$<, a so called macro, is short for the Fortran file.
One can use variables in make, here **OBJS** and **FFLAGS**.

```
OBJS   = main.o sub.o
FFLAGS = -O3

run: $(OBJS)
        g95 -o run $(FFLAGS) $(OBJS)

.SUFFIXES: .f90
.f90.o:
        g95 -c $(FFLAGS) $<
```

OBJS (for objects) is a variable and the first line is an assignment to it. **$(OBJS)** is the value (i.e. **main.o sub.o**) of the variable **OBJS**. **FFLAGS** is a standard name for flags to the Fortran compiler. I have switched on optimization in this case. Note that we have changed the suffix rule as well.

Make knows about certain variables, like **FFLAGS**. Suppose we would like to use the **ifort**-compiler instead. When compiling the source files, make is using the compiler whose name is stored in the variable **FC** (or possible **F90** or **F90C**). We write:

```
OBJS   = main.o sub.o
FC     = ifort
FFLAGS = -O3

run: $(OBJS)
        $(FC) -o run $(FFLAGS) $(OBJS)

.SUFFIXES: .f90
.f90.o:
        $(FC) -c $(FFLAGS) $<
```

It is usually important to use the same compiler for compiling and linking (or we may get the wrong libraries). It may also be important to use the same Fortran flags.

Sometimes we wish the recompile all files (we may have changed **$(FFLAGS)** for example). It is common to have the target **clean**. When having several targets we can specify the one that should be made:

```
OBJS   = main.o sub.o
FC     = g95
FFLAGS = -O3

run: $(OBJS)
     $(FC) -o run  $(FFLAGS) $(OBJS)

# Remove objects and executable
clean:
     rm -f $(OBJS) run

.SUFFIXES: .f90
.f90.o:
        $(FC) -c $(FFLAGS) $<
```

Without **-f**, **rm** will complain if some files are missing.

We type:

```
% make clean
rm -f main.o sub.o run
```


Suppose we like to use a library containing compiled routines. The new makefile may look like:

```
OBJS   = main.o sub.o
FC     = g95
FFLAGS = -O3
LIBS   = -lmy_library

run: $(OBJS)
     $(FC) -o run  $(FFLAGS) $(OBJS) $(LIBS)

.SUFFIXES: .f90
.f90.o:
        $(FC) -c $(FFLAGS) $<
```

If you are using standard functions in C sin, exp etc. you must use the math-library:

```
cc ... -lm
```

The equivalent makefile for C-programs looks like:

```
OBJS   = main.o sub.o
CC     = cc
CFLAGS = -O3
LIBS   = -lmy_library -lm

run: $(OBJS)
     $(CC) -o run  $(CFLAGS) $(OBJS) $(LIBS)

clean:
     rm -f $(OBJS) run
```


For the assignments it is easiest to have one directory and one makefile for each. It is also possible to have all files in one directory and make one big makefile.

```
OBJS1  = main1.o sub1.o
OBJS2  = main2.o sub2.o
CC     = cc
CFLAGS = -O3
LIBS1  = -lm
LIBS2  = -lmy_library

all: prog1 prog2

prog1: $(OBJS1)
     $(CC) -o $@  $(CFLAGS) $(OBJS1) $(LIBS1)

prog2: $(OBJS2)
     $(CC) -o $@  $(CFLAGS) $(OBJS2) $(LIBS2)

clean:
     rm -f $(OBJS1) $(OBJS2) prog1 prog2
```

When one is working with (and distributing) large projects it is common to use make in a recursive fashion. The source code is distributed in several directories. A makefile on the top-level takes care of descending into each sub-directory and invoking make on a local makefile in each directory.

There is much more to say about make. See e.g. the O'Reilly-book, Robert Mecklenburg, Managing Projects with GNU Make, 3rd ed, 2004.

Computer Architecture

Why this lecture?

Some knowledge about computer architecture is necessary:

- to understand the behaviour of programs
- in order to pick the most efficient algorithm
- to be able to write efficient programs
- to know what computer to run on
  (what type of architecture is your code best suited for)
- to read (some) articles in numerical analysis
- when looking for the next computer to buy
  (and to understand those PC-ads., caches, GHz, RISC...)

The change of computer architecture has made it necessary to re-design software, e.g Linpack ⇒ Lapack.

A very simple (and traditional) model of a computer:

The CPU contains the ALU, arithmetic and logic unit and the control unit. The ALU performs operations such as +, -, *, / of integers and Boolean operations.
The control unit is responsible for fetching, decoding and executing instructions.
The memory stores instructions and data. Instructions are fetched to the CPU and data is moved between memory and CPU using buses.
I/O-devices are disks, keyboards etc.

The CPU contains several registers, such as:

- PC, program counter, contains the address of the next instruction to be executed
- IR, instruction register, the executing instruction
- address registers
- data registers

The memory bus usually consist of one address bus and one data bus. The data bus may be 64 bits wide and the address bus may be $\geq$ 32 bits wide. With the introduction of 64-bit computers, buses tend to become increasingly wider. The Itanium 2 uses 128 bits for data and 44 bits for addresses.

Operations in the computer are synchronized by a clock.
A modern CPU may run at a few GHz (clock frequency). The buses are usually a few (4-5) times slower.

A few words on 64-bit systems

Why 64 bit?

- A larger address range, can address more memory.
  With 32 bits we can (directly) address 4 Gbyte, which is rather limited for some applications.
- Wider busses, increased memory bandwidth.
- 64-bit integers.

Be careful when mixing binaries (object libraries) with your own code. Are the integers 4 or 8 bytes?

```
% cat kind.c
#include <stdio.h>

int main()
{
  printf("sizeof(short int) = %d\n", sizeof(short int));
  printf("sizeof(int)       = %d\n", sizeof(int));
  printf("sizeof(long int)  = %d\n", sizeof(long int));

  return 0;
}
```

```
% gcc kind.c              On the student system
% a.out
sizeof(short int) = 2
sizeof(int)       = 4
sizeof(long int)  = 4

% a.out                  On another Opteron-system
sizeof(short int) = 2
sizeof(int)       = 4
sizeof(long int)  = 8    4 if gcc -m32
```


CISC (Complex Instruction Set Computers) before ≈ 1985.
Each instruction can perform several low-level operations, such as a load from memory, an arithmetic operation, and a memory store, all in a single instruction.

Why CISC?
For a more detailed history, see the literature.

- Advanced instructions simplified programming
  (writing compilers, assembly language programming).
  Software was expensive.
- Memory was limited and slow so short programs were good.
  (Complex instructions ⇒ compact program.)

Some drawbacks:

- complicated construction could imply a lower clock frequency
- instruction pipelines hard to implement
- long design cycles
- many design errors
- only a small part of the instructions was used
  According to Sun: Sun's C-compiler uses about 30% of the available 68020-instructions (Sun3 architecture). Studies show that approximately 80% of the computations for a typical program requires only 20% of a processor's instruction set.

When memory became cheaper and faster, the decode and execution on the instructions became limiting.

Studies showed that it was possible to improve performance with a simple instruction set and where instructions would execute in one cycle.

RISC - Reduced Instruction Set Computer

- IBM 801, 1979 (publ. 1982)
- 1980, David Patterson, Berkeley, RISC-I, RISC-II
- 1981, John Hennessy, Stanford, MIPS
- ≈ 1986, commercial processors

A processor whose design is based on the rapid execution of a sequence of simple instructions rather than on the provision of a large variety of complex instructions.

