - ▶ use Fortran 2003 **access='stream'**
  - ▶ use I/O libraries

# How much precision do I need?



- ▶ Single, Double or Quad?
  - ▶ maybe single is too much!
  - ▶ computations get (much) slower when increasing precision, storage increases and power supply too
- ▶ Famous story
  - ▶ Patriot missile incident (2/25/91) . Failed to stop a scud missile from hitting a barracks, killing 28
  - ▶ System counted time in 1/10 sec increments which doesn't have an exact binary representation. Over time, error accumulates.
  - ▶ The incident occurred after 100 hours of operation at which point the accumulated errors in time variable resulted in a 600+ meter tracking error.
- ▶ **Wider floating point formats turn compute bound problems into memory bound problems!**

# How much precision do I need? / 2



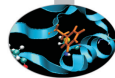
- ▶ Programmers should conduct mathematically rigorous analysis of their floating point intensive applications to validate their correctness
- ▶ Training of modern programmers often ignores numerical analysis
- ▶ Useful tricks
  - ▶ Repeat the computation with arithmetic of increasing precision, increasing it until a desired number of digits in the results agree
  - ▶ Repeat the computation in arithmetic of the same precision but rounded differently, say Down then Up and perhaps Towards Zero, then compare results
  - ▶ Repeat computation a few times in arithmetic of the same precision but with slightly different input data, and see how widely results vary



- ▶ A “correct” approach
- ▶ Interval number: possible values within a closed set

$$\mathbf{x} \equiv [x_L, x_R] := \{x \in \mathbb{R} \mid x_L \leq x \leq x_R\}$$

- ▶ e.g.,  $1/3=0.33333$  ;  $1/3 \in [0.3333, 0.3334]$
- ▶ Operations
  - ▶ Addition  $x + y = [a, b] + [c, d] = [a + c, b + d]$
  - ▶ Subtraction  $x - y = [a, b] - [c, d] = [a - d, b - c]$
  - ▶ ...
- ▶ Properties are interesting and can be applied to equations
- ▶ Interval Arithmetic has been tried for decades, but often produces bounds too loose to be useful
- ▶ A possible future
  - ▶ chips supporting variable precision and uncertainty tracking
  - ▶ runs software at low precision, tracks accuracy and reruns computations automatically if the error grows too large.



- ▶ N.J. Higham, Accuracy and Stability of Numerical Algorithms 2nd ed., SIAM, capitoli 1 e 2
- ▶ D. Goldberg, What Every Computer Scientist Should Know About Floating-Point Arithmetic, ACM C.S., vol. 23, 1, March 1991 [http://docs.oracle.com/cd/E19957-01/806-3568/ncg\\_goldberg.html](http://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html)
- ▶ W. Kahan <http://www.cs.berkeley.edu/~wkahan/>
- ▶ Standards: <http://grouper.ieee.org/groups/754/>

# Hands-on: Compensated sum



- ▶ The code in `summation.cpp/f90` initializes an array with an ill-conditioned sequence of the order of

`100, -0.001, -100, 0.001, . . . . .`

- ▶ Simple and higher precision summation functions are implemented
- ▶ Implement Kahan algorithm in C++ or Fortran
- ▶ Compare the accuracy of the results