USI Report No. 3

High-Performance Matrix Multiplication

Nelson H. F. Beebe
Center for Scientific Computing
Department of Mathematics
220 South Physics Building
University of Utah
Salt Lake City, UT 84112
USA

Tel: (801) 581-5254
FAX: (801) 581-4148

Internet: Beebe@science.utah.edu

29 November 1990

Abstract

This document describes techniques for speeding up matrix multiplication on some high-performance computer architectures, including the IBM RS-6000, the IBM 3090/600S-VF, the MIPS RC3240 and RC6280, the Stardent 3040, and the Sun SPARCstation. The methods illustrate general principles that can be applied to the inner loops of scientific code.

Contents

1 Introduction

This report is derived from material in a much larger document [2] describing the results of a series of matrix multiplication benchmarks, written in Fortran and C, that have been carried out by the author on a variety of machines of interest to the University of Utah campus computing community.

That document supplies many interesting comparisons between commonly-available computing systems, and it also contains several surprises.

It is the purpose of this report to discuss in detail some conclusions that have been drawn from the benchmarking activity. Examples exist where rather straightforward rearrangements of program code can change performance by over a factor of twenty.

This suggests that *many scientific programs ported to some newer high-performance machines may perform far below their potential*. Achieving high performance requires an understanding of the underlying hardware architecture, and how to exploit it in programs written in high-level languages, like C and Fortran.

The tables in this report have been derived automatically from the benchmark output files using the UNIX `awk` utility [1]. Most of them do not appear in [2], although the data presented in them could be (laboriously) deduced from the extensive tables presented there.

2 Matrix multiplication as a benchmark

Matrix multiplication was chosen as a convenient benchmark of machine performance for several reasons. It is easily scaled for a wide range of performance, because the work grows like N^3 for matrices of order N. There are three nested loops in the code; the inner loop is short, consisting, in the simplest implementation, of a single multiply and add. With care, the loops can be parallelized or vectorized to achieve high performance on many machines. Versions of the benchmark can be prepared for different data types (the large report uses short and long integers, and short and long floating-point precision), and in different languages.

Comparisons between performance figures for *different data types* on the same machine prove interesting; on some of these machines, long floating-point precision is fastest by far. Some RISC architectures lack integer multiply hardware, and such machines may accordingly exhibit unusually low integer performance.

Comparisons between *programming languages* on the same machine may be beneficial in selecting a target language for implementation of new software. On some machines, C code is significantly faster, and on others, Fortran wins.

3 The matrix multiplication algorithm

Matrix multiplication, usually written compactly as $\boldsymbol{A} = \boldsymbol{BC}$, in component form is defined by

$$A_{ij} = \sum_{k=1}^{N} B_{ik} C_{kj} \qquad (i = 1 \ldots N, \quad j = 1 \ldots N)$$

This translates straightforwardly into Fortran, but SFTRAN3 [10] structured Fortran syntax will be used here to avoid the need for ugly statement labels:

```
DO FOR J = 1,N
     DO FOR I = 1,N
          DO FOR K = 1,N
               A(I, J) = A(I, J) + B(I, K)*C(K, J)
          END FOR
     END FOR
END FOR
```

Note, however, that this is not the only way to compute the matrix product; the mathematical form says nothing whatever about the order in which the A_{ij} elements are to be computed. Since all such elements are independent of one another, in the best case, they could be computed in parallel in one multiply-add time by N^2 processors. The order of the three loops does not matter mathematically, but it does affect performance; see the discussion of virtual memory faults below.

The legacy of Fortran loops unfortunately leads the non-expert programmer into thought patterns of sequential one-element-at-a-time processing, resulting in programs that are far from optimal on some architectures. We shall see later that performance boosts may require computing several elements at once, and this can be done even on scalar machines.

4 Can matrix multiplication be speeded up?

As we have written it, the matrix multiplication algorithm has complexity $O(N^3)$,[1] because N^2 product elements must be computed, each from a dot product of N-element vectors.

There is a class of matrix multiplication algorithms, known by the names of Strassen, Pan, and others [9, 16], whose complexity grows by smaller powers of N; the best result known by 1985 was $O(N^{2.496})$. While at first sight this is a very attractive result, these methods have remained largely of theoretical interest. They are quite difficult to implement, they are much more susceptible to virtual memory paging problems, and the constant multiplying the power of N is quite large, so that in practice, they do not become competitive with the standard $O(N^3)$ method until the matrices are very large.

