Writing efficient programs

Thomas Ericsson
Computational Mathematics
Chalmers University of Technology
and the University of Gothenburg
thomas@chalmers.se

PDC Summer School 2010

Contents

Your situation

- A large and old code which has to be optimized. Even a slight speedup would be of use, since the code may be run on a daily basis.
- A new project, where language and data structures have to be chosen.

  C/C++ usually slower than Fortran for floating point.
  Java? Can be slow and use large amounts of memory.
  See the article (compendium) for an example.

  Should it be parallel?

  Test a simplified version of the computational kernel.
  Fortran for floating point, C/C++ for the rest.
- Things that are done once. Let the computer work.
  Unix-tools, Matlab, Maple, Mathematica ...

The optimization process

Basic: Use an efficient algorithm.

Simple things:

- Use (some of) the optimization options of the compiler.
  Optimization can give large speedups
  (and new bugs, or reveal bugs).
  - Save a copy of the original code.
  - Compare the computational results before and after optimization.
    Results may differ in the last bits and still be OK.
- Read the manual page for your compiler.
  Even better, read the tuning manual for the system.
- Switch compiler and/or system.

The next page lists the compiler options, flags, of the Intel Fortran90-compiler. There are more than 300 flags. The names are not standardized, but it is common that `-c` means "compile only, do not link". To produce debug information `-g` is used.

Some of the flags are passed on to the preprocessor (locations and names of header files) and to the linker (locations and names of libraries). The most important flags in this course are those for optimization. `-O[n]` usually denotes optimization on level `n`. There may be an option, like `-fast`, that gives a combination of suitable optimization options. Here a *few* av the more than 1000 lines produced by `icc -help` and `ifort -help`.

There is a user and reference guide, PDF (> 3800 pages, for Fortran, C++-manual 1894 pages).

• Optimization
...
-O2 optimize for maximum speed (DEFAULT)
-O3 optimize for maximum speed and enable more aggressive optimizations that may not improve performance on some programs
-O same as **-O2**
...
-O0 disable optimizations
-fast enable **-xHOST -O3 -ipo -no-prec-div -static...**
-fno-alias assume no aliasing in program
...
• Code Generation
-x<code1> generate specialized code to run exclusively on processors indicated by **<code>** as described below

• Interprocedural Optimization (IPO)
-[no-]ip enable(DEFAULT)/disable single-file IP optimization within files
-ipo[n] enable multi-file IP optimization between files
...

• Advanced Optimizations
...
-[no-]vec enables(DEFAULT)/disables vectorization
...
Here is an incomplete list of the remaining categories:
• Profile Guided Optimization (PGO)
• Optimization Reports
• OpenMP* and Parallel Processing
• Floating Point
• Inlining
• Output, Debug, PCH (pre compiled header files)
• Preprocessor
• Compiler Diagnostics
• Linking/Linker

If you are willing to work more...

- Decrease number of disk accesses (I/O, virtual memory)
- (LINPACK, EISPACK) → LAPACK
- Use numerical libraries tuned for the specific system, BLAS

Find bottlenecks in the code (profilers).
Attack the subprograms taking most of the time.
Find and tune the important loops.

Tuning loops has several disadvantages:

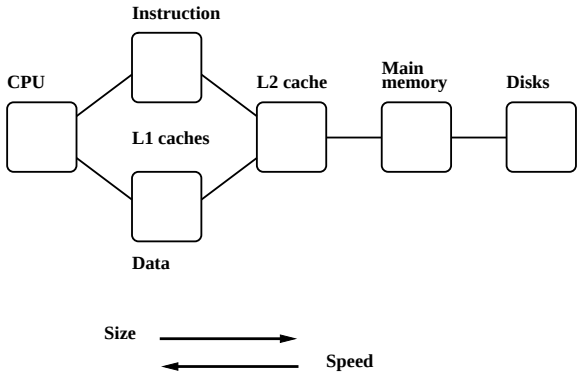
- The code becomes less readable and it is easy to introduce bugs. Compare computational results before and after tuning.
- Detailed knowledge about the system, such as cache configuration, may be necessary.
- What is optimal for one system need not be optimal for another; faster on one machine may actually be slower on another.
  This leads to problems with portability.
- Code tuning is not a very deterministic business.
  The combination of tuning and the optimization done by the compiler may give an unexpected result.
- The computing environment is not static; compilers become better and there will be faster hardware of a different construction.
  The new system may require different (or no) tuning.

What should one do with the critical loops?

The goal of the tuning effort is to keep the FPU(s) busy.

Accomplished by efficient use of the

- memory hierarchy
- parallel capabilities



Superscalar: start several instructions per cycle.
Pipelining: work on an instruction in parallel.
Vectorization: parallel computation on short arrays.

