Writing efficient programs

Thomas Ericsson

Computational Mathematics
Chalmers University of Technology
and the University of Gothenburg

PDC Summer School 2013

Table of Contents I

Table of Contents II

- How does one get good performance from a computer system?
- Focus on floating point performance on one core.
- To get maximum performance from a parallel code it is important to tune the code running on each core.
- General advice and not specific systems.
- Fortran, some C (hardly any C++) and some Matlab.

5: 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.
  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 ...

6: 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.

7: The optimization process — The Intel compiler

- Compiler options, flags, of the Intel Fortran90-compiler, more than 300.
- Names not standardized.
- 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).
- There are user and reference guides in PDF (thousands of pages).
- Here a few av the more than 1000 lines produced by **`icc -help`** and **`ifort -help`**.
- Other compilers have similar options (often with the same names).

8: The optimization process — The Intel compiler

- 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 may be slower, TE's comment
-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 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)
-c compile to object (.o) only, do not link
-S compile to assembly (.s) only, do not link
-o <file> name output file
-g produce symbolic debug information in object file (implies **-O0** when another optimization option is not explicitly set)
• Preprocessor
• Compiler Diagnostics
• Linking/Linker

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

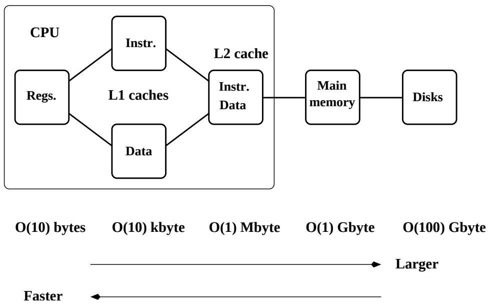
Find bottlenecks in the code (profilers). Attack the subprograms taking most of the time. Find and tune the important loops.
Tuning loops has several disadvantages:

- The code becomes less readable and you may introduce bugs.
- 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.

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 and prevent vectorization

Keywords: memory locality, data dependencies

- 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.

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++. Now a toy example.

```
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 some platforms the Fortran routine can be 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, this is called aliasing.

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:

```
// Assuming 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 due to the data dependency.
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,  etc.)
```

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

21: Choice of language

An alternative is to use compiler flags, **-fno-alias** , **-xrestrict** etc. supported by <u>some</u> compilers.
If you "lie" (or use a Fortran routine with aliasing) you probably get the wrong answer!
According to an Intel article, their 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 we modify **add**.

```
void add_f(double *a, double *b, double *c,
           double *f, int n)
{
  for(int k = 0; k < n; k++) {
    c[k] = a[k] + *f * b[k];  // *f
    *f += 1e-7;               // update f
  }
}
```

22: Choice of language

Now for a test. n=5000 and calling the routines 100000 times.
add is the original routine.
add_f is the routine above.
add_f_tmp uses a temporary **f**, local to the function.

```
  double tmp = *f;  // tmp is local to add_f_tmp


  for(int k = 0; k < n; k++) {
    c[k] = a[k] + tmp * b[k];
    tmp += 1e-7;
  }
  *f = tmp;
```

add_f_res like **add_f** with **restrict**.
add_dep calls **add(a, c, &c[1], f, n-1);**

23: Choice of language

Here are some times on an Opteron, Bulldozer, using four different compilers with suitable compiler options (but assuming aliasing).
Fortran is like **add** and Fortran_f like **add_f**.

Code	**Intel**	**PathScale**	**PGI**	**GNU**
Fortran	0.4	0.5	0.4	0.5
Fortran_f	0.4	0.8	0.8	0.9
add	0.4	0.5	0.6	0.5
add_f	2.9	2.9	3.1	2.9
add_f_tmp	0.4	0.8	0.8	0.9
add_f_res	0.4	2.9	3.1	2.9
add_dep	3.5	2.7	2.7	3.5

24: Choice of language

It is instructive to compare the assembly output of **add_f** and **add_f_tmp**.
gcc -O3 -S prog.c gives assembly output on **prog.s**.
I used **gcc** since it generates simple code.
We expect at least two loads, **a[k]** and **b[k]**, and one store, **c[k]**, for each iteration in the loop.
First **1e-7** is stored in a register, then come the loops:

```
         add_f                                    add_f_tmp
  movsd  .LC0(%rip), %xmm1                 movsd  .LC0(%rip), %xmm2
  ...                                      ...
.L15:                                    .L20:
  mulsd  (%rsi,%rax,8), %xmm0              movsd  (%rsi,%rax,8), %xmm0
  addsd  (%rdi,%rax,8), %xmm0              mulsd  %xmm1, %xmm0
  movsd  %xmm0, (%rdx,%rax,8)              addsd  %xmm2, %xmm1
  addq   $1, %rax                          addsd  (%rdi,%rax,8), %xmm0
  cmpl   %eax, %r8d                        movsd  %xmm0, (%rdx,%rax,8)
  movsd  (%rcx), %xmm0 load f              addq   $1, %rax
  addsd  %xmm1, %xmm0                      cmpl   %eax, %r8d
  movsd  %xmm0, (%rcx) store f             jg     .L20
  jg     .L15                              movsd  %xmm1, (%rcx)
```

Note the two extra memory references in **add_f**.

A few more comments.

You can read aliased values in Fortran, but you must not change them.

In C: variables local to a function are not aliased (inside the function), e.g:

```
double func( ...
{
  double a[100], b[100]; // not aliased (in func)
  double *pa, *pb;

  pa = malloc(n * sizeof(double));  // not aliased
  pb = malloc(n * sizeof(double));  // (in func)

  ...
```


Now **add** with complex numbers using Fortran (complex is built-in) and C++ ("C-arrays" of **complex<double>**).
-fno-alias does not help **icpc**.

Code	Intel	PathScale	PGI	GNU
Fortran	1.1	1.0	1.0	1.4
Fortran_f	1.3	6.4	1.4	1.3
add	3.7	14.9	11.2	2.4
add_f	5.1	17.8	14.1	3.1
add_f_tmp	4.4	17.9	13.3	3.1
add_dep	8.9	14.8	10.9	5.7

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).

The timings below are for Matlab version R2012a on a 2.27GHz Intel Xeon. Matlab 6.5 (and newer) has a JIT-accelerator (Just In Time) which is quite effective.

- Use the built-in compiled routines. The Matlab-language is interpreted (unless JIT can be applied).
- 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**.
It may make a difference if the loops are packaged in a script-file or in a function (may be faster).
Say you want to save a large number of vectors for later analysis.


```
x = rand(n, 1);
A = zeros(n);   % preallocate
for k = 1:n
  A(:, k) = x;  % would have different arrays
end             0.03 s

clear A
for k = 1:n
  A(:, k) = x;  % terrible in R2010 and earlier
end             0.08 s

A = [];
for k = 1:n
  A = [A, x];  % same speed as this one
end            0.07s, 56 s if in a script-file
```

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

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

1.2 s                        0.0003 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);          % inner product
  v(1) = v(1) + 1e-50;    % added to the loops
end                       % below as well
2.5 s

for k = 1:100000
  c = v' * w;
end
0.1 s
```



```
for k = 1:100000
  c = cross(v, w);
end
Takes 8.5 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 0.06 s
```


Many modern CPUs have vector units which can work in parallel on the elements of short arrays, .e.g. Intel's SSE (Streaming SIMD Extensions). SIMD = Single Instruction Multiple Data.
Arrays consist of two double precision numbers or four single precision numbers.

In 2011 Intel released its Sandy Bridge CPU, which can perform four double precision (eight single) multiply-adds in parallel, AVX (Advanced Vector Extensions). AMD's Bulldozer CPU also supports AVX.

The vector-arithmetic may have different roundoff properties compared to the usual FPU (x87 in an Intel CPU).

If you do not vectorize but use the usual FPU.

- 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.
- May have several computational cores as well as vector units.

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 (x87). Using vectorization, single is usually faster.

