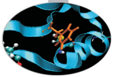
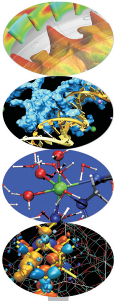


Debugging and Optimization of Scientific Applications

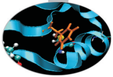
G. Amati P. Lanucara
V. Ruggiero

CINECA Rome - SCAI Department



Rome, 20-22 April 2015

AGENDA



20th April 2015

9.00-9.30 Registration
9.30-10.30 Architectures
10.30-13.00 Cache and Memory System + Esercises
14.00-15.00 Pipelines + Exercises
15.00-17.00 Profilers + Exercises

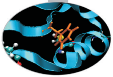
21st april 2015

9.30-13.00 Compilers+Exercises
14.00-15.30 Scientific Libraries + Exercises
15.00-17.00 Floating-point + Exercises

22nd april 2015

9.30-11.00 Makefile + Exercises
11.00-13.00 Debugging+Exercises
14.00-17.00 Debugging+Exercises

Outline

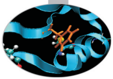


Compilers and Code optimization

Scientific Libraries

Floating Point Computing

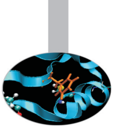
Programming languages



- Many programming languages were defined...
- http://foldoc.org/contents/language.html

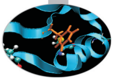
```
20-GATE; 2.PAK; 473L Query; 51forth; A#; A-0; a1; a56;
Abbreviated Test Language for Avionics Systems; ABC;
ABC ALGOL; ABCL/1; ABCL/c+; ABCL/R; ABCL/R2; ABLE;
ABSET; abstract machine; Abstract Machine Notation;
abstract syntax; Abstract Syntax Notation 1;
Abstract-Type and Scheme-Definition Language;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

Programming languages



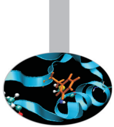
- Interpreted language
    - statement by statement translation during code execution
    - no way to perform optimization between different statements
    - easy to find semantic errors
    - e.g. scritping languages, Java (bytecode),. . .
- Compiled language
    - Che code is translated by the compiler before the execution
    - possibility to perform optimization between different statements
    - e.g. Fortran, C, C++

CPU/1



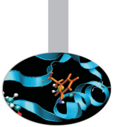
- It is composed by (first approximation):
  - registers: instruction operands
  - Functional units: performs instructions
- Functional units
  - logical operations (bitwise)
  - integer arithmetic
  - floating-point arithmetic
  - computing address
  - load & store operation
  - branch prediction and branch execution

CPU/2



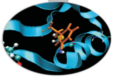
- RISC: Reduced Instruction Set CPU
  - simple "basic" instructions
  - one statement → many istructions
  - simple decode and execution
- CISC: Complex Instruction Set CPU
  - many "complex" instructions
  - one statement → few istructions
  - complex decode and execution
- in these days now CISC like-machine split instruction in micro RISC-line ones

Architecture vs. Implementation



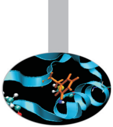
- Architecture:
  - instruction set (ISA)
  - registers (integer, floating point, . . .)
- Implementation:
  - physical registers
  - clock & latency
  - # of functional units
  - Cache's size & features
  - Out Of Order execution, Simultaneous Multi-Threading, ...
- Same architecture, different implementations:
  - Power: Power3, Power4, . . ., Power8
  - x86: Pentium III, Pentium 4, Xeon, Pentium M, Pentium D, Core, Core2, Athlon, Opteron, . . .
  - different performances
  - different rules to improve performance

The Compiler



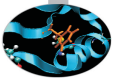
- It "translate" source code in executable
- It rejects code with sintax errors
- It (sometimes) warns about "semantic" problems
- It (if allowed) try to optimize the code
    - code independent optimization
    - code dependent optimization
    - CPU dependent optimization
    - Cache & Memory oriented optimization
    - Hint to the CPU (branch prediction)
- It is:
    - powerfull: can save programmer's time
    - complex: can perform "complex" optimization
    - limited: it is an expert system but can be fooled by the way you write the code . . .

Building an executable



- It is a three-step process:
- Pre-processing:
  - every source code is analyzed by the pre-processor
    - MACROs substitution (**`#define`**)
    - code insertion for **`#include`** statements
    - code insertion or code removal (**`#ifdef`** . . .)
    - removing comments . . .
- Compiling:
  - each code is translated in object files
    - object files is a collection of "symbols" that refere to variables/function defined in the program
- Linking:
  - All the object files are put together to build the finale executable
  - Any symbol in the program must be resolved
    - the symbols can be defined inside your object files
    - you can use other object file (e.g. external libraries)

Example: gfortran compilation



- With the command:

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

all the three steps (preprocessing, compiling, linking) are performed

- Pre-processing

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

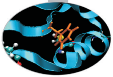
  - `-E -cpp` options force `gfortran` to stop after pre-processing
  - no need to use `-cpp` if file extension is `*.F90`

- Compiling

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

  - `-c` option force `gfortran` only to pre-processing and compile
  - from every source file an object file `*.o` is created

Example: gfortran linking



- Linking: we must use object files

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

- To solve symbols from external libraries
  - suggest the libraries to use with option `-l`
  - suggest the directory where the libraries are with option `-L`
- How link `libdsp.a` library located in `/opt/lib`

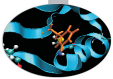
```
user@caspur$> gfortran file1.o file2.o -L/opt/lib -ldsp -o dsp.exe
```

- How create and link a static library

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

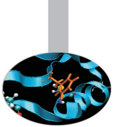
  - `ar` creates the archive `libdsp.a` containing `dsp.o`
  - `ranlib` builds the library

Compiler: what it can do



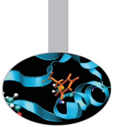
- It performs these code modifications
  - Register allocation
  - Register spilling
  - Copy propagation
  - Code motion
  - Dead and redundant code removal
  - Common subexpression elimination
  - Strength reduction
  - Inlining
  - Index reordering
  - Loop pipelining , unrolling, merging
  - Cache blocking
  - . . .
- Everything to maximize performances!!

Compiler: what it cannot do



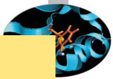
- Global optimization of "big" source code, unless switch on interprocedural analisys (IPO) but it is very time consuming . . .
- Understand and resolve complex indirect addressing
- Strenght reduction (with non-integer values)
- Common subexpression elimination through function calls
- Unrolling, Merging, Blocking with:
  - functions/subroutine calls
  - I/O statement
- Implicit function inlining
- Knowing at run-time variabile's values

Optimizations: levels



- All compilers have
  "predefined" optimization levels **`-O<n>`**
  - with **`n`** from 0 a 3 (IBM up to 5)
- Usually :
  - **`-O0`**: no optimization is performed, simple translation (tu use with **`-g`** for debugging)
  - **`-O`**: default value
  - **`-O1`**: basic optimizations
  - **`-O2`**: memory-intensive optimizations
  - **`-O3`**: more aggressive optimizations, it can alter the instruction order (see floating point section)
- Some compilers have **`-fast`** option (**`-O3`** plus more options)

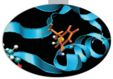
Intel compiler: –O3 option



```
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)

From source code to executable

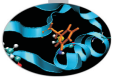


- Executable (i.e. istructions performed by CPU) is very very different from what you think writing a code
- Example: matrix-matrix production

```
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
```

- Computational kernel
  - load from memory three numbers
  - perform one product and one sum
  - store back the result

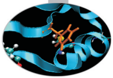
Hands-on: download code



- Examples at the same place

```
https://hpc-forge.cineca.it/files/CoursesDev/public/2015/...
...Debugging_and_Optimization_of_Scientific_Applications/Rome/

Compilers_codes.tar

Libraries_codes.tar

FloatingPoints_codes.tar
```

