

# A COMPARISON OF THE FLOATING-POINT PERFORMANCE OF CURRENT COMPUTERS

Steven H. Langer

Department Editor:

**Paul F. Dubois**

*dubois1@llnl.gov*

A comparison of the floating-point performance of commonly available personal computers and workstations, which will be described in this article, has shown unambiguously that in many situations, today's PCs can compute as well as yesterday's supercomputers. Good computational science can be done today on the desktop.

The performance comparison has been carried out with the help of several floating-point benchmark programs. The results should be interpreted with care. If you are a scientist using one of the computers described in this article, you should remember that the benchmark results are not definitive unless your program does exactly what the benchmark program does. The results do, however, provide insight into the performance of today's computers and help in identifying both the hardware characteristics and the programming practices that lead to good floating-point performance.

Benchmark results are presented for PCs, Apple Macintosh computers, and various Unix workstations. Most of the results are from machines designed for individual users, although there are also some results from high-end multiprocessor computers. High-end machines (supercomputers and massively parallel computers) are frequently purchased to run parallel jobs, whereas all the benchmark programs used in this article utilize single processors. A different benchmark suite is required to assess parallel performance. Whenever possible I have tested the latest available models, but in some instances newer models may have appeared by the time this article is published. The source code for the benchmarks is available at the *Computers in Physics* site on the World Wide Web (<http://www.aip.org/cip>), so that you can test new machines as they come out.

The benchmark results cannot be understood without knowing something about modern computer architecture.

---

*Steven H. Langer is a physicist in the Inertial Confinement Fusion Program at Lawrence Livermore National Laboratory, Livermore, CA 94550. E-mail: [shl@icf.llnl.gov](mailto:shl@icf.llnl.gov)*

The next section provides a brief description of the central processing unit (CPU) and memory system of a typical computer. The third section presents the results of a general-purpose benchmark, the Livermore Fortran Kernels (LFK; run in both Fortran and C, despite the name). The fourth section presents a simple cache-benchmark program that directly shows the effects of the memory system on floating-point performance. An analysis of the cache-benchmark results suggests several ways of writing a program that can lead to better performance. Finally, I summarize the results and emphasize the criteria that are important to consider when buying a desktop computer for scientific applications.

## Modern architecture

Most modern computers have a CPU that is contained on a single chip. This chip contains registers, the integer and floating-point arithmetic units, and often some high-speed cache memory. Most computers also have some cache memory that is not part of the CPU chip. The main memory is comparatively slow and is connected to the CPU by a high-speed bus or crossbar switch.

The Pentium processors from Intel Corp. (and various compatible chips) have a complex-instruction-set computer (CISC) architecture. The Pentium II is the latest member of a family that also includes the Pentium, Pentium MMX, and Pentium Pro. The Pentium has four integer registers, four address registers, and an eight-register floating-point stack, and it uses 32-bit address registers. The top register on the stack is always one of the operands for a floating-point operation, and so it is sometimes necessary to swap registers before an operation can be performed.

All other currently popular microprocessors have a reduced-instruction-set computer (RISC) architecture. Such processors have a large number of general-purpose integer and floating-point registers (often 32 of each), which can be used in any arithmetic instruction. These processors include the PowerPC family used in the Macintosh, IBM Unix workstations, and IBM AS-400 minicomputers; the SPARC chips used in Sun workstations; the PA-RISC chips used in Hewlett-Packard (HP) Unix workstations; the Alpha chip used in Unix workstations and Windows NT personal computers from Digital Equipment Corp. (DEC); and the MIPS chips used in Silicon Graphics Inc. (SGI) workstations. Some versions of all these chips have 64-bit address registers. An interesting



chip that is due to appear in computers in 2000 is the Merced chip from Intel and HP. The Merced chip is fundamentally a RISC chip, but it will also execute Pentium instructions in hardware. One advantage of having a large number of registers on a RISC chip is that they can hold many intermediate results and thus reduce accesses to memory.

Another type of computer familiar to many scientists is the vector supercomputer, such as the Cray Y/MP. The defining feature of these computers is that they can issue a single instruction that operates on all elements of a vector. Modern CPU chips issue one or more floating-point operations per cycle, just like a vector computer. One advantage of a Y/MP is that it can fetch a 64-bit word from memory every clock cycle, even if the words are not next to one another in memory. The Y/MP also has vector gather/scatter hardware that allows it to move a vector at high speed between the CPU and memory locations with irregular separations. Computers based on the Pentium and RISC chips cannot access main memory at irregularly spaced locations quickly. As a result, a Y/MP is better than a workstation at handling irregular memory-access patterns for large arrays. However, the memory system of a Y/MP is extremely expensive, and so the most cost-effective strategy is usually to modify a program to run well with workstation-style memory systems.

