Contents

Performance engineering

Thomas Ericsson

Chalmers University of Technology, Mathematical Sciences, SE-412 96 Göteborg, Sweden

1 Introduction

This chapter is an overview of how to get good performance from a computer system. We will concentrate on systems having one CPU, but most of this chapter is applicable to parallel systems as well. To get maximum performance from a parallel code it is important to tune the code running on each CPU. The goal of code optimization is usually to get good floating point performance and that is the focus of this chapter. There will be no discussion of applications from graphics, audio or video, or details about the special operations available in some CPUs for such applications. We avoid discussing specific hardware, compilers and tuned non-portable I/O-libraries, so the reader who is interested in maximum performance should study the tuning guides available for the specific systems and languages, see for example [4,10,9]. These comprehensive guides, often spanning several hundred pages, can dig deeper and discuss more tuning tips than what is possible in these thirty pages. For an example of a more technical tuning guide (closer to the hardware) see [23]. A general optimization book, also dealing with parallel programs, is [13]. A delightful book about algorithms in general is [5].

Most computational codes are written in some dialect of Fortran or C, and this chapter contains examples from both language groups, but the emphasis will be on Fortran. Matlab, C++, and Java are mentioned. A few basic tuning guidelines, together with illustrating examples, will be presented. Many compilers have become so good at optimizing code so it is hard to give short and working tuning-examples. We have left out many of the simple tuning strategies which most compilers will take care of without any help from the programmer.

Tuning can have different starting points. A large and old code which has to be optimized, such as a research- or industrial code which is run more or less continuously through the years. Researchers, graduate students and engineers add new modules to the code, but data structures and programming languages are fixed. Even a slight speedup would be of use, since the code is run on a daily basis. The other extreme is when a task is to be done once or a few times. In such a case it may be optimal to spend more computer time instead programmer time; not all code must consist of highly optimized Fortran. Tools like awk, sed [12] or perl [33] are very capable and time saving and so are powerful systems like Matlab, `<http://www.mathworks.com/>`, Maple, `<http://www.maplesoft.com/>` and Mathematica, `<http://www.`

wri.com/>. By "misusing" these systems it is easy to produce very slow code. Using an explicit for-loop for summing the elements of an array in Matlab instead of using the builtin sum-function is hundreds of times slower. The reason is that the loop is interpreted and the function consists of compiled C-code. Matlab 6.5, and later version, can increase the speed by using a just-in-time-accelerator [25].

Yet another another scenario would be a new project, where language and data structures have to be chosen. It may be necessary to decide on some parallel system such as MPI or OpenMP as well. When making these decisions the construction of a benchmark is probably time well spent. The construction of the benchmark may not be so time consuming as the 90/10-rule often applies: "90% of the time is spent in 10% of the code." So perhaps it is possible to code and test a simplified version of the computational kernel using the different languages and data structures which are under consideration. A common solution is to use Fortran for the computational part and C/C++ for computer graphics and the interface to the operating system.

1.1 The optimization process

When it has been decided that the code should be optimized there are several options available. Very important is to use an efficient algorithm, of course. Assuming that is the case, the most obvious is to check the optimization options of the compiler. Unoptimized code is very slow and optimization can give large speedups, a factor of ten in some cases. See the case studies in this book for examples. The tuning guide or the manual page for the compiler should be consulted for finding suitable options. Compilers contain bugs like all non-trivial programs, so the result of the computation must be checked after optimization. This is true for all tuning, of course. Optimization may change the order of execution and can thereby reveal bugs in ones own program as well. If aggressive optimization has been switched on the result may not be bit-wise identical to what is produced by the unoptimized version, since the order of floating point operations may have changed, roundoff differs *etc.* Switching on all possible optimization options can occasionally generate slower code. If other hardware and/or compilers are available one may try to switch. Compilers do vary in quality as does the sophistication of hardware.

The optimization steps following hereafter require more work. Disks are very slow devices as compared to the CPU, so excessive disk usage will give poor performance. Tuning I/O and reducing the use of virtual memory (simulating a larger memory using disk) are important steps.

Several numerical packages use routines optimized for CISC-systems. Linear solvers have been fetched from LINPACK [11] and eigenvalue-solvers from EISPACK [14]. The Linpack-routines are based on vector operations which give poor performance on most modern computers. The LAPACK-solvers [3] are based on matrix-matrix-operations (when possible) which give much better performance on modern systems. Replacing a LINPACK/EISPACK-routine by the corresponding LAPACK-routine is usually a straightforward operation. Specially tuned platform-specific numerical libraries may be available. These libraries also contain the BLAS-routines for basic matrix operations such as matrix multiply (there are more details about BLAS on page 24). There may be solvers for sparse linear systems, FFT-routines *etc.* LAPACK and BLAS are standard libraries and give no problems with portability. Some of these platform-specific packages are: IBM's Engineering and Scientific Subroutine Library (ESSL) `<http://www.redbooks.ibm.com/>`, SGI's Scientific and Mathematical Library (complib.sgimath) `<http://techpubs.sgi.com/>`, the Sun Performance Library (Sunperf) `<http://docs.sun.com/>`, HP's MLIB `<http://devresource.hp.com/>` and Intel's Performance Libraries `<http://developer.intel.com/>`.

The next step requires rewriting of the code, usually critical loops. The task of finding these bottlenecks in the code and measuring performance are discussed in the chapter dealing with programming tools. 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.

Despite all these complications, code tuning can be very rewarding. It is, however, important to restrict the tuning effort to important loops (the others are not worth the effort).

The goal of the tuning effort is to keep the FPUs (Floating Point Unit(s)) busy. This means that the memory hierarchy and parallel capabilities of the system must be used in a proper manner. Memory is arranged in a hierarchy. The CPU contains registers after which comes a sequence of cache memories then main memory and finally disks. Distancing us from the CPU, the memories become larger and slower. The access times of registers and disks differ by several orders of magnitude. The same is true for the storage capacities. So in order to get good performance data must be "close" to the CPU and communication with slower parts must be avoided as much as possible. Locality of reference is the key phrase. Data which has been fetched to the L1-cache should be re-used, if possible and memory should be accessed in byte order. The programmer has a large influence over how memory is used, and it is often possible to help the compiler to produce more efficient code.

Modern CPUs are superscalar, so they can start several instructions per clock cycle and using pipelining the CPU works on (some) individual instructions in parallel as well. See the chapter on computer architecture for details. The instructions must be arranged so that parallel computation is feasible, and this is where we may help the compiler doing a better job.

The tuning advice on the following pages may or may not give a speedup. The outcome is too dependent on the actual piece of code, hardware, programming language and compiler. One is probably more likely to make an impact on a program written in C than in Fortran. It is easier to improve on the job of a poor compiler; some compilers are so good that it is almost impossible to get an improvement. A code tuner must be willing to use trial and error.

The code examples on the following pages have been run on a fair number of different systems (hardware and compilers). The results have been included to give the reader an idea of the improvements that can be expected. It should be kept in mind that there are significant differences between systems and that compilers and hardware improve with time. The author had to scrap some old favourite examples since the new compilers could cope without any external tuning help.

2 C versus Fortran

The two dominating languages for high performance numerical computation are Fortran 90 (plus older dialects) and C/C++. Fortran is more adapted to numerical computations and provides complex numbers and array operations. C/C++ has support for more advanced data structures and is almost the only choice when it comes to low level UNIX programming. C++ supports object oriented programming. There are some excellent Fortran compilers due to the competition between manufacturers and the design of the language. It is harder to generate fast code from C/C++ and it is easy to write inefficient programs in C++. The difference in performance between C and Fortran can be illustrated by the following simple add-routine; here is the C-version.

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

  for(k = 0; k < n; k++)
    c[k] = a[k] + 2.0 * b[k];
}
```

The analogous subroutine was written in Fortran 90 (disregarding the fact that Fortran 90 can perform array operations). The length of the vectors, `n`, was chosen such that the three vectors would fit in the L1-cache, all at the same time. This way we could concentrate on CPU-performance not having to worry about the bandwidth of the memory system. On the two systems tested the Fortran 90 routine was **twice** as fast (the routine was called many times to give accurate timings). This is a reasonable rule of thumb as well, at least when it comes to numerical computations.

One reason Fortran is faster is that the compiler can make several assumptions about the code and it can therefore apply more optimization. In particular, it may disregard so-called aliasing (overlap of arrays or pointers). If two arguments to a procedure overlap (partially or completely) it is illegal for the procedure to change the overlapping elements (see the Fortran 95 standard [24]; a more easily accessible source is [28]).

Since the iterations in the our loop are independent from each other, the compiler may produce code that works on several iterations in parallel.