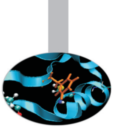
Hands-on: compiler optimization flags



- Matrix-Matrix product, 1024×1024, double precision
- main loop cache friendly
- code in **`matrixmul`** directory (both C & Fortran)
- to load compiler: (**`module load profile/advanced`**):
  - GNU –> **`gfortran, gcc`** : **`module load gnu`**
  - Intel –> **`ifort, icc`** : **`module load intel`**
  - PGIi –> **`pgf90, pgcc`**: **`module load pgi`**
  - You can load one compiler at time, **`module purge`** to remove previous compiler

	GNU	Intel	PGI	GNU	Intel	PGI
flags	seconds	seconds	seconds	GFlops	GFlops	GFlops
-O0						
-O1						
-O2						
-O3						
-O3 -funroll-loops		——	——		——	——
-fast	——			——		

Hands-on: Solution

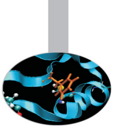


- Matrix-Matrix product, 1024×1024, double precision
- 2 esa-core XEON 5645 Westmere CPUs@2.40GHz
- Fortran results

	GNU	Intel	PGI	GNU	Intel	PGI
flags	seconds	seconds	seconds	GFlops	GFlops	GFlops
default	7.78	0.76	3.49	0.27	2.82	0.61
-O0	7.82	8.87	3.43	0.27	0.24	0.62
-O1	1.86	1.45	3.42	1.16	1.49	0.63
-O2	1.31	0.73	0.72	1.55	2.94	2.99
-O3	0.79	0.34	0.71	2.70	6.31	3.00
-O3 -funroll-loops	0.65	——	——	3.29	——	——
-fast	——	0.33	0.70	——	6.46	3.04

- Open question:
  - Why this behaviour?
  - Which is he best compiler?

Matmul: performance



- Size 1024×1024, duoble precision
- Fortran core, cache friendly
  - FERMI: IBM Blue Gene/Q system, single-socket PowerA2 with 1.6 GHz, 16 core
  - PLX: 2 esa-core XEON 5650 Westmere CPUs 2.40 GHz

FERMI - xlf

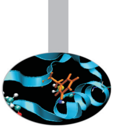
Option	seconds	Mflops
-O0	65.78	32.6
-O2	7.13	301
-O3	0.78	2735
-O4	55.52	38.7
-O5	0.65	3311

PLX - ifort

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

- Why ?

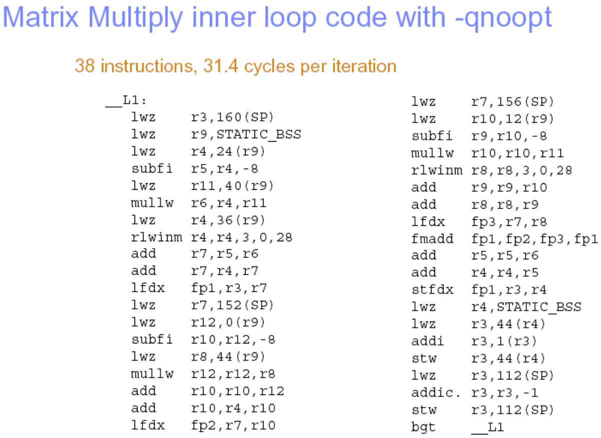
Compiler: report



- What happens at different optimization level?
  - Why performance degradation using `-O4`?
- Hint: use report flags to investigate
- Using IBM `-qreport` flag for `-O4` level shows that:
  - The compiler understant matrix-matrix pattern (it is smart) ad perform a substitution with external BLAS function (`__xl_dgemm`)
  - But it is slow because it doesn't belong to IBM optimized BLAS library (ESSL)
  - At `-O5` level it decides not to use external library
- As general rule of thumb performance increase as the optimization level increase . . .
  - . . . but it's bettet to check!!!

Take a look to assembler

- Very very old example (IBM Power4) but usefull

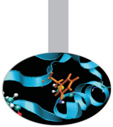


Matrix Multiply inner loop code with -qnoopt

38 instructions, 31.4 cycles per iteration

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

Load,store and floating point

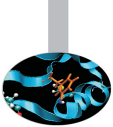


Matrix Multiply inner loop code with -qnoopt

necessary instructions

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

Loop control instructions



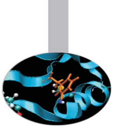
Matrix Multiply inner loop code with -qnoopt

necessary instructions   loop control

```
__L1:
   lwz     r3,160(SP)
   lwz     r9,STATIC_BSS
   lwz     r4,24(r9)
   subfi   r5,r4,-8
   lwz     r11,40(r9)
   mullw   r6,r4,r11
   lwz     r4,36(r9)
   rlwinm  r4,r4,3,0,28
   add     r7,r5,r6
   add     r7,r4,r7
   lfdx    fp1,r3,r7
   lwz     r7,152(SP)
   lwz     r12,0(r9)
   subfi   r10,r12,-8
   lwz     r8,44(r9)
   mullw   r12,r12,r8
   add     r10,r10,r12
   add     r10,r4,r10
   lfdx    fp2,r7,r10
```

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

Data addressing



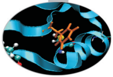
Matrix Multiply inner loop code with -qnoopt

necessary instructions   loop control   addressing code

```
__L1:
   lwz     r3,160(SP)
   lwz     r9,STATIC_BSS
   lwz     r4,24(r9)
   subfi   r5,r4,-8
   lwz     r11,40(r9)
   mullw   r6,r4,r11
   lwz     r4,36(r9)
   rlwinm  r4,r4,3,0,28
   add     r7,r5,r6
   add     r7,r4,r7
   lfdx    fp1,r3,r7
   lwz     r7,152(SP)
   lwz     r12,0(r9)
   subfi   r10,r12,-8
   lwz     r8,44(r9)
   mullw   r12,r12,r8
   add     r10,r10,r12
   add     r10,r4,r10
   lfdx    fp2,r7,r10
```

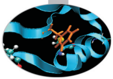
```
lwz     r7,156(SP)
lwz     r10,12(r9)
subfi   r9,r10,-8
mullw   r10,r10,r11
rlwinm  r8,r8,3,0,28
add     r9,r9,r10
add     r8,r8,r9
lfdx    fp3,r7,r8
fmadd   fp1,fp2,fp3,fp1
add     r5,r5,r6
add     r4,r4,r5
stfdx   fp1,r3,r4
lwz     r4,STATIC_BSS
lwz     r3,44(r4)
addi    r3,1(r3)
stw     r3,44(r4)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

?????

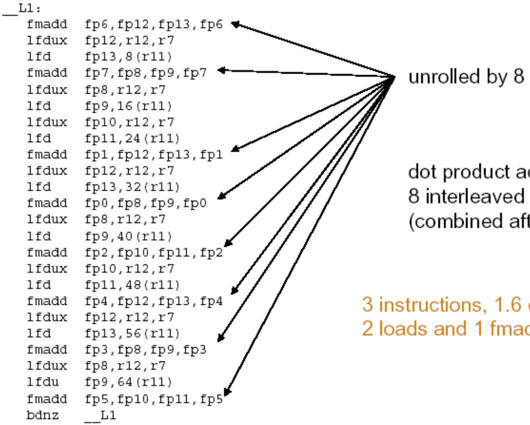


- Memory addressing operations are predominant (30/37)
- Hint:
  - the loop access to contigous memory locations
  - memory address can be computed in easy way from the first location adding a constant
  - use one single memory address operation to address more memory locations
- A (smart) compiler can perform all in automatic way

Optimization/1



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



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

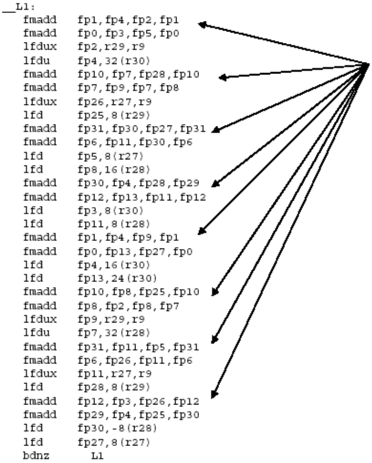
unrolled by 8

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

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

Optimization/2

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

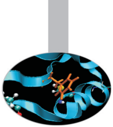