The Sun 4/330 workstation I worked with in 1989 had a 25-MHz clock rate. The fastest CPU I use today is a 600-MHz Digital Alpha chip. The performance for floating-point calculations of these two machines is roughly 2 Mflops and 140 Mflops as measured by the LFK, a factor-of-70 improvement. The clock rate has gone up only by a factor of 24—the rest of the improvement was achieved by reducing the number of clock cycles required to complete an instruction and by executing multiple instructions simultaneously.

All the chip designers have faced the same problems in trying to increase performance, and many of their solutions, as reflected in the features of the chips listed above, are quite similar. The biggest problem in recent years is the large disparity in speed between CPUs and the dynamic-random-access-memory (DRAM) chips used for main memory. One clock cycle for a CPU is now roughly 1.5–3 ns long, whereas the access time (the delay between when the CPU supplies the address and when the data are ready) for a DRAM chip is typically 60 ns. This means that the CPU has to wait 20 or more cycles between requesting data from main memory and receiving them. Another problem with DRAM chips is that they have a “recharge time” of around 100 ns after a memory access before they can be read again. DRAM chips do have a capability called “burst access,” however, in which accesses to consecutive addresses require 60 ns for the first read and around 10 ns for each read from the next sequential address.

There is a faster form of memory called static random-access memory (SRAM). SRAM is more expensive, consumes more energy, and has fewer bits per chip than DRAM. Today’s computer designers try to work around the slowness of DRAM by using a limited amount of SRAM memory as a high-speed “cache” of information recently accessed from the much larger main memory. A cache takes advantage of the locality of data access in many programs. A program that reads a word from address  $N$  is likely in the near future to read a word from address  $N + 1$ . Moreover, if a program reads address  $N$ , this address often is used again several times in that portion of the program. As a result, caching the most recently used memory locations in faster memory and fetch-

ing several adjacent memory locations via burst mode can speed average memory-access times. Some programs satisfy 95% or more of their memory requests from the cache and run nearly as fast as if all the computer’s memory was SRAM. Other programs have irregular memory-access patterns and may not run well on a cache-based computer.

As CPUs become ever faster, they demand ever faster effective memory-access times. Today’s computers often have multiple levels of cache. The first level of cache (called L1) is usually located on the CPU chip and

is (currently) limited in size to 8–32 kbytes. The Alpha from DEC has a 96-kbyte secondary (L2) cache that is also on the CPU chip, helping the chip to cope with its very high clock rates. In most computers the L2 cache (L3 on the Alpha) is external to the CPU. Sizes for external caches run from 256 kbytes to 8 Mbytes. Access times get longer for each successive cache level.

Another feature of caches is that they do not store individual bytes of memory. Instead, they store a “cache line” of (typically) 32 consecutive bytes. Consider a hypothetical computer with no on-chip cache, a 32-byte cache line, 256 kbytes of external “direct-mapped” cache, and 32-bit addresses. The 5 least significant bits of an address select a location within a cache line, the next 13 bits select a specific cache line, and all addresses that differ only in the upper 14 bits map to the same cache location. Each cache line has “tag” bits that store the 14 upper bits of the address currently being cached there. When the CPU accesses memory, the hardware uses 13 bits of the address to select a specific cache line. It then compares the upper 14 bits of the requested address with the tag bits for the cache line. If there is a match, the request is satisfied from the cache, and the CPU quickly has the data it needs. If there is a mismatch, the cache line containing the desired address is read from DRAM using burst mode, and the tag bits for the cache line are changed to match the new location. This is much slower than if there was a cache “hit,” but if the CPU then reads the next consecutive memory location, it will be ready in the cache.

---

*As CPUs  
become ever faster,  
they demand ever faster  
effective memory-  
access times.*

---

One other potential problem with either cache or main memory is that neither one may be able to supply data as fast as the CPU can process them. Some computers use bus widths

of 128 bits to obtain a high bandwidth to memory without having to go to an extremely fast bus clock.

The computers on which the benchmarks were run are shown in Table I. The first column shows the model name, and the second column shows the operating system (in some cases the same kind of computer can run two different operating systems). The machines are grouped by the kind of CPU chip. The final column lists cache sizes. If there is only one number, it is for an external cache. If there are two numbers, the first is for the

on-chip cache and the second is for the external cache. If there are three numbers, the first two are for the two on-chip caches and the third is for the external cache. The main memory size is not shown, but in all cases it was large compared to the size of the data used in the benchmark.