- Locality of reference, data reuse
- Avoid data dependencies and other constructions that give pipeline stalls

What can you hope for?

- Many compilers are good.
  May be hard to improve on their job.
  We may even slow the code down.
- Depends on code, language, compiler and hardware.
- Could introduce errors.
- But: can give significant speedups.

Not very deterministic, in other words.

- Do not rewrite all the loops in your code.
- Save a copy of the original code. If you make large changes to the code, use som kind of version control system.
- Compare computational results before and after tuning.

Choice of language

Fortran, C/C++ dominating languages for high performance numerical computation.
There are excellent Fortran compilers due to the competition between manufacturers and the design of the language.
It may be harder to generate fast code from C/C++ and it is easy to write inefficient programs in C++

```
void add(const double a[], const double b[],
         double c[], double f, int n)
{
  int k;

  for(k = 0; k < n; k++)
    c[k] = a[k] + f * b[k];
}
```

`n`, was chosen such that the three vectors would fit in the L1-cache, all at the same time.
On the two systems tested (in 2005) the Fortran routine was twice as fast.

From the Fortran 90 standard (section 12.5.2.9):

> "Note that if there is a partial or complete overlap between the actual arguments associated with two different dummy arguments of the same procedure, the overlapped portions must not be defined, redefined, or become undefined during the execution of the procedure."

Not so in C. Two pointer-variables with different names may refer to the same array.

A Fortran compiler may produce code that works on several iterations in parallel.

```
c(1) = a(1) + f * b(1)
c(2) = a(2) + f * b(2)  ! independent
```

Can use the pipelining in functional units for addition and multiplication.

The assembly code is often unrolled this way as well.
The corresponding C-code may look like:

```
/* This code assumes that n is a multiple of four*/
    for(k = 0; k < n; k += 4) {
      c[k]   = a[k]   + f * b[k];
      c[k+1] = a[k+1] + f * b[k+1];
      c[k+2] = a[k+2] + f * b[k+2];
      c[k+3] = a[k+3] + f * b[k+3];
    }
```

A programmer may write code this way, as well. Unrolling gives:

- fewer branches (tests at the end of the loop)
- more instructions in the loop; a compiler can change the order of instructions and can use prefetching

If we make the following call in Fortran, (illegal in Fortran, legal in C), we have introduced a data dependency.

```
call add(a, c, c(2), f, n-1)
         |  |   |
         a  b   c

c(2) = a(1) + f * c(1)  ! b and c overlap
c(3) = a(2) + f * c(2)  ! c(3) depends on c(2)
c(4) = a(3) + f * c(3)  ! c(4) depends on c(3)
```


If that is the loop you need (in Fortran) write:

```
do k = 1, n - 1
  c(k + 1) = a(k) + f * c(k)
end do
```

This loop is slower than the first one (slower in C as well).

In C, aliased pointers and arrays are allowed which means that it may be harder for a C-compiler to produce efficient code.
The C99 `restrict` type qualifier can be used to inform the compiler that aliasing does not occur.

```
void add(double * restrict a, double * restrict b,
         double * restrict c, int n)
```

It is not supported by all compilers and even if it is supported it may not have any effect (you may need a special compiler flag, e.g. `-std=c99`).

An alternative is to use compiler flags, `-fno-alias`, `-xrestrict` etc. supported by some compilers. If you "lie" (or use a Fortran routine with aliasing) you may get the wrong answer!

According to an Intel article, the C/C++-compiler can generate dynamic data dependence testing (checking addresses using if-statements) to decrease the problem with aliasing.

To see the effects of aliasing one may need more complicated examples than `add`. I have kept it because it is easy to understand. On the next page is a slightly more complicated example (Horner's method for polynomials), but still only a few lines of code, i.e. far from a real code.


```
subroutine horner(px, x, coeff, n)
  integer           j, n
  double precision  px(n), x(n), coeff(0:4), xj

  do j = 1, n
    xj    = x(j)
    px(j) = coeff(0) + xj*(coeff(1) + xj*(coeff(2) &
          + xj*(coeff(3) + xj*coeff(4))))
  end do
end
```

Using `n = 1000` and calling the routine 10^6 times the speed advantage of Fortran was a factor of 1 up to 4.
If `-fno-alias` is used, C $\approx$ Fortran.