There is also a variant of the standard method, due to Winograd [18]. It trades half the multiplications for the same number of additions, as follows:

$$A_{ij} = \sum_{k=1}^{N/2}(B_{i,2k-1} + C_{2k,j})(B_{i,2k} + C_{2k-1,j}) - (\xi_i + \eta_j) + ((1-(-1)^N)/2)B_{iN}C_{Nj}$$

$$\xi_i = \sum_{k=1}^{N/2} B_{i,2k-1}B_{i,2k}$$

[1] $O(n)$ means *of the order of n.*

$$\eta_j = \sum_{k=1}^{N/2} C_{2k-1,j} C_{2k,j}$$

Unfortunately, the method is numerically unstable if $\|B_{i,2k-1}/C_{2k,j}\|$ is much larger, or much smaller, than one. Also, the tendency on modern machines is to make floating-point addition and multiplication execute in one clock cycle, so the savings from this method are not expected to be large.

In this section, we shall therefore restrict ourselves to the programming problem of improving the performance of matrix multiplication by code modifications that implement the standard $O(N^3)$ algorithm.

Matrix multiplication benchmarks tend to *overemphasize* performance, because they are compute bound, I/O free, and have a simple inner loop that most compilers can generate good code for.

Nevertheless, on some architectures, it is possible to significantly increase the speed of a matrix multiplication, without having to resort to assembly language coding. Why is this the case? The answer lies in an understanding of some general features of modern computer architectures, and some details of the specific machine that code is being optimized for.

Modern architectures have a *memory hierarchy*:

- At the lowest level, computations are usually done in *registers*, or between registers and memory. Only a few architectures, such as the DEC VAX, support memory-to-memory operations.

  Registers can be thought of as a form of very fast, but small, memory; indeed, on the DEC-20, they are mapped into the first few words of memory, but implemented in faster hardware.

  Registers may be general purpose, or dedicated to certain instructions (notably in the Intel 80xxx family), and there may or may not be separate sets of registers for integer and floating-point computations, and for different processes.

  The Intel 80xxx CPUs have 8 80-bit floating-point registers, but only one 32-bit integer register can be used for arithmetic.

  The MIPS R3000 has 32 32-bit integer and 16 64-bit floating-point registers.

  The IBM RS/6000 has 32 32-bit integer and 32 64-bit floating-point registers.

  The IBM 360 architecture (including the 3090) has 16 scalar 32-bit general registers used for integer arithmetic, and 8 32-bit floating-point registers (giving 4 pairs of 64-bit registers). The vector facility adds 8 vector registers, each with 256 64-bit elements.

  The Sun SPARC has 24 32-bit integer registers (8 for input arguments, 8 for locals, and 8 for called routine arguments), and 32 32-bit (or 16 64-bit) floating-point registers, for a single routine. However, the architecture supports up to 520 registers that are used to form overlapping *register windows* in nested calls; registers only need to be saved when no more register windows are available. Current implementations of the SPARC architecture support 7 or 8 windows.

  Registers can be accessed by the CPU in less than one machine cycle.

- *Cache memory* is at the next level. It is conceptually similar to normal memory, but is implemented in faster (and more expensive) hardware. Machines that have caches load registers from the cache, which in turn references main memory. Caches vary in size from 8 bytes to 256KB.[2] A cache may hold both instructions and data, or there may be separate caches for each. On most architectures, a process context swap flushes the cache.

  Cache access typically takes few (less than 10) machine cycles; the fastest machines, such as the IBM 3090, can fetch a `DOUBLE PRECISION` number from cache memory in one cycle.

  Cache memory is transparent to the programmer, and to most of the operating system. No machine instructions are available to the user to manipulate the cache, or even to determine its size.

- *Random-access memory* (RAM), or *central memory*, is still called *core memory* by old-timers. It holds the working storage of programs, and on most machines (other than the smaller members of the Intel 80xxx family), is a linearly addressable array. Addressing is usually by byte, but some older machines, and Cray supercomputers, use word addressing.

  Older machines usually had memories limited to at most a few hundred thousand words; the IBM 360 introduced byte addressing and a 24-bit (16MB) address space. The IBM 370-XA extended architecture increased this to 31-bit addressing (2GB). Most machines developed since the mid-1970s have used 32-bit addressing (4GB), but the IBM RS/6000 uses 52-bit addressing (4.5×10^{15} bytes); that appears to be the largest address space of any current machine.

  The increased address space of the IBM RS/6000 is certainly timely, because with memory prices falling below US$40 per megabyte in some markets in late 1990, 4GB of memory might only cost about US$160 000; that is small compared to the costs of the largest machines today.[3]

  Current memory price trends seem likely to continue, so it is reasonable to predict that by the end of this decade, such an amount of memory may cost no more than a workstation does today.

  Access time of a single byte is typically of the order of 80 nsec to 150 nsec; interleaving of consecutive bytes in separate memory banks allows several bytes to be fetched in essentially constant time.