## Livermore Fortran Kernels

Frank McMahon wrote the first version of the LFK,<sup>1</sup> or the Livermore Loops, benchmark program in 1970. This benchmark consists of 24 loops extracted from physics simulation codes written at Lawrence Livermore National Laboratory. Each kernel is run many times inside a timed loop, and the results are used to compute a floating-point performance rating for each kernel. The benchmark program reports a number of statistics describing the run. Our experience indicates that the geometric mean of the speeds in megaflops of the 24 kernels is the best single number for characterizing a computer, and it is the only result reported in this paper. However, the output from a run contains much other useful information and is worth studying. Measured geometric-mean LFK performance has correlated well with computational speeds on our simulation codes, provided the codes use cache efficiently.

I have chosen to use the version of the LFK with 8-byte floating-point numbers (double precision on all but Cray computers) because that is the precision required in many physics simulation codes. The performance in single precision is usually only 10–20% faster than in double precision for the LFK. Single precision is helpful when trying to fit many variables into the available memory and reduces the required memory bandwidth for problems that do not fit in the cache. One nice feature of the LFK benchmark is that the 24 kernels are quite different from one another, and so compiler updates that improve the benchmark results usually improve the performance on real programs as well.

The results of the LFK measurements are shown in Table II. The benchmark was run in both a Fortran and an ANSI C version. The rates for the Fortran and C versions are close on most computers, with the Fortran version generally being faster. The two languages are not always this close—in many more-complex programs, C is significantly slower. The problem is often that the C compiler cannot decide whether two iterations of a loop modify the same storage location. To guarantee code correctness, the C compiler makes sure that one iteration of the loop completes before allowing the next loop to start. It is not possible to keep the floating-point pipeline full with this type of code. The Fortran compiler is allowed, by the terms of the ANSI standard, to assume that the memory used by array arguments does not overlap, and so it generates code that keeps the pipeline busy by working on more than one loop iteration at a time. This problem with C compil-

*The LFK,  
or Livermore  
Loops, bench-  
mark consists of  
24 loops extract-  
ed from physics  
simulation  
codes.*

**Table 1. Computer characteristics.**

Model	Operating System	CPU	Clock (MHz)	L1/L2/L3 Cache (kbytes)
DEC 433au	Digital Unix	EV 5.6	433	8/96/2048
DEC 500au NT	Alpha NT	EV 5.6	500	8/96/2048
DEC 8400/600	Digital Unix 4.0d	EV 5.6	600	8/96/4096
Sun Ultra 10	Solaris 2.5.1	Ultra Sparc III	300	16/512
Sun Ultra 5	Solaris 2.5.1	Ultra Sparc III	270	16/256
Sun Ultra 2	Solaris 2.5	Ultra Sparc II	296	16/1024
HP J 282	HP Unix 10.20	PA-RISC 8000	180	1024
HP C200	HP Unix 10.20	PA-RISC 8200	200	1024
SGI Octane	Irix 6.4	R10000	195	32/1024
SGI O2	Irix 6.3	R5000	200	32/1024
IBM ThinkPad 760XD	Windows 95	Pentium MMX	166	32/256
HP Vectra XU 6/150	Windows NT 4.0	Pentium Pro	150	8/256
Dell XPS Pro 180n	Windows NT 4.0	Pentium Pro	180	8/256
Dell XPS H266	Linux 2.0.30	Pentium II	266	16/512
HP Kayak XW	Windows NT 4.0	Pentium II	300	16/512
PowerMac 9500	MacOS 8	PPC 604e	180	32/512
PowerMac 7300	MacOS 8	PPC 604e	200	32/256
PowerMac G3 266	MacOS 8	PPC 750	266	32/512
IBM SP2	AIX 4.2	PPC 604e	332	32/256

## Feedback Lightning and the Snake

Recently I have been trying to produce a facility that makes it easier to extend Python with C++. I'll get to that work below, but first I should mention Todd Veldhuizen's GNU-licensed matrix/vector class library for C++, which he calls "Blitz++." At the bottom of these remarks are some notes about programming in Fortran 9X.