```
__L1:
   fmadd    fp1,fp4,fp2,fp1
   fmadd    fp0,fp3,fp5,fp0
   lfdux    fp2,r29,r9
   lfdu     fp4,32(r30)
   fmadd    fp10,fp7,fp28,fp10
   fmadd    fp7,fp9,fp7,fp8
   lfdux    fp26,r27,r9
   lfd      fp25,8(r29)
   fmadd    fp31,fp30,fp27,fp31
   fmadd    fp6,fp11,fp30,fp6
   lfd      fp5,8(r27)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

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

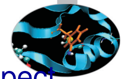
2 instructions, 1.0 cycles per iteration
balanced: 1 load and 1 fmadd per iteration

Instruction to be performed



- Instruction to be performed for the statement `c(i,j) = c(i,j) + a(i,k)*b(k,j)`
- -O0: 24 instructions
  - 3 load/1 store, 1 floating point multiply+add
  - flop/instructions 2/24 (i.e. 8% if peak performance)
- -O2: 9 instructions (more efficent data addressing)
  - 4 load/1 store, 2 floating point multiply+add
  - flop/instructions 4/9 (i.e. 44% if peak performance)
- -O3: 150 instructions (unrolling)
  - 68 load/34 store, 48 floating point multiply+add
  - flop/instructions 96/150 (i.e. 64% if peak performance)
- -O4: 344 instructions (unrolling&blocking)
  - 139 load/74 store, 100 floating point multiply+add
  - flop/instructions 200/344 (i.e. 54% if peak performance)

Who does the dirty work?



- option **-fast** (ifort on PLX) produce a $\simeq$ 30x speed-up respect to option **-O0**
  - many different (and complex) optimizations are done . . .
- **Hand-made optimizations?**
- The compiler is able to do
  - Dead code removal: removing branch

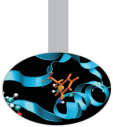
```
b = a + 5.0;
if ((a>0.0) && (b<0.0)) {
   ......
}
```

  - Redudant code removal

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

- But coding style can fool the compiler

Loop counters



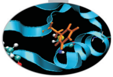
- Always use the correct data type
- If you use as loop index a real type means to perform a implicit casting real → integer every time . . .
- I should be an error according to standard, but compilers sometimes are sloppy...

```
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
```

Time in seconds

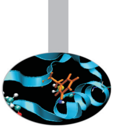
compiler/level	integer	real
(PLX) gfortran -O0	9.96	8.37
(PLX) gfortran -O3	0.75	2.63
(PLX) ifort -O0	6.72	8.28
(PLX) ifort -fast	0.33	1.74
(PLX) pgif90 -O0	4.73	4.85
(PLX) pgif90 -fast	0.68	2.30
(FERMI) bgxlf -O0	64.78	104.10
(FERMI) bgxlf -O5	0.64	12.38

Compilers limitations



- A compiler can do a lot of work . . . but it is a program
- It is easy to fool it!
    - loop body too complex
    - loop values not defined a compile time
    - to much nested **`if`** structure
    - complicate indirect addressing/pointers

index reordering



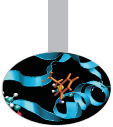
- For simple loops there's no problem
  - . . . using appropriate optimization level

```
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
```

- Time in seconds

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

index reordering/2



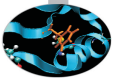
- For more complicated loop nesting could be a problem . . .
  - also at higher optimization levels
  - solution: always write cache friendly loops, if possible

```
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
```

- Time in seconds

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

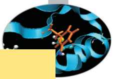
Cache & subroutine



```
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
```

- non unitary access (stride MUST be $\simeq$ 1)

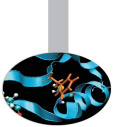
Cache & subroutine/2



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

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

- "same" results as the the previous one
- stride $= 1$
- Sometimes compiler can perform the transformations, but `inlining` option must be activated

Inlining



- means to substitue the functon call with all the instruction
  - no more jump in the program
  - help to perform interpocedural analysis
- the keyword **`inline`** for C and C++ is a “hint” for compiler
- Intel (n: 0=disable, 1=inline functions declared, 2=inline any function, at the compiler’s discretion)

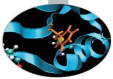
```
-inline-level=n
```

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

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

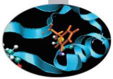
- It varies from compiler to compiler, read the manpage . . .

Common Subexpression Elimination



- Reusing common Subexpression for intemediate results:
  A= B+C+D
  E= B+F+C
- 4 load, 2 store, 4 sums
  A=(B+C) + D
  E=(B+C) + F
- 4 load, 2 store, 3 sums
- WARNING: with floating point arithmetics results can be different
- “Scalar replacement” if you access to a vector location many times
  - compilers can do that (at some optimization level)

Functions & Side Effects



- Functions returns a values but
  - sometimes global variables are modified
  - I/O operations can prduce side effects
- side effects can “stop” compiler to perform inlining
- Example (no side effect):

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

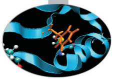
so `f(x)+f(x)+f(x)` it is equivalent to `3*f(x)`

- Example (side effect):

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

so `f(x)+f(x)+f(x)` it is different from `3*f(x)`

CSE & function



- reordering function calls can produce different results
- It is hard for a compiler understand is there’s side effects
- Example: 5 calls to functons, 5 products:

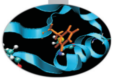
```
x=r*sin(a)*cos(b);
y=r*sin(a)*sin(b);
z=r*cos(a);
```

- Example: 4 calls to functons, 4 products, 1 tempory variable:

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

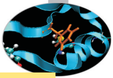
- Correct if there’s no side effect!

CSE: limitations



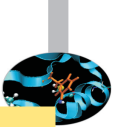
- Core loop too wide:
  - Compiler is able to handle a fixed number of lines: it could not realize that there's room for improvement
- Functions:
  - there is a side effect?
- CSE mean to alter order of operations
  - enabled at "high" optimization level (**`-qnostrict`** per IBM)
  - use parentheis to "inhibit" CSE
- "register spilling": when too much intermediate values are used

What can do a compiler?



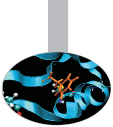
```
do k=1,n3m
   do j=n2i,n2do
      jj=my_node*n2do+j
      do i=1,n1m
         acc =1./(1.-coe*aciv(i)*(1.-int(forclo(nve,i,j,k))))
         aci(jj,i)= 1.
         api(jj,i)=-coe*apiv(i)*acc*(1.-int(forclo(nve,i,j,k)))
         ami(jj,i)=-coe*amiv(i)*acc*(1.-int(forclo(nve,i,j,k)))
         fi(jj,i)=qcap(i,j,k)*acc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

. . . this . . .



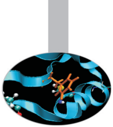
```
do k=1,n3m
   do j=n2i,n2do
      jj=my_node*n2do+j
      do i=1,n1m
         temp = 1.-int(forclo(nve,i,j,k))
         acc =1./(1.-coe*aciv(i)*temp)
         aci(jj,i)= 1.
         api(jj,i)=-coe*apiv(i)*acc*temp
         ami(jj,i)=-coe*amiv(i)*acc*temp
         fi(jj,i)=qcap(i,j,k)*acc
      enddo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

. . . but not that!!! (20% faster)



```
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
...
...
!  the same for the other loop
```

Array syntax



- in place 3D-array translation (512^3)
- Explixcit loop (Fortran77): 0.19 seconds
  - CAVEAT: the loop order is “inverse” in order not to overwirte data

```
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
```

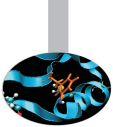
- Array Syntax (Fortran90): 0.75 seconds
  - According to the Standard → store in an intermediate array to avoid to overwrite data

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

- Array syntax with hint: 0.19 seconds

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

Ottimizzazione Report/1



- A report of optimization performed can help to find “problems”
- Intel

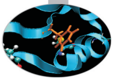
```
-opt-report[n]     n=0(none),1(min),2(med),3(max)
-opt-report-file<file>
-opt-report-phase<phase>
-opt-report-routine<routine>
```

- one or more `*.optrpt` file are generated

```
...
Loop at line:64 memcopy generated
...
```

- Is this `memcopy` necessary?