```
c(1) = a(1) + 2.0 * b(1)
c(2) = a(2) + 2.0 * b(2)   ! c(2) does not depend on c(1) etc.
```

The functional units for addition and multiplication are pipelined which implies that the compiler can produce code such that `c(1)`, `c(2)` *etc.* can be worked on simultaneously. In Fortran, the computation of `c(2)` does not have to wait for `c(1)` to be available. This situation changes completely if we make the **illegal** call, (in Fortran),

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

Let us write out the first few iterations to see why (recall that `c(2)` is passed as the **address** to `c(2)`):

```
c(2) = a(1) + 2.0 * c(1)   ! b and c overlap
c(3) = a(2) + 2.0 * c(2)   ! c(3) depends on c(2)
c(4) = a(3) + 2.0 * c(3)   ! c(4) depends on c(3)
```

We have introduced a data dependency. Several systems will give us the wrong answer in this case, at least if we ask for full optimization, since the compiler still assumes there is no overlap.

This does **not** mean that we cannot have a loop like:

```
do k = 1, n - 1
  c(k + 1) = a(k) + 2.0 * c(k)
end do
```

If this is what we need, we have to write a new subroutine (and we should not expect a very good performance, due to the data dependency). What it means is that we are not allowed to "fool" the compiler into thinking that the arrays are independent when they are not.

In C, aliased pointers and arrays are allowed which means that it is harder for a C-compiler to produce efficient code. Some systems provide for directives (put in the code) or compiler options which inform the compiler that variables are not aliased. This may allow the compiler to generate more efficient code.

3 A word on Java

Java is an object oriented language developed by Sun Microsystems in the early 1990's (publicly available in 1995). It is probably fair to say that Java is not primarily used for HPC although there are activities in that area, see e.g. `<http://math.nist.gov/javanumerics>` and `<http://www.oonumerics.org/oon>`.

In the author's view Java has problems with execution time, memory usage and support for numerical computing. The small example below has been included to show the nature of these problems. Java has advantages as well, one being the portability.

Suppose we would like to compute an inner product between the double precision complex vectors `x` and `y`. We have compared codes written in Fortran 90, C++ and Java. Complex is a builtin data type in Fortran 90. Here is the loop:

```
do j = 1, n                          ! Fortran90
  s = s + conjg(x(j)) * y(j)
end do
```

The C++ standard library has support for complex data and the loop may look like:

```
for (int j = 0; j < n; j++)          // C++
  s += conj(x[j]) * y[j];
```

Since C++ (and Fortran 90) supports overloading of operators it is possible to write `*` for complex multiplication, *etc.* This is not supported in Java so a code may look like:

```
for (int j = 0; j < n; j++)          // Java 1
  s = s.add(x[j].conj().mul(y[j]));
```

The lack of overloading can lead to hard-to-read code. Complex numbers are not available in the standard Java library. We used `JavaComplex.tgz` from Netlib (`http://www.netlib.org/java/`) in this test.

It is well-known [35] that working with many small objects may cause severe performance degradation. This is obvious when we compare the object oriented code (Java 1) to one using separate arrays for real- and imaginary parts (Java 2 below). In this code we have inlined (see page 32 for details) `add`, `mul` and `conj` as well. The major performance bottleneck in the complex code should not be due to the many calls of `add` *etc.* since the Java HotSpot Virtual Machine can perform inlining (see [1] or Appendix B in [35]).

```
for (int j = 0; j < n; j++) {        // Java 2
  s_re = s_re + x_re[j] * y_re[j] + x_im[j] * y_im[j];
  s_im = s_im + x_re[j] * y_im[j] - x_im[j] * y_re[j];
}
```

We tested the codes on two 900 MHz systems using the latest Java versions available (the Sun Microsystem Java HotSpot Server VM 1.5.0 on System 1 and the 64-bit version 1.4.2 on System 2). "small n" in the table uses $n = 10^3$ calling the inner product routines 10^5 times. "large n" takes $n = 10^5$ and making 10^3 calls (so the number of floating point operations stays the same). Note the poor performance of Java 1 as compared to Fortran 90 and

Table 1. A complex benchmark.

	System 1		System 2	
	small n	large n	small n	large n
Fortran 90	0.43 s	0.53 s	0.68 s	0.69 s
C++	0.46 s	0.53 s	0.60 s	2.59 s
Java 1	18.1 s	25.1 s	23.0 s	29.5 s
Java 2	1.8 s	4.8 s	1.3 s	3.0 s

C++ and the huge speedup when using separate arrays in Java 2. One reason Java is slower than Fortran 90 and C++ is that Java checks so that array subscripts are within bounds. A large n gives an increased number of cache misses leading to longer execution times.

In Java it is not possible to create an array of complex objects, but only an array of references (pointers) to complex objects. Pointers are a useful tool when dealing with sparse matrices, for example, but if we are dealing with dense arrays we probably would like to create an array of complex numbers (objects). This is possible in Fortran 90 and C++.

On System 1, a reference requires four bytes and on System 2 eight bytes. There is an overhead for each complex object as well. System 1 requires eight bytes extra (apart from the sixteen bytes for the real- and imaginary parts) and System 2 sixteen bytes. So, a complex array of n elements requires $n \cdot (4 + 8 + 16)$ bytes and $n \cdot (8 + 16 + 16)$ respectively. Fortran 90 and C++ require $n \cdot 16$ bytes; this is the case for the arrays in Java 2 as well. This is pretty bad news when using the 64-bit system, the memory requirements are more than doubled. This is not a big deal in the benchmark, but it could be a major problem in some applications. The storage overhead is partly responsible for the poor performance, as well. More bytes must pass through the memory hierarchy.

One advantage with Java is the portability of the compiled code (the byte code). Java is available on some cell phones, as well, and just for fun, the author implemented a Linpack code and a QL-code for computing all the eigenvalues of a real symmetric matrix. Computing all the eigenvalues of a 97×97-matrix (the largest that would fit in memory) took three minutes. Not HPC, but rather impressive for such a small device. This author's telephone has no support for floating point and he used [27].

4 Tuning Matlab programs

Since Matlab is a common tool for numerical computations, we have included some tuning tips. See the Matlab documentation for more information. The timings below are for Matlab version 6. Computation should be done using the builtin compiled routines. This means that one should try to work on the matrix/vector-level (vectorizing the code), rather than using operations on an element-level. Nested for-loops working on the element-level in matrices can take quite some time, since the Matlab-language is interpreted. As was mentioned in the introduction Matlab 6.5 and later versions can increase the speed by using a JIT-accelerator [25]. It still pays to vectorize the code though.

The other important factor is to use the dynamic memory allocation features in an efficient way. If an element of an array does not exist and we assign a value to this element, the dimensions of the array will be enlarged to accommodate the new element. It is much more efficient to preallocate the necessary memory.

Here come a few examples, `n = 1500` in all cases. To show how for-loops can slow down a program, we have formed the matrix sum:

```
for j = 1:n
  for k = 1:n
    A(j, k) = A(j, k) + B(j, k);
  end
end
```

The code above takes 45 s while the builtin compiled addition operation, `A = A + B;`, requires 0.1 s. If we remove the innermost line from the loops, just leaving the empty loops, the run takes 2.2 s.

Suppose `A` does not exist when we run the following loop. In each iteration we add a column to the matrix (in a real application we would have different arrays `x`). Allocating more memory this way is very time consuming, and the run takes 154 seconds.

```
for k = 1:n
  A(:, k) = x;
end
```

If we put the statement, `A = zeros(n);`, before the loop we have preallocated the required memory for the matrix `A` and the time for the loops goes down to less than 0.3 s.

Parentheses can be used to change the order of evaluation. This is very important in certain situations. In this example `W` is a 1500×10-matrix and `x` is a column vector having 1500 elements.

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

The left form takes 0.4 s and the right takes less than 0.03 s. In the left case a temporary 1500×1500-matrix is formed which is then multiplied by the vector. In the right version only the temporary vector, `(W' * x)`, is formed. For large `n` the temporary matrix, `W * W'`, can easily consume the available memory.

5 Basic arithmetic and elementary functions

There is a standard for floating point arithmetic [21,19,15], which covers storage formats, basic arithmetic (`+, -, *, /, sqrt`) rounding, exceptions and some other topics. Single precision uses 32 bits and double precision 64 bits as shown in Tab. 2 (subnormal is a synonym for denormalized). Most modern computers adhere to this standard although there are exceptions (some CRAY computers, for example).

Table 2. Floating point numbers, approximate ranges.

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

It is common that the FPU (Floating Point Unit) can perform an addition and a multiplication in parallel. An operation like `a+b*c` can often be performed with one round-off and the combined multiply-add is often called MADD or FMA. There are machines having multiple FPUs, so the CPU may be able to execute two FMAs in parallel, for example. Most systems have pipelines for addition and multiplication, so provided we can get the operands to the FPU in time, a sum and a product can be produced every clock cycle. Note that we cannot expect to get two sums (or two products) per clock cycle (unless the machine has several FPUs). Division is not usually pipelined and may require around twenty clock cycles.

Using single- instead of double precision can give better performance, as 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, and this is very noticeable when it comes to gradual underflow (the handling of denormalized numbers). See [19,15] for details. Consider the following Matlab-example (`tic/toc` takes the time):

```
>> A = rand(200); B = A;     % random matrices with elements
>> tic; C = A * B; toc       % in (0, 1).
elapsed_time = 0.0432        % time for the multiplication
```