### Blitz++

Blitz++ uses the expression-template technologies previously discussed in several articles in *Scientific Programming*, most recently in the May/June issue (CIP 12:3, 1998, p. 259). The Web page for Blitz++ is <http://seurat.uwaterloo.ca/blitz/>. The latest benchmark results are that the median performance of Blitz++ over 23 Livermore loop kernels on the Cray T3E is 95–98% of that of Fortran 77/90. Compilers are now catching up to this technology, and Todd says that Blitz++ compiles reliably under recent snapshots of EGCS, the new free GNU C++ compiler. See <http://egcs.cygnus.com/> and also the platform notes at <http://seurat.uwaterloo.ca/blitz/platforms/>.

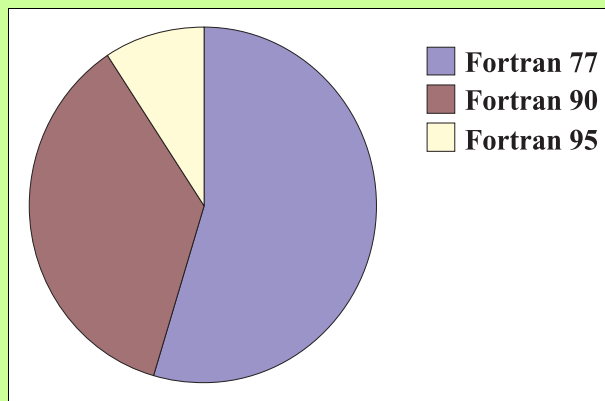
### Python/C++

Normally, Python would be extended with C, and although this is relatively easy to do, the opportunity of making mistakes, particularly in keeping the correct reference counts for the Python objects, is painfully large. Such bugs are hard to find. To do everything correctly takes a ton of defensive coding, which leads to programs that are hard to read and maintain. Such a situation is just what C++ was made for. I've written a new facility for writing Python extensions in C++ called "CXX." The documentation is at <http://xfiles.llnl.gov>. I've submitted a paper on it for the Seventh International Python Conference (November 9–13, Houston; see <http://www.python.org>).

The technical highlight of the work was being able to use the "Curiously Recursive Template Pattern" (CRTP). A class inherits from some class templated on itself:

```
class MyClass: public SomeBase<MyClass> {...}
```

It doesn't seem possible that this is legal C++, but it is. The CRTP allows you to "reuse" code that depends on the class for which it is to be written. In my case, a user just makes MyClass inherit from PythonExtension<MyClass> in order to make MyClass a Python extension, and then makes some calls to set behaviors. All the technically difficult part is hidden in class PythonExtension; but I am unable to write PythonExtension for the user without knowing the class for which I am writing it. The CRTP solves this dilemma.



Comparison of time and effort required to learn various versions of Fortran, starting with Fortran 77.

### Fortran 95 lecture series

Beginning in April I gave a series of four lectures at Lawrence Livermore National Laboratory entitled "Large-Scale Scientific Simulations with Fortran 95: An Object-Based Approach." I was inspired by the article in this column written by Mark Gray and Randy Roberts from Los Alamos, and by papers graciously sent to me by Viktor Decyk (UCLA) and Charles Norton (JPL). My goal was to offer a "field upgrade" to our working scientists and engineers. You can find the course materials at <ftp://ftp-icf.llnl.gov/OBF90>, including the slides and example code.

I confess freely that I once (in a fit of object-oriented passion) replied to a question at a seminar that I had no intention of learning Fortran 90. Ahem. Wrong answer. The figure shows how surprisingly large a change from Fortran 77 is involved. Yet, the main new ideas are easy to learn and easy to apply to existing code. (Side note: the nature of the changes from Fortran 90 to Fortran 95 are relatively minor from a compiler writer's point of view, and so I don't expect much delay in seeing compilers catch up to the latest standard.)

This diagram is adapted from Digital Equipment Corp.'s *Fortran Language Reference Manual*, which I obtained when I got Digital's Visual Fortran environment for Windows 95/NT. This is an integrated development environment that includes the editor, compiler, debugger, browser, and documentation. It is a wonderful way to learn Fortran 95.

**Paul F. Dubois**

ers is referred to as "pointer aliasing." If a C program is performing poorly compared to a similar program in Fortran, a good first step is to try a compiler flag that tells the compiler to ignore the possibility of pointer aliasing. This problem does not arise in the C version of the LFK because it stores arrays

in global variables, and the compiler knows that these variables do not overlap.