  RAM access may take from 8 to 25 times longer than register access.

- *External storage* (slow memory, disk, tape, …) provides potentially unlimited storage for computer programs, at the expense of special programming to access it. Access times are often in milliseconds, and benchmarked transfer rates on the machines included in this collection range from about 25 KB/sec to about 5 MB/sec.

[2] 1 byte = 8 bits, 1KB = 1024 (2^{10}) bytes, 1MB = 1 048 576 (2^{20}) bytes.

[3] The actual cost of memory today for the IBM RS/6000 is far higher, because of an uncommon memory chip configuration, and lack of third-party competition. At present, IBM is a sole supplier, and charges about US$600 per megabyte.

> With the exception of personal computers, and the Cray supercomputer, almost all modern machines support the notion of *virtual memory*; physical central memory may be quite limited, but to the programmer, it appears as if the entire address space is available. Pages of memory are swapped in and out to external storage as needed.

This four-level memory hierarchy has a significant effect on performance; data access should be restricted to the lowest levels possible.

Ideally, one would like the entire computation of inner loops to reside in registers, with no memory accesses, no subroutine calls (they can invalidate the cache, causing it to be flushed and reloaded from main memory), and no I/O. Alas, few useful computations fall into this category, with the possible exception of fractal computations for things like the Mandelbrot set and the Julia set.

Compilers with good optimizers are usually capable of recognizing scalar variables in loops, and assigning them to registers. Currently, only a few compilers are good enough to recognize array references as candidates for such assignments. That is why the inner loop of the matrix multiplication benchmark programs is written as

```
SUM = SUM + B(I, K)*C(K, J)
```

instead of as

```
A(I, J) = A(I, J) + B(I, K)*C(K, J)
```

Most programming languages give the user no control over whether a variable is assigned to a register or not; the C language is a notable exception. Thus, a good optimizing compiler is essential for high-performance computing.

If you think about what kinds of machine instructions have to be generated for the matrix multiplication inner loop, you can see that there have to be loads, possibly array index arithmetic, a floating-point multiplication, a floating-point addition, and a test and branch back to the top of the loop. Of all of these instructions, the only 'useful' ones are the floating-point operations. *To get higher performance, we have to increase the fraction of floating-point instructions in the inner loop code*.

One way to do this is *loop unrolling*; that is, we compute several terms of the summation in one iteration of the loop. Here is the equivalent code, unrolled to a depth of five:

```
      SUM = 0.0
      DO FOR K = 1, (N/5)*5, 5
           SUM = SUM +
     X        B(I, K + 0)*C(K + 0, J) +
     X        B(I, K + 1)*C(K + 1, J) +
     X        B(I, K + 2)*C(K + 2, J) +
     X        B(I, K + 3)*C(K + 3, J) +
     X        B(I, K + 4)*C(K + 4, J)
      END FOR
      DO FOR K = (N/5)*5 + 1, N
```

```
      SUM = SUM + B(I, K)*C(K, J)
END FOR
A(I, J) = SUM
```

The first loop steps K through 1, 6, 11, …, accomplishing ten floating-point operations in each pass through the loop. The second loop handles the remaining 0 to 4 elements, in the event that N was not a multiple of 5.

Loop unrolling usually helps to reduce loop overhead, provided there are sufficient registers available, and that the loop code is not made so long that it is too big to fit in the instruction cache. The reason is that many compilers will be able to compute the base address of `B(I,K+0)` once, and obtain the following 4 elements without further address arithmetic.

RISC machines have pipelined functional units, and branch execution can often be made to overlap that of loads, so there is very little penalty for a branch. With a good compiler, almost all of the array indexing arithmetic is moved outside the loop, and consequently, loop unrolling is likely to be of less use on RISC systems. On some vector machines (e.g. IBM 3090/xxx-VF), unrolling may inhibit vectorization, while on others (e.g. Cray), it may enhance vectorization. *There is no hard-and-fast rule*; either benchmarks, or examination of compiler-generated code, or vendor literature, must be consulted to determine the best strategy for any given system.

There is no explicit control over cache available to the programmer, but it is nevertheless possible to program in such a way that the cache is not reloaded unnecessarily. There are two steps that can be taken to reduce cache misses:

- In array processing, use a *stride* of one. That is, successive array elements should be fetched from consecutive memory locations.

  Unit stride is desirable because caching is not done on individual memory words, but instead on a *line*. A line is a group of consecutive memory words that are fetched from central memory into the cache. The first element referenced causes a cache load, but subsequent elements come directly from the cache, until the next cache line is reached, and central memory must again be referenced.

  For matrix multiplication, achieving a stride of one means that one should transpose one of the factors first (an N^2 operation) so that the N^3 multiplication can be done more efficiently.