```
>> A = 1e-320 * A;          % denormalized numbers in A
>> tic; C = A * B; toc
elapsed_time = 158.1941     % time for the multiplication
```

In this particular machine denormalized numbers are handled in software, which explains the enormous amount of time used for the second multiplication. Note that this is **not** Matlab's fault, it is a consequence of the design of the computer. It is usually possible to turn off the handling of denormalized numbers when compiling a program but this leads to abrupt underflow (instead of gradual underflow). There exist CPUs that handle denormalized numbers in hardware. Denormalized numbers are usually quite rare, but there are a few occasions when they are common, so the Matlab-example is a bit extreme.

For better performance one can sometimes replace a *division* by a multiplication. Suppose, for example, that `v` is an array having `n` elements, and that `norm` is the length of the vector. The normalized vector can be computed by `v / norm` requiring `n` divisions. A faster alternative is to compute `v * (1.0 / norm)` which requires one division and `n` multiplications. Changing divisions to multiplications this way may give slightly different results. There are a few cases where there can be major differences. For example, (`d` denotes double precision exponent): `v(j)/1d-320`, with `v(j)=1d-100`, becomes `1d220`, but `v(j) * (1d0/1d-320)` becomes `Inf`, since `1d-320` is a denormalized number and `1d0/1d-320` overflows giving an `Inf`. Another example is given by `a/b/c` which can written as `a/(b*c)` which is faster. Say that all three numbers are roughly `1d200`, then the first form will give something like `1d-200` but the other will give zero, since `b*c` will overflow.

An increasing number of CPUs have SIMD-capabilities (Single Instruction Multiple Data) making it possible for the CPU to work on short vectors in parallel. Two, of several, such technologies are the SSE-family [22,7,2] (Streaming SIMD Extensions) and AltiVec [30,31]. The data types and operations supported by the technologies vary.

To make use of the SIMD-assembly instructions special compiler options may have to be used. Speedup varies, from no speedup, or even a slowdown, to perhaps a factor of two or more. The actual speedup depends, as all optimization, on the application, data types, compiler and hardware so the resulting performance is not easy to predict.

Integer multiplication and multiply-add are often (but not always) slower (by a factor of two to five, perhaps) than their floating point equivalents. Changing four byte integers to single precision floating point seems natural, but proper care must be taken. The overflow properties are different and the floating point number has fewer bits in the mantissa (using double precision is an option of course). If, for example, `k = 1234567` is a four-byte integer, then `k * k` will give integer overflow and the result will probably print as `-557712591`. In single precision we get an approximation, `1.524155679e12`, of the exact value, `1524155677489`. If we let `k = 12345` the square will be computed correctly, `152399025`, using integers but the single precision value comes out as the approximate `152399024`.

The author has seen a few programs where the computation of *elementary functions* require almost half of the execution time. In one extreme example (antenna design) the total runtime was three CPU-weeks and trigonometric functions accounted for one week. Elementary functions (with the exception of `sqrt` which may be implemented in hardware) are often coded in C and the routines usually reside in the mathematics library (`libm`). A function is typically evaluated in three steps. Let us take the computation of $\sin x$ as example. It is usually hard to approximate a function on a large interval, so the first step is argument reduction. In the sine-case this is done by subtracting a suitable integer multiple of $\pi/2$ from x giving us the reduced value r. Using standard trigonometric identities we see that it is sufficient to know $\sin r$ on the interval $r \in [0, \pi/2]$. It is even possible to approximate $\sin r$ on $[0, \pi/4]$ if the cosine function is used, since $\sin(\pi/2 - r) = \cos r$. The second step consists of approximating the function by a high-degree minimax-polynomial (or a rational approximation for some functions) on this short interval. For functions like $\exp x$ it is necessary to transform the reduced value, $\exp r$, back to the original $\exp x$.

This has some implications. Computing elementary functions may take **much more** time than a few multiplications and additions. Since argument reduction is not always needed, different arguments require different amounts time (a factor of twenty in time is not impossible).

The computation of elementary functions is not covered by the IEEE-standard and the quality of implementations do differ. The author has used systems where a computed sine-value may have all digits wrong, and on some machines it is possible that $\sin x > 10^7$ for some large x. Some implementations return a good approximation to $\sin(fl(x))$ where $fl(x)$ is the closest machine number to x (but note that x and $fl(x)$ may be very far apart in absolute terms).

Some systems have faster but less accurate versions of the elementary functions, see for example the MASS library in [4]. It is common with vector versions of the elementary functions. These procedures take an array of arguments and return an array of results. These procedures may be substantially faster than using the standard calls.

The author has seen a few instances where it has been possible to decrease the number of function references by using trigonometric identities and tables containing a few precomputed values.

It is important to move constant function references from loops. This is something that most compilers will do when it comes to simple arithmetic expressions.

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

```
do k = 1, 1000000
  x(k) = exp(2.0) * y(k)
end do
```

In the first example `2.0 * pi + 3.0` will be evaluated once and re-used in the loop. In the second example one compiler computes `exp(2.0)` once and stores the computed value in a register. Another compiler recomputes `exp(2.0)` in every iteration. If we replace `exp` by a function of our own construction, `f` say, a compiler **must** recompute `f(2.0)`, even though the argument is constant. This is because the function could have side-effects (performing I/O, changing parameters or global variables, for example). This is considered bad programming practice, and in Fortran 95 it is possible to assert that a procedure has no side-effects by using the `PURE` keyword [28]. Using `PURE`, an interface block and full optimization some, but not all, compilers the author tested would compute `f(2.0)` once.

6 Using disk

Disk is used when performing I/O but also in connection with virtual memory. Since disks are very slow in comparison to the CPU, it is essential to use them sparingly and in an efficient way. We start with a section on paging after which come some tuning recommendations for I/O.

6.1 Virtual memory and paging

Virtual memory makes it possible for programs to use more memory than what is available physically in the computer. The larger memory is simulated using disk. Virtual memory is divided into pages, perhaps 4 or 8 kbyte, each. Moving pages between disk and physical memory is known as paging. Some paging is natural and does no harm, but excessive paging, so-called thrashing, can bring a program to a standstill. Having large data structures where some parts are used infrequently is a useful application of virtual memory. The unused parts will be paged out on disk and paged in only occasionally.

Having large data structures where all parts are accessed frequently will lead to severe problems, as is shown in the following Matlab-example. The test-rogram was run on a machine with only 64 Mbyte memory, in order to show the effects of paging very clearly.

Note that $m * n^2$ is constant so the number of additions are the same in all three cases. Note also that 91.6 Mbyte is much larger than the size of the physical memory. Not all the 64 Mbyte in physical memory is available for storing matrices either, since the operating system, Matlab *etc.* takes up a whole lot of space.

```
>> 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
```

There are several tools that can be used to find out if the program is making excessive use of virtual memory. If the computer is standing on your own desk and paging is done to the local disk, you can listen for noise from the disk. A disk that never rests is usually a sure sign of paging (provided no other I/O is taking place, of course). A common tool is `vmstat` (virtual memory statistics). It was run in another window in parallel with Matlab session, and the table below shows an edited version of the printout. Each line represents the activity over the previous second (`vmstat p` prints the average over the `p` previous seconds).

```
% 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 three minutes.

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

We see that `pi` and `po` go up when the third test run is made. `us` stands for user time (time spent doing the computations in Matlab), `sy` is system time (executing operating-system routines on behalf of Matlab) and `id` is doing nothing. The system is essentially spending all time waiting for disk, so this example is somewhat extreme. Also the time depends on the speed of the disk and how it is connected to the machine (is it a local disk or a disk connected through a network). Many systems have fast and large local disks that can be used for paging and for storing scratch files.

6.2 Input-output, I/O

It is often necessary to use files to store large data sets, temporarily or for later use. The following part shows a few different ways this can be done in Fortran. The most space- and time-consuming is to use formatted I/O, and writing one number per line. Numbers must be converted from binary format to text format which requires many calls of the routines in the I/O-library. A further disadvantage is that the conversion may introduce rounding errors. Advantages are the portability (to other systems) and that it makes debugging easier, since it is easy to inspect the contents of a formatted file.

When performance is important unformatted files are the only choice. It is also important to group data in fairly large records, instead of writing separate numbers.

Suppose we need to store $5 \cdot 10^6$ double precision numbers in a file. Tab. 3 shows the performance using a few different methods. A local disk was used for the tests. `10` is the unit of the file (different tests may require different `open`-statements).

Table 3. Comparison between different I/O.

Test	Statement	time (s)	size (Mbyte)
1	`write(10, '(1pe23.16)') x(k)`	51.5	114.4
2	`write(10) x(k)`	12.0	76.3
3	`write(10) (vec(j), j = 1, 10000)`	0.4	38.2
4	`write(10) vec(1:10000)`	0.4	38.2
5	`write(10) vec`	0.4	38.2
6	`write(10, rec = k) vec`	0.3	38.1

In the first test we use formatted writing with 23 characters per number and one number per line. `k` in `x(k)` is the loop index (going from 1 to 5000000). The file size can be computed as follows:

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

The `ls`-command can be used to find the size as well, of course.

The second test shows how unformatted writing behaves. Each number requires eight bytes, but each record (number in this case) requires a header and a trailer, each requiring four bytes, making a total of $5 \cdot 10^6 \cdot (8+4+4)/2^{20} \approx 76.3$. It is obviously better to have larger records.

