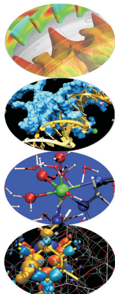
CINECA SCAI
SuperComputing Applications and Innovation

SAPIENZA UNIVERSITÀ DI ROMA

Programmazione Avanzata

Vittorio Ruggiero

(v.ruggiero@cineca.it)



Roma, Marzo 2017

Outline

Compilers and Code optimization

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:
  - 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:
  - 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: hold 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 $\rightarrow$ many istructions
  - simple decode and execution
- CISC: Complex Instruction Set CPU
  - many "complex" instructions
  - one statement $\rightarrow$ 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 way to improve performance

The Compiler

- "Translate" source code in an executable
- Rejects code with syntax errors
- Warns (sometimes) about "semantic" problems
- Try (if allowed) 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

A three-step process:

1. 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 . . .
2. Compiling:
   - each code is translated in object files
     - object files is a collection of "symbols" that refere to variables/function defined in the program
3. 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.x
```

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

- 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.x
```

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

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

- 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 many 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 is done 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 compiler up to 5)
- Usually :
  - `-O0`: no optimization is performed, simple translation (tu use with `-g` for debugging)
  - `-O`: default value (each compiler has it’s own default)
  - `-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/-Ofast` 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

Matmul: performance

- Size 1024×1024, duoble precision
- Fortran core, cache friendly loop
  - 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
```

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

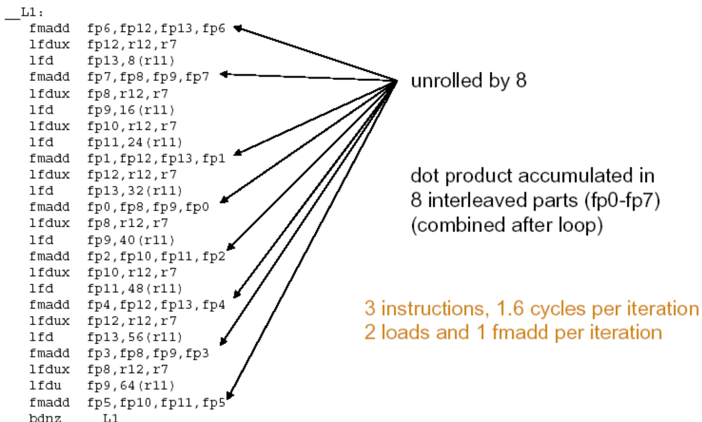
```
__L1:
      lwz     r3,160(SP)
      lwz     r9,STATIC_BSS
      lwz     r4,24(r9)
      subfi   r5,r4,-8
      lwz     r11,40(r9)
      mullw   r6,r4,r11
      lwz     r4,36(r9)
      rlwinm  r4,r4,3,0,28
      add     r7,r5,r6
      add     r7,r4,r7
      lfdx    fp1,r3,r7
      lwz     r7,152(SP)
      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 it in automatic way

Optimization/1

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



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 $\rightarrow$ integer every time
- I should be an error according to standard, but compilers are (sometimes) 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

- Using Common Subexpression for intermediate results:
  A= B+C+D
  E= B+F+C
- ask for: 4 load, 2 store, 4 sums
  A=(B+C) + D
  E=(B+C) + F
- ask for 4 load, 2 store, 3 sums, 1 intermediate result.
- 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 produce 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 are 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 temporary 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
```

Optmization 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?

Optimization 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

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. segmentation fault)
  - bash: to check

```
ulimit -a
```

  - bash: to modify

```
ulimit -s unlimited
```

Dynamic allocation using C/1

- Using C matrix $\rightarrow$ 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 present very different latencies

- sum: few clock cycles
- product: few clock cycles
- sum+product: few clock cycles
- division: many clock cycle ($O(10)$)
- sin,sos: 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:
  - many system calls
  - pipeline goes dry
  - cache coerency can be destroyed
  - it is very slow (HW limitation)

- 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)   ! dump (1)

write(69) a                                  ! dump (2)
```

Different I/O: some figures

	seconds	Kbyte
formatted	81.6	419430
binary	81.1	419430
by colunm	60.1	268435
by 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

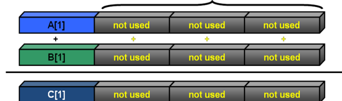
- Golden Rule #1: NEVER mix computing with I/O operations
- Golden Rule #2: NEVER read/write a single data, pack them in a block
- For HPC is possibile use:
  - I/O libraries: MPI-I/O, HDF5, NetCDF,...

Vector units

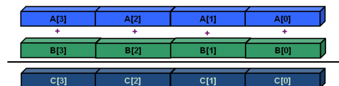
- We are not talking of vector machine
- Vector Units performs parallel floating/integer point operations on dedicate units (SIMD)
  - Intel: MMX, SSE, SSE2, SSE3, SSE4, AVX, AVX2
- i.e.: summing 2 arrays of 4 elements in one single instruction

```
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 (from Intel Core/AMD Opteron)
  - 4 floating/integer operations in single precision
  - 2 floating/integer operations in double precision
- AVX: 256 bit register (from 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)
  - 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 a 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 vectorized 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];
```

- “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)
...
```

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                  automatic parallelization
-qsmp=openmp:noauto    no automatic parallelization
```