- If the cache is reasonably large (capable of holding at least a few thousand numbers), it will be useful to process arrays in strips, such that the combined strip sizes are smaller than the cache size. This technique is known as *strip mining*.

  In the matrix multiplication context, this means that instead of 3 nested loops, one may expand the code to 6 nested loops, where the inner 3 run over contiguous subsets of the index ranges.

  Strip-mining is tedious, and error-prone, to code. It therefore makes sense to implement suitable primitives that can hide the messiness, and reduce the number of times the code has to be written. The higher-level Basic Linear Algebra Subroutines (BLAS) [5, 6, 8, 11] are a good step in this direction.

> Rare is the compiler that can automatically strip mine arrays, but one may expect that more will do so in the future.

Memory references can be reduced by *maximizing re-use* of data, provided the data can be held in registers, or at least in cache, during the loop.

Virtual memory page faults should be avoided, since they introduce relatively enormous delays (equivalent to tens of thousands, or more, lost cycles). This can be achieved by localizing consecutive array element accesses, stepping through the array in storage order.

In Fortran, this means that one should have the *first* (row) subscript varying most rapidly; in most other languages, including Ada, C, and Pascal, it should be the *last* (column) subscript. The Fortran loops above were written in J-I-K order to minimize page faults.

This detail is very important when translating programs between these languages. The commercial Fortran-to-C translator used to prepare the C versions of the benchmark programs reverses the index order of all arrays so as to preserve the memory access patterns. The EISPACK library [7, 15] was based on Algol procedures that were developed and published during the 1960s [17], when few machines, other than the IBM 360, had virtual memory. In the hand translation of Algol to Fortran, array indexes were *not* exchanged, so EISPACK code can sometimes cause serious page thrashing. This deficiency is remedied in the developing LAPACK project, which will supercede both EISPACK and LINPACK [4].

Storage access order considerations may need modification on vector machines, because there are sometimes tradeoffs between vector register re-use and memory access; see [12, p. 67] for details.

5 Selected case studies

The IBM RS/6000 [3, 13] provides a good case study for investigation of the effect of the memory hierarchy described in the preceding section, because it has a large register set, a large cache, and a good optimizing compiler, but is still a scalar machine. We will compare it with IBM 3090/600S-VF results with and without vectorization, and also with the Stardent 3040 (a machine that is both parallel and vector), two MIPS machines, and the Sun SPARCstation.

Although the long report [2] contains data for four different data types, and many different compilers and optimization levels, in these case studies, we will show only the `DOUBLE PRECISION` results, compiled at the highest optimization level, unless otherwise stated.

The normal inner loop of the benchmark programs looks like

```
DO FOR K = 1,N
     SUM = SUM + B(I,K)*C(K,J)
END FOR
```

On the IBM RS/6000, the code generated for this loop has only two incrementing loads, one multiply-add, and one branch; that is *perfect* assembly code for that loop. For the

100×100 `matmr8` benchmark, on the model 530 (25 MHz, 40 nsec cycle time), the maximum speed reached is 18 Mflops. Unrolling the loops to include five multiplications reduces the rate to 16 Mflops.

Note that the $\boldsymbol{B}$ matrix is being accessed with non-unit stride, i.e. across the rows. Replacing the matrix with its transpose has little effect on the RS/6000, because all three matrices fit in the cache for this example, so the non-unit stride did not cause a serious penalty.

In order to improve performance further, we need to increase the ratio of floating-point instructions to loads in the loop. The way this can be done is *to compute more than one element of the product matrix at a time*:

```
DO FOR K = 1,N
     SUM0 = SUM0 + B(I,K)*C(K,J+0)
     SUM1 = SUM1 + B(I,K)*C(K,J+1)
     ...
     SUMP = SUMP + B(I,K)*C(K,J+P)
END FOR
```

This of course requires that the surrounding `J` loop be modified to work in steps of $P+1$, and that a following loop be installed to handle the remaining `J` values.

The main reason for the increase is that we are able to re-use data in the inner loop; as the loop is written above, it needs $P + 1$ loads, and the same number of multiply-add instructions. Since the pipelining permits the overlap of the loads and the multiply-adds, the loop becomes dominated by floating-point instructions, and pushes towards peak performance of 2 flops per instruction executed.

An alternate scheme is to compute $P \times P$ blocks in the inner loop. A 2×2 case would look like

```
DO FOR K = 1,N
     SUM00 = SUM00 + B(I+0,K)*C(K,J+0)
     SUM10 = SUM10 + B(I+1,K)*C(K,J+0)
     SUM01 = SUM01 + B(I+0,K)*C(K,J+1)
     SUM11 = SUM11 + B(I+1,K)*C(K,J+1)
END FOR
```

This is even better than computing several row or column elements individually, because now each loaded element gets used *twice* on average, instead of just once. In fact, there are P^2 multiply-adds for $2P$ loads, so the data re-use grows like P. However, large P values mean that several columns of $\boldsymbol{C}$ have to be accessed, so there is a tradeoff against cache refilling.