It is easy to fix the C-code without using `-fno-alias`

```
...
  double          xj, c0, c1, c2, c3, c4;

  /* no aliasing with local variables*/
  c0 = coeff[0]; c1 = coeff[1]; c2 = coeff[2];
  c3 = coeff[3]; c4 = coeff[4];

  for (j = 0; j < n; j++) {
    xj    = x[j];
    px[j] = c0 + xj*(c1 + xj*(c2 + xj*(c3 + xj*c4)));
  }
...
```

It is instructive to compare the assembly ouput of the two Horner routines. `gcc -O3 -S horner.c` gives assembly on `horner.s`.

```
Original routine (just the loop)
.L4:
        movl    12(%ebp), %edx
        fldl    (%edx,%eax,8)
        movl    -16(%ebp), %edx
        fld     %st(0)
        fmull   (%edx)
        movl    8(%ebp), %edx
        faddl   (%ecx)
        fmul    %st(1), %st
        faddl   (%esi)
        fmul    %st(1), %st
        faddl   (%edi)
        fmulp   %st, %st(1)
        faddl   (%ebx)
        fstpl   (%edx,%eax,8)
        addl    $1, %eax
        cmpl    20(%ebp), %eax
        jne     .L4

Modified code
.L11:
        fldl    (%ecx,%eax,8)
        fld     %st(0)
        fmul    %st(2), %st
        fadd    %st(3), %st
        fmul    %st(1), %st
        fadd    %st(4), %st
        fmul    %st(1), %st
        fadd    %st(5), %st
        fmulp   %st, %st(1)
        fadd    %st(5), %st
        fstpl   (%ebx,%eax,8)
        addl    $1, %eax
        cmpl    %edx, %eax
        jne     .L11
```


I used `gcc` instead of `icc` which vectorizes the code and makes it very hard to read.

Now to Horner with complex numbers using Fortran (complex is built-in) and C++ (using "C-arrays" of `complex<double>`).

I got the following times, on three systems using `n = 1000` and calling the routine 10^5 times.

Compiling using `-O2` or `-O3`, whatever is best.
`i` = Intel, `pg` = Portland Group, `g` = GNU.

ifort	icpc	pgf90	pgCC	g95	g++
0.6	7.4	na	na	5.0 (†)	5.4
1.9	4.5	2.3	14.1	3.1	7.0
1.0	1.9	1.2	3.9	1.2 (‡)	2.6

(†) `g77` instead of `g95`. (‡) `gfortran` instead of `g95`.

The tables do show that is important to test different systems, compilers and compile-options.

The behaviour in the above codes changes when `n` becomes very large. *CPU-bound* (the CPU limits the performance) versus *Memory bound* (the memory system limits the performance).

Tuning Matlab programs

The timings below are for Matlab version 7.7.0.471 (R2008b) on Ferlin. Matlab 6.5 (and newer) has a JIT-accelerator (Just In Time). Older versions can be much slower.

- Use the built-in compiled routines.
  The Matlab-language is interpreted.
- Work on the matrix/vector-level, not on element-level.
  Different programming style.
- Take care when using the dynamic memory allocation.
  Preallocate.

Some examples, `n = 2500` everywhere on this page:

```
for i = 1:n
  for j = 1:n
    for k = 1:n
    end
  end
end
Takes 47 s      % loops take time

clear A
x = rand(n, 1);
for k = 1:n
  A(:, k) = x;  % could have different arrays
end
Takes 71 s

A = zeros(n);   % preallocate
for k = 1:n
  A(:, k) = x;
end
Takes 0.04 s
```


`W` is a 8000 × 15-matrix and `x` is a column vector having 8000 elements.

```
y = W * W' * x;          y = W * (W' * x);

Takes 1.9 s                0.0004 s
```

Note that it may be impossible just to form `W * W'` even though `y = W * (W' * x);` gives no problem.

Do not use more general functions than necessary (inline):

```
v = rand(3, 1);  w = rand(3, 1);

for k = 1:100000
  d = dot(v, w);
  v(1) = v(1) + 1e-50;  % added to the loops
end                     % below as well
Takes 1.9 s

for k = 1:100000
  c = v' * w;
end
Takes 0.07 s

for k = 1:100000
  c = cross(v, w);
end
Takes 5.9 s

for k = 1:100000
  c = [v(2)*w(3)-v(3)*w(2); v(3)*w(1)-v(1)*w(3); ...
       v(1)*w(2)-v(2)*w(1)];
end
Takes 0.4 s
```

Basic arithmetic and elementary functions

Many modern CPUs have vector units which can work in parallel on the elements of short arrays, e.g. adding two vectors. Intel has the SSE (Streaming SIMD Extensions, SIMD = Single Instruction Multiple Data).

The arrays usually consist of two double precision numbers or four single precision numbers.

The arithmetic may have different roundoff properties compared to the usual FPU (x87 in an Intel CPU) since the x87 uses extra digits (to satisfy the requirements in the IEEE floating point standard).

- Common that the (x87) FPU can perform `+` and `*` in parallel.
- `a+b*c` can often be performed with one round-off, multiply-add MADD or FMA.
- `+` and `*` usually pipelined, so one sum and a product per clock cycle in the best of cases (not two sums or two products). Often one sum every clock cycle and one product every other.
- `/` not usually pipelined and may require 15-40 clock cycles.
- Two FMAs in a few machines.
- Many modern CPUs have several computational cores as well as vector units.