In case three to five we write the file using 500 records each consisting of 10000 numbers. So the statements in the table are repeated for `k = 1, 500`. Note the difference in speed. The size of the file is given by $500 \cdot (8 \cdot 10000 + 4 + 4)/2^{20} \approx 38.2$ In case five `vec` has 10000 elements. The implied do loop, case three, may be slower on some systems.

Up to now we have only looked at sequential files. In the final test (no. 6) we are using a direct access file with a record length of 80000 bytes. In this case no record separators are needed and the size of the file is $500 \cdot 8 \cdot 10000/2^{20} \approx 38.1$. Direct access files allow us to access the records in any order but the record length must be the same for all the records in the file.

To appreciate the difference between the last two cases, we have tested writing and reading a larger file. Keeping the same record length we wrote 5000 records and then read the file ten times. The file size is roughly 381.5 Mbyte in both cases and the read time is for one reading of the file. The result is shown in Tab. 4.

Table 4. The difference of performance between direct access files and sequential files.

Statement	write (s)	read (s)
`write(10) vec`	5.1	6.0
`write(10, rec = k) vec`	3.4	4.6

Operating system I/O-buffers are typically sized as a power of two. By using a record length which is adapted to this length (do not forget possible headers and trailers) we may get further improvement (see the tuning guides for the different systems for details). So 10000 is not necessarily optimal. Having very large records cause problems for some systems. If `A` is an enormous matrix, it is probably unwise to write the whole matrix in one record, `write(10) A`. Writing in chunks is safer, perhaps one column at a time, so something like:

```
do k = 1, n
  write(10) A(:, k)
end do
```

Most systems have commands such as `iostat` or `dkstat` which can provide I/O-statistics.

One important disadvantage with binary files is that they may not be portable between different systems, even if all the systems involved follow the IEEE-standard for floating point. The structures of the files may differ (the interpretation of record delimiters, for example) and the byte order can be different, little-endian or big-endian (the terms come from J. Swift's "Gulliver's Travels" [17] via [8]). In big-endian machines the most significant byte has the lowest address ("big-end-first"). The Intel processors are little-endian ("little-end-first") for example.

In the following example we opened a binary file and executed the following write-statement on a big-endian machine. The number 10 is the unit, and `1.0d300` is the double precision constant 10^{300} *etc.* using the notation from Fortran 77.

```
write(10) -1.0d-300, -1.0d0, 0.0d0, 1.0d0, 1.0d300
```

When we read the file on a little-endian system we got the following values (and no warning):

```
2.11238712E+125  3.04497598E-319  0.
3.03865194E-319 -1.35864115E-171
```

Even if both systems use the same byte order care should be taken, as the file layout may differ.

There are libraries for machine-independent formats, such as NetCDF (Network Common Data Form) [29]. The web-page in the above reference contains links to other such formats as well.

7 Memory locality and caches

As we said in the introduction, it is essential to keep data close to the CPU. This implies that one should try to minimize the number of accesses to main memory, and once data has been brought into registers or the L1 cache it should be re-used, if possible. The following examples show some different techniques.

7.1 The function of caches.

We start this section with a simple cache example. The algorithm is very slow and primitive, but the plots are informative.

We have a matrix (a table), in Fortran, with four integers per column and a varying number of columns (the table size). The first integer in each column is a key and we search sequentially through the table to find a particular key. In this example it is always the **last** column that matches. We repeat the search 99 times after which we add a column to the table and start over. For each table size we have computed the time for the search divided by the number of columns. A first guess may be that this time should be fairly constant, independent of table size. This is not the case, however, since we will not have equal access time to all keys in the table. In Fig. 1 we have plotted time/table size scaled so that the minimum quotient is one. The shape of the curve depends very much on the particular system, of course.

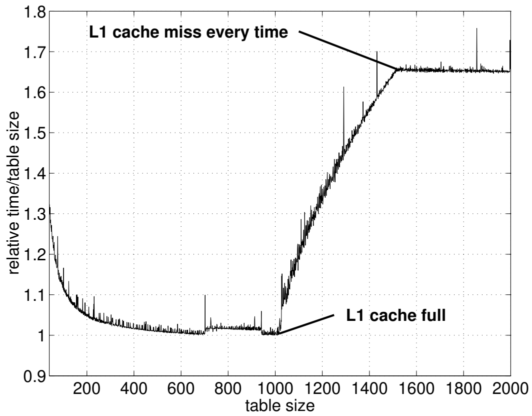

Fig. 1. Behaviour of the L1 cache.

Suppose the table size is such that the whole table fits into the L1 data cache. The first time we search the table we will get cache misses for all the columns, but the table will reside in the cache when we perform the remaining 99 searches. When we add columns to the table, there will come a point when the whole table cannot fit into the cache. This means that we will get some cache misses in every one of the remaining 99 searches. The time will increase with the number of misses (the number of columns). The worst that can happen is that we get a cache miss for every column in the table. In this case the time curve will flatten out, because things cannot be worse (although for large enough tables the L2 cache will play a role as well). The increase in time for small tables is partly due to loop overhead.

To understand the details of the figure, we can think of one column of the table as an L1 cache line. The reason is that Fortran stores matrices by columns, so a column of four, four byte, integers requires 32 consecutive bytes in memory and an L1 cache line on this particular system consists of 32 bytes as well (this is the reason we chose four integers per column). (Note that C stores matrices by row.) The L1 data cache is two-way set-associative and consists of two sets with 512 lines each. The replacement policy is LRU (Least Recently Used). This means that the cache will be full when the table consists of 1024 columns. So we expect the time to increase when the table has 1025 columns or more. The curve flattens out again, when we get a cache miss for every column. This happens when the table has 1536 columns ($3 \cdot 512$). We urge the reader to think through the smaller example with twelve columns in the table and a cache holding two sets of four lines each.

Fig. 2 shows what happens when we increase the number of columns. The leftmost dip can be recognised as Fig. 1 but on a different scale. On the tested machine, the L2-cache has two sets of 4096 lines, each with a length of 132 bytes. We can make roughly the same analyzis as for the L1 cache (the L2 cache stores code as well).

Using a performance tool, specific to the particular system, the author has come to the conclusion that the middle bump is due to "way prediction" (the hardware starts to mispredict which L2-set to put the cache line in).

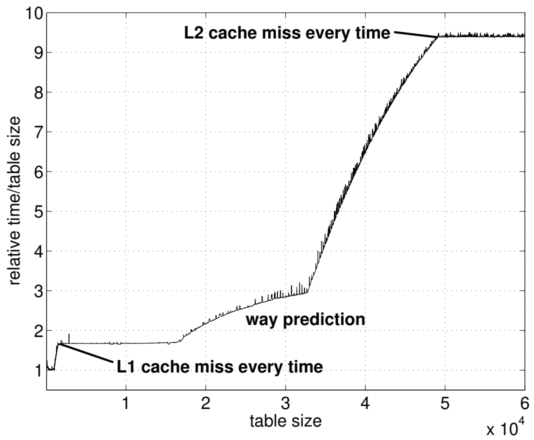

Fig. 2. Performance for large tables.

Notice that the fastest and slowest case differ by a factor of 9.5. Sequential search is not a good choice when the tables are so large, so in this case we should change algorithm and data structure to speed things up. One way to increase performance even if we keep the sequential search is to used so-called **blocking**. Suppose we are given a set of keys, instead of one key at a time. Using the first key we search in as much of the table as fits into the L1 cache. This will still give us cache misses. However, we then perform the search for the other keys in the set, but we search only in the part of the table that resides in cache (giving no additional cache misses). We repeat the search with the remaining keys (some may have been found) of the next block of the table that fits into cache. This procedure is iterated until all the keys have been found. Blocking is an efficient technique for data re-use and it is used in many matrix algorithms.

7.2 More data re-use; loop fusion.

Blocking is one of several methods to accomplish data re-use. In the following min/max-computation we merge loops instead.