Among the computers we tested, DEC, HP, SGI, and IBM all have models with performance levels that exceed 100 Mflops (see Table II). To put this in perspective, a Cray Y/MP

produces 40 Mflops for the LFK geometric mean. A 300-MHz Pentium II achieves over 35% of the speed of a 600-MHz DEC Alpha and is 50% faster than a Y/MP. This means that you do not need an expensive computer to get good performance for single-processor jobs. It should be noted, however, that because the LFK loops use cache effectively, they run well on cache-based computers. A program that does not use cache efficiently can be much faster on a Y/MP due to the fast (and expensive) main memory in the Y/MP.

Many programs are able to take advantage of multiple processors. One advantage of Unix computers over PCs and Macs is that all the major workstation vendors have machines with eight or more CPUs in a single computer, and these machines run parallel jobs much faster than the fastest PC.

The LFK results generally show that performance of computers using the same CPU chip is directly proportional to the speed of the CPU (provided that the compiler is the same). Much larger differences would be seen in programs that do not use cache efficiently, because there are often large differences in memory bandwidth between inexpensive and expensive systems using the same chip.

The choices of compiler and compiler flags are important. An example of the effect of a compiler on performance can be found in the 109 Mflops delivered by the 332-MHz PowerPC 604e in the IBM SP2 versus the 36 Mflops delivered by the Absoft compiler on a Macintosh with the 200-MHz version of the same chip. The IBM compiler is 80% faster after adjusting for the clock rate. Setting the compiler switch that creates code for the specific CPU chip in a system instead of building a program that can run on all computers made by the vendor often produces faster code. Subtle compiler switches can also make a big difference. Telling gcc to align double-precision numbers on 8-byte boundaries, instead of the default 4 bytes, doubles the rate for some LFK kernels on the Pentium II. The rates reported in this paper use favorable settings of all the compiler switches known to make a significant difference in floating-point performance.

## Cache benchmark

To explore the effects of the memory system on floating-point performance, a cache-benchmark suite of programs has been developed. The main portion of this suite contains four different versions of a dot product. The first version, when written in C, looks like this:

```
for(i= 0, sum= 0.0; i < n; i++) sum += x[i]*y[i];
```

The remaining three versions all access the array indirectly (this is typical behavior, for example, for a hydrodynamics code with an arbitrarily connected grid). The indirect dot product looks like this:

```
for(i= 0, sum= 0.0; i < n; i++) sum += x[ndx[i]]*y[ndx[i]];
```

In the first indirect-dot-product case, `ndx[i]` is set to `i` and the pattern of memory access is identical to that for the direct dot product (except for the need to read the index array). In the second case, the `ndx` array is randomly sorted, and in the third case the `ndx` array is randomly sorted in chunks of eight consecutive double-precision numbers. The performance for these three dot-product loops is equal when everything fits in

**Table II. Floating-point performance of computers running Fortran (F77) and C versions of the LFK benchmark program.<sup>a</sup>**

Model	Measured geometric-mean LFK performance in megaflops	
	F77 version	C version
DEC 433au	103.69	77.53
DEC 500au NT	113.03	94.36
DEC 8400/600	158.05	125.64
Sun Ultra 10	68.44	53.25
Sun Ultra 5	49.71	43.46
Sun Ultra 2	76.08	54.71
HP J 282	89.83	94.38
HP C200	99.27	107.38
SGI Octane	100.25	81.68
SGI O2	46.27	34.48
IBM ThinkPad 760XD	10.44	11.35
HP Vectra XU 6/150	29.98	27.79
Dell XPS Pro 180n	34.1	28.99
Dell XPS H266	46.86	48.83
HP Kayak XW	58.65	60.93
PowerMac 9500	34.85	22.61
PowerMac 7300	35.69	27.18
PowerMac G3 266	52.13	38.07
IBM SP2	108.9	93.09

*a. The rates in this table should be repeatable to a few percent. In some cases the machines were idle except for the benchmark run, but in other cases there were additional processes running on the computer. One of the rules for running the benchmark is that all kernels in the LFK must be compiled with the same compiler flags. Compiler flags were set to achieve the highest performance, consistent with correct results, that could be obtained by testing a reasonable number of performance flags.*

the on-chip cache. The random-access loop performs poorly when the data set is larger than the cache, whereas the random-access-in-chunks loop has nearly the same performance as the indirect loop. The random dot product uses only 8 bytes from each cache line it fetches, whereas the chunked loop uses more of each cache line and thus takes advantage of burst fetches from memory.