Some RISC-characteristics:

- load/store architecture; `C = A + B`

```
LOAD  R1,A
LOAD  R2,B
ADD   R1,R2,R3
STORE C,R3
```

- fixed-format instructions (the op-code is always in the same bit positions in each instruction which is always one word long)
- a (large) homogeneous register set, allowing any register to be used in any context and simplifying compiler design
- simple addressing modes with more complex modes replaced by sequences of simple arithmetic instructions
- one instruction/cycle
- hardwired instructions and not microcode
- efficient pipelining
- simple FPUs; only +, -, *, / and $\sqrt{}$.
  sin, exp etc. are done in software.

Advantages: Simple design, easier to debug, cheaper to produce, shorter design cycles, faster execution, easier to write optimizing compilers (easier to optimize many simple instructions than a few complicated with dependencies between each other).

CISC - short programs using complex instructions.
RISC - longer programs using simple instructions.

So why is RISC faster?

The simplicity and uniformity of the instructions make it possible to use pipelining, a higher clock frequency and to write optimizing compilers.

Will now look at some techniques used in all RISC-computers:

- instruction pipelining
  work on the fetching, execution etc. of instructions in parallel
- cache memories
  small and fast memories between the main memory and the CPU registers
- superscalar execution
  parallel execution of instructions (e.g. two integer operations, *, + floating point)

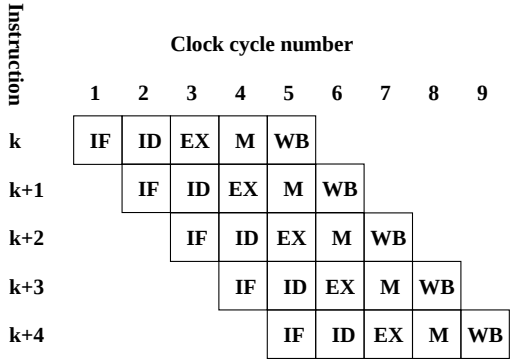
The most widely-used type of microprocessor, the x86 (Intel), is CISC rather than RISC, although the internal design of newer x86 family members is said to be RISC-like. All modern CPUs share some RISC characteristics, although the details may differ substantially.

Pipelining - performing a task in several steps, stages

Analogy: building cars using an assembly line in a factory.

Suppose there are five stages (can be more), .e.g

IF	Fetch the next instruction from memory.
ID	Instruction decode.
EX	Execute.
M, WM	Memory access, write to registers.



So one instruction completed per cycle once the pipeline is filled.

Not so simple in real life: different kind of hazards, that prevent the next instruction from executing during its designated clock cycle. Can make it necessary to stall the pipeline (wait cycles).

- Structural hazards arise from resource conflicts, e.g.
  - two instructions need to access the system bus (fetch data, fetch instruction),
  - not fully pipelined functional units (division usually takes 10-20 cycles, for example).
- Data hazards arise when an instruction depends on the results of a previous instruction (will look at some cases in later lectures) e.g.

```
a = b + c
d = a + e   d depends on a
```

  The second addition must not start until `a` is available.
- Control hazards arise from the pipelining of branches (if-statements).

An example of a control hazard:

```
if ( a > b - c * d ) then
  do something
else
  do something else
end if
```

Must wait for the evaluation of the logical expression.

If-statements in loops may cause poor performance.

Several techniques to minimize hazards (look in the literature for details) instead of just stalling. Some examples:

Structural hazard:

Add hardware. If the memory has only one port `LOAD adr,R1` will stall the pipeline (the fetch of data will conflict with a later instruction fetch). Add a memory port (separate data and instruction caches).

Data hazards:

- Forwarding: `b + c` available after EX, special hardware "forwards" the result to the `a + e` computation (without involving the CPU-registers).
- Instruction scheduling. The compiler can try and rearrange the order of instruction to minimize stalls.
  Try to change the order between instructions using the wait-time to do something useful.

```
a = b + c
d = a + e

load b
load c
add  b + c  has to wait for load c to complete

load b
load c
load e      give the load c time to complete
add  b + c  in parallel with load e
```


Control hazards: (many tricks)

- Add hardware; can compute the address of the branch target earlier and can decide whether the branch should be taken or not.
- Branch prediction; try to predict, using "statistics", the way a branch will go. Compile-time/run-time. Can work very well. The branch att the end of a `for`-loops is taken all the times but the last.
- Branch delay slot: let the compiler rearrange instructions so that something useful can be done while it is decided whether we should branch or not.

```
loop: instr                 loop: instr
      instr                       instr
       ...            -->         ...
      FADD R1, R2, R3             if j < n goto loop
      if j < n goto loop          FADD R1, R2, R3
```

Perform the FADD-instruction while waiting for the if to complete.

A more general construction, speculative execution: Assume the branch not taken and continue executing (no stall). If the branch is taken, must be able do undo.

Superscalar CPUs

Fetch, decode and execute more than one instruction in parallel. More than one finished instruction per clock cycle. There may, e.g. be two integer ALUs, one unit for floating point addition and subtraction one for floating point multiplication. The units for +, - and * are usually piplined (they need several clock cycles to execute).
There are also units for floating point division and square root; these units are not (usually) pipelined.

```
MULT  xxxxxxxx
MULT   xxxxxxxx
MULT    xxxxxxxx
```

Compare division; each `xxxxxxxxxx` is 22 cycles (on Sun):

```
DIV  xxxxxxxxxx
DIV            xxxxxxxxxx
DIV                      xxxxxxxxxx
```

How can the CPU keep the different units busy?
The CPU can have circuits for arranging the instructions in suitable order, dynamic scheduling (out-of-order-execution).

To reduce the amount of hardware in the CPU we can let the compiler decide a suitable order. Groups of instructions (that can be executed in parallel) are put together in packages. The CPU fetches a whole package and not individual instructions. VLIW-architecture, Very Long Instruction Word.

The Intel & HP Itanium CPU uses VLIW (plus RISC ideas). Read the free chapter from: W. Triebel, Itanium Architecture for Software Developers. See the first chapter in: IA-32 Intel Architecture Optimization Reference Manual for details aboute the Pentium 4. Read Appendix A in the Software Optimization Guide for AMD64 Processors. See the web-Diary for links.

More on parallel on floating point operations.

flop = floating point operation.
flops = plural of flop or flop / second.
In numerical analysis a flop may be an addition-multiplication pair. Not unreasonable since (+, *) often come in pairs, e.g. in an inner product.

Top floating point speed =
# of processors × flop / s =
# of processors × # flop / clock cycle × clock frequency

On the student machines (AMD64) the theoretical top performance is $2 \cdot 10^9$ additions and multiplications per second. Note that the student machines are running PowerNow!, AMD's technology for dynamically changing the clock frequency of the CPU (to save energy, to reduce the rpm of the cooling fan etc). There are three possible clock frequencies: 2GHz, 1.8GHz and 1GHz. More details are available from the homepage.

An Intel Pentium4 can finish an addition every clock cycle and a multiplication every other.

Some Intel, AMD and Motorola CPUs have another unit (a kind of vector unit) that can work on short vectors of numbers. These technologies have names like: SSE3, 3DNow! and AltiVec. More details later in the course.

To use the the vector unit you need a compiler that can vectorize. The vector unit may not be IEEE 754 compliant (not correctly rounded). So results may differ between the vectorized and unvectorized versions of the same code.
See `www.spec.org` for benchmarks with real applications.

Why do we often only get a small percentage of these speeds?
Is it possible to reach the top speed (and how)?