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

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

It is better to merge the loops into one loop (loop fusion), since we can re-use `v(k)`. There will also be less overhead for the loop. A suitable loop body is given by:

```
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
```

but on some systems the following 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)
```

In some situations the opposite, loop splitting, may be an advantage. If we have put too many statements referring to unrelated arrays in one loop, loop splitting may result in better cache usage.

7.3 The importance of small strides.

In the two previous examples we have looked at the importance of data re-use. In some cases data cannot be re-used, but it is still important to use the caches in an efficient manner. Locality of reference is the key phrase.

Suppose that `v` is an array and that we have to operate on its elements once, and that we can choose the order of access. One way to bring down the number of references to main memory is to access the elements in `v` using stride one, `v(1), v(2), v(3),...`. The stride is distance between successive elements. In a more general setting we say that, `v(k+j*m)`, accesses the elements with stride `m` if `k` is constant and we loop over `j = 1, 2, ...`. Stride one is good since when `v(j)` is needed (and does not already reside in cache), a whole cache line is fetched from memory and not just the single variable. This means that `v(j+1)` often can be found in the cache-system and the access to main memory is avoided (see the chapter on computer architecture for details). Cache lines have a limited length, so if we use a large stride, the next variable will not be in the same cache lines as `v(j)`.

Suppose we have to compute the sum of all the elements in a square matrix of order `n`. Here are two alternatives (`row` and `col` are of type integer):

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

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

The alternative to the right will be faster, since we are accessing the elements in `A` using stride one (note the different order of the loops). As was discussed in the first example, Fortran stores matrices by column (so `A(row+1, col)` follows `A(row, col)` in memory). The left alternative is using stride `n` since `A(row, col+1)` and `A(row, col)` are one column apart.

Some compilers can switch loop order (loop interchange) provided we turn on enough optimization. We should make certain that the right loop order is used, wrong order can be slower by a large factor, five to fifteen, perhaps; it depends very much on compiler, hardware and the value of `n`.

In C, matrices are stored by row, so the leftmost alternative will be the faster.

Tab. 5 shows how this program performs on three different systems. We have turned on full optimization on the compilers, and the first two Fortran compilers can switch loop order, the third cannot. Notice the difference between Fortran and C.

Table 5. The importance of loop order.

	System 1		System 2		System 3	
	C	Fortran 90	C	Fortran 90	C	Fortran 90
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

7.4 Blocking and large strides.

Sometimes loop interchange is of no use. Suppose we would like to compute the following sum:

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

In this case we have good locality for B but not for A. Loop interchange will not improve things. As we saw in the first example, blocking is good for data re-use, but it can also be used in connection with large strides.

Let us partition A and B in square sub-matrices each having the same order, the block size. In a real application the dimensions of A and B may not be divisible by the block size in which case some cleanup code may be necessary. The idea is to 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.

In Fig. 3 we have zoomed in on a pair of sub-matrices, block (j, k) in A and block (k, j) in B, say.

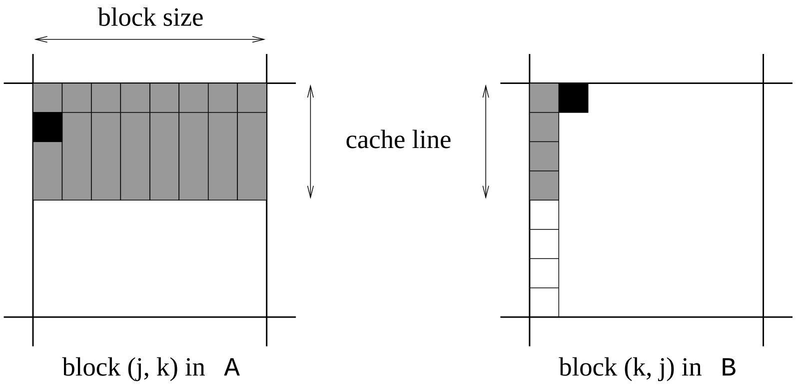

Fig. 3. Blocking.

Let us assume that each cache line consists of four double precision numbers. When we fetch the (1, 1)-element of the B-block we will get four numbers (assuming that the number in questions is the first in its cache line). The cache line has been marked with a grey pattern. When the corresponding A-element is accessed a cache line is fetched (a grey column). For the next iteration, the (2, 1)-element of the B-block is already in cache but the (1, 2)-element of the A-block has to fetched from main memory (it will not be in cache, since we are accessing A by **row**). It continues this way until we have processed the first column of the B-block. When we access the (1, 2)-element (filled with black), in B, the corresponding (2, 1)-element (filled with black) of the A-block is already in cache.

It is essential that the block size is not too large. For this blocking technique to succeed we must be able to hold all the grey elements in A in cache (until they have been used). If n is large and we run through the first column of B we will not be able to store the first four rows of A in the cache so when we start with the second column of B the elements in the second row of A must be fetched again.

The Fortran code may look like this (note that using the min-statements this works even if n is not divisible by the block size).

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

s = 0.0
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
```

Fig. 4 shows two testruns. The left plot comes from a machine with a cache system of simpler design, while the right system has more advanced hardware and larger caches. The order of the matrices were 2000. The solid lines shows the scaled time (time divided by the minimum time) as a function of block size. The dashed horizontal lines show how the unblocked algorithms behave (they are independent of block size, of course). Note the speedups (4.5 and 7.2). In the left image the time increases rapidly above a certain block size. This particular block size depends on n (for small n it moves to the right).

8 More on the BLAS.

BLAS, the Basic Linear Algebra Subprograms arose in stages, driven by the development of hardware. The original BLAS, level-one BLAS (also called BLAS1), contains vector operations such as inner product, scaling of vector and linear combinations of vectors. Level-two BLAS (BLAS2) contains matrix-vector operations such as matrix-vector-multiply. BLAS3, the third level, deals with matrix-matrix operations such as matrix multiply. Good references for the different BLAS-levels is [16] and <http://www.netlib.org/blas/index.html> (this web page contains a quick reference guide, blasqr.ps, for the BLAS as well).

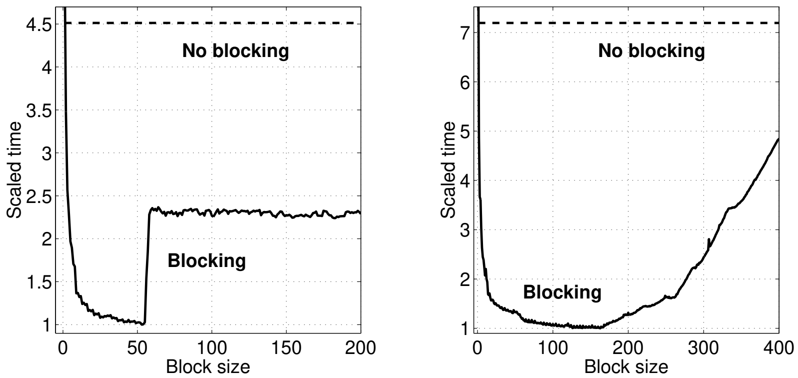

Fig. 4. How blocking performs on two systems. A solid line shows the time as a function of block size, and a dashed line shows the time using no blocking.

Let us study one routine from each of the BLAS-levels. `a` and `b` are scalars, `x` and `y` are vectors having `n` elements, `A`, `B`, and `C` are square matrices of order `n` (the routines can be used in more general situations than is shown below). All floating point variables are double precision.

To compute `y := a*x + y` one would use `daxpy` (double `a` times `x` plus `y`). The initial `d` stands for double precision (`s` is for single, `c` for complex and `z` for double complex). `dgemv`, in BLAS2, can compute `y := a*A*x + b*y`, and `dgemm`, in BLAS3, forms `C := a*A*B + b*C`. If `a = 1` and `b = 0` this is an ordinary matrix multiply.

A call to `daxpy` involves $\mathcal{O}(n)$ data and performs $\mathcal{O}(n)$ operations. This is the main reason why BLAS1-routines are slow on RISC-systems. There are too few floating point operations as compared to the number of memory accesses. The same is true for `dgemv` where we have $\mathcal{O}(n^2)$ data and $\mathcal{O}(n^2)$ operations. The BLAS3-routine differs, however, as we have $\mathcal{O}(n^2)$ data and $\mathcal{O}(n^3)$ operations. The ratio between operations and data is to our favour. We can see the opportunities for data re-use in a slightly different way. When we compute `C = A * B`, `A(j, k)` is used when forming all elements in row `j` of `C` (similarly for the elements in `B`). One may think that it would fairly straightforward to write a fast matrix multiplication routine, but this is unfortunately not the case. The basic algorithm taught in most linear algebra classes, "row times column", `C(j, k) = A(j, :) * B(:, k)` (to use Matlab notation) gives very bad performance unless one is using a really superb compiler (or a preprocessor, which recognizes the code and replaces it with something faster). There are actually several research projects working on methods to, more or less automatically, generate fast BLAS-routines, see for example [6,34,32] and [18,26] for related problems. One reason this is so important, is that the BLAS-routines are used as building blocks in other programs, for example in LAPACK. We have not sufficient room in this chapter to describe how to write a fast `dgemm`-routine. The basic ingredient is blocking, see [4] for a thorough discussion.

To show the difference between different implementations, we have tested `C = A * B` for varying dimension, `n`, of the square matrices. (see Fig. 5).

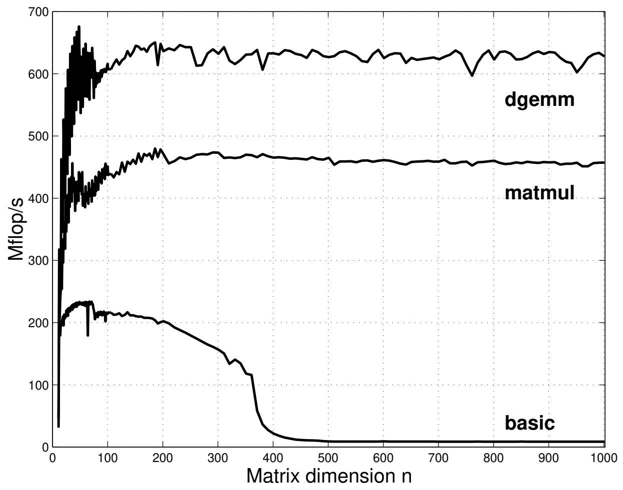

Fig. 5. Performance of different matrix multiplication routines.