Ottimizzazione Report/2



- There’s no equivalent flag for GNU compilers
  - Best solution:

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

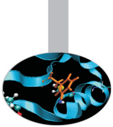
  - dump all compiler operations
  - very hard to understand
- PGI compilers

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

Info at standard output

SuperComputing Applications and Innovation

Give hints to compiler



- Loop size known at compile-time o run-time
  - Some optimizations (like unrolling) can be inhibited

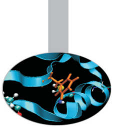
```
real a(1:1024,1:1024)
real b(1:1024,1:1024)
real c(1:1024,1:1024)
...
read(*,*) i1,i2
read(*,*) j1,j2
read(*,*) k1,k2
...
do j = j1, j2
do k = k1, k2
do i = i1, i2
c(i,j)=c(i,j)+a(i,k)*b(k,j)
enddo
enddo
enddo
```

- Time in seconds (Loop Bounds Compile-Time o Run-Time)

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

- WARNING: compiler dependent...

Static vs. Dynamic allocation

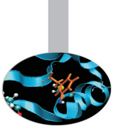


- Static allocation gives more information to compilers
  - but the code is less flexible
  - recompile every time is really boring

```
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))
```

Static vs. Dynamic allocation/2



- for today compilers there’s no big difference
  - Matrix-Matrix Multiplication (time in seconds)

	static	dynamic
(PLX) ifort -O0	6.72	18.26
(PLX) ifort -fast	0.34	0.35

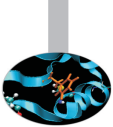
- With static allocation data are put in the “stack”
  - at run-time take care of stacksize (e.g. sementation fault)
  - using bash: to check

```
ulimit -a
```

  - using bash: to modify

```
ulimit -s unlimited
```

Dynamic allocation using C/1



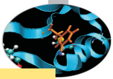
- Using C matrix → arrays of array
  - with static allocation data are contiguos (columnwise)

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

- with dynamic allocation ....
  - “the wrong way”

```
/* 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;
}
```

Dynamic allocation using C/2



- allocate a linear array

```
/* 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;
}
```

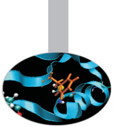
- using as a matrix (with index linearization)

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

- MACROs can help
- also use pointers

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

Aliasing & Restrict

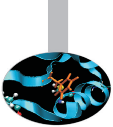


- Aliasing: when two pointers point at the same area
- Aliasing can inhibit optimization
  - you cannot alter order of operations
- C99 standard introduce **`restrict`** keyword to point out that aliasing is not allowed

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

- C++: aliasing not allowed between pointer to different type (strict aliasing)

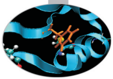
Different operations, different latencies



For a CPU different operations could present different latencies

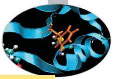
- Sum: few clock cycles
- Product: few clock cycles
- Sum+Product: few clock cycles
- Division: many clock cycle ($O(10)$)
- Sin,Cos: many many clock cycle ($O(100)$)
- exp,pow: many many clock cycle ($O(100)$)
- I/O operations: many many many clock cycles ($O(1000 - 10000)$)

Input/Output



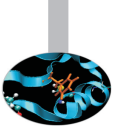
- Handled by the OS
  - system calls
  - pipeline goes dry
  - cache coerency can be destroyed
  - it is very slow
- Golden Rule #1: NEVER mix computing with I/O operations
- Golden Rule #2: NEVER read/write a single data, pack them in a block

Different I/O



```
do k=1,n  ;  do j=1,n  ;  do i=1,n
write(69,*) a(i,j,k)                              ! formatted I/O
enddo     ;  enddo     ;  enddo

do k=1,n  ;  do j=1,n  ;  do i=1,n
write(69) a(i,j,k)                                ! binary I/O
enddo     ;  enddo     ;  enddo

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

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

write(69) (((a(i,j,k),i=1,n),j=1,n),k=1,n)        ! dump1

write(69) a                                       ! dump2
```

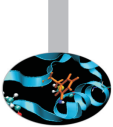
Different I/O: some figures



	seconds	Kbyte
formatted	81.6	419430
binary	81.1	419430
by colunm	60.1	268435
nt matrix	0.66	134742
dump (1)	0.94	134219
dump (2)	0.66	134217

- WARNING: the filesystem used could affect performance (e.g. RAID)...

I/O



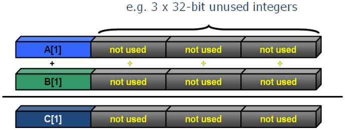
- Read/write operations are slow
- Read/write format data are very very slow
- ALWAYS Read/write binary data
- Golden Rule #1: NEVER mix computing with I/O operations
- Golden Rule #2: NEVER read/write a single data, pack them in a block
- For HPC is possibile use:
  - I/O libraries: MPI-I/O, HDF5, NetCDF,...

Vector units

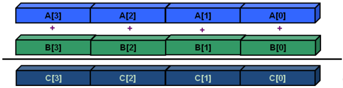
- We are not talking of vector machine
- Vector Units performs parallel floating/integer point operations on dedicate SIMD units
  - Intel: MMX, SSE, SSE2, SSE3, SSE4, AVX
- Example: summing 2 arrays of 4 elements in one single instruction

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

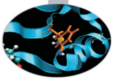
no vectorization



vectorization

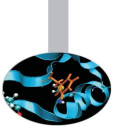


SIMD - evolution



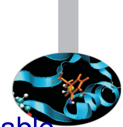
- SSE: 128 bit register (Intel Core - AMD Opteron)
  - 4 floating/integer operations in single precision
  - 2 floating/integer operations in double precision
- AVX: 256 bit register (Intel Sandy Bridge - AMD Bulldozer)
  - 8 floating/integer operations in single precision
  - 4 floating/integer operations in double precision
- MIC: 512 bit register (Intel Knights Corner - 2013)
  - 16 floating/integer operations in single precision
  - 8 floating/integer operations in double precision

Vectorization issues



- Vectorization is a key issue for performance
- To be vectorized single loop iteration must be independent: no data dependence
- coding style can inhibit vectorization
- Some issues for vectorization:
  - Countable
  - Single entry-single exit (no break or exit)
  - Straight-line code (no branch)
  - Only internal loop can be vectorized
  - no function call (unless math or inlined)
- WARNING: due to floating point arithmetic results could differ . . .

Algorithm & Vectorization



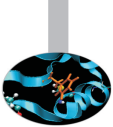
- Different algorithm for the same problem could be vectorazable or not
  - Gauss-Seidel: data dependencies, cannot be vectorized

```
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: no data dependence, can be vectorized