Example on a 167 MHz Sun; top speed 334 Mflops:

```
Instr.  fl. p. registers

fmuld   %f4,%f2,%f6             faddd   %f4,%f2,%f6
faddd   %f8,%f10,%f12           faddd   %f8,%f10,%f12
fmuld   %f26,%f28,%f30          faddd   %f4,%f2,%f6
faddd   %f14,%f16,%f18          faddd   %f8,%f10,%f12
fmuld   %f4,%f2,%f6             faddd   %f4,%f2,%f6
faddd   %f8,%f10,%f12           faddd   %f8,%f10,%f12
                        ...

331.6 Mflops                    166.1 Mflops
```

```
fdivd   %f4,%f2,%f6             fdivd   %f4,%f2,%f6
faddd   %f8,%f10,%f12           fdivd   %f8,%f10,%f12
fdivd   %f4,%f2,%f6             fdivd   %f4,%f2,%f6
faddd   %f8,%f10,%f12           fdivd   %f8,%f10,%f12
                        ...

15.1 Mflops                     7.5 Mflops
```

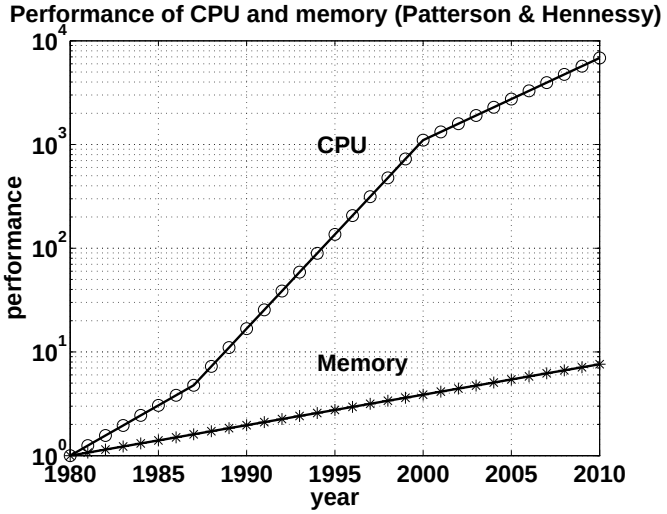
Addition and multiplication are pipelined. Division is not pipelined (so divides do not overlap) and takes 22 cycles for double precision.

$$\frac{167 \cdot 10^6 s^{-1}}{22} \approx 7.6 \cdot 10^6/s$$

So, the answer is sometimes

- provided we have a suitable instruction mix and that
- we do not access memory too often

Memory is the problem - caches

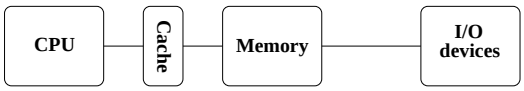


CPU: increase 1.35 improvement/year until 1986, and a 1.55 improvement/year thereafter.

DRAM (dynamic random access memory), slow and cheap, 1.07 improvement/year.

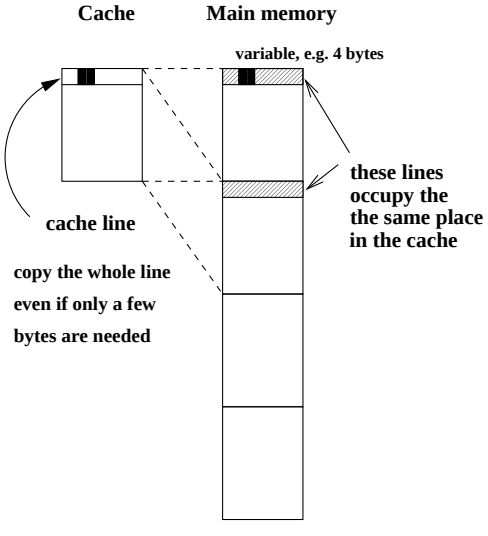
Use SRAM (static random access memory) fast & expensive for cache.

Direct mapped cache

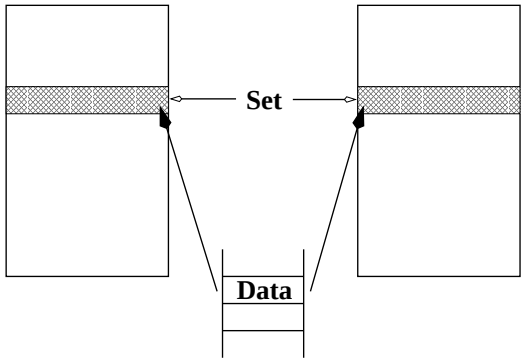


The cache is a small and fast memory used for storing both instructions and data.

This is the simplest form of cache-construction.

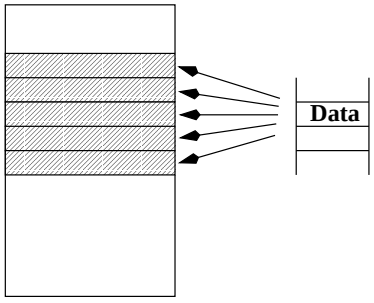


There are more general cache constructions.
This is a two-way set associative cache:



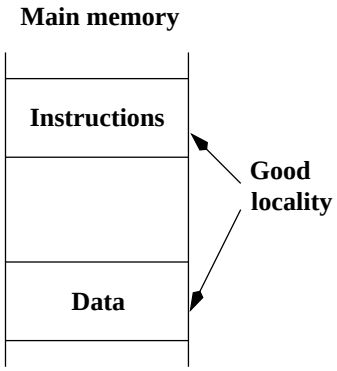
A direct mapped cache is one-way set associative.

In a fully associative cache data can be placed anywhere.



To use a cache efficiently locality is important.

- instructions: small loops, for example
- data: use part of a matrix (blocking)

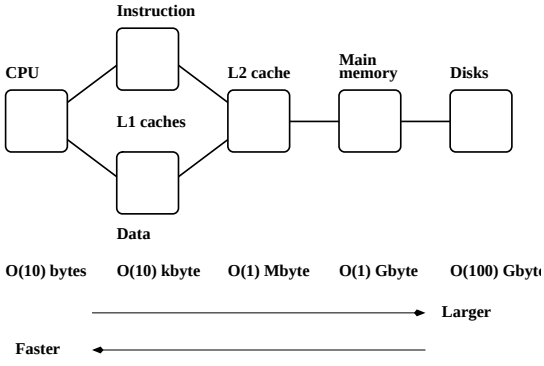


Not necessarily good locality together.

Make separate caches for data and instructions.

Can read instructions and data in parallel.

L1 and L2 caches



Memory hierarchy.

Newer machines even have an L3 cache.

The AMD64 (student machines)

(Some) Intel and AMD-CPUs have an instruction, **cpuid**, that gives details about the CPU, such as model, SSE-features, L1- and L2-cache properties. These values can be hard to find just reading manuals. Some parameters are available in **/proc/cpuinfo** .

Unfortunately one has to code in assembler to access this information. **gcc** supports inlining of assembly code using the **asm**-function. **asm** makes it possible to "connect" registers with C-variables. It may look like this (note that I have broken the string):

```
...
 unsigned long before, after;

 /* Does the CPU support the cpuid-instruction?*/
 asm("pushfl; popl %%eax; movl %%eax, %0;
      xorl $0x40000, %%eax; pushl %%eax; popfl; pushfl
      popl %%eax; movl %%eax, %1; pushl %0; popfl "
      : "=r" (before), "=r" (after)  /* output */
      :                              /* input  */
      : "eax"             /* changed registers */
    );
 if ( before != after ) {
   ...                    /* Support. Test more */
 }
...
```

One can call **cpuid** with a set of different "arguments", and **cpuid** then returns bit patterns in four registers. Reading the AMD-manual "CPUID Specification" one can interpret the bits. Bits 31-24 in the ECX-register contain the size of L1-data cache in kbyte, for example.