Since this kind of optimization is likely to be useful on other register-rich architectures, like the MIPS RISC system, I expanded the benchmark code during early November, 1990, to include $P \times P$ unrolling ($P = 2 \ldots 5$), and to repeat the tests with a transposed $\boldsymbol{B}$ matrix, denoted by $\boldsymbol{B^T}$. Since the transposition can be done in $O(N^2)$ operations, except for virtual memory paging, it should have little impact on the overall performance, and the time to do so has been excluded from the benchmark figures. Transposition would be unnecessary in the case of symmetric matrices.

The following subsections summarize the results of the case studies, using the expanded benchmark programs.

5.1 IBM RS/6000

The RS/6000 has 32 32-bit integer registers and 32 64-bit floating-point registers. Models 320 and 520 have 32 KB of cache, and models 530, 540, 550, 730, and 930 have 64 KB of cache. A cache line holds 128 bytes (8 `DOUBLE PRECISION` numbers).

On the IBM RS/6000-530, the benchmark results are as follows:

IBM RS/6000-530
`DOUBLE PRECISION`
100×100 Matrix Multiplication

Loop type	no-opt Mflops	opt Mflops	opt/ no-opt
	$\boldsymbol{A = BC}$		
normal	1.24	18.09	14.6
unrolled	2.46	16.25	6.6
2×2	2.21	30.61	13.9
3×3	3.09	33.17	10.7
4×4	3.92	44.22	11.3
5×5	4.55	31.84	7.0
	$\boldsymbol{A = B^T C}$		
normal	1.27	18.09	14.2
unrolled	2.75	15.92	5.8
2×2	2.15	28.43	13.2
3×3	3.09	33.17	10.7
4×4	3.90	39.80	10.2
5×5	4.60	33.17	7.2

There are several things to notice in this table:

- The speed-ups in the last column show that *compiler code optimization is absolutely essential* to obtain high performance on this system. This is generally the case for most RISC and vector architectures.
- Linear loop unrolling to a depth of 5 doubles performance in the unoptimized code, but reduces performance of optimized code.
- $P \times P$ unrolling improves performance of both optimized and unoptimized code up to $P = 4$, and then performance falls. With $P = 5$, there are 25 partial sums and 10 matrix elements in the loop. This is too many to hold in the 32 available floating-point registers, so some values must be stored back into the cache. The significant performance jumps that happen with increasing P suggest that it should be possible to make use of even more floating-point registers than the 32 available in the RS/6000 architecture.

- The optimized 4×4 unrolling result of 44 Mflops is reasonably close to the RS/6000-530's peak performance of 50 Mflops, and was obtained in portable Fortran without having to resort to assembly coding.
- Changing the normal matrix multiplication code to the 4×4 unrolling improves performance by a factor of 2.4; this demonstrates the sensitivity of this architecture to the problem's algorithm.
- The best results for the RS/6000-530 are *10 times faster* than those for the top-end Sun 4/490 (4.2 Mflops), and *775 times faster* than those of the entry-level Sun 3/50 (0.057 Mflops). The latter is based on the Motorola 68020 architecture used in numerous other workstations, and in many Apple Macintosh personal computers. Given that the Sun 3/50 was introduced to the market about 1986, this is a stunning performance improvement in four years.

Surprisingly, transposition of the first product matrix does not help in most cases for this matrix.

For the 100×100 case, there are a total of 30 000 `DOUBLE PRECISION` numbers, requiring 240KB of memory. This is substantially larger than the cache size of 64KB on the model 530. However, computation of one column of $\boldsymbol{A}$ requires all of $\boldsymbol{B}$ and only one column of $\boldsymbol{C}$, so the main demand on cache is from only one of the matrices. This suggests that larger matrices might show the effect of cache overflow. I therefore made a special version of the benchmark program, with the following results:

IBM RS/6000-530 DOUBLE PRECISION Matrix Multiplication			
		$\boldsymbol{A = BC}$	$\boldsymbol{A = B^T C}$
Loop type	Size	Mflops	Mflops
normal	100	13.27	18.95
	200	2.17	17.35
	300	1.86	17.71
	400	1.69	18.29
	500	1.49	18.28
unrolled	100	14.21	16.58
	200	2.17	15.65
	300	1.87	16.66
	400	1.68	16.77
	500	1.49	17.15
2×2	100	26.53	28.43
	200	5.41	29.56
	300	5.45	30.29
	400	5.30	30.44
	500	4.87	30.51
3×3	100	28.43	33.13
	200	9.56	31.60
	300	10.43	32.77
	400	9.26	32.70
	500	8.40	33.17
4×4	100	39.81	44.18
	200	16.80	40.41
	300	17.25	42.62
	400	15.50	43.34
	500	14.56	43.40
5×5	100	36.19	30.61
	200	19.46	31.29
	300	18.92	31.90
	400	17.19	32.57
	500	16.66	33.06

Notice the drastic plunge in performance for the normal and unrolled cases when the matrix size increases beyond 100×100; this happens because the matrices can no longer fit in cache memory. One way to fix this performance problem would be to introduce strip mining of the outer loops.

$P \times P$ unrolling reduces the effect of the cache overflow, but performance of the normal multiplication, $\boldsymbol{A = BC}$, is still often less than half of that of the transposed case, $\boldsymbol{A = B^T C}$. It appears that $P = 4$ is a good choice for this machine, provided the first matrix is transposed before the multiplication.

The sharp-eyed reader may have noticed that the performance for the 100×100 case is lower than that given in the preceding table. The reason is that in the last table, all matrices were dimensioned for the 500×500 case; this leaves memory gaps between successive columns for the smaller matrices, and results in some virtual memory paging degradation.

5.2 IBM 3090/600S-VF

Let us now examine the same benchmarks on the IBM 3090/600S-VF. The 3090/600S has a 128KB cache, and a 15 nsec cycle time, but the architecture has only eight 32-bit (or four 64-bit) floating-point registers. With the vector facility, there are eight vector registers, each holding 256 `DOUBLE PRECISION` words. Transfer of a `DOUBLE PRECISION` word from cache to a register takes 1 cycle; memory-to-register time is 22 cycles.

Transfers between the 2GB extended memory and the 256MB central memory are done in 4KB pages at the rate of one `DOUBLE PRECISION` word per cycle, but there is a startup overhead of 70 μsec, so a page takes $4096/8 \times 15 \text{ nsec} + 70 \ \mu\text{sec} = 78 \ \mu\text{sec}$, or an average of one `DOUBLE PRECISION` word every 10 cycles. On the IBM 3090, extended memory is normally used by the operating system for virtual memory paging storage, avoiding much slower disk accesses.

IBM 3090/600S-VF DOUBLE PRECISION Matrix Multiplication							
		$A = BC$			$A = B^TC$		
Loop type	Size	Mflops scalar	Mflops vector	vector/ scalar	Mflops scalar	Mflops vector	vector/ scalar
normal	100	9.35	70.06	7.5	12.22	32.53	2.7
	200	7.84	61.99	7.9	11.24	35.53	3.2
	300	5.31	57.26	10.8	11.30	35.49	3.1
	400	3.40	59.05	17.4	11.35	38.24	3.4
	500	3.27	60.95	18.6	11.45	40.51	3.5
unrolled	100	11.73	63.10	5.4	13.05	16.54	1.3
	200	9.79	53.84	5.5	12.15	20.04	1.6
	300	6.77	49.47	7.3	12.16	23.12	1.9
	400	4.29	50.99	11.9	12.27	24.98	2.0
	500	4.09	53.13	13.0	12.41	27.11	2.2
2×2	100	9.62	46.38	4.8	10.80	22.14	2.0
	200	9.13	48.11	5.3	10.60	25.87	2.4
	300	8.09	52.02	6.4	10.64	26.11	2.5
	400	6.84	54.25	7.9	10.68	27.79	2.6
	500	6.67	56.47	8.5	10.72	28.66	2.7
3×3	100	10.34	19.33	1.9	10.90	16.94	1.6
	200	10.13	23.76	2.3	10.82	20.52	1.9
	300	9.39	26.89	2.9	10.83	20.58	1.9
	400	8.47	27.28	3.2	10.85	21.82	2.0
	500	8.21	28.17	3.4	10.90	22.50	2.1
4×4	100	10.79	15.37	1.4	10.98	16.08	1.5
	200	10.53	20.11	1.9	11.05	19.44	1.8
	300	10.02	22.64	2.3	11.02	19.14	1.7
	400	9.44	24.16	2.6	11.02	20.29	1.8
	500	9.23	25.33	2.7	11.04	20.71	1.9
5×5	100	10.84	13.41	1.2	11.03	15.32	1.4
	200	10.70	18.32	1.7	11.09	18.24	1.6
	300	10.38	20.78	2.0	11.05	17.90	1.6
	400	10.06	22.38	2.2	11.10	18.92	1.7
	500	9.94	23.59	2.4	11.11	19.16	1.7

The results for the IBM 3090 are qualitatively different from those for the RS/6000. With vectorization, unrolling linearly, or in $P \times P$ blocks, worsens performance; the 3090 vectorizer clearly does best when presented with the simplest code. In scalar mode, linear unrolling is best, but the 3090/600S is substantially out-performed by the RS/6000-530. Vectorization can boost scalar performance by a factor of nearly twenty, and the `-O3V` results are the best of any machine tested up to early November, 1990.