```
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];
```

Optimization & Vectorization



- “coding tricks” can inhibit vectorization
  - can be vectorized

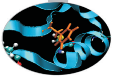
```
for( i = 0; i < n-1; ++i ){
   b[i] = a[i] + a[i+1];
}
```

  - cannot be vectorized

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

- You can help compiler’s work
  - removing unnecessary data dependencies
  - using directives for forcing vectorization

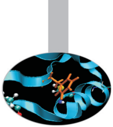
Directives



- You can force to vectorize when the compiler doesn't want using directive
- they are "compiler dependent"
  - Intel Fortran: **`!DIR$ simd`**
  - Intel C: **`#pragma simd`**
- Example: data dependency found by the compiler is apparent, cause every time step **`inow`** is different from **`inew`**

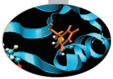
```
     do k = 1,n
!DIR$ simd
        do i = 1,l
...
          x02 = a02(i-1,k+1,inow)
          x04 = a04(i-1,k-1,inow)
          x05 = a05(i-1,k  ,inow)
          x06 = a06(i  ,k-1,inow)
          x11 = a11(i+1,k+1,inow)
          x13 = a13(i+1,k-1,inow)
          x14 = a14(i+1,k  ,inow)
          x15 = a15(i  ,k+1,inow)
          x19 = a19(i  ,k  ,inow)

          rho =+x02+x04+x05+x06+x11+x13+x14+x15+x19
...
          a05(i,k,inew) = x05 - omega*(x05-e05) + force
          a06(i,k,inew) = x06 - omega*(x06-e06)
...
```

Hands-on: Vectorization



- Compare performances w/o vectorization **`simple_loop.f90`** using PGI and Intel compilers
  - **`-fast`**, to inibhit vectorization use **`-Mnovect`** (PGI) or **`-no-vec`** (Intel)
- Program **`vectorization_test.f90`** contains 18 different loops
  - Which can be vectorized?
  - check with PGI compiler with reporting flag **`-fast -Minfo`**
  - check with Intel compiler with reporting flag **`-fast -opt-report3 -vec-report3`**
  - check with GNU compiler with reporting flag **`-ftree-vectorizer-verbose=n`**
  - Any idea to force vectorization?

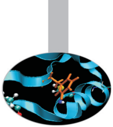
Hands-on: Vectorization/2



	PGI	Intel
Vectorized time		
Non-Vectorized time		

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

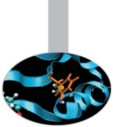
Hands-on: Vectorization Results



	PGI	Intel
Vectorized time	0.79	0.52
Non-Vectorized time	1.58	0.75

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

Handmade Vectorization



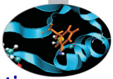
- It is possible to insert inside the code vectorized function
- You have to rewrite the loop making 4 iteration in parallel . . .

```
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);
  }
}
```

- Non-portable tecnique...

Automatic parallelization



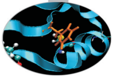
- Some compilers are able to exploit parallelism in an automatic way
- Shared Memory Parallelism
- Similar to OpenMP Paradigm without directives
  - Usually performance are not good . . .
- Intel:

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

- IBM:

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

Outline

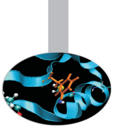


Compilers and Code optimization

Scientific Libraries

Floating Point Computing

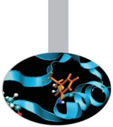
Static and Dynamic libraries



- you have to link with
  `-L<library_directory> -l<library_name>`

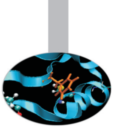
- Static library:
  - **`*.a`**
  - all symbols are included in the executable at linking
  - if you built a new library that use an other external library it doesn't contains the other symbols: you have to explicitly linking the library
- Dynamic Library:
  - **`*.so`**
  - Symbols are resolved at run-time
  - you have to set-up where find the requested library at run-time (i.e. setting **`LD_LIBRARY_PATH`** environment variable)
  - **`ldd <exe_name>`** gives you info about dynamic library needed

Scientific Libraries



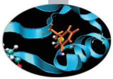
- A (complete?) set of function implementing different numeric algorithms
- A set of basic function (e.g. Fasr Fourier Transform, . . .)
- A set of low level function (e.g. scalar products or random number generator), or more complex algorithms (Fourier Transform or Matrix diagonalization)
- (Usually) Faster than hand made code (i.e. sometimes written in assembler)
- Proprietary or OpenSource
- Sometimes developer for a particular compiler . . .

Pros & Cons

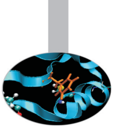


- Pros:
  - helps to moudularize the code
  - portability
  - efficient
  - ready to use
- Cons:
  - some details hidden (e.g. Memory requirements)
  - you don't have the complete control
  - . . .

Which library?



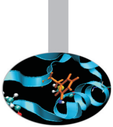
- It is hard to have a complete overview of Scientific libraries
  - many different libraries
  - still evolving . . .
  - . . . especially for “new architectures” (e.g GPU, MIC)
- Main libraries used in HPC
  - Linibear Algebra
  - FFT
  - I/O libraries
  - Parallel Computing
  - Mesh decomposition
  - . . .



Linear Algebra

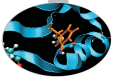
- Different parallelization paradigm
  - Shared memory (i.e. multi-threaded) or/and Distributed Memory
- Shared memory
  - BLAS
  - GOTOBLAS
  - LAPACK/CLAPACK/LAPACK++
  - ATLAS
  - PLASMA
  - SuiteSparse
  - ...
- Distributed Memory
  - Blacs (only decomposition)
  - ScaLAPACK
  - PSBLAS
  - Elemental
  - ...

BLAS



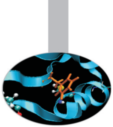
- BLAS: Basic Linear Algebra Subprograms
  - it is one of the first library developed for HPC (1979, vector machine)
  - it includes basic operations between vectors, matrix and vector, matrix and matrix
  - it is used by many other high level libraries
- It is divided into 3 different levels
  - BLAS lev. 1: basic subroutines for scalar-vector operations (1977-79, vector machine)
  - BLAS lev. 2: basic subroutines for vector-matrix operations (1984-86)
  - BLAS lev. 3: subroutine for matrix-matrix operations (1988)

BLAS/2



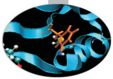
- It apply to real/complex data, in single/double precision
- Old Fortran77 style
- Level 1: scalar-vector operations (O(n))
  - *SWAP vector swap
  - *COPY vector copy
  - *SCAL scaling
  - *NRM2 L2-norm
  - *AXPY sum: a*X+Y (X,Y are vectors)
- Level 2: vector-matrix operations (O(n^2))
  - *GEMV product vector/matrix (generic)
  - *HEMV product vector/matrix (hermitian)
  - *SYMV product vector/matrix (simmetric)

BLAS/3



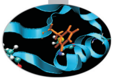
- Level 3: matrix-matrix operations ($O(n^3)$)
  - *GEMM product matrix/matrix (generic)
  - *HEMM product matrix/matrix (hermitian)
  - *SYMM product matrix/matrix (simmetric)
- GOTOBLAS
  - optimized (using assembler) BLAS library for different supercomputers. Develped by Kazushige Goto, now at Texas Advanced Computing Center, University of Texas at Austin).

LAPACK & Co.



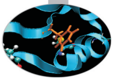
- LAPACK: Linear Algebra PACKage
  - Linear algebral solvers (linear systems of equations, Ordinary Least Square, eigenvalues, . . .)
  - evolution of LINPACK e EISPACK
- ATLAS: Automatically Tuned Linear Algebra Software
  - BLAS and LAPACK (but only some subroutine) implementations
  - Automatic optization of Software paradigm
- PLASMA: Parallel Linear Algebra Software for Multi-core Architectures
  - Similare to LAPACK (less subroutines) developed to be efficent on multicore systems.
- SuiteSparse
  - Sparse Matrix

Linear Algebra/2



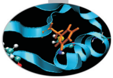
- Eigenvalues/Eigenvectors
  - EISPACK: with specialized version for matrix fo different kinf (real/complex, hermitia, simmetrich, tridiagonal, . . .)
  - ARPACK: eigenvalus for big size problems. Parallel version use BLACs and MPI libraries.
- Distributed Memory Linear Algebra
  - BLACS: linear algebra oriented message passing interface
  - ScaLAPACK: Scalable Linear Algebra PACKage
  - Elemental: framework per algebra lineare densa
  - PSBLAS: Parallel Sparse Basic Linear Algebra Subroutines
  - . . .

Input/Output Libraries



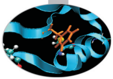
- I/O Libraries are extremely important for
  - interoperability: C/Fortran, Little Endian/Big Endian, . . .
  - visualizzazion
  - Sub-set data analysis
  - metadata
  - parallel I/O
- HDF5: “is a data model, library, and file format for storing and managing data”
- NetCDF: “NetCDF is a set of software 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”

Other Libraries



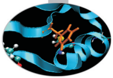
- MPI: Message Passing Interface
  - De facto standard for Distributed Memory Parallelization (MPICH/OpenMPI)
- Mesh decomposition
  - METIS e ParMETIS: “can partition a graph, partition a finite element mesh, or reorder a sparse matrix”
  - Scotch e PT-Scotch: “sequential and parallel graph partitioning, static mapping and clustering, sequential mesh and hypergraph partitioning, and sequential and parallel sparse matrix block ordering”

Other Scientific computing libraries



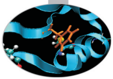
- Trilinos
  - object oriented software framework for the solution of large-scale, complex multi-physics engineering and scientific problems
  - A two-level software structure designed around collections of packages
  - A package is an integral unit developed by a team of experts in a particular algorithms area
- PETSc
  - It is a suite of data structures and routines for the (parallel) solution of applications modeled by partial differential equations.
  - It supports MPI, shared memory pthreads, and GPUs through CUDA or OpenCL, as well as hybrid MPI-shared memory pthreads or MPI-GPU parallelism.

Specialized Libraries



- 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
  - Optimized functions for AMD processors. It includes 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, Sparse Solvers, FFT e may other. The Parallel version uses MPI

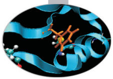
How to call a library



- first of all the sintax should be correct (read the manual!!!)
- always check for the right version
- sometimes for proprietary libraries linking could be “complicated”
- e.g. Intel ScaLAPACK

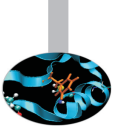
```
mpif77 <programma> -L$MKLROOT/lib/intel64 \
   -lmkl_scalapack_lp64  -lmkl_blacs_openmpi \
   -lmkl_intel_lp64 -lmkl_intel_thread -lmkl_core \
   -liomp5 -lpthread
```

Interoperability



- Many libreries are written using C, many others using Fortran
- This can produce some problems
  - type matching: C **`int`** is not granted to be the same with Fortran **`integer`**
  - symboli Match: Fortran e C++ “alter” symbol’s name producing object file (e.g. Fortran put an extra _)
- Brute force approach
  - hand-made match all types and add _ to match all librarie’s objects.
  - **`nm <object_file>`** lists all symbols
- Standard Fortran 2003 (module **`iso_c_binding`**)
  - The most important library gives you Fortran2003 interface
- In C++ command **`extern "C"`**

Interoperability/2

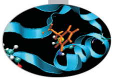


- To call libraries from C to Fortran and viceversa
- Example: **`mpi`** written using C/C++:
  - old style: **`include "mpif.h"`**
  - new style: **`use mpi`**
  - the two approach are not fully equivalent: using the module implies also a compile-time check type!
- Example: **`fftw`** written using C
  - legacy : **`include "fftw3.f"`**
  - modern:

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

- Example: **`BLAS`** written using Fortran
  - legacy: call **`dgemm_`** instead of **`dgemm`**
  - modern: call **`cblas_dgemm`**
- Standardization still lacking...
  - Read the manual . . .

BLAS: Interoperability/1



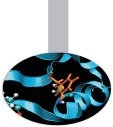
- Take a look at “netlib” web site