This is some of the facts I found out about the caches (I read some manuals as well):

The L1 data cache is 64 kbyte, 2-way associative and has a cache line size (length) of 64 bytes. Cache-line replacement is based on a least-recently-used (LRU) replacement algorithm.

The L2 cache is "on-die" (on-chip), 512 kbyte and 16-way associative. The cache line size is 64 bytes.

A note on reading assembly output

In the lecture and during the labs I said it was sometimes useful to look at the assembler code produced by the compiler.
Here comes a simple example. Let us look at the the following function.

```
double dot(double x[], double y[], int n)
{
  double s;
  int k;

  s = 0.0;
  for (k = 0; k < n; k++)
    s += x[k] * y[k];

  return s;
}
```

First some typical RISC-code from a Sun ULTRA-Sparc CPU. I used **gcc** and compiled the code by:

```
gcc -S -O3 dot.c
```

-S produces the assembler output on **dot.s**.
Here is the loop (code for passing parameters, setting up for the loop, and returning the result is not included).

```
.LL5:                          My translation
  ldd     [%o0+%g1], %f8     %f8  = x[k]
  ldd     [%o1+%g1], %f10    %f10 = y[k]
  add     %g2, 1, %g2        k    = k + 1
  fmuld   %f8, %f10, %f8     %f8  = %f8* %f10
  cmp     %o2, %g2           k == n? Set status reg.
  faddd   %f0, %f8, %f0      %f0  = %f0 + %f8
  bne     .LL5               if not equal, go to .LL5
  add     %g1, 8, %g1        increase offset
```


Some comments.

%f8 and **%f10** are registers in the FPU. When entering the function, the addresses of the first elements in the arrays are stored in registers **%o0** and **%o1**. The addresses of **x[k]** and **y[k]** are given by **%o0 + 8k** and **%o1 + 8k**. The reason for the factor eight is that the memory is byte addressable (each byte has an address). The offset, **8k**, is stored in register **%g1**.

The offset, **8k**, is updated in the last **add**. It looks a bit strange that the **add** comes after the branch, **bne**. The add-instruction is, however, placed in the branch delay slot of the branch-instruction, so it is executed in parallel with the branch.

add is an integer add. **faddd** is a "floating point add double". It updates **%f0**, which stores the sum. **%f0** is set to zero before the loop. **cmp** compares **k** with **n** (the last index) by subtracting the numbers. The result of the compare updates the Z-bit (Z for zero) in the integer condition code register. The branch instruction looks at the Z-bit to see if the branch should be taken or not.

We can make an interesting comparison with code produced on the AMD64. The AMD (Intel-like) has both CISC- and RISC-characteristics. It has fewer registers than the Sparc and it does not use load/store in the same way. The x87 (the FPU) uses a stack with eight registers. In the code below, **eax** etc. are names of 32-bit CPU-registers. (in the assembly language a **%** is added).

```
.L5:
  fldl    (%ebx,%eax,8)
  fmull   (%ecx,%eax,8)
  faddp   %st, %st(1)
  incl    %eax
  cmpl    %eax, %edx
  jne     .L5
```


When the loop is entered **%ebx** and **%ecx** contain the addresses of the first elements of the arrays. Zero has been pushed on the stack as well (corresponds to **s = 0.0**).

fldl (%ebx,%eax,8) loads a 64 bit floating point number. The address is given by **%ebx + %eax*8**. The number is pushed on the top of the stack, given by the stackpointer **%st**.

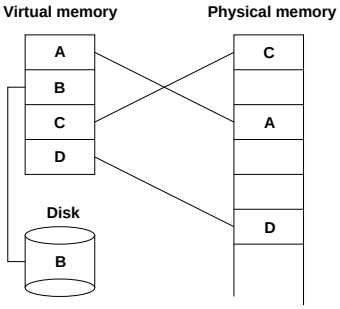
Unlike the Sparc, the AMD can multiply with an operand in memory (the number does not have to be fetched first). So the **fmull** multiplies the top-element on the stack with the number at address **%ecx + %eax*8** and replaces the top-element with the product.

faddp %st, %st(1) adds the top-elements on the stack (the product and the sum, **s**), pops the stack, the **p** in **faddp**, and replaces the top with the new value of **s**.

incl increases **k** (stored in **%eax**) and **cmpl** compares it to **n**. **jne** stands for jump if not equal.

Virtual memory

Use disk to "simulate" a larger memory. The virtual address space is divided into pages e.g. 4 kbytes. A virtual address is translated to the corresponding physical address by hardware and software; address translation.



A page is copied from disk to memory when an attempt is made to access it and it is not already present (page fault). When the main memory is full, pages must be stored on disk (e.g. the least recently used page since the previous page fault). Paging.
(Swapping; moving entire processes between disk and memory.)

Some advantages of virtual memory:

- simplifies relocation (loading programs to memory), independece of physical addresses;
  several programs may reside in memory
- security, can check access to protected pages, e.g. read-only data; can protect data belonging to other processes
- allows large programs to run on little memory; only used sections of programs need be present in memory; simplifies programming (e.g. large data structures where only a part is used)

Virtual memory requires locality (re-use of pages) to work well, or thrashing may occur.

A few words on address translation

The following lines sketch one common address translating technique.

A virtual address is made up by two parts, the virtual page number and the page offset (the address from the top of the page).
The page number is an index into a page table:

```
physical page address =
    page_table(virtual page number)
```

The page table is stored in main memory (and is sometimes paged). To speed up the translation (accessing main memory takes time) we store part of the table in a cache, a translation lookaside buffer, TLB which resides in the CPU ($\mathcal{O}(10) - \mathcal{O}(1000)$ entries).

Once again we see that locality is important. If we can keep the references to a few pages, the physical addresses can found in the TLB and we avoid a reference to main memory. If the address is not available in the TLB we get a TLB miss (which is fairly costly, taking tens of clock cycles).

Reading the actual data may require a reference to main memory, but we hope the data resides in the L1 cache.

Second best is the L2 cache, but we may have to make an access to main memory, or worse, we get a page fault and have to make a disk access (taking millions of clock cycles).

Code Optimization

- How does one get good performance from a computer system?
- Focus on systems with one CPU and floating point performance.
- To get maximum performance from a parallel code it is important to tune the code running on each CPU.
- Not much about applications from graphics, audio or video. One example of SSE2 (Streaming SIMD Extensions 2).
- General advice and not specific systems.
- Fortran, some C (hardly any C++).
  Some Java in the Springer chapter.

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 $\approx$ 2 Fortran, C++ $\approx$ 4 Fortran (for floating point).
  Java? Can be slow and use large amounts of memory.
  See the article (Springer chapter) 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 ...

More about unix-tools:

- shell scripts (**sh**, **csh**, **tcsh**, **ksh**, **bash**)
  (**for**, **if**, **|** pipes and lots more)
- **awk** (developed by Alfred Aho, Peter Weinberger, and Brian Kernighan in 1978)
- **sed** (stream editor)
- **grep** (after the qed/ed editor subcommand "g/re/p", where re stands for a regular expression, to Globally search for the Regular Expression and Print)
- **tr** (translate characters)
- **perl** (Larry Wall in 1987; contains the above)
- etc.

Some very simple examples:

Counting the number of lines in a file(s):

```
% wc file     or    wc -l file
% wc files    or    wc -l files
```

Finding a file containing a certain string

```
% grep string files                e.g.
% grep 'program matrix'*.f90       or
% grep -i 'program matrix'*.f90    etc.
```

The **grep**-command takes many flags.