Along the y-axis we have million multiply-add operations per second, Mflop/s, (in a matrix-multiplication, * and + appear in pairs). In some contexts, Mflop/s, is defined as number of million floating point operations per second, which would double the performance numbers used in the plot. On this particular system the fastest routine is a tuned platform-specific `dgemm`, after which comes the Fortran 90 builtin `matmul`. The basic algorithm is the slowest of them all. We used full optimization on the compiler. This particular machine is running at 375 MHz and can start two FMAs per clock cycle, so the theoretical top speed is 750 million FMAs per second. Note that `dgemm` comes rather close to this top speed. The curve for the basic algorithm is quite typical for matrix routines based on vector operations, such as the LINPACK-solvers for linear systems. For short vectors there is a fair amount of loop overhead and the pipelining does not perform too well. In a middle region the routine behaves in a reasonable way, but for large matrices the absence of blocking leads to poor cache usage and performance drops to a miserable level. The author recommends using LAPACK and the tuned libraries.

9 Optimal performance of loops and loop unrolling

In this section we examine how compilers and programmers can reshape the code so as to make use of the parallel capabilities of the CPU. To better appreciate the problems involved we will study assembly code produced for a fictitious CPU. This can be interesting to do for real machines as well, and it is sometimes the only way to understand why a piece of code behaves in a particular way (it is fairly standard that the `-S` option turns on the generation of assembly code).

Let $\boldsymbol{T}$ be a symmetric, tridiagonal and positive definite matrix of order n. Suppose we have a large number of systems, $\boldsymbol{T}\boldsymbol{x}_j = \boldsymbol{b}_j, j = 1, 2, \ldots, n_{rhs}$ to solve (fixed coefficient matrix but several right hand sides). We need to know if all the right hand sides are available from the beginning, or if we are given one $\boldsymbol{b}_j$ at a time. The latter case occurs if $\boldsymbol{b}_j$ depends on $\boldsymbol{x}_{j-1}$, for example. If we have access to all the right hands sides we have more room for optimization and can possibly construct a faster code (but it is easy to make a slower one, as well). Let us assume that we get one right hand side at a time. How should we solve this problem in an efficient way? The easy way is to use the numerical package LAPACK [3]. `dpttrf` computes the factorization, and `dptts2` solves a system given the factored matrix. Using the manufacturers implementation of LAPACK may enhance performance even further. But let us try to do something ourselves.

The standard way to solve the linear system is to compute a triangular factorization of $\boldsymbol{T}$, $\boldsymbol{T} = \boldsymbol{C}\boldsymbol{C}^T$, the Cholesky factorization or $\boldsymbol{T} = \boldsymbol{L}\boldsymbol{D}\boldsymbol{L}^T$ ($\boldsymbol{D}$ is diagonal, $\boldsymbol{C}$ and $\boldsymbol{L}$ are lower triangular and bidiagonal, with $\boldsymbol{L}_{k,k} = 1$) see [16,19] for details. It turns out that computing $\boldsymbol{C}$ takes more time since `sqrt` is used and the solver part uses more operations. So let us concentrate on the $\boldsymbol{L}\boldsymbol{D}\boldsymbol{L}^T$-factorization. Given $\boldsymbol{D}$ stored as a one-dimensional array `d` and the subdiagonal of $\boldsymbol{L}$ stored as the array `e(1:n-1)` the following algorithm overwrites $\boldsymbol{b}$ with the solution $\boldsymbol{x}$:

```
do k = 2, n
  b(k) = b(k) - e(k - 1) * b(k - 1)
end do

b(n) = b(n) / d(n)
do k = n - 1, 1, -1
  b(k) = b(k) / d(k) - e(k) * b(k + 1)
end do
```

Let us analyze the **first** loop. There are two problems with this loop, from a performance perspective. In each iteration, `e(k-1)` must be fetched from memory and `b(k)` must be both loaded and stored (any reasonable compiler will store `b(k-1)` in a register between iterations). So we have three memory accesses per FMA. The other problem is the data dependency, `b(k)` depends on the previous `b(k-1)`. The amount of parallelism in the loop is thus reduced, which is a nuisance if we have multiple FPUs or a parallel computer (several right hand sides would help here).

Let us now find out the maximum performance of the loop. To do this we must know something about the hardware. Let us assume that the CPU is superscalar and can start four instructions every clock cycle, one load or store, one FMA and two integer instructions (including a branch). Just issuing the required two loads and one store requires three clock cycles. Provided that the memory system can keep up with this pace, we can start an FMA every third clock cycle. So, this loop will, at most, run at one third of top speed (since top speed is obtained if one FMA is issued every clock cycle). The loop can run much slower if the FPU must wait for memory, which is often the case.

To make a more detailed analysis let us assume that a load of a double precision number from L1-cache requires three cycles. The FMA executes in three cycles, where `e(k-1)` and `b(k-1)` must be available at the start of the FMA-instruction; `b(k)` can come one cycle later. Integer instructions require one cycle. This describes a fictitious and simplified CPU, and a user who would like to see more realistic examples can consult [4,9]. Compiling the loop without using any optimization may produce code like in Tab. 6, where we have omitted initialization code. The cycles have been numbered from the start of the loop, and inst 1-4 denote the four instruction streams. The extra horizontal lines have been added for readability. We assume that the

Table 6. Instruction schedule.

cycle	inst 1	inst 2	inst 3	inst 4	comment
0	load `e(k-1)`				
1	load `b(k)`				
2					wait for loads
3		FMA			
4					`b(k)` available
5					wait for FMA
6	store `b(k)`		branch	`k = k+1`	

load/store-instructions have sufficient flexibility so that we can write `load e(k-1)` *etc.* If this is not the case we can have separate counters for each array (there are plenty of free cycles to update these counters).

Provided that the arrays are placed in the L1-cache this code will compute one FMA every seven cycles, which is far from the optimal one FMA in three cycles. By rearranging the code we can reach the optimal speed, and Tab. 7 shows one possible solution (of several). Before starting the loop, `b(1)`, `e(1)` and `b(2)` should be available in registers, and `k` should be set to `3`. The FMA computes `b(k-1) = b(k-1)-e(k-2)*b(k-2)` and the store instruction is also lagging behind, storing `b(k-2)`. Some cleanup is necessary. When the

Table 7. Optimized instruction schedule.

cycle	inst 1	inst 2	inst 3	inst 4	comment
0	load `e(k-1)`	FMA			`b(k-1)-e(k-2)*b(k-2)`
1	load `b(k)`				
2	store `b(k-2)`		branch	`k = k+1`	

arrays are large or if they have not been accessed for some time we cannot assume that they reside in the L1-cache. If the requested elements reside in the L2-cache there will be a delay of perhaps ten cycles. If main memory has to be accessed there will be an additional twenty cycles (this depends very much on the system). In some cases we may get a really long delay, several hundred cycles, if we get an unfavourable case of TLB-miss (some misses just take tens of cycles). The TLB (Translation Lookaside Buffer) is used for virtual to physical address translation. See the chapter on computer architecture, [20] and the tuning guides for more details. To decrease the waiting time, compilers use prefetching (fetching data before it is needed). This may give the memory system the necessary time to deliver the data in time to the CPU. A programmer can use this technique as well, but we refer the reader to the tuning guides for details.

A standard technique, used by compilers and programmers, to make it easier to exploit the parallel capabilities of the CPU is **loop unrolling**. This means performing fewer iterations but having more work in each loop. Apart from less loop overhead there is more room for optimization (more to work with in each iteration). In the next piece of code we have taken our loop and unrolled it twice.

```
m = modulo(n, 2)                          ! even or odd?
if ( m == 0 ) b(2) = b(2) - e(1) * b(1) ! cleanup

do k = 3 - m, n, 2
  b(k)     = b(k)     - e(k - 1) * b(k - 1)
  b(k + 1) = b(k + 1) - e(k)     * b(k)
end do
```

One can unroll more, but unrolling too much will slow the code down, since the CPU will run out of registers. It is usually necessary with a cleanup loop, before or after. Since it is hard to predict the outcome of the combined effort of programmer and compiler, it is usually necessary to test a few alternatives. In our case, the loop is so simple, so it is not certain that we can improve on the job of the compiler. To see that unrolling can make a difference we have tested using less optimization when compiling. Tab. 8 shows how one system performs for a combination of different optimization options and levels of unrolling of our particular loop.

Table 8. Unrolling.

optimization	no unroll	unroll=2	unroll=3	unroll=4
no opt.	49.9	33.7	27.7	25.6
some	12.7	11.0	12.2	11.6
more	12.7	10.1	11.1	10.3
full	7.3	7.0	7.1	7.0

As can be seen from the table, it is not easy to improve on the compiler's job in this case. To make a real impact when solving, $\boldsymbol{T}\boldsymbol{x}_j = \boldsymbol{b}_j, j = 1, 2, \ldots, n_{rhs}$, we can compute `q(k) = 1.0 / d(k)` once and for all, replacing the divisions in the second loop with multiplications. This way our code can beat several of the tuned LAPACK-libraries.