```
        http://www.netlib.org/blas/
```

- BLAS was written in Fortran 77, some compilatori gives you interfaces (types check, F95 features)
    - Using Intel e MKL

```
    use mkl95_blas
```

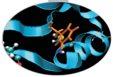
BLAS:Interoperability/2



- C (legacy):
  - add underscore to function's name
  - Fortran: argoments by reference, it is mandatory to pass pointers
  - Type matching (compiler dependent): probably **`double`**, **`int`**, **`char`** → **`double precision`**, **`integer`**, **`character`**
- C (modern)
  - use interface **`cblas`**: GSL (GNU) or MKL (Intel)
  - include header file **`#include <gsl.h>`** or **`#include<mkl.h>`**

`http://www.gnu.org/software/gsl/manual/html_node/GSL-CBLAS-Library.htm`

Hands-on: BLAS



- make an explicit call to **DGEMM** routine (BLAS).
- **DGEMM** It perform double precision matrix-matrix multiplication
- **DGEMM** : **http://www.netlib.org/blas/dgemm.f**

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

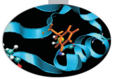
- Fortran: GNU, use **acml**:
  - **gfortran64** (serial)
  - **gfortran64_mp** (multi-threaded)

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

- Fortran: Intel, use **mkl**:
  - **sequential** (serial)
  - **parallel** (multi-threaded)

```
module load intel
module load mkl
ifort -O3 -mkl=sequential matrixmulblas.F90
```

Hands-on: BLAS/2



- C: Intel (MKL with cblas)
  - include header file **`#include<mkl.h>`**
  - try **`-mkl=sequential`** e **`-mkl=parallel`**

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

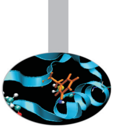
- C: GNU (GSL with cblas)
  - include l'header file **`#include <gsl/gsl_cblas.h>`**

```
module load profile/advanced
module load gnu/4.8.0 gsl/1.15--gnu--4.8.0
gcc -O3 -L$GSL_HOME/lib -lgslcblas matrixmulblas.c -I$GSL_INC
```

  - Compare with performance obtained with **`-o3`**/**`-fast`**
- Write the measured GFlops for a matrix of size 4096x4096

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

Hands-on: solutions



- Fortran:

```
call DGEMM('n','n',N,N,N,1.d0,a,N,b,N,0.d0,c,N)
```

- C (cblas):

```
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
            nn, nn, nn, 1., (double*)a, nn, (double*)b,
            nn, 0., (double*)c, nn);
```

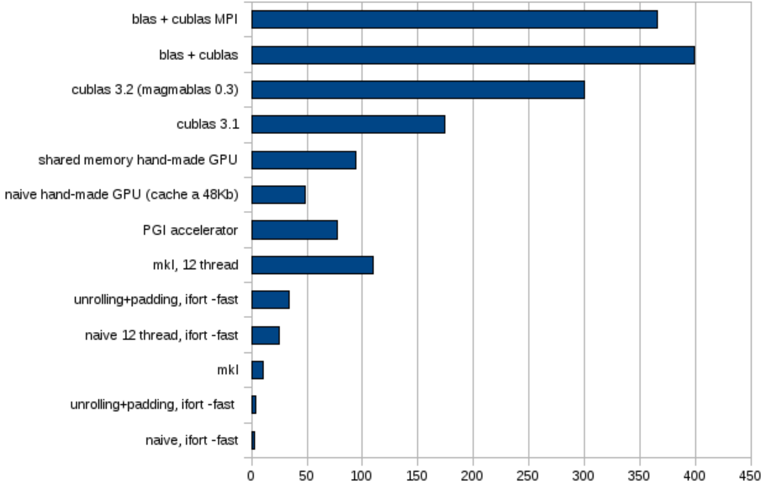
- C (legacy):