Example: interchange the two blank-separated columns of numbers in a file:

```
% awk '{print $2, $1}' file
```

Example: sum the second columns of a set of datatfiles. Each row contains: **number number text text**
The files are named **data.01**, **data.02**, ...

foreach? is the prompt.

```
% foreach f ( data.[0-9][0-9] )
foreach? echo -n $f': '
foreach? awk '{s += $2} END {print s}' $f
foreach? end
data.01: 30
data.02: 60
data.03: 77
data.20: 84
```

Another possibility is:

```
awk '{s += $2} END {print FILENAME ": " s}' $f
```


Just the other day (two years ago) ...

You have ≈ 600 files each consisting of ≈ 24000 lines (a total of $\approx 14 \cdot 10^6$ lines) essentially built up by:

```
<DOC>
<TEXT>
Many lines of text (containing no DOC or TEXT)
</TEXT>
</DOC>
<DOC>
<TEXT>
Many lines of text
</TEXT>
</DOC>
etc.
```

There is a mismatch between the number of **DOC** and **TEXT**.
Find it!

We can localize the file this way:

```
% foreach f ( * )
foreach? if ( `grep -c "<DOC>" $f` !=   \
              `grep -c "<TEXT>" $f` ) echo $f
foreach? end
```

Not so efficient; we are reading each file twice.
Takes ≈ 3.5 minutes.
We used binary search to find the place in the file.

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).

  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 of the Sun Fortran90/95-compiler. The names are not standardized, but it is common that **-c** means "compile only, do not link".
To produce debug information **-g** is used.

-O[n] usually denotes optimization on level **n**. There may be an option, like **-fast**, that gives a combination of suitable optimization options. In Sun's case **-fast** is equivalent to:

- **-xtarget=native** optimize for host system (sets cache sizes for example).
- **-O5** highest optimization level.
- **-libmil** inline certain math library routines.
- **-fsimple=2** aggressive floating-point optimizations. May cause many programs to produce different numeric results due to changes in rounding.
- **-dalign** align data to allow generation of faster double word load/store instructions.
- **-xlibmopt** link the optimized math library. May produce slightly different results; if so, they usually differ in the last bit.
- **-depend** optimize DO loops better.
- **-fns** for possibly faster (but nonstandard) handling of floating-point arithmetic exceptions and gradual underflow.
- **-ftrap=common** option to set trapping on common floating-point exceptions (this is the default for f95).
- **-pad=local** option to improve use of cache (alignment).
- **-xvector=yes** enable use of the vectorized math library.
- **-xprefetch=yes** enable generation of prefetch instructions on platforms that support it.

```
f95 | f90 [ -a ] [ -aligncommon[=a] ] [ -ansi ]
 [ -autopar ] [ -Bx ] [ -C ] [ -c ] [ -cg89 ] [ -cg92 ]
 [ -copyargs ] [ -Dnm[=def] ] [ -dalign ] [ -db ]
 [ -dbl_align_all[=yes|no] ] [ -depend ] [ -dryrun ]
 [ -d[y|n] ] [ -e ] [ -erroff=taglist ] [ -errtags[=yes
 [ -explicitpar ] [ -ext_names=e ] [ -F ] [ -f ]
 [ -fast ] [ -fixed ] [ -flags ] [ -fnonstd ]
 [ -fns=yes|no ] [ -fpover=yes|no ] [ -fpp ] [ -free ]
 [ -fround=r ] [ -fsimple[=n] ] [ -ftrap=t ] [ -G ]
 [ -g ] [ -hnm ] [ -help ] [ -Idir ] [ -inline=rl ]
 [ -Kpic ] [ -KPIC ] [ -Ldir ] [ -libmil ] [ -loopinfo
 [ -M dir ] [ -mp=x ] [ -mt ] [ -native ] [ -noautopar
 [ -nodepend ] [ -noexplicitpar ] [ -nolib ] [ -nolibmi
 [ -noqueue ] [ -noreduction ] [ -norunpath ] [ -O[n] ]
 [ -o nm ] [ -onetrip ] [ -openmp ] [ -p ] [ -pad[=p] ]
 [ -parallel] [ -pg ] [ -pic ] [ -PIC ] [ -Qoption pr l
 [ -qp ] [ -R list ] [ -r8const ] [ -reduction ] [ -S ]
 [ -s ] [ -sb ] [ -sbfast ] [ -silent ] [ -stackvar ]
 [ -stop_status=yes|no ] [ -temp=dir ] [ -time ] [ -U ]
 [ -Uname ] [ -u ] [ -unroll=n ] [ -V ] [ -v ] [ -vpara
 [ -w ] [ -xa ] [ -xarch=a ] [ -xautopar ] [ -xcache=c
 [ -xcg89 ] [ -xcg92 ] [ -xchip=c ] [ -xcode=v ]
 [ -xcommonchk[=no|yes] ] [ -xcrossfile=n ] [ -xdepend
 [ -xexplicitpar ] [ -xF ] [ -xhasc[=yes|no] ] [ -xhelp
 [ -xia[=i] ] [ -xildoff ] [ -xildon ] [ -xinline=rl ]
 [ -xinterval=i ] [ -xipo[=0|1] ] [ -xlang=language[,la
 [ -xlibmil ] [ -xlibmopt ] [ -xlicinfo ]
 [ -xlic_lib=sunperf ] [ -Xlist ] [ -xloopinfo ] [ -xma
 [ -xmemalign[=ab] ] [ -xnolib ] [ -xnolibmil ] [ -xnol
 [ -xO[n] ] [ -xopenmp ] [ -xpad ] [ -xparallel ] [ -xp
 [ -xpp=p ] [ -xprefetch=a[,a]] [ -xprofile=p ] [ -xrec
 [ -xreduction ] [ -xregs=r ] [ -xs ] [ -xsafe=mem ]
 [ -xsb ] [ -xsbfast ] [ -xspace ] [ -xtarget=t ] [ -xt
 [ -xtypemap=spec ] [ -xunroll=n ] [ -xvector=yes|no ]
   source file(s) ...  [ -lx ]
```


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:

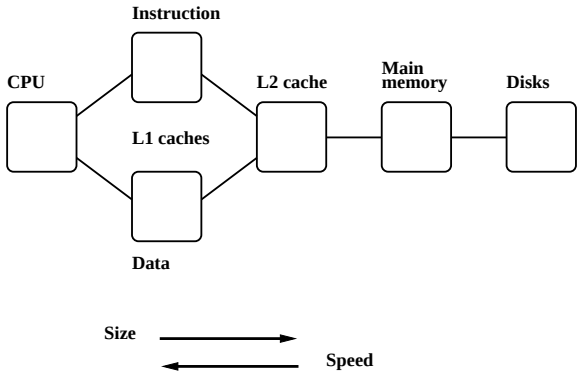
- The code becomes less readable and it is easy to introduce bugs.
- Detailed knowledge about the system, such as cache configuration, is often necessary.
- What is optimal for one system need not be optimal for another; faster on one machine may actually be slower on another.
  This leads to problems 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

- memory hierarchy
- parallel capabilities



Superscalar: start several instructions per cycle.
Pipelining: work on an instruction in parallel.

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.

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!

The compilers on Lucidor (Itanium 2 at PDC) have improved since 2005, so `restrict` or `-fno-alias` are not needed (for `add`).
Restricted pointers give a slight improvement on Lenngren (Intel Xeon) from 2s to 1.3s (10^5 calls of `add` with `n` = 10000).

So this is not a static situation. Compilers and hardware improve every year, so 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, but still only a few lines of code, i.e. far from a real code.

Here is a polynomial evaluation using Horner's method:

```
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
```

On Lucidor the Fortran code takes 1.6s and the C-code 6.4s.
(`n = 1000` and calling the routine 10^6 times).
I compiled using `icc -O3 ...`. Lowering the optimization can help (it did in 2007), but not this year.
On Lenngren the times were 2.9s (Fortran) 4.8s (C).