Floating point formats

Type	min denormalized	min normalized	max	bits in mantissa
IEEE 32 bit	$1.4 \cdot 10^{-45}$	$1.2 \cdot 10^{-38}$	$3.4 \cdot 10^{38}$	24
IEEE 64 bit	$4.9 \cdot 10^{-324}$	$2.2 \cdot 10^{-308}$	$1.8 \cdot 10^{308}$	53

- Using single- instead of double precision can give better performance. Fewer bytes must pass through the memory system.
- The arithmetic may not be done more quickly since several systems will use double precision for the computation regardless.

The efficiency of FPUs differ (this on a 2 GHz Opteron).

```
>> A = rand(1000); B = A;
>> tic; C = A * B; toc
Elapsed time is 0.780702 seconds.

>> A = 1e-320 * A;
>> tic; C = A * B; toc
Elapsed time is 43.227665 seconds.
```


For better performance it is sometimes possible to replace a division by a multiplication.

```
vector / scalar          vector * (1.0 / scalar)
```

Integer multiplication and multiply-add are often slower than their floating point equivalents.

```
...
  integer, dimension(10000) :: arr = 1
  integer                   :: s   = 0

  do k = 1, 100000
    s = s + dot_product(arr, arr)
  end do
...
```

Change types to `real` and then to `double precision` here are the times on three systems:

integer	single	double
1.7	0.58	0.39
1.0	1.6	1.6
0.92	0.22	0.44

Elementary functions

Often coded in C, may reside in the `libm`-library.

- argument reduction
- approximation
- back transformation

Can take a lot of time.

```
>> v = 0.1 * ones(1000, 1);
>> tic; for k = 1:1000, s = sin(v); end; toc
elapsed_time =
    0.039218

>> v = 1e10 * ones(1000, 1);
>> tic; for k = 1:1000, s = sin(v); end; toc
elapsed_time =
    0.717893
```

```
program ugly
  double precision :: x = 2.5d1
  integer          :: k

  do k = 1, 17, 2
    print'(1p2e10.2)', x, sin(x)
    x = x * 1.0d2
  end do

end program ugly
```

```
% a.out
  2.50E+01 -1.32E-01
  2.50E+03 -6.50E-01
  2.50E+05 -9.96E-01
  2.50E+07 -4.67E-01
  2.50E+09 -9.92E-01
  2.50E+11 -1.64E-01
  2.50E+13  6.70E-01
  2.50E+15  7.45E-01
  2.50E+17  4.14E+07   <---
```

Some compilers are more clever than others, which is shown on the next page.

You should know that, unless x is an integer, v^x is computed using something like:

$$v^x = e^{\log(v^x)} = e^{x \log v}, \quad 0 < v, x$$


```
subroutine power(vec, n)
  integer                          :: k, n
  double precision, dimension(n) :: vec

  do k = 1, n
    vec(k) = vec(k)**1.5d0  ! so vec(k)^1.5
  end do

end
```

Times with `n = 10000` and called `10000` on a 2 GHz AMD64.

Compiler `-O3`	power	opt. power
Intel	1.2	1.2
`g95`	8.2	1.6
`gfortran`	8.1	1.6

Looking at the assembly output from Intel's compiler:

```
 ...
       fsqrt                    <---- NOTE
       fmulp     %st, %st(1)    <---- NOTE
 ...
```

`g95` and `gfortran` call `pow` (uses exp and log).

In "opt. power" I have written the loop this way:

```
 ...
  do k = 1, n
    vec(k) = sqrt(vec(k))* vec(k)
  end do
```


There may be vector versions of elementary functions as well as slightly less accurate versions. AMD's ACML and Intel's MKL both have vector-versions.
Here an example using MKL's VML (Vector Mathematics Library). Read the manual for details (how to use `vmlSetMode` to set the accuracy mode, for example).

```
...
  include mkl_vml.fi

  integer, parameter             :: n = 100000
  double precision, dimension(n) :: v, sinv

   v = ...
   call vdsin(n, v, sinv) ! vector-double-sin
...
```

Performance depends on the type of function, range of arguments and vector length. Here are a few examples runs (1000 repetitions with `n` as above). The routines are threaded but seemed to perform best on one thread.

Function	loop	vec	less acc. vec	prec
sin	2.3	0.49	0.40	single
exp	1.6	0.36	0.33	
atan	2.1	0.83	0.51	
sin	3.0	1.3	1.3	double
exp	2.1	0.8	0.8	
atan	7.2	2.2	2.0	

loop means using the standard routine and a loop (or equivalently `sinv = sin(v)`). vec uses the vector routine from VML and less acc. uses the less accurate version.