```
dgemm_(&transpose1, &transpose2, &n, &n, &n, &alfa,
      (double*)a, &n, (double*)b, &n, &beta, (double*)c, &n);
```

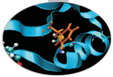
GNU -O3	Intel -fast	GNU-ACML/GSL seq	Intel-MKL seq
1.5	6.3	5.3/1.2	9.1
—	Intel -fast -parallel	GNU-ACML par	Intel-MKL par
—	75	61	75

Which performance can I reach?

- A factor 100x!!!!!



Outline

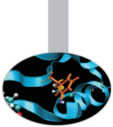


Compilers and Code optimization

Scientific Libraries

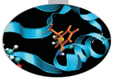
Floating Point Computing

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/not known
  - portability
  - exceptions
  - surprising behaviours with arithmetic

Bits and Integers



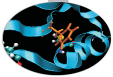
- Computers handle bits $(0/1)$
- An integer number n is stored as a sequence of bits (r)
- You have a range

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

- 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 compromizing 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

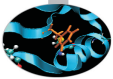
$$0 \leq i \leq 3623878656 > 2^{31} = 2147483648$$

Bits and Integers/2



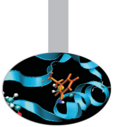
- Fortran “officially” does not let you specify the size of declared data
  - you request `kind` and the language do it for you
  - in principle very good, but interoperability must be considered with attention
- C standard types do not match exact sizes, too
  - look for `int`, `long int`, `unsigned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

Bits and Reals



- 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)

Floating point approach

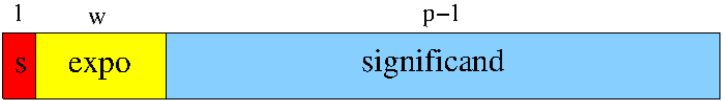


- Scientific notation:

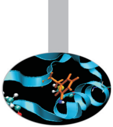
$$n = (-1)^s \cdot m \cdot \beta^e$$

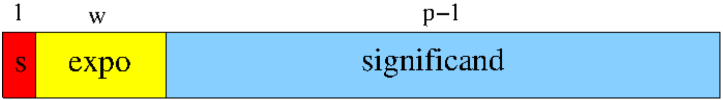
$$0.0046367 = (-1)^0 \cdot 4.6367 \cdot 10^{-3}$$

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



Exponent and Mantissa

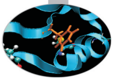


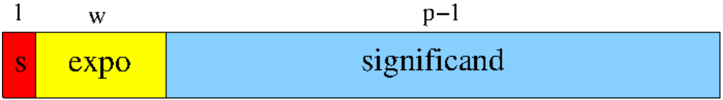


- Exponent
  - unsigned biased exponent
  - $e_{min} \leq e \leq e_{max}$
  - e_{min} must be equal to $(1 - e_{max})$
- Mantissa
  - precision p, the digits x_i are $0 \leq x_i < \beta$
  - “hidden bit” format used for normal values: 1.xx...x

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

Mantissa



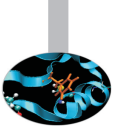


- Cons:
  - only “some” real numbers are floating point numbers (see later)
- Pros:
  - constant relative resolution (relative precision), each number is represented with the same *relative error* which is the distance from one point to the closest one divided by the number (see later)
  - wide range: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

$$1 \leq m < \beta$$

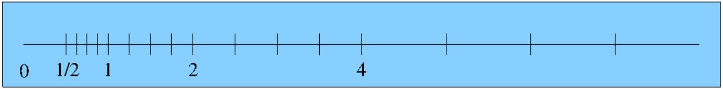
i.e. using “normal” floating-point numbers

Resolution

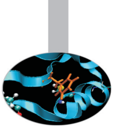


- The distance among “normal” numbers is not constant
- E.g., $\beta = 2$, $p = 3$, $e_{min} = -1$ and $e_{max} = 2$:
  - 16 positive “normalized” floating-point numbers

```
e = -1 -> 1/2 ; m = 1 + [0:1/4:2/4:3/4] ==> [4/8:5/8:6/8:7/8]
e =  0 -> 1   ; m = 1 + [0:1/4:2/4:3/4] ==> [4/4:5/4:6/4:7/4]
e = +1 -> 2   ; m = 1 + [0:1/4:2/4:3/4] ==> [4/2:5/2:6/2:7/2]
e = +2 -> 4   ; m = 1 + [0:1/4:2/4:3/4] ==> [4/1:5/1:6/1:7/1]
```



Relative Resolution



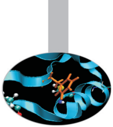
- What does it mean “constant relative resolution”?
- Given a number $N = m \cdot \beta^{e}$ the nearest number has distance

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

  - 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 $m \simeq \beta/2$)

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

Intrinsic Error

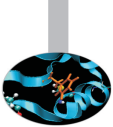


- WARNING: not any real number can be expressed as a floating point number
  - because you would need a larger exponent
  - or because you would need a larger precision
- The resolution is directly related to the intrinsic error
  - if $p = 4$, 3.472 may approximate numbers between 3.4715 and 3.4725, 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 ε is also called “machine epsilon” or “relative precision”

Measuring error



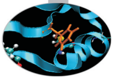
- When performing calculations, floating-point error may propagate and exceed the intrinsic error

```
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

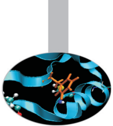
```
relative error   = 3.17−3.14145 = 0.02855
machine epsilon  = 10/2*0.001   = 0.005
relative error   = 5.71 ε
```

Handling errors



- 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 (mayby for intermediate results . . .)
  - exponent range is symmetric: if possibile, perform calculations around 1 is a good idea

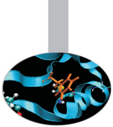
Types features



IEEE Name	min	max	ε	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 trascendental functions and increase overflow
- Extended and quad support depends on architecture and compiler: often emulated and so really 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

Error propagation



- 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

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

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

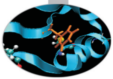
- Second idea: compute the difference with $\mathbf{p}$ digits

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

$$x - y = 0.02 \cdot 10^1 = 2,00 \cdot 10^{-2}$$

the error is 30 ulps!

Guard digit



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

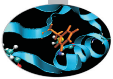
$$x = 1.010 \cdot 10^1$$

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

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

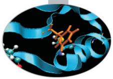
- Theorem: if x and y are floating-point numbers in a format with parameters and p, and if subtraction is done with p + 1 digits (i.e. one guard digit), then the relative rounding error in the result is less than 2 ε.

Cancellation



- 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 $70 ulps$
  - the subtraction did not introduce any error, but rather exposed the error introduced in the earlier multiplications.

Cancellation/2

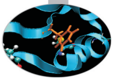


- The expression $x^2 - y^2$ is more accurate when rewritten as $(x-y)(x+y)$ because a catastrophic cancellation is replaced with a benign one
  - replacing a catastrophic cancellation by a benign one may be not worthwhile if the expense is large, because the input is often an approximation
- Eliminating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

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

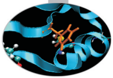
  - if $b^2 >> ac$ then $b^2 - 4ac$ does not involve a cancellation
  - but, if $b > 0$ the addition in the formula will have a catastrophic cancellation.
  - to avoid this, multiply the numerator and denominator of x_1 by $-b - \sqrt{b^2 - 4ac}$ to obtain $x_1 = (2c)/(-b - \sqrt{b^2 - 4ac})$ where no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

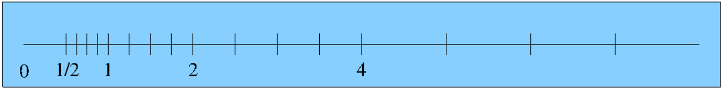
Special values



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

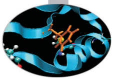
Zero and Denormals

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



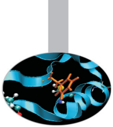
- At least we need to add the zero value
  - there are two zeros: **+0** and **−0**
- When a computation result is less than the minimum value, it could be rounded to zero or to the minimum value

Zero and Denormals/2



- 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!