The $A = B^TC$ case has surprisingly low performance; further investigation is

needed to find out why. I expected that it would be uniformly better than the $\boldsymbol{A = BC}$ form because it ensures unit stride.

Optimal code for the 3090 and RS/6000 machines is quite different, suggesting that *naive porting of code from one to the other will usually suffer significant performance penalties.*

This machine has sufficient memory that the first time the arrays are referenced, they very likely become resident in physical memory; the benchmark programs actually perform the multiplication several times, and then average the performance results.

The plunging scalar performance seen for larger matrices in the $\boldsymbol{A = BC}$ case is due mainly to cache overflow; further details can be found in an excellent paper by two IBM researchers [12], on matrix multiplication, and in an earlier report from this Institute [14] on factorization of large matrices.

5.3 MIPS RC3240 and RC6280

MIPS Computer Systems is the company that was formed to commercialize the Stanford RISC CPU developed by John Hennessy and colleagues. Sun's SPARC grew out of the work by David Patterson's group at UC/Berkeley. MIPS produces workstations and mainframes, and also sells CPU chips and compiler technology to other vendors. Their Fortran and C compilers provide four levels of scalar optimization.

At present, MIPS chips are used in products from Stardent (15xx and 30xx series), DEC (DECstation series), and Silicon Graphics. They are also used by AT&T in telephone switching circuitry. DEC and Silicon Graphics use MIPS compilers; Stardent developed its own.

All MIPS Computer Systems' own products so far have been single-CPU scalar processors; Stardent produces vector and parallel machines, and Silicon Graphics, parallel processors.

MIPS Computer Systems has produced at least four generations of CPUs, the R2000, R3000, R4000, and R6000, with associated floating-point coprocessors, Rx010. The R4000 and R6000 have so far been available only in machines built by MIPS. The R6000 is implemented in bipolar ECL, to date the only commercially-available RISC chip to use this advanced technology.

Benchmark results for the MIPS R4000- and R6000-based products therefore give a hint of things to come from other vendors that currently use the R2000 and R3000 chip generations. The MIPS R6000 and IBM RS/6000 chips give the highest scalar performance in the industry today in machines with list prices under US$150 000.

Here are results for the RC3240 (25 MHz R3000 CPU and R3010 FPU) and the RC6280 (66.7 MHz R6000 CPU and R6010 FPU). MIPS literature quotes `DOUBLE PRECISION` Fortran LINPACK results of 3.9 Mflops and 10.3 Mflops, respectively, for these machines. The machine on which the benchmarks were run had a reduced clock rate of 60 MHz; that lowers the last number to 9.26 Mflops, which agrees with the results shown here.

MIPS RC3240 DOUBLE PRECISION 100 × 100 Matrix Multiplication				
Loop type	`-O0` Mflops	`-O1` Mflops	`-O2` Mflops	`-O3` Mflops
$\boldsymbol{A = BC}$				
normal	0.66	0.76	3.80	3.85
unrolled	1.05	1.86	2.89	2.95
2×2	0.99	1.87	3.48	3.48
3×3	1.26	2.30	4.12	4.09
4×4	1.51	2.56	3.69	3.66
5×5	1.67	2.77	4.04	4.00
$\boldsymbol{A = B^T C}$				
normal (B transpose)	0.66	0.76	3.92	3.92
unrolled (B transpose)	1.16	1.87	3.00	2.98
2×2 (B transpose)	1.00	1.85	3.37	3.37
3×3 (B transpose)	1.28	2.29	4.15	4.16
4×4 (B transpose)	1.52	2.57	3.87	3.86
5×5 (B transpose)	1.64	2.75	4.22	4.23

MIPS RC6280 DOUBLE PRECISION 100 × 100 Matrix Multiplication				
Loop type	`-O0` Mflops	`-O1` Mflops	`-O2` Mflops	`-O3` Mflops
$\boldsymbol{A = BC}$				
normal	1.46	1.65	8.65	8.63
unrolled	2.50	4.20	7.01	6.82
2×2	2.21	3.86	7.06	6.94
3×3	2.84	4.95	8.68	8.75
4×4	3.38	5.73	8.20	8.11
5×5	3.75	5.88	8.96	8.99
$\boldsymbol{A = B^T C}$				
normal (B transpose)	1.46	1.67	8.80	8.80
unrolled (B transpose)	2.76	4.25	7.01	7.17
2×2 (B transpose)	2.23	3.90	7.09	7.06
3×3 (B transpose)	2.85	4.91	8.63	8.70
4×4 (B transpose)	3.38	5.67	8.48	8.50
5×5 (B transpose)	3.73	5.82	9.36	9.42

For both of these machines, $P \times P$ unrolling more than doubles performance with `-O0`, and triples performance with `-O1`. With the two highest optimization levels, it

produces only about a 5% improvement. High-performance code would normally be compiled at the highest optimization level, so unrolling may not be worthwhile in most programs.