Newer Intel compilers use vectorized routines automatically.

An SSE-example

We need an optimizing compiler that produces code using the special vector instructions (or we can program in assembly). For example (using the default compiler):

```
% ifort -O3 -xSSE3 -vec_report3    files...
dot_ex.f90(34) : (col. 3) remark: LOOP WAS VECTORIZED.
```

```
! A simple benchmark
  s = 0.0
  do k = 1, 10000
    s = s + x(k) * y(k)
  end do
```

Called 100000 times. Here are some typical times on two systems:

single		double	
no vec	vec	no vec	vec
1.60	0.38	1.80	0.92
0.83	0.41	0.99	0.80

Some compilers vectorize automatically.

Sppedup may differ, also not all codes can be vectorized.

Disadvantage: the x87-FPU uses double extended precision, 64 bit mantissa. SSE2 uses 24 bits (single precision) or 53 bits (double precision). You may get different results.

Eliminating constant expressions from loops

```
pi = 3.14159265358979d0
do k = 1, 1000000
  x(k) = (2.0 * pi + 3.0) * y(k)  ! eliminated
end do

do k = 1, 1000000
  x(k) = exp(2.0) * y(k)      ! probably eliminated
end do


do k = 1, 1000000
  x(k) = my_func(2.0) * y(k)  ! cannot be eliminated
end do
```

Should use **PURE** functions, **my_func** may have side-effects.

Virtual memory and paging

- Simulate larger memory using disk.
- Virtual memory is divided into pages, perhaps 4 or 8 kbyte.
- Moving pages between disk and physical memory is known as paging.
- Avoid excessive use. Disks are slow.
- Paging can be diagnosed by using your ear (if you have a local swap disk), or using the **sar**-command, **sar -B interval count**, so e.g. **sar -B 1 3600** . **vmstat** works on some unix-systems as well and the **time**-command built into **tcsh** reports a short summary.

Input-output

We need to store 10^8 double precision numbers in a file.
A local disk was used for the tests. Intel's Fortran compiler on an Intel Core Duo. Roughly the same times in C.

Test	Statement	time (s)	size (Gbyte)
1	`write(10, '(1pe23.16)') x(k)`	415.1	2.24
2	`write(10) x(k)`	274.4	1.49
3	`write(10) (vec(j), j = 1, 10000)`	1.1	0.74

In the third case we write $10^8/10^4$ records of 10^4 numbers each.

File sizes:

$$1: \quad \underbrace{10^8}_{\#\text{ of numbers}} \cdot \underbrace{(23+1)}_{\text{characters + newline}} / \underbrace{2^{30}}_{\text{Gbyte}} \approx 2.24$$

$$2: \quad \underbrace{10^8}_{\#\text{ of numbers}} \cdot \underbrace{(8+4+4)}_{\text{number + delims}} / \underbrace{2^{30}}_{\text{Gbyte}} \approx 1.49$$

$$3: \quad \left[\underbrace{10^8}_{\#\text{ of numbers}} \cdot \underbrace{8}_{\text{number}} + (10^8/10^4) \cdot \underbrace{(4+4)}_{\text{delims}} \right] / \underbrace{2^{30}}_{\text{Gbyte}} \approx 0.74$$

Portability of binary files?

- Perhaps
- File structure may differ
- Byte order may differ
- Big-endian, most significant byte has the lowest address ("big-end-first").
- The Intel processors are little-endian ("little-end-first").

```
On a big-endian machine
write(10) -1.0d-300, -1.0d0, 0.0d0, 1.0d0, 1.0d300


Read on a little-endian
2.11238712E+125  3.04497598E-319  0.
3.03865194E-319 -1.35864115E-171
```

Optimizing for locality, a few examples
Data re-use; loop fusion

```
v_min = v(1)
do k = 2, n
  if ( v(k) < v_min ) v_min = v(k) ! fetch v(k)
end do

v_max = v(1)
do k = 2, n
  if ( v(k) > v_max ) v_max = v(k) ! fetch v(k) again
end do
```

Merge loops data re-use, less loop overhead.

```
v_min = v(1)
v_max = v(1)
do k = 2, n
  if ( v(k) < v_min ) then     ! v(k) is fetched here
    v_min = v(k)
  elseif ( v(k) > v_max ) then ! and re-used here
    v_max = v(k)
  end if
end do
```