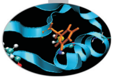
Walking Through



- Double precision: w=11 ; p=53

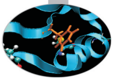
```
0x0000000000000000  +zero
0x0000000000000001  smallest subnormal
...
0x000fffffffffffff  largest subnormal
0x0010000000000000
...
0x001fffffffffffff  smallest normal
0x0020000000000000  2 X smallest normal
...
0x7fefffffffffffff  largest normal
0x7ff0000000000000  +infinity
```

Walking Through



```
0x7ff0000000000001  NaN
...
0x7fffffffffffffff  NaN
0x8000000000000000  -zero
0x8000000000000001  negative subnormal
...
0x800fffffffffffff  ’largest’ negative subnormal
0x8010000000000000  ’smallest’ negative normal
...
0xfff0000000000000  -infinity
0xfff0000000000001  NaN
...
0xffffffffffffffff  NaN
```

Error-Free Transformations

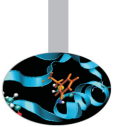


- An error-free transformation (EFT) is an algorithm which determines the rounding error associated with a floating-point operation
- E.g., addition/subtraction

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

  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**

EFT for Addition



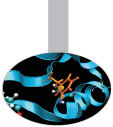
- FastTwoSum: compute $a + b = s + t$ where

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

$$s = a \oplus b$$

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

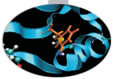
EFT for Addition/2



- 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;
```

Summation techniques

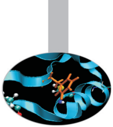


- Condition number

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

- 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

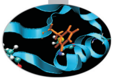


$$s = \sum_{i=0}^{n} x_i$$

```
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

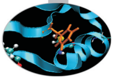
Kahan summation



- 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

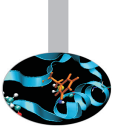
```
double Kahan( const double* a, const int n ) {
   double s = a[ 0 ];          //  sum
   double t = 0.0;             //  correction term
   for(int i=1; i<n ; i++) {
       double y = a[ i ] - t; //  next term "plus" correction
       double z = s + y;      //  add to accumulated sum
       t = (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

More



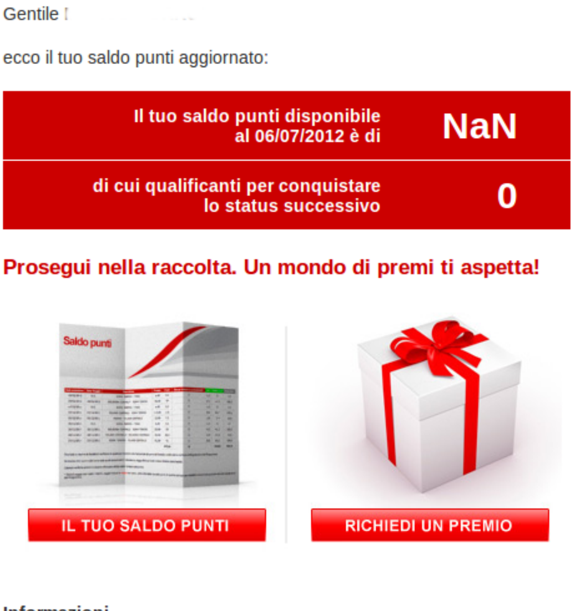
- Many variations known (Knutht, Priest,...)
- Sort the values and sum starting from smallest values (for positive numbers)
- Other techniques (distillation)
- Use a greater precision or emulate it (long accumulators)
- Similar problems for Dot Product, Polynomial evaluation,...

Exceptions (IEEE 754-2008)



- 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

Exception in real life . . .



Gentile I [redacted]

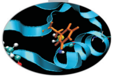
ecco il tuo saldo punti aggiornato:

Il tuo saldo punti disponibile al 06/07/2012 è di	NaN
di cui qualificanti per conquistare lo status successivo	0

Prosegui nella raccolta. Un mondo di premi ti aspetta!

Informazioni

Handling exceptions



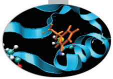
- Let us say you may produce a NaN
- What do you want to do in this case?

- 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)
```

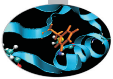
  - to be safe I should pass the domain of f but the it could be difficult to do
  - I may prefer to check each grid point **`g(i)`**
  - if the function is not defined somewhere, I will get a NaN (or other exception) but I do not care: the maximum value will be correct

Handling exceptions/2



- 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

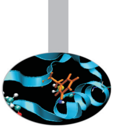
Handling exceptions/3



- (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):

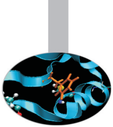
```
int IsFiniteNumber(double x) {
    return (x <= DBL_MAX && x >= -DBL_MAX);
}
```

Floating-point control



- 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!
  - $x - x = 0.0$ ...
  - $x * 0.0 = 0.0$ ...

Floating-point control/2

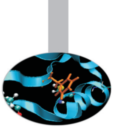


- An important case: reassociation is not safe with floating-point numbers
  - $(x+y)+z = x+(y+z)$: reassociation is not safe
  - compare

    $$-1.0 + 1.0e-13 + 1.0 \;=\; 1.0 - 1.0 + 1.0e-13 \;=\; 1.0e-13 + 1.0 - 1.0$$

  - $a * b/c$ 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)

Floating-point control/3



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

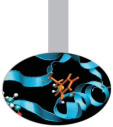
```
-f[no-]fast-math
```

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

- For more detail

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

Floating-point control/4



- Intel options:

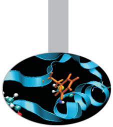
```
-fp-model <type>
```

  - 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
```

Endianness

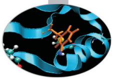


- Which is the ordering of bytes in memory? E.g.,

```
-1267006353 ===> 10110100011110110000010001101111
```

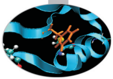
  - Big endian: `10110100 01111011 00000100 01101111`
  - Little endian: `01101111 00000100 01111011 10110100`
  - Other exotic layouts (VAX,...) nowadays unusual
  - Limits portability
- Possibile solutions
  - conversion binary to text and text to binary
  - compiler extensions(Fortran):
    - Intel: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

C and Fortran data portability



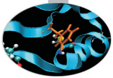
- For C Standard Library a file is written as a stream of byte
- In Fortran file is a sequence of records:
  - each read/write refer to a record
  - there is record marker before and after a record (32 or 64 bit depending on file system)
  - remember also the different array layout from C and Fortran
- 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?



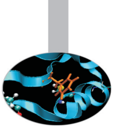
- Single, Double or Quad?
  - maybe single is too much!
  - computations get (much) slower when increasing precision, storage increases and power supply too
- Famous story
  - Patriot missile incident (2/25/91) . Failed to stop a scud missile from hitting a barracks, killing 28
  - System counted time 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

Interval arithmetic

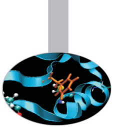


- A “correct” approach
- Interval number: possible values within a closed set

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

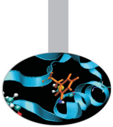
  - e.g., 1/3=0.33333 ; 1/3 $\in$ [0.3333,0.3334]
- Operations
  - Addition x + y = [a, b] + [c, d] = [a + c, b + d]
  - Subtraction x + y = [a, b] + [c, d] = [a -d, b -c]
  - . . .
- Properties are interesting and can be applied to equations
- Interval Arithmetic has been tried for decades, but often 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.

References



- 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
- W. Kahan http://www.cs.berkeley.edu/ wkahan/
- Standards: http://grouper.ieee.org/groups/754/

Hands-on: Compensated sum

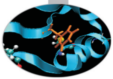


- The code in **`summation.cpp/f90`** initializes an array with an ill-conditioned sequence of the order of

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

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

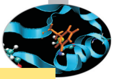
Hands-on: C++ Solution



```
REAL_TYPE summation_kahan( const REAL_TYPE a[],
                           const size_t n_values )
{
  REAL_TYPE s = a[ 0 ];          //  sum
  REAL_TYPE t = 0;               //  correction term
  for( int i = 1; i < n_values; i++ ) {
      REAL_TYPE y = a[ i ] - t;  //  next term "plus" correction
      REAL_TYPE z = s + y;       //  add to accumulated sum
      t = ( z - s ) - y;         //  t <- -( low part of y )
      s = z;                     //  update sum
  }
  return s;
}
```

```
Summation simple  :   35404.960937500000000
Summation Kahan   :   35402.851562500000000
Summation higher  :   35402.855468750000000
```

Hands-on: Fortran Solution



```fortran
function sum_kahan(a,n)
    integer :: n
    real(my_kind) :: a(n)
    real(my_kind) :: s,t,y,z

    s=a(1)                 ! sum
    t=0._my_kind           ! correction term
    do i=1,n
       y = a(i) - t        ! next term "plus" correction
       z = s + y           ! add to accumulated sum
       t = (z-s) - y       ! t <- -( low part of y )
       s = z               ! update sum
    enddo
    sum_kahan = s
end function sum_kahan
```

```
Summation simple:    7293.98193359375000
Summation Kahan:     7294.11230468750000
Summation Higher:    7294.10937500000000
```