Below is an example where unrolling **may** give a significant speedup (more than a factor of six on some systems). On some systems the unrolled code shows no improvement. Note that no "no-aliasing" flag has been used when compiling the codes. The following routine uses Horner's method to evaluate a polynomial of degree four, defined by `coeff`, in `x[j]` storing the results in `px[j]`.

```
void horner(double px[], double x[], double coeff[], int n)
{
  int            j;
  double         xj;

  for (j = 0; j < n; j++) {
    xj    = x[j];
    px[j] = coeff[0] + xj * (coeff[1] + xj * (coeff[2] +
            xj * (coeff[3] + xj * coeff[4])));
  }
}
```

Here is an unrolled version. A cleanup loop is necessary in case eight does not divide `n`. Removing the local variables `c0` to `c4`, using `coeff` instead, gives a marked decrease in performance on several systems. On some systems the introduction of these local variables is the reason for the speedup, but on other systems it is the combination of the local variables and the unrolled loop.

```
void horner(double px[], const double x[],
            const double coeff[], int n)
{
  int            j, rem;
  double         xj, xjp1, xjp2, xjp3, xjp4, xjp5,
                 xjp6, xjp7, c0, c1, c2, c3, c4;
```

```
 /* Local variables, no aliasing. */
  c0 = coeff[0]; c1 = coeff[1]; c2 = coeff[2];
  c3 = coeff[3]; c4 = coeff[4];

  /* cleanup loop if 8 does not divide n */
  if ( rem = (n % 8) ) {  /* not a multiple of eight */
    for (j = n - rem; j < n; j++) {
      xj    = x[j];
      px[j] = c0+xj*(c1+xj*(c2+xj*(c3+xj*c4)));
    }
    n -= rem;
  }

  /* Unroll the loop */
  for (j = 0; j < n; j += 8) {
    xj   = x[j];   xjp1 = x[j+1]; xjp2 = x[j+2];
    xjp3 = x[j+3]; xjp4 = x[j+4]; xjp5 = x[j+5];
    xjp6 = x[j+6]; xjp7 = x[j+7];

    px[j]   = c0+xj  *(c1+xj  *(c2+xj  *(c3+xj  *c4)));
    px[j+1] = c0+xjp1*(c1+xjp1*(c2+xjp1*(c3+xjp1*c4)));
    px[j+2] = c0+xjp2*(c1+xjp2*(c2+xjp2*(c3+xjp2*c4)));
    px[j+3] = c0+xjp3*(c1+xjp3*(c2+xjp3*(c3+xjp3*c4)));
    px[j+4] = c0+xjp4*(c1+xjp4*(c2+xjp4*(c3+xjp4*c4)));
    px[j+5] = c0+xjp5*(c1+xjp5*(c2+xjp5*(c3+xjp5*c4)));
    px[j+6] = c0+xjp6*(c1+xjp6*(c2+xjp6*(c3+xjp6*c4)));
    px[j+7] = c0+xjp7*(c1+xjp7*(c2+xjp7*(c3+xjp7*c4)));
  }
}
```

10 Indirect addressing

Indirect addressing (pointers) is a standard tool when working with data-structures such as storage schemes for sparse matrices and PDE-meshes. Since indirect addressing tends to give bad memory locality we expect poor cache performance.

Consider the following `daxpy`-like routine where `ix` contains a permutation of the numbers one through `n` and `x` and `y` contain `n` elements each.

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

Tab. 9 shows the execution times, on three systems, for calling the routine many times. "random" means that the `ix(k)` are in random order, "ordered" means using pointers but where `ix(k)=k` or `ik(k)=n+1-k` (the two alternatives behave roughly the same way), and "no `ix`" is not using any pointers.

Table 9. The effect of indirect addressing.

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

Once again we see how important it is with locality of reference. Indirect addressing with ordered data does not cause such a performance degradation.

11 If-statements

If-statements in a loop may stall the pipeline in the processor, and should be moved outside the loop if possible. Sometimes it is feasible to rewrite the code so that if-statements are avoided, as in the following example:

```
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, ..., n
  else                            end do
    statements
  end if
end do
```

This can be generalized if there are more special cases in the loop.

If we have a sequence of if-tests, `if-then-elseif ...`, one should try to order the alternatives according to frequency with the most probable alternative first.

In logical tests like, `if (a(k) .and. b(k)) then`, where `a` and `b` are logical arrays, the least likely alternative should be tested first (the most likely in an or-test). According to the Fortran 90 standard it is not necessary for the CPU to evaluate and test all parts of the expression. If `a(k)` is a time-consuming function reference, it is probably better to place it last.

12 Inlining and overloading of operators

It is not uncommon to have short procedures that are called many times in in a loop. Since there is a substantial overhead in calling a routine, one can move the body of the procedure into the loop, so-called **inlining**. On the author's computer a call (inside a loop) of a four-parameter subroutine, doing an immediate return, takes 17 μs.

Inlining can be done by hand, but this reduces the readability of the code and it is error prone due to possible conflicts with variable names. Usually it is possible to ask the compiler or a preprocessor to inline routines, naming specific routines or asking the compiler to make a choice. Some compilers will inline suitable routines provided that the calling and called routine are residing in the same file and that the optimization level is high enough. Some systems provide for inlining-directives in the code.

It does not pay to inline everything, the code will swell and the performance may be degraded. So, only short routines that are called a huge number of times are suitable candidates.

Short functions are often produced when overloading operators. To be explicit, suppose we are using complex arithmetic and assume that multiplication and division are the two dominating operations in our application. By storing complex numbers in polar form, $re^{i\phi}$, instead of in rectangular form, $a + ib$, we will get better performance. Multiplication of two numbers in polar form only requires one multiplication and one addition. It may be argued that less rounding error will result by keeping ϕ in $(0, 2\pi)$ at all times, and this can be solved by an if-statement and one (or two) subtraction(s).

In C++ it is straightforward to construct a new complex type (class) and to bind the arithmetic operations to functions. If `z`, `w` and `q` are variables (objects) of this type, the multiplication- and division-routines would be called in an expression like `z = z * w / q`. The same thing could be accomplished in Fortran 90 using modules, although there is a minor problem since `complex` is a predefined type in Fortran.

We implemented a new complex type in Fortran 90 using overload operators. Tab. 10 shows the times for a huge number of `p = p * v(k)` and `p = p * v(k) / v(j)` in the second column, where `v` is a vector and `p` is a scalar of the new complex type. To make Tab. 10 more readable the times have been normalized so that the minimum time is one. The line for Complex shows the time using the builtin Fortran 90 type. The lines with (if) shows the time when we keep ϕ in $(0, 2\pi)$ (using an if-statement as discussed above).

Table 10. The effect of using inlining.

	Time: *	Time: * and /
No inline (if)	27	56
Inline (if)	6	11
No inline	23	50
Inline	1	9
Complex	7	45

Notice the speedup due to inlining and the degradation caused by the if-statement and the divide operation.

13 Alignment

Consider the following lines of Fortran-code where we allocate 100 001 bytes as a general work area (`integer*1` is non-standard Fortran for one-byte integers). In the call of the subroutine, the address of `work(2)` is passed as a parameter and received as the address of the first element of a double precision array.

```
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
```

If we store the subroutine in a separate file this code will compile (if everything is kept in one file, a good compiler should complain, due to the mix of data types). Several UNIX-systems will give "Bus error" when we try to execute the code. This error is more likely if we ask the compiler to optimize the code. This behaviour is caused by alignment problems, how data is placed relative to byte addresses in memory. 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). Suppose, for example, that a double precision number is allowed two span two cache lines (the beginning of the number is stored in the end of one line and the end of the number is stored in the beginning of the next line). This would force the memory system to load two lines instead of one. So, for efficient memory usage certain alignment rules should (and must, on certain systems) be obeyed. To get the maximum rate of transfer between CPU and memory data should be aligned to fit caches, buses and assembly language instructions (for example double-word load/store instructions). The slowdown caused by misalignment may easily be a factor of 10 or 100. The above example can be written in C as well, in which case the `work`-array may have been allocated using `malloc`.

Several compilers have misalignment-flags (to warn the compiler about misalignment), in which case very conservative loads and stores must be used for data (one byte at a time, for example). There may also be alignment flags telling the compiler that data has a proper alignment. Another potential source of alignment problems is the common-construct in Fortran 77. `common /block/ byte_variable, double(100)` should be reordered as `common /block/ double(100), byte_variable` to give the double-vector a proper alignment. The variables in a common block should be ordered according to the size in bytes, having "large" types (many bytes) first.

14 Closing notes

We have dealt with two basic tuning principles in this chapter. One is to improve the memory access pattern of a program by using caches and virtual memory in an efficient way; locality of reference and data re-use are the key phrases. This may be accomplished by stride minimization, blocking, proper alignment and the avoidance of indirect addressing.

The other principle is to increase the opportunity for making use of the parallel capabilities of the CPU. This can be done by decreasing data dependencies, unrolling loops, inlining short procedures and eliminating if-statements.