On some systems the following loop body is faster

```
  vk = v(k)                 ! optional
  if(v_min < vk) v_min = vk ! can use v(k) instead
  if(v_max > vk) v_max = vk
```

or

```
  vk    = v(k)
  v_min = min(v_min, vk)
  v_max = max(v_max, vk)
```


When dealing with large, but unrelated, data sets it may be faster to split the loop in order to use the caches better. Here is a contrived example:

```
  integer, parameter                :: n = 5000
  double precision, dimension(n, n) :: A, B, C, D
  ...
  sum_ab = 0.0
  sum_cd = 0.0
  do col = 1, n
    do row = 1, n  ! the two sums are independent
      sum_ab = sum_ab + A(row, col)* B(col, row)
      sum_cd = sum_cd + C(row, col)* D(col, row)
    end do
  end do
!
! Split the computation
!
  sum_ab = 0.0
  do col = 1, n
    do row = 1, n
      sum_ab = sum_ab + A(row, col)* B(col, row)
    end do
  end do

  sum_cd = 0.0
  do col = 1, n
    do row = 1, n
      sum_cd = sum_cd + C(row, col)* D(col, row)
    end do
  end do
```

When `n = 5000` the first loop requires 4.9 s and the second two 0.84 s (together) on a 2.4 GHz, 4 Gbyte, Opteron.

The importance of small strides

If no data re-use, try to have locality of reference.

Small strides.
`v(1), v(2), v(3),..`, stride one
`v(1), v(3), v(5),..`, stride two

```
  slower                      faster
  s = 0.0                     s = 0.0
  do row = 1, n               do col = 1, n
    do col = 1, n               do row = 1, n
      s = s + A(row, col)         s = s + A(row, col)
    end do                      end do
  end do                      end do

  A(1, 1)
  A(2, 1)
    ...    first column
  A(n, 1)
----------
  A(1, 2)
  A(2, 2)
    ...    second column
  A(n, 2)
----------
    ....
----------
  A(1, n)
  A(2, n)
    ...    n:th column
  A(n, n)
```

Some compilers can switch loop order (loop interchange).
In C the leftmost alternative will be the faster.

Performance on three systems. Compiling using `-O3` in the first test and using `-O3 -ipo` in the second.

	C	Fortran	C	Fortran	C	Fortran
By row	0.7 s	2.9 s	0.6 s	2.4 s	0.5 s	1.5 s
By column	4.6 s	0.3 s	2.4 s	0.6 s	1.6 s	0.5 s
By row `-ipo`	0.3 s	0.3 s	0.6 s	0.6 s	0.5 s	0.5 s
By column `-ipo`	2.9 s	0.3 s	0.6 s	0.6 s	1.5 s	0.5 s

`-ipo`, interprocedural optimization i.e. optimization between routines (even in different files) gives a change of loop order, at least for Fortran, in this case. Some Fortran compilers can do this just specifying `-O3`, and this happens Ferlin if we put the main-program and the subroutines in the same file.

```
ferlin > ifort -O3 main.f90 sub.f90    Separate files
sub.f90(27): remark: LOOP WAS VECTORIZED.

ferlin > ifort -O3 -ipo main.f90 sub.f90
ipo: remark #11000: performing multi-file optimizations
ipo: remark #11005: generating object file /tmp/ipo_ifo
main.f90(13): remark: PERMUTED LOOP WAS VECTORIZED.
main.f90(19): remark: LOOP WAS VECTORIZED.

ferlin > ifort -O3 all.f90           One file
all.f90(13): remark: PERMUTED LOOP WAS VECTORIZED.
all.f90(20): remark: LOOP WAS VECTORIZED.
all.f90(52): remark: LOOP WAS VECTORIZED.
```

Blocking and large strides

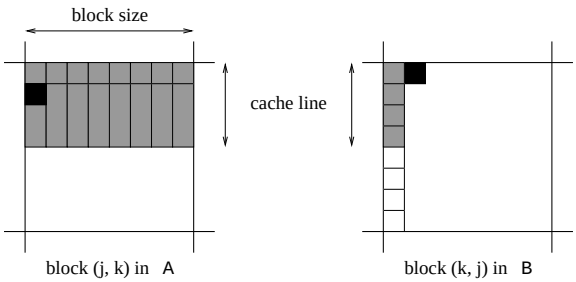
Sometimes loop interchange is of no use.

```
s = 0.0
do row = 1, n
  do col = 1, n
    s = s + A(row, col) * B(col, row)
  end do
end do
```

Blocking is good for data re-use, and when we have large strides.

Partition **A** and **B** in square sub-matrices each having the same order, the block size.