On Ferlin (672 compute nodes, each with two quad-core Intel 2.66GHz CPUs with a 1333 MHz front-side bus, and 8Gbyte memory), the times were 1.7s (Fortran) and and 3s (C).
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)));
  }
...
```

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 (using Intel's compilers), `n = 1000` and calling the routine 10^5 times:

System, compiler	`ifort -O3`	`icpc -O2`	`icpc -O3`
Lucidor	0.62	32.5	11.0
Lenngren	1.9	4.3	4.7
Ferlin	1.0	2.6	2.6
Opteron	2.1	28.5	28.5

The Portland group compilers, on Lenngren, took 2.2s (Fortran) and 14.1s (C++).
Here are some tests of the GNU-compilers on the same code.

System, compiler	`g95 -O3`	`gfortran -O3`	`g++ -O3`
Lucidor	NA	NA	26.0
Lenngren	NA	NA	1.9
Ferlin	NA	1.1	2.6
Opteron	2.6	1.9	3.5

These times may not be representative, my experience is that the gcc-family of compilers produce slower code than Intel's.

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

Basic arithmetic and elementary functions

- Common that the FPU can perform `+` and `*` in parallel.
- `a+b*c` can often be performed with one round-off, multiply-add MADD or FMA.
- Several FMAs in parallel on some machines.
- `+` and `*` usually pipelined, so one sum and a product per clock cycle in the best of cases
  (not two sums or two products).
- `/` not usually pipelined and may require around twenty clock cycles.

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 one can sometimes 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.

```
program int_vs_float
  integer, parameter :: n = 10000
  integer            :: k
  real, dimension(n) :: arr
  real               :: s = 0.0

  arr = 1
  do k = 1, 100000
    s = s + product(arr)
  end do

  print*, s

end program int_vs_float
```

```
% time a.out
 100000.0
5.36u 0.05s 0:05.58 96.9%
```

Change to:

```
  integer, dimension(n) :: arr
  integer               :: s = 0
```

```
% time a.out
 100000
44.15u 0.02s 0:44.33 99.6%
```

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 =
    0.8840

>> v = 1e10 * ones(1000, 1);
>> tic; for k = 1:1000, s = sin(v); end; toc
elapsed_time =
    9.2969
```



```
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
```


"The Mathematical Acceleration SubSystem" MASS (IBM).

High performance versions of some intrinsic Fortran functions. Sacrifices a small amount of accuracy (last bit).

There are also vector versions of some of the functions. Must link with special libraries (scalar-MASS, vector-MASS).

```
program mass_test
  integer, parameter             :: n = 10000
  double precision, dimension(n) :: v, sinv
...
    v = ...
    call vsin(sinv, v, n)  ! vector-sin
...
end
```

Performance depends on the type of function, range of arguments and vector length (when using the vector library). Two examples (normalised times, n = 5000):

Function	default	scalar MASS	vector MASS
sin	4.9	3.5	1
exp	4.5	2.8	1

SSE2, Streaming SIMD Extensions 2

Some CPUs have built-in "vector computers".
The Pentium 4 SSE2 can do e.g. vector multiplies: `a = a .* b` (using Matlab notation) where `a` and `b` contain 4 single precision or 2 double precision numbers.

We need an optimizing compiler that produces code using the special vector instructions. For example:

```
% ifc -vec_report3 -O3 -tpp7 -xW    files...
(15) vector dependence: assumed FLOW dependence ...
     loop was not vectorized
(23) LOOP WAS VECTORIZED.

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

Called 100000 times. Times (in s) on a 2.8 GHz Intel Xeon:

single		double		
No SSE2	SSE2	No SSE2	SSE2	cos/sin
1.9	0.4	1.9	0.8	no
132.1	12.0	132.2	36.9	yes

Why so fast? `__libm_sse2_sin`(and `cos`) is used.

Disadvantage: the x87-FPU uses double extended precision, 64 bit mantissa. SSE2 uses 24 bits (single precision) or 53 bits (double precision). Does not support denormalized numbers.

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.

This test-program was run on a machine with only 64 Mbyte memory, $m * n^2$ is constant, so same number of additions

```
>> type test          % list the program
clear A B C           % remove the matrices
tic                   % start timer
for k = 1:m           % repeat m times
  A = ones(n);        % n x n-matrix of ones
  B = ones(n);        % all are 64-bit numbers
  C = A + B;
end
toc                   % stop timer

% Run three test cases
>> n = 500; m = 16; test  % 5.7 Mbyte for A, B and C
elapsed_time = 1.1287     % roughly ONE SECOND

>> n = 1000; m = 4; test  % 22.9 Mbyte
elapsed_time = 1.1234     % roughly the same as above

>> n = 2000; m = 1; test  % 91.6 Mbyte
elapsed_time = 187.9      % more than THREE MINUTES
```



```
% vmstat 1  (edited)
      page          cpu
   pi     po     us  sy  id
    0      0      0  0  100
    0      0      0  0  100
  352    128      0  0  100  <-- third test is run
  616    304      0  6   94
  608    384      0  2   98
  712    256      0  2   98   etc. for over 3 minutes

  pi = kilobytes paged in  / second
  po = kilobytes paged out / second
```

Input-output

Need to store $5 \cdot 10^6$ double precision numbers in a file.
A local disk was used for the tests. Intel's Fortran compiler on AMD64. Roughly the same times in C.

Test	Statement	time (s)	size (Mbyte)
1	`write(10, '(1pe23.16)') x(k)`	29.4	114.4
2	`write(10) x(k)`	19.5	76.3
3	`write(10) (vec(j), j = 1, 10000)`	0.1	38.2
4	`write(10) vec(1:10000)`	0.1	38.2
5	`write(10) vec`	0.1	38.2

File sizes:

$$1: \quad \underbrace{5 \cdot 10^6}_{\#\text{ of numbers}} \cdot \underbrace{(23+1)}_{\text{characters + newline}} \Big/ \underbrace{2^{20}}_{\text{Mbyte}} \approx 114.4$$

$$2: \quad \underbrace{5 \cdot 10^6}_{\#\text{ of numbers}} \cdot \underbrace{(8+4+4)}_{\text{number + delims}} \Big/ \underbrace{2^{20}}_{\text{Mbyte}} \approx 76.3$$

$$3-5: \quad \left[\underbrace{5 \cdot 10^6}_{\#\text{ of numbers}} \cdot \underbrace{8}_{\text{number}} + 500 \cdot \underbrace{(4+4)}_{\text{delims}} \right] \Big/ \underbrace{2^{20}}_{\text{Mbyte}} \approx 38.2$$

In 5 **vec** has 10000 elements and we write the array 500 times.

g95 and **gfortran** were slower in all cases but case 2, where **g95** took 6s.

Portability of binary files?

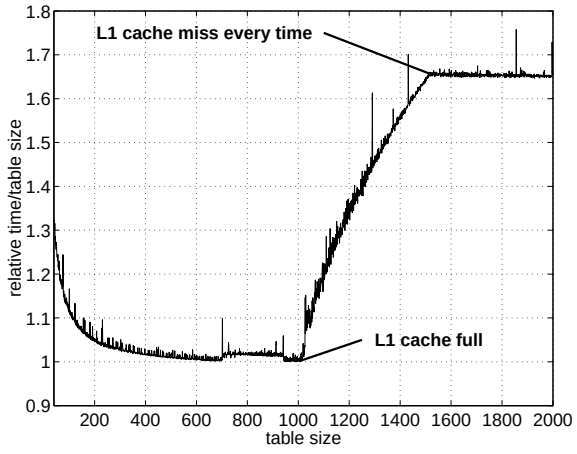
- Perhaps
- File structure may differ
- Byte order may differ
- Big-endian, most significant byte has the lowest address ("big-end-first").
- The Intel processors are little-endian ("little-end-first").