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

15 Exercises. Note that the web-pages do not exist.

The exercises below try to illustrate some of the more important points of the chapter. In order to decrease the programming effort some of the exercises are somewhat contrived. It can be interesting to run a program on different systems. Testing different compilers, languages and optimization levels is educational as well. When testing code it is easy to be fooled by a good compiler. Some advice of how to construct good test programs (benchmarks) can be found on this book's homepage, `<http://www.pdc.kth.se/???>`. Analyze your results. The exercises have been constructed with dialects of Fortran and C in mind, but any compiled language should do.

1. You have a three-dimensional array. Which order of access will give the best locality (for the language you are using)?
2. The following code initializes a square matrix. Improve the code.

```
do row = 1, n ! n = order of matrix
  do col = 1, n
     A(row, col) = 1.5 ! off-diagonals
     if ( row == col ) A(row, col) = 10.5 ! diagonal
   end do
 end do
```

3. You have three **separate** loops performing the following operations. What are the relative speeds (explain)? What happens if you change `*` to `+`?

- `y(k) = x(k) * x(k)`, `k` is the loop variable
- `y(k) = x(k) * y(k)`
- `y(k) = x(k) / y(k)`

4. A Householder matrix (common in numerical analysis) is defined by $\boldsymbol{H} = \boldsymbol{I} - 2\boldsymbol{u}\boldsymbol{u}^T/(\boldsymbol{u}^T\boldsymbol{u})$, where $\boldsymbol{u}$ is a column vector with n elements and $\boldsymbol{I}$ is the identity matrix of order n. Write a **fast** Matlab-function, `Hx(u, x)`, that computes $\boldsymbol{y} = \boldsymbol{H}\boldsymbol{x}$.
5. How do the following loops perform (`x` and `s` are scalar double precision variables)?

```
s = 1.0; x = 0.0
do j = 1, many_times
  x = x + 1.0e-6
  s = s + x + x + x
end do
```

```
s = 1.0; x = 0.0
do j = 1, many_times
  x = x + 1.0e-6
  s = s + x * x * x
end do
```

6. Implement a function, `poly(x, c, n)`, that evaluates a polynomial in `x` (scalar variable). `n` is the order of the polynomial and the array `c` contains the coefficients.
   Suppose you have to perform, `y(k) = poly(x(k), c, n)`, **many** times in a loop. Discuss different alternatives to speed things up. Now suppose that you **know** that `n = 3`, always. How much execution time do you save by writing a special `poly_n_eq_3` that is fully unrolled (no loop)?
7. On the book's homepage, `<http://www.pdc.kth.se/???>`, you can find code for the Cholesky factorization (this code is often presented in text books). Compare the speed of this algorithm with `dpotrf` from LAPACK.
8. Check if your C-compiler supports a `restrict` keyword (used to inform the compiler that there is no aliasing in a function). If that is the case, see if you can speed up the example on page 6.
9. Write a program that computes $\boldsymbol{A} = (\boldsymbol{A} + \boldsymbol{A}^T)/2$, where $\boldsymbol{A}$ is a large square matrix. First do it in the three obvious ways (using the Fortran 90 `transpose`-function, then writing row-column- and column-row-oriented do-loops). Finally, try writing a faster routine using blocking.

New versions of some of the books, listed in the references below, have appeared since the first version of this article. Since the author has not read the new versions the references list the old ones with a note about a new version being available.

References

1. White paper: The java hotspot virtual machine, v1.4.1, 2002. `<http://java.sun.com/products/hotspot/docs/whitepaper/Java_Hotspot_v1.4.1/JHS_141_WP_d2a.pdf>`.
2. Homepage of AMD. `<http://www.amd.com>`.

3. E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, 3d edition, 1999. `<http://www.netlib.org/lapack/index.html>`.
4. S. Andersson, R. Bell, J. Hague, H. Holthoff, P. Mayes, J. Nakano, D. Shieh, and J. Tuccillo. *RS/6000 Scientific and Technical Computing: POWER3 Introduction and Tuning Guide*. IBM Corp., 1998. Document Number SG24-5155-00, `<http://www.redbooks.ibm.com>`.
5. J. L. Bentley and J. Bentley. *Programming pearls*. Addison Wesley Publishing Company, 2nd edition, 1999.
6. J. Bilmes, K. Asanović, J. Demmel, D. Lam, and C.W. Chin. Homepage of PHiPAC (Portable High Performance Ansi C for BLAS3 compatible fast matrix matrix multiply). `<http://www.icsi.berkeley.edu/~bilmes/phipac>`.
7. D. Boggs, A. Baktha, J. Hawkins, D.T. Marr, J.A. Miller, P. Roussel, R. Singhal, and K.S Toll, Band Venkatraman. The microarchitecture of the intel® pentium® 4 processor on 90nm technology. *Intel Technology Journal*, 8(1):9–25, 2004. See also *IEEE Computer*, March 1981.
8. D. Cohen. On holy wars and a plea for peace, April 1, 1980. `<ftp://ftp.chalmers.se/rfc/ien/ien137.txt>`.
9. D. Cortesi et al. *Origin2000 (TM) and Onyx2 (TM) Performance Tuning and Optimization Guide*. Silicon Graphics, Inc., 1998. Document Number 007-3430-002, `<http://techpubs.sgi.com>`.
10. I. Crawford and K. Wadleigh. *Software Optimization for High Performance Computing: Creating Faster Applications*. Prentice Hall, 2000.
11. J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *LINPACK User's Guide*. SIAM, Philadelphia, 1979.
12. D. Dougherty and A. Robbins. *sed & awk*. O'Reilly & Associates, Inc., 2nd edition, 1997.
13. K. Dowd and C. Severance. *High Performance Computing*. O'Reilly & Associates, Inc., 2nd edition, 1998.
14. B. S. Garbow, J. M. Boyle, J. J. Dongara, and C. B. Moler. Matrix eigensystem routines-eispack guide extension. In *Lecture Notes in Computer Science*. Springer Verlag, New York, 1977.
15. D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 18(1):5–48, March 1991. Available on many www-sites, try `<http://docs.sun.com>` for example.
16. G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, 3d edition, 1996.
17. L. Gulliver. *Travels into several remote nations of the world*. Benjamin Motte, London, 1726. `<http://www.jaffebros.com/lee/gulliver/index.html>`.
18. F. G. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM Journal of research and development*, 41(6):737–756, 1997. `<http://www.research.ibm.com/journal>`.
19. M. T. Heath. *Scientific Computing - An introductory survey*. McGraw-Hill, 1997. A new edition is available.
20. J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann, 2nd edition, 1995. A new edition is available.
21. IEEE. IEEE standard 754-1985 for binary floating-point arithmetic. *SIGPLAN Notices*, 22(2):9–25, 1987. See also *IEEE Computer*, March 1981.

22. Homepage of Intel. `<http://www.intel.com>`.
23. Intel Corp. *Intel® Pentium® 4 Processor Optimization, Reference manual.* Order Number 248966, `<http://developer.intel.com>`.
24. International Organization for Standardization. *ISO/IEC 1539-1:1997, Information technology–Programming languages–Fortran–Part 1: Base Language*, 1997.
25. Accelerating matlab: the matlab jit-accelerator. *Newsletters - MATLAB Digest*, 10(5), 2002. See `<http://www.mathworks.com/company/newsletters/digest/sept02/accel_matlab.pdf>`.
26. B. Kågström and P. Ling. Homepage of GEMM-based BLAS. `<http://www.netlib.org/blas/gemm_based>`.
27. R. Lauritzsen. Real - java floating point library for midp devices. `<http://gridbug.ods.org/Real.html>`.
28. M. Metcalf and J. Reid. *Fortran 90/95 Explained.* Oxford University Press, 2nd edition, 1999. A new version is available.
29. R. K. Rew, G. P. Davis, S. Emmerson, and H. Davies. *NetCDF User's Guide for C, An Interface for Data Access, Version 3*, 1997. See `<http://www.unidata.ucar.edu/packages/netcdf/>` and the links on this page.
30. P. Seebach. *Unrolling AltiVec, Part 1: Introducing the PowerPC SIMD unit*, 2005. `<ftp://www6.software.ibm.com/software/developer/library/pa-unrollav1.pdf>`.
31. Homepage of SIMD. `<http://www.simdtech.org/home>`.
32. R. A. van de Geijn, J. Gunnels, and Henry E. Quintana-Orti G. Homepage of FLAME (Formal Linear Algebra Methods Environment). `<http://www.cs.utexas.edu/users/flame>`, see also `<http://www.cs.utexas.edu/users/flame/ITXGEMM>`.
33. L. Wall, T. Christiansen, and R. L. Schwartz. *Programming Perl.* O'Reilly & Associates, Inc., 2nd edition, 1996. A new edition is available.
34. R. C. Whaley, A. Petitet, and J. J. Dongarra. Homepage of ATLAS (Automatically Tuned Linear Algebra Software). `<http://math-atlas.sourceforge.net>`.
35. S. Wilson and J. Kesselman. *Java Platform Performance: Strategies and Tactics.* Addison-Wesley Professional, 1st edition, 2000. `<http://java.sun.com/docs/books/performance>`.