Treat pairs of blocks, one in **A** and one in **B** such that we can use the data which has been fetched to the L1 data cache.
Looking at two blocks:



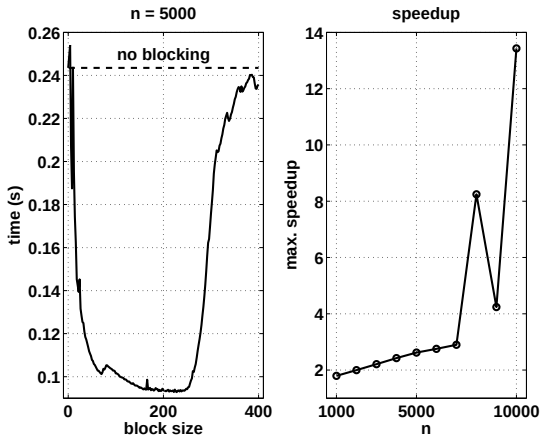
The block size must not be too large. Must be able to hold all the grey elements in **A** in cache (until they have been used).

This code works even if **n** is not divisible by the block size).

```
! first_row = the first row in a block etc.

do first_row = 1, n, block_size
  last_row = min(first_row + block_size - 1, n)
  do first_col = 1, n, block_size
    last_col = min(first_col + block_size - 1, n)
    do row = first_row, last_row      ! sum one block
      do col = first_col, last_col
        s = s + A(row, col) * B(col, row)
      end do
    end do
  end do
end do
```

Tested for $n = 10^3, 2 \cdot 10^3, \ldots, 10^4$, using **ifort -O3** on an Intel Core Duo. The left plot shows time for $n = 5000$ and different block sizes. The second figure shows the speedup for the different n-values and the optimal block size.



One can study the behaviour in more detail.

PAPI = Performance Application Programming Interface
http://icl.cs.utk.edu/papi/index.html
PAPI uses hardware performance registers, in the CPU, to count different kinds of events, such as L1 data cache misses and TLB-misses.

TLB = Translation Lookaside Buffer, a cache in the CPU that is used to improve the speed of translating virtual addresses into physical addresses.

See the Springer article for an example.

Two important libraries

BLAS (the Basic Linear Algebra Subprograms) are the standard routines for simple matrix computations.
(**s** single, **d** double, **c** complex, **z** double complex). Examples:

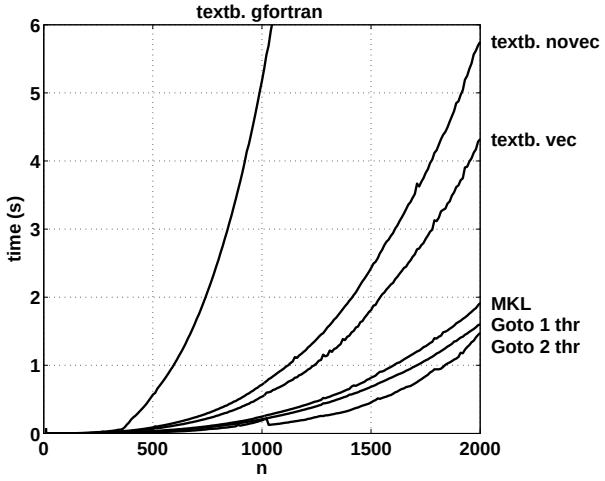
BLAS1: **y := a*x + y** one would use **daxpy**

BLAS2: **dgemv** can compute **y := a*A*x + b*y**

BLAS3: **dgemm** forms **C := a*A*B + b*C**

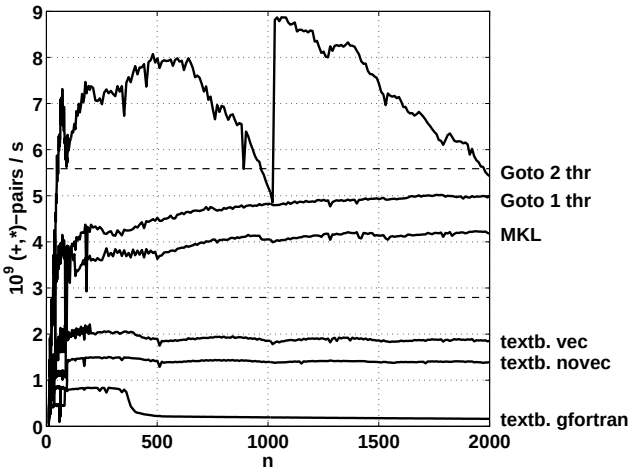
daxpy: $\mathcal{O}(n)$ data, $\mathcal{O}(n)$ operations
dgemv: $\mathcal{O}(n^2)$ data, $\mathcal{O}(n^2)$ operations
dgemm: $\mathcal{O}(n^2)$ data, $\mathcal{O}(n^3)$ operations, data **re-use**

Multiplication of $n \times n$-matrices, Intel Core Duo.



Tested textbook "row times column" using **gfortran** and **ifort** with and without vectorization. MKL is Intel's MKL-library. Goto is Goto-BLAS by Kazushige Goto.
The fast codes use blocking and other tricks. A goal of Goto-BLAS is to minimize the number of TLB-misses.
Goto-BLAS on two threads is roughly equal to MKL on two threads.