The efficiency of FPUs differ (this on a 2.66 GHz Intel Xeon).

```
» A = rand(1000); B = A;
» tic; C = A * B; toc % takes 0.19 seconds.
» A = 1e-320 * A;
» tic; C = A * B; toc % takes 64 seconds.
```


Try to avoid division:

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

Integer multiplication and multiply-add can be 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**. A few tests:

integer (32 bit)	single	double
0.5	0.18	0.36
0.7	1.0	1.0
1.0	1.6	1.6
0.92	0.22	0.44
0.74	0.20	0.27

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

- argument reduction
- approximation
- back transformation

Can take a lot of time (much more than **+**, *****).

```
» v = 0.1 * ones(10000, 1);
» tic; for k = 1:1000, s = sin(v); end; toc
Elapsed time is 0.089619
% time increases after pi/4
» v = 1e5 * ones(10000, 1); tic etc.
Elapsed time is 0.352703 seconds.

» v = 1e10 * ones(10000, 1); tic etc.
Elapsed time is 1.711913
```

```
double precision :: x = 2.5d1

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

% 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. Unless x is an integer, v^x is computed like this:

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

```
double precision, dimension(n) :: vec

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

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

Compiler `-O3`	code above	my opt. code
Intel	1.2	1.2
`gfortran`	8.1	1.6

Looking at the assembly output from Intel's compiler:

```
fsqrt
fmulp     %st, %st(1)
```

`gfortran` calls `pow` (uses exp and log).
In my optimized routine I have written the loop this way:

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

Interesting when dealing with $1/r^2$-forces.

$$F = c\frac{r/|r|}{|r|^2} = \frac{c\,r}{|r|^3} = \frac{c\,r}{\left(\sqrt{r_1^2 + r_2^2 + r_3^2}\right)^3} = \frac{c\,r}{(r_1^2 + r_2^2 + r_3^2)^{1.5}}$$

Vector versions of elementary functions as well as slightly less accurate versions area available in AMD's ACML and Intel's MKL. Performance depends on the type of function, range of arguments and vector length. With n=100000 and 1000 repetitions (one one thread, seems optimal).

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: standard routine and a loop (or `sinv = sin(v)`).
vec: vector routine from VML and less acc: less accurate version.
Newer Intel compilers use vectorized routines automatically.

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

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

Called 100000 times. Here are some typical times on three systems (the last has 256-bit SSE-instructions):

single		double	
no vec	vec	no vec	vec
1.60	0.38	1.80	0.92
0.83	0.41	0.99	0.80
1.54	0.28	1.53	0.46

Some compilers vectorize automatically. Speedup may differ. You may get different results using vectorization (due to different round-off properties).

Not all codes can be vectorized:

```
double precision              :: a, b
double precision, dimension(n) :: v

do k = 2, n - 1
  v(k + 1) = a * v(k) + b * v(k - 1)
end do

% ifort -c -O3 -vec_report=3 rec.f90
  loop was not vectorized: existence of
    vector dependence.
  vector dependence: assumed FLOW dependence
    between v line 8 and v line 8.

% pgf90 -c -O3 -Mvect -Mneginfo=vect rec.f90
    Loop not vectorized: data dependency
```



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

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

- 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.

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)	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:

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

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

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

- 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").
- Compilers may have conversion flags.

```
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
```


Compute **min(v)** and **max(v)**, where **v** is a vector.

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

In **v_min = v(k)**, **v(k)** is stored in a register and not fetched again.

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

49: Optimizing for locality, data re-use, loop fusion

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     ! fetch
    v_min = v(k)
  elseif ( v(k) > v_max ) then ! re-use
    v_max = v(k)
  end if
end do

  if(v_min < vk) v_min = v(k) ! may be faster
  if(v_max > vk) v_max = v(k) ! on some systems

  v_min = min(v_min, v(k)) ! or like this
  v_max = max(v_max, v(k))
```

50: Optimizing for locality, loop-splitting

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 = 30000
double precision, dimension, allocatable(:,:) &
  :: A,B,C,D,E,F ! 40 Gbyte matrix storage
...
allocate(A(n, n)) ! allocate (the matrices)
A = 1.0d0         ! and initialize
sum_ab = 0.0;   sum_cd = 0.0;   sum_ef = 0.0
do col = 1, n
  do row = 1, n  ! independent sums
    sum_ab = sum_ab + A(row, col) * B(col, row)
    sum_cd = sum_cd + C(row, col) * D(col, row)
    sum_ef = sum_ef + E(row, col) * F(col, row)
  end do
end do
```

51: Optimizing for locality, loop-splitting

```
  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
! and similarly for sum_cd and sum_ef
```

On a 48 Gbyte 2.66 GHz Intel Xeon 5650 the first loop took **126.8** s and the second three $\mathbf{3} \times \mathbf{19.5} = \mathbf{58.5}$ s (together). Speedup depends on **n**, hardware and compiler.