The final portion of the cache-benchmark suite evaluates a fifth-degree polynomial for all elements of a vector. In the case of the dot product, two double-precision numbers (and an element of the index array when needed) are read from memory for every individual multiply and add. Even the on-chip cache may not be fast enough to keep up with a calculation that does so little arithmetic for each word fetched from memory. The polynomial evaluation, however, requires one read and one write of a double-precision number per five multiplies and five adds. The result is that the polynomial evaluation does more floating-point operations per second than the dot product because it does not spend as much time waiting for memory.



Table III lists cache-benchmark performance results for both C and Fortran. Only the minimum and maximum rates for the direct dot product and the polynomial evaluation are included in Table III; the complete results are available at the *Computers in Physics* Web site. For all computers except the DEC workstations, the results for C and Fortran dot products are essentially the same. The dot-product benchmark is, as indicated above, dominated by memory bandwidth for all sizes, yielding performance rates that are always well below the maximum rate at which the chip could issue instructions. The measured performance for the polynomial evaluation is always higher than for the dot product, but only slightly so on the Pentium-based machines. The Pentium has only eight floating-point registers, and so it is unable to hold enough intermediate results in registers to keep the CPU busy.

In a number of cases the Fortran rate for polynomial evaluation is much higher than the C rate. This may be because the C compiler was not willing to operate on more than one element of the array at a time. (The flag telling the compiler to assume that arrays are not overlapping was not explicitly set in the Makefile. However, compilers performing high levels of optimization sometimes make this assumption.) In the case of the DEC 433au, the theoretical maximum

rate is 866 Mflops, and so the Fortran compiler did a very good job. The HP, SGI, and IBM Fortran compilers also did a good job. The ratio between the maximum and minimum rates for the Fortran polynomial evaluation is between 3 and 5 for all machines except those based on the Pentium (the floating-point unit in the Pentium Pro and II is much better than the one in the Pentium).

Figure 1 shows the results of a run of the cache benchmark on a DEC 433au, a computer with a 433-MHz Alpha CPU chip. This chip has an 8-kbyte and a 96-kbyte data cache on the CPU chip and an additional 2-Mbyte external cache. The drop in speed for the direct dot product at a vector size of approximately 500 is due to the difference in speed between the 8-kbyte and 96-kbyte on-chip caches. The slowdown at roughly 6000 happens when data must be loaded from the external cache. The slowdown at about 120,000 happens when the data do not fit in the 2-Mbyte external cache. The rate for the indirect dot product is less than half the rate for the direct dot product until the vector no longer fits in cache.

On most of the other computers, the rates for direct and indirect dot products are very similar when they fit in the on-chip cache. For the largest vectors, the ratio is roughly 3:2

**Table III. Floating-point performance of computers running Fortran (F77) and C versions of the cache-benchmark suite of programs. Individual columns list the minimum (min) and maximum (max) performance on the direct-dot-product and polynomial portions of the suite.**

Model	Measured cache-benchmark performance in megaflops							
	C version				F77 version			
	direct dot product		polynomial		direct dot product		polynomial	
	max	min	max	min	max	min	max	min
DEC 433au	157.14	25.11	345.60	111.63	360.01	36.09	687.25	128.13
DEC 500au NT	224.20	28.55	394.76	117.06	163.52	38.73	405.16	142.42
DEC 8400/600	230.01	29.49	507.73	107.91	518.10	30.38	980.59	103.77
Sun Ultra 10	193.80	28.65	99.55	61.66	195.99	29.10	335.02	112.14
Sun Ultra 5	174.03	25.65	89.55	53.96	176.07	25.80	302.66	95.54
Sun Ultra 2	191.61	30.99	98.47	62.34	191.97	32.39	334.13	119.73
HP J282	115.76	30.80	333.22	85.45	119.79	33.03	476.81	107.76
HP C200	133.32	37.38	376.19	104.72	133.40	37.51	532.17	103.74
SGI Octane	194.03	43.93	268.22	111.17	194.6	41.63	359.18	121.02
SGI O2	78.73	10.24	68.61	31.34	79.27	10.13	198.87	42.13
IBM ThinkPad 760XD	12.63	4.56	18.19	12.55	28.01	5.02	22.13	12.48
HP Vectra XU 6/150	83.46	17.13	72.55	38.12	112.97	18.45	77.96	45.12
Dell XPS Pro 180n	98.85	16.65	86.57	46.09	138.89	15.74	93.94	45.78
Dell XPS H266	146.71	17.45	158.09	53.57	150.82	17.55	168.07	53.21
HP Kayak XW	168.16	42.58	147.58	62.62	164.36	43.98	156.91	64.00
PowerMac 9500	116.81	8.97	134.77	34.00	116.81	9.46	190.05	31.73
PowerMac 7300	130.81	11.35	151.80	36.36	130.64	10.91	211.64	35.82
PowerMac G3 266	117.69	18.40	147.31	76.45	111.47	17.72	197.34	71.64
IBM SP2	219.80	21.76	254.97	106.05	219.58	23.07	456.69	106.54

(the ratio of variables fetched per loop iteration).