The following figure shows the number of (+, *)-pairs executed per second. The dashed lines show the clock frequency and twice the frequency.



LAPACK is the standard library for (dense):

- linear systems
- eigenvalue problems
- linear least squares problems

There is no support for large sparse problems, although there are routines for banded matrices of different kinds.

LAPACK is built on top of BLAS (BLAS3 where possible). When using LAPACK, it is important to have optimized BLAS.

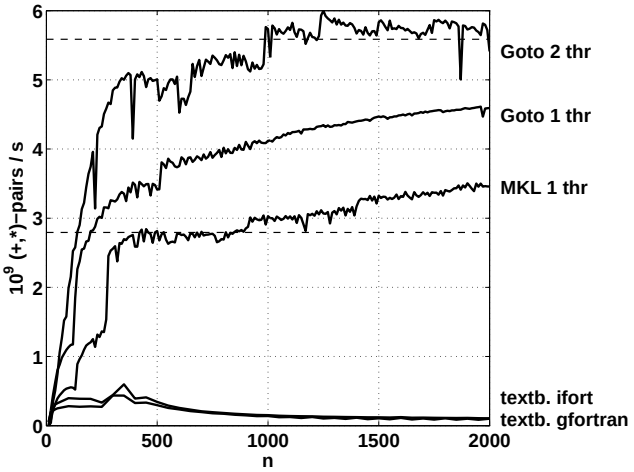
In this example we compute the Cholesky decomposition of a symmetric and positive definite matrix A, so $A = CC^T$, where C is undertriangular.

"textbook", in the figure on the next page, is one common way, often presented in textbooks, for computing C.
Here is a Matlab-code:

```
n = length(A);
for k = 1:n
  A(k, k) = sqrt(A(k, k) - sum(A(k, 1:k-1).^2));
  for i = k+1:n
    A(i, k) = (A(i, k) - ...
          sum(A(i, 1:k-1) .* A(k, 1:k-1))) / A(k, k);
  end
end
```

The number of + and * is roughly $n^3/6$.

The following figure shows the results of five runs. The textbook algorithm compiled with **gfortran** and **ifort**.
dpotrf is an LAPACK routine for computing C and the tests used **gfortran**, Goto-BLAS (and LAPACK) with one and two threads respectively. Intel's MKL using **dpotrf** on one thread.



Do not use simplistic algorithms from textbooks!

Inlining

moving the body of a short procedure to the calling routine.

Calling a procedure or a function takes time and may break the pipelining. So the compiler (or the programmer) can move the body of a *short* subprogram to where it is called. Some compilers do this automatically when the short routine resides in the same file as the calling routine. A compiler may have a flag telling the compiler to look at several files. Using some compilers you can specify which routines are to be inlined.

Indirect addressing, pointers

Sparse matrices, PDE-meshes...
Bad memory locality, poor cache performance.

```
do k = 1, n
  j    = ix(k)
  y(j) = y(j) + a * x(j)
end do
```

system	random **ix**	ordered **ix**	no **ix**
1	39	16	9
2	56	2.7	2.4
3	83	14	10

If-statements

If-statements in a loop may stall the pipeline. Modern CPUs and compilers are rather good at handling branches, so there may not be a large delay.

```
Original version          Optimized version

do k = 1, n               take care of k = 1
  if ( k == 1 ) then      do k = 2, n
    statements              statements for k = 2 to n
  else                    end do
    statements
  end if
end do
```

```
if ( most probable ) then
  ...
else if ( second most probable ) then
  ...
else if ( third most probable ) then
  ...
```

`if (a(k) .and.  b(k)) then`, least likely first

`if (a(k) .or.  b(k)) then`, most likely first

Alignment

```
integer*1  work(100001)
...
! work(some_index) in a more general setting
call do_work(work(2), 12500) ! pass address of work(2
...
end

subroutine do_work(work, n)
integer           n
double precision  work(n)

work(1) = 123
...
```

May produce “Bus error”.
Alignment problems.

It is usually required that double precision variables are stored at an address which is a multiple of eight bytes (multiple of four bytes for a single precision variable).

The slowdown caused by misalignment may easily be a factor of 10 or 100.

Closing notes

Two basic tuning principles:

- Improve the memory access pattern
  - Locality of reference
  - Data re-use

  Stride minimization, blocking, proper alignment and the avoidance of indirect addressing and aliasing.
- Use parallel capabilities of the CPU
  - Avoid data dependencies and aliasing
  - Loop unrolling
  - Inlining
  - Elimination of if-statements

Choosing a good algorithm and a fast language, handling files in an efficient manner, getting to know ones compiler and using tuned libraries are other very important points.