Loop splitting is worth trying only if the matrices are large.

52: The importance of small strides

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

Some compilers can switch loop order (loop interchange).

```
First loop     Second loop
A(1, 1)        A(1, 1)        First column
A(2, 1)        A(2, 1)
A(3, 1)        A(3, 1)
  ...            ...
A(n, 1)        A(n, 1)
A(1, 2)        A(1, 2)        Second column
A(2, 2)        A(2, 2)
  ...            ...
A(n, 2)        A(n, 2)
          ...
A(1, n)        A(1, n)        n:th column
A(2, n)        A(2, n)
  ...            ...
A(n, n)        A(n, n)
```

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** (if **s** is local or the return value of a function).


```
function add1(A, n) result(s)
  integer :: n, i, j, k
  double precision, dimension(n, n, n) :: A
  double precision  :: s

  s = 0.0d0
  do i = 1, n
    do j = 1, n
      do k = 1, n
        s = s + A(i, j, k)
      end do
    end do
  end do

end
```



```
function add2(A, n) result(s)
 ...
  s = 0.0d0
  do k = 1, n
    do j = 1, n
      do i = 1, n
        s = s + A(i, j, k)
      end do
    end do
  end do
```

With $n = 500$, **add1** takes 2.3s on an Intel Core Duo using gfortran. **add2** takes 0.18s.
On an AMD Bulldozer the times are 1.1s and 0.24s.
Some compilers give equal times, due to loop interchange (**ifort -O3** gives 0.12s for both loops on the AMD).

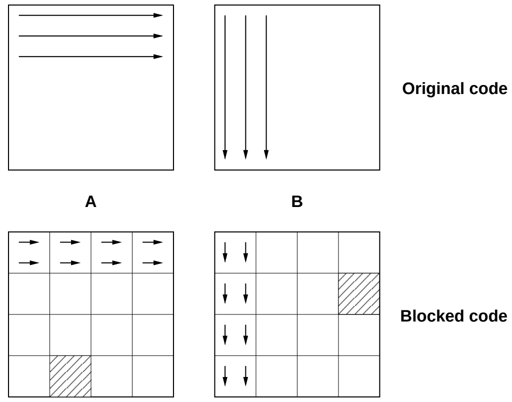
57: Blocking and large strides

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
```

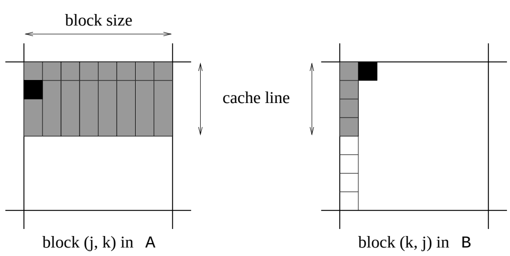
Bad locality for **A** good for **B**.
Blocking is good for data re-use, and when we have large strides.
Partition **A** and **B** in square sub-matrices each having the same order, the block size.
Treat pairs of blocks, one in **A** and one in **B** such that we can use the data which has been fetched to the L1 data cache.

58: Blocking and large strides



59: Blocking and large strides

Looking at two (shaded) 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).

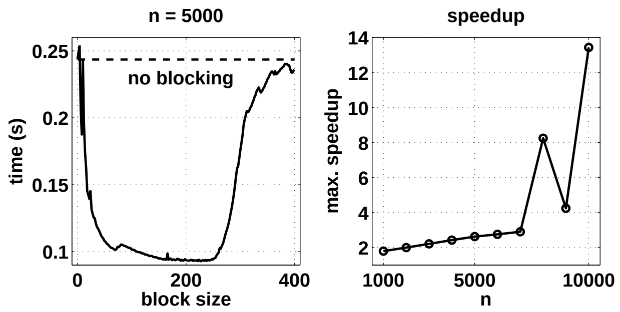
60: Blocking and large strides

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)

    ! sum one block
    do row = first_row, last_row
      do col = first_col, last_col
        s = s + A(row, col) * B(col, row)
      end do
    end do

  end do
end do
```

61: Blocking and large strides

Left plot, $n = 5000$, different block sizes using `ifort -O3` on an Intel Core Duo. Right plot, speedup for $n = 10^3, 2 \cdot 10^3, \dots, 10^4$ with optimal block size.