All three indirect loops execute at the same rate when the loops are small enough to fit in the on-chip caches. The indirect dot product with random indexing suffers a large performance drop when the data spill over into the external cache, whereas the chunked random calculation does not slow as much (it uses most of each cache line).

These results show that a program runs fastest when the data it needs are stored contiguously in memory. Sometimes changing the order in which loops are nested can facilitate the accessing of consecutive memory locations. In cases in which the variables needed in a calculation are spread around memory, execution of a program can often be sped up by gathering the variables into contiguous memory, performing the update, and then spreading the results back to their scattered locations.

The minimum performance numbers for the DEC 433au under Fortran exceed those for the DEC 8400/622, whereas the maximum performance numbers are smaller, scaling approximately with the clock ratio of 433:622. The 8400 is an eight-processor computer with a shared main-memory bus. Even though the bus on the 8400 is very fast, it has to share access among eight CPUs. The net result is that the 433au has a higher effective memory-transfer rate (given the load on the 8400 at the time that these tests were run) and thus is faster for large vectors.

The rate for the polynomial evaluation starts out higher than that for the dot product and remains quite high even for

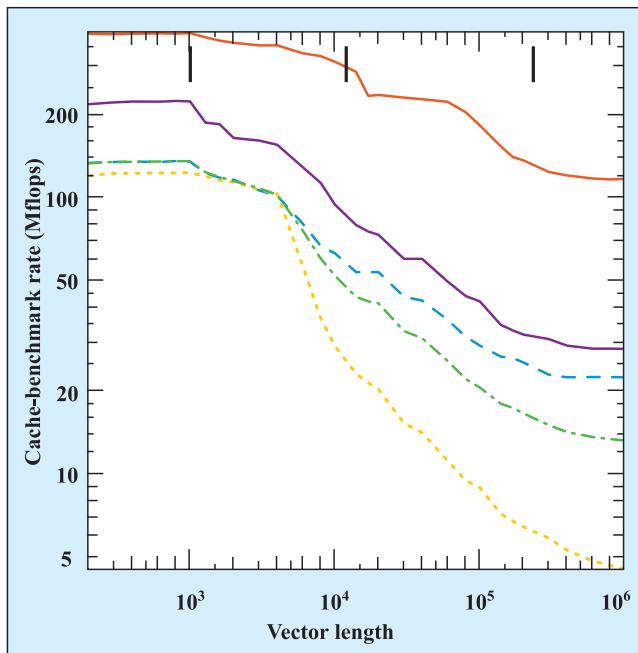


Figure 1. Results of the cache benchmark for a Digital 433au workstation. The solid orange curve is the rate for polynomial evaluation, the solid purple curve is for a direct dot product, the dashed curve is for an indirect dot product in sequential order, the dotted curve is for an indirect dot product done in random order, and the dot-dashed curve is for an indirect dot product with chunks done in random order. The short vertical lines show approximate locations where the data for the polynomial evaluation will no longer fit in the 8-kbyte on-chip cache, the 96-kbyte on-chip cache, and the 2-Mbyte external cache.

the largest vectors. The rate for polynomial evaluation of large vectors is three to four times the rate for evaluation of a dot product on the Unix workstations. A polynomial evaluation requires five times as many floating-point operations per memory access as a dot product, which means that the memory accesses per second are similar for both. The results for the largest vectors are controlled mostly by the rate at which data can be moved to and from main memory.

A cache-benchmark comparison of the Power Mac 7300 and the Power Mac G3 demonstrates that both the CPU chip and the memory system have an effect on floating-point performance. For short vectors, the 7300 is faster because the 604e chip is faster for floating-point computations than the 750 chip at the same clock speed. For long vectors, the G3 is faster because it can read numbers from main memory faster.

The almost tenfold change in rate between the smallest and largest vector sizes for the direct dot product on the DEC is much larger than the observed difference on the LFK benchmark between the 600-MHz Alpha and a 300-MHz Pentium II. This means that making effective use of the cache can be more important than the choice of computer. There are often ways of changing the algorithm used to solve a problem so that it uses the cache more efficiently. An example is the *dgefa/dgesl* routine in LINPACK,<sup>2</sup> which uses Gaussian elimination to solve a system of linear equations.