```
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
```


Memory locality and caches

```
  1  2                        # of columns
---------------  ...  -----------------
|  |  |  |  |  |       |  |  |  |  |  |   <-- keys
---------------  ...  -----------------
|  |  |  |  |  |       |  |  |  |  |  |
---------------  ...  -----------------
|  |  |  |  |  |       |  |  |  |  |  |
---------------  ...  -----------------
|  |  |  |  |  |       |  |  |  |  |  |
---------------  ...  -----------------

Search -->
```



Analysis

- Fortran stores matrices by columns
- (C stores matrices by rows)
- L1 data cache is two-way set-associative, two sets with 512 lines each (MIPS R10000, SGI)
- Replacement policy is LRU (Least Recently Used)
- One column per L1 cache line
- When ≤ 1024 columns only cache misses in the first search

Suppose we have two sets of four cache lines, instead.

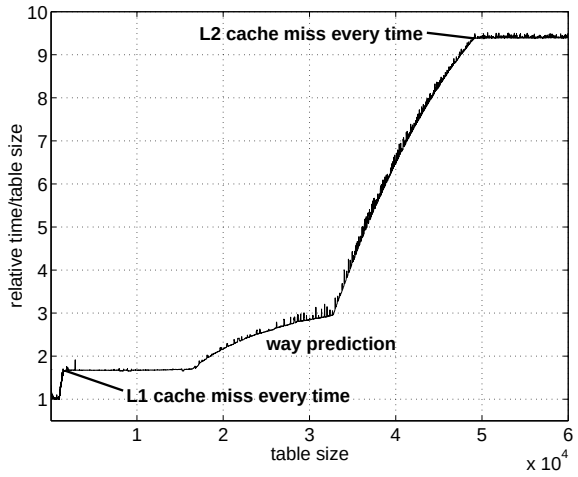
Assume we have nine columns.

```
  sets
|1|  |5|     |9|  |5|     |5|  |1|     |5|  |9|
!2|  |6|     |2|  |6|     |2|  |6|     |2|  |6|
|3|  |7|     |3|  |7|     |3|  |7|     |3|  |7|
|4|  |8|     |4|  |8|     |4|  |8|     |4|  |8|

 after 8     after 9     after 8      after 9
 cols        cols        next search
```

Assume we have twelve columns.

```
   first search                 next search
|1| |5|   | 9| |5|    | 9| |1|   |5|  |1|   |5|  | 9|
!2| |6|   |10| |6|    |10| |2|   |6|  |2|   |6|  |10|
|3| |7|   |11| |7|    |11| |3|   |7|  |3|   |7|  |11|
|4| |8|   |12| |8|    |12| |4|   |8|  |4|   |8|  |12|

after 8   after 12    after 4    after 8    after 12
```

L2-cache, two sets of 4096 lines, each with a length of 132 bytes.



Notice that the fastest and slowest case differ by a factor of 9.5.

Change algorithm and data structure. If not: blocking.

Blocking is an efficient technique for data re-use and is used in many matrix algorithms.

More data re-use; loop fusion

Blocking is one method for data re-use.

```
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 different systems.
Full optimization on the compilers.

	System 1		System 2		System 3	
	C	Fortran	C	Fortran	C	Fortran
By row	0.12 s	0.093 s	0.36 s	0.31 s	0.87 s	2.9 s
By column	1.32 s	0.093 s	1.08 s	0.31 s	3.69 s	0.68 s

The first two Fortran compilers can switch loop order, the third cannot. Notice the difference between Fortran and C.

Blocking and large strides

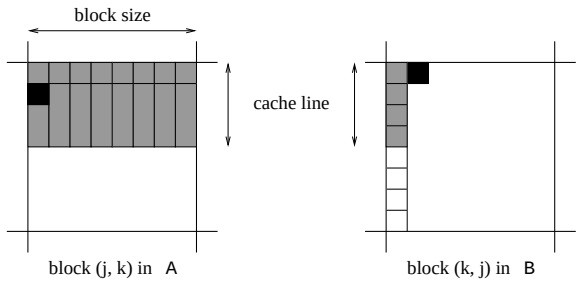
Sometimes loop interchange is of no use.

```
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.

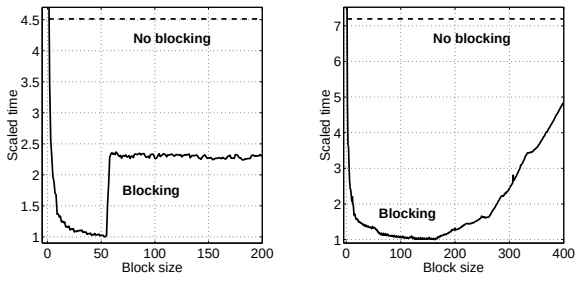
Treat pairs of blocks, one in **A** and one in **B** such that we can use the data which has been fetched to the L1 data cache.
Looking at two blocks:



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
```



$n = 2000$.

Note the speedups (4.5 and 7.2).

More on the BLAS

(the Basic Linear Algebra Subprograms).

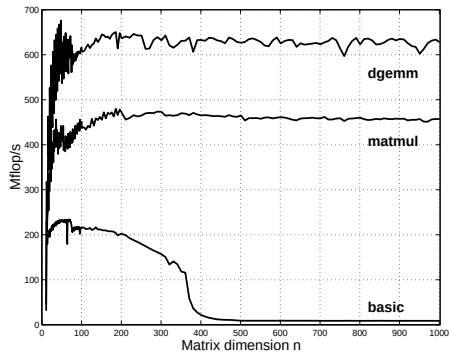
BLAS1: `y := a*x + y` one would use **daxpy**

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**

Matrix multiplication: "row times column", slow.
Blocking is necessary.



375 MHz machine, start two FMAs per clock cycle, top speed is 750 million FMAs per second.

LAPACK. Tuned libraries.

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

Inlining and overloading of operators

Inlining: moving the body of a short procedure to the calling routine. Here comes a slightly contrived example:

```
module types
  type Point3D
    double precision :: x, y, z
  end type Point3D
end module types

function norm(point) result(res)
  use types
  type(Point3D)    :: point
  double precision :: res

  res = sqrt(point%x**2 + point%y**2 + point%z**2)

end

program main
  ... code
! points is an array of type(Point3D)
  do k = 1, MANY_TIMES
    s = s + norm(points(k))  ! or something
  end do                     ! more realistic
  ... code
end
```

Changing `norm(points(k))` in the main program to `sqrt(points(k)%x**2 + points(k)%y**2 + points(k)%z**2)`, will give faster code.

Inlining, this way, by hand is error-prone, so a compiler can usually do it for you. Some compilers are not doing a very good job of it, though. It can make a difference if routines are stored in different files.

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.
- Use parallel capabilities of the CPU
  - Avoid data dependencies
  - 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.

Low level profiling

valgrind and PAPI are two tools for counting cache misses.

http://valgrind.org/, **man valgrind** and **/usr/share/doc/valgrind-3.1.1/html/index.html**

From 22nd stanza in "Grímnismál" (poetic Edda). In old Icelandic and Swedish:

Valgrind heitir,	Valgrind den heter,
er stendr velli á	som varsnas på slätten,
heilög fyr helgum dyrum;	helig framför helig dörrgång;
forn er sú grind,	fornåldrig är grinden,
en ■at fáir vitu,	och få veta,
hve hon er í lás lokin.	hur hon i lås är lyckt.

and a reasonable (I believe) English translation:

Valgrind is the lattice called,
in the plain that stands,
holy before the holy gates:
ancient is that lattice,
but few only know
how it is closed with lock.