62: Blocking and large strides

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.

Intel's VTune Amplifier and AMD's CodeAnalyst are other tools for performance analysis. Wikipedia has a list of such tools.

63: Two important libraries

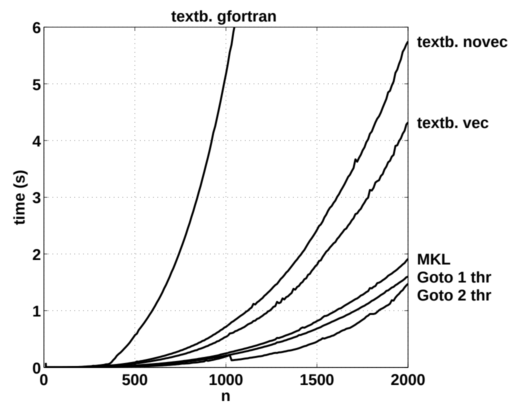
BLAS (the Basic Linear Algebra Subprograms) are the standard routines for simple matrix computations.
(**`s`** single, **`d`** double, **`c`** complex, **`z`** double complex).

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

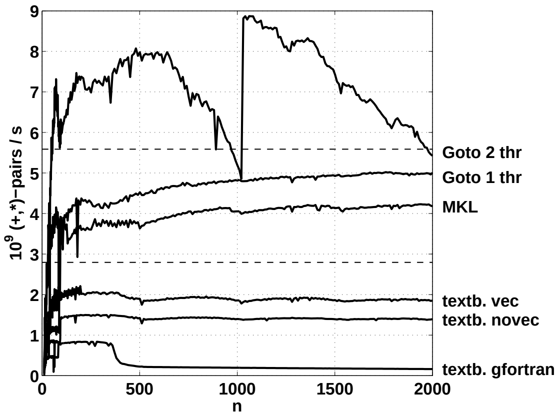
64: Two important libraries

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.
Other fast BLAS: AMD's ACML and OpenBLAS (based on Goto).

The next 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

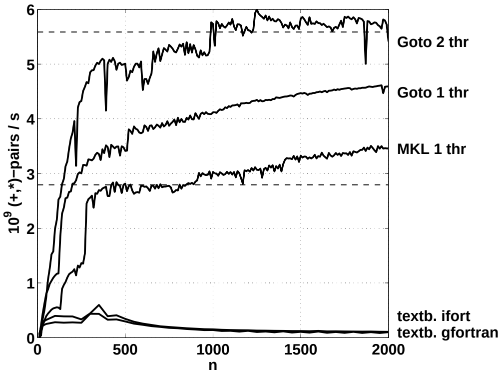
No support for large sparse problems, but there are routines for banded matrices of different kinds.
LAPACK is built on top of BLAS (BLAS3 where possible).
When using LAPACK, it is important to have optimized BLAS.

In this example we compute the Cholesky decomposition of a symmetric and positive definite matrix $\boldsymbol{A}$, so $\boldsymbol{A} = \boldsymbol{C}\boldsymbol{C}^T$, where $\boldsymbol{C}$ is undertriangular.
"textbook", in the figure on the next page, is one common way, often presented in textbooks, for computing $\boldsymbol{C}$.
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**.
Using LAPACK's **dpotrf** with Goto-BLAS on one and two threads.
Using MKL's **dpotrf** on one thread.

69: Two important libraries



Do not use simplistic algorithms from textbooks!

70: Inlining

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.

71: Indirect addressing, pointers

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

```
do k = 1, n
  j    = ix(k)   ! a sparse daxpy
  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

72: If-statements

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

Original version

```
do k = 1, n
  if ( k == 1 ) then
    statements
  else
    statements
  end if
end do
```

Optimized version

```
take care of k = 1
do k = 2, n
  statements for
  k = 2 to n
end do
```

73: If-statements

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

Suppose **f** and **g** are (time consuming) logical functions.
if (f(k) .and. g(k)) then , least likely first
if (f(k) .or. g(k)) then , most likely first

Make sure that **g** does not have side-effects.

74: Closing notes

Two basic tuning principles:

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

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

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.