Figure 2 shows the performance rating for a double-precision matrix solve on a Pentium Pro computer as a function of matrix size. When the matrix is larger than roughly  $180 \times 180$ , it no longer fits in the 256-kbyte cache and the LINPACK performance drops 50%. The solid curve shows the performance for the *dgesv* routine in LAPACK.<sup>3</sup> This routine uses Gaussian elimination, but it performs the calculations in an order that results in good cache utilization. The LAPACK performance improves significantly as the matrix becomes larger.

### Faster computers present new choices

The gap in CPU performance between inexpensive personal computers and high-end workstations has largely disappeared. A Pentium II is faster than the Cray Y/MP supercomputer on the geometric mean of the LFK. Today any scientist can run on a desktop (single-CPU) computer programs that only a few years ago would have required access to a major computer center.

Achieving this performance does, however, place a burden on the programmer. A high-performance program must be written so that it allows the cache to mask the intrinsic slowness of main memory. In many cases this can be achieved through careful data layout and proper nesting of loops. In other cases it may require the development of new algorithms. When it is not possible to make a program “cache friendly,” using a Unix workstation with a fast memory system can help performance.

The benchmarks presented in this paper should help users to choose a computer and understand the characteristics affecting its performance. Floating-point performance is not the only thing to consider in buying a computer. A PC such as the HP XW Kayak is more expensive than many Pentium II machines, but it can hold two CPU chips, has high-speed disks, and can be equipped with fast 3D-graphics cards.

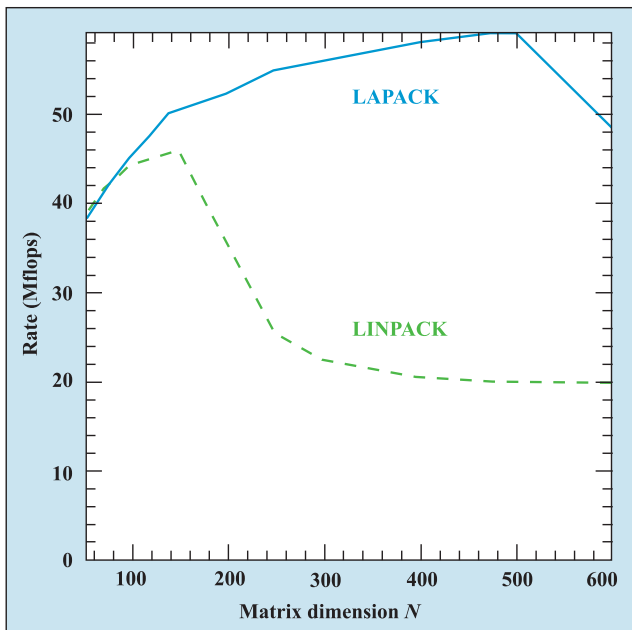


Figure 2. Rate at which a double-precision  $N \times N$  linear system was solved on a Pentium Pro computer as a function of matrix size. Dashed curve is obtained with the LINPACK Gaussian elimination solver, whereas the solid curve is obtained with the equivalent LAPACK solver. Because LAPACK uses cache memory efficiently, it performs much better for large matrices.

Another potentially critical feature is the maximum amount of memory that a computer can hold. Support and compatibility are also important. If two Unix workstations have similar performance, it makes sense to buy from the vendor whose machines you already own. Once you have chosen an operating system and type of CPU, look carefully at the size of the external cache and the speed of the main memory bus—they have a significant effect on floating-point performance. The capabilities of PCs and workstations have improved so much over the past few years that there is little need to compromise on floating-point performance just so that you can get the other features you need.

### Acknowledgements

I would like to thank Hewlett-Packard, Silicon Graphics, Digital Equipment, and Sun Microsystems for providing access to their latest PCs and/or workstations. The views represented are those of the authors and do not represent those of Lawrence Livermore National Laboratory, the University of California, or the United States government. This work was performed under the auspices of the U.S. Department of Energy by the Lawrence Livermore National Laboratory under Contract No. W-7405-ENG-48.

### References

1. Frank H. McMahon, "The Livermore Fortran Kernels," UCRL-53475, 1986.
2. J. J. Dongarra, C. B. Moler, J. R. Bunch and G. W. Stewart, *The LINPACK User's Guide* (SIAM, Philadelphia, PA, 1979).
3. E. Anderson, *The LAPACK User's Guide* (SIAM, Philadelphia, PA, 1992).