The main gate of Valhall (Eng. Valhalla), hall of the heroes slain in battle.

From the manual:
"valgrind is a flexible program for debugging and profiling Linux executables. It consists of a core, which provides a synthetic CPU in software, and a series of "tools", each of which is a debugging or profiling tool."

The memcheck tool performs a range of memory-checking functions, including detecting accesses to uninitialized memory, misuse of allocated memory (double frees, access after free, etc.) and
detecting memory leaks.

We will use the cachegrind tool:
cachegrind is a cache simulator. It can be used to annotate every line of your program with the number of instructions executed and cache misses incurred.

```
valgrind --tool=toolname program args
```

Call the following routine

```
void sub0(double A[1000][1000], double*s)
{
  int j, k, n = 1000;

  *s = 0;

  for (j = 0; j < n; j++)
    for (k = 0; k < n; k++)
       *s += A[k][j];
}
```

Compile with **-g**:

```
% gcc -g main.c sub.c
```

I have edited the following printout:

```
% valgrind --tool=cachegrind a.out

==5796== Cachegrind, an I1/D1/L2 cache profiler.
==5796== Copyright (C) 2002-2005, and GNU GPL'd,
         by Nicholas Nethercote et al.
==5796== For more details, rerun with: -v
9.990000e+08 6.938910e-01
```



```
I   refs:     46,146,658
I1  misses:          756
L2i misses:          748
I1  miss rate:      0.00%
L2i miss rate:      0.00%

D   refs:     21,073,437  (18,053,809 rd+3,019,628 wr)
D1  misses:      255,683  (   130,426 rd+  125,257 wr)
L2d misses:      251,778  (   126,525 rd+  125,253 wr)
D1  miss rate:       1.2% (       0.7%  +      4.1%  )
L2d miss rate:       1.1% (       0.7%  +      4.1%  )

L2 refs:         256,439  (   131,182 rd+  125,257 wr)
L2 misses:       252,526  (   127,273 rd+  125,253 wr)
L2 miss rate:        0.3% (       0.1%  +      4.1%  )
```

valgrind produced the file, **cachegrind.out.5796** (**5796** is a pid). To see what source lines are responsible for the cache misses we use **cg_annotate -pid source-file** I have edited the listing and removed the columns dealing with the instruction caches (the lines are too long otherwise).

```
% cg_annotate --5796 sub.c
       Dr    D1mr    D2mr         Dw D1mw D2mw
        .       .       .          .    .    .  void sub0
        0       0       0          2    0    0  {
        0       0       0          1    0    0  int j, k,
        1       0       0          2    0    0  *s = 0;
    3,002       0       0          1    0    0  for (j =
3,002,000       0       0      1,000    0    0  for (k =
7,000,000 129,326 125,698 1,000,000    0    0 *s += A[k
        3       0       0          0    0    0  }
```

Dr: data cache reads (ie. memory reads), **D1mr**: L1 data cache read misses **D2mr**: L2 cache data read misses **Dw**: D cache writes (ie. memory writes) **D1mw**: L1 data cache write misses **D2mw**: L2 cache data write misses

To decrease the number of **Dw**:s we use a local summation variable (no aliasing) and optimze, **-O3**.

```
double local_s = 0;
for (j = 0; j < n; j++)
  for (k = 0; k < n; k++)
    local_s += A[k][j];

*s = local_s;
```

We can also interchange the loops. Here is the counts for the summation line:

```
      Dr    D1mr    D2mr
7,000,000 129,326 125,698 *s += A[k][j]; previous
1,000,000 125,995 125,696  local_s, -O3
1,000,000 125,000 125,000  above + loop interchange
```

Dw = D1mw = D2mw = 0

valgrind cannot count TLB-misses, so switch to PAPI, which can.
PAPI = Performance Application Programming Interface
http://icl.cs.utk.edu/papi/index.html
PAPI requires root privileges to install, so I have tested the code at PDC.

PAPI uses hardware performance registers, in the CPU, to count different kinds of events, such as L1 data cache misses and TLB-misses. Here is (a shortened example):


```
% icc main.c sub.c
% papiex -m -e PAPI_L1_DCM -e PAPI_L2_DCM \
         -e PAPI_L3_DCM -e PAPI_TLB_DM -- ./a.out

Processor:              Itanium 2
Clockrate:              1299.000732
Real usecs:             880267
Real cycles:            1143457807
Proc usecs:             880000
Proc cycles:            1143120000

PAPI_L1_DCM:            2331
PAPI_L2_DCM:            3837287
PAPI_L3_DCM:            3118846
PAPI_TLB_DM:            24086796

Event descriptions:
Event: PAPI_L1_DCM: Level 1 data cache misses
Event: PAPI_L2_DCM: Level 2 data cache misses
Event: PAPI_L3_DCM: Level 3 data cache misses
Event: PAPI_TLB_DM: Data TLB misses
```

The values change a bit between runs, but the order of magnitude stays the same. Here are a few tests. I call the function 50 times in a row. **time** in seconds. **cycl** = 10^9 process cycles. **L1**, **L2**, **L3** and **TLB** in kilo-misses. **local** using a local summation variable.

	icc -O0	icc -O3	icc -O3 local	icc -O3 loop interc	
time:	3.5	0.6	0.07	0.3	
cycl:	4.6	0.8	0.09	0.4	Giga
L1:	13	4	3	4	kilo
L2:	3924	3496	1923	2853	kilo
L3:	3169	3018	1389	2721	kilo
TLB:	24373	24200	24	24	kilo

time and **cycl** are roughly the same, since the clockrate is 1.3 GHz. Note that the local summation variable, in column three, makes a dramatic difference. This is the case for loop interchange as well (column four) where we do not have a local summation variable (adding one gives essentially column three).

Note the drastic reduction of TLB-misses in the fast runs.

Here comes PAPI on the blocking example,
s = s + A(i, k) * B(k, j), with **ifort -O3**.
n = 5000 and ten calls.

On the Itanium:

bs:	NO BL	16	32	40	64	128	
time:	5.6	2.0	1.6	1.5	1.6	5.1	
L1:	69	46	41	43	44	52	kilo
L2:	306	51	48	52	54	59	Mega
L3:	31	33	38	38	36	35	Mega
TLB:	257	19	12	10	15	267	Mega

Note again the drastic reduction of TLB-misses.

Profiling on a higher level

Most unix systems have **prof** and **gprof** which can be used to find the most time consuming routines. **gcov** (Linux) (**tcov** Sun) can find the loops (statements), in a routine, that are executed most frequently.
man prof, **man gprof**, **man gcov** for details.

This is how you use **gprof** on the student system.
The flags are not standardised, so you have to read the documentation, as usual.

```
ifort -O3 -qp prog.f90 sub.f90
icc   -O3 -qp prog.c   sub.f90

gfortran -O3 -pg prog.f90  sub.f90
g95      -O3 -pg prog.f90  sub.f90
gcc      -O3 -pg prog.c    sub.c
g++      -O3 -pg prog.cc   sub.c

./a.out  produces gmon.out
gprof
```

One can use other options, of course, and have more than two files. One should link with the profiling options as well since it may
include profiled libraries.
Profiling disturbs the run; it takes more time.

The Intel compilers have support for "Profile-guided Optimization", i.e. the information from the profiled run can be used by the
compiler (the second time you compile) to generate more efficient code.