5.4 Stardent 3040

Let us now look at a rather different architecture, the Stardent 3040, formerly known as the Ardent Titan-4 P-3. This machine has four MIPS R3000 CPUs, with 16 64-bit scalar floating-point registers.

Each CPU has a 64-bit vector unit whose design is quite unusual: it can be configured as one register with 1024 elements, two registers with 512 elements, four registers with 256 elements, …, 1024 registers of 1 element. Current versions of the Fortran and C compilers choose to subdivide it into 32 vector registers, each with 32 64-bit elements. In conversations with the compiler developers, I have been told that memory and cache access are often the limiting factor in reaching peak performance on this machine; the actual length of the vector registers is not critical, provided that the pipeline can be kept full of data. A choice of 32 vector registers seems to be rich enough to cover almost all practical applications, allowing all vector quantities in the inner loop to be kept in vector registers.

The compiler has four optimization levels:

`-O0` Turn off all optimizations.

`-O1` Perform common subexpression elimination and instruction scheduling.

`-O2` Perform `-O1` and vectorization.

`-O3` Perform `-O2` and parallelization.

The interesting results will be found in comparisons of the three highest levels, since this illustrates the ratios of scalar : vector : vector + parallel:

Stardent 3040 DOUBLE PRECISION 100 × 100 Matrix Multiplication							
Loop type	-O0 Mflops	-O1 Mflops	-O2 Mflops	-O3 Mflops	-O2/ -O1	-O3/ -O2	-O3/ -O1
$A = BC$							
normal	2.75	2.74	8.31	28.16	3.03	3.39	10.26
unrolled	3.40	3.28	4.13	13.09	1.26	3.17	3.99
2×2	4.56	4.75	10.62	29.85	2.24	2.81	6.28
3×3	4.69	4.78	10.55	30.46	2.21	2.89	6.38
4×4	3.17	4.03	12.04	31.09	2.99	2.58	7.72
5×5	3.65	4.10	6.75	16.86	1.65	2.50	4.11
$A = B^T C$							
normal	2.77	2.77	7.71	23.14	2.78	3.00	8.34
unrolled	3.27	3.33	4.38	15.47	1.32	3.53	4.65
2×2	4.66	4.90	9.82	32.45	2.00	3.30	6.62
3×3	4.86	4.85	10.05	28.16	2.07	2.80	5.81
4×4	3.91	4.09	11.22	33.54	2.74	2.99	8.20
5×5	3.47	4.04	6.80	16.58	1.68	2.44	4.11

From the `-O0` and `-O1` columns, it is evident that modest speed-ups are obtained by loop unrolling.

From the `-O2` column, which is vectorization on a single processor, 4×4 unrolling is a clear winner

From the `-O3` column, which is vectorization and parallelization across four processors, 4×4 unrolling is again the best strategy.

The behavior is very much like the IBM RS/6000 and *unlike* the IBM 3090/600S-VF.

The `-O2/-O1` column shows the speed-ups from vectorization on a single processor; they are much smaller than we found on the IBM 3090/600S-VF.

The `-O3/-O2` column shows the speed-up from parallelization; the limit is 4.0, so these numbers are really quite good.

Finally, the last `-O3/-O1` column shows the overall speed-up obtained by vectorization and parallelization over a purely scalar MIPS R3000 processor. These extra facilities give very definite performance boosts.

Stardent has a processor upgrade for the 3040 that more than doubles the floating-point performance that we have shown here. In an advertisement in the July 1990 issue of Supercomputing Review, Stardent claims a LINPACK 1000×1000 result of 77 Mflops. By contrast, the IBM 3090/180S-VF (one processor) achieves 92 Mflops (theoretical peak: 133 Mflops), and the IBM 3090/600S-VF (six processors parallelized) reaches 518 Mflops (theoretical peak: 798 Mflops). On the 1000×1000 tests, vendors are permitted to make system-dependent changes to the code, including rewriting the loop structure, or coding in assembly language. The theoretical peak rates are determined by assuming that every cycle generates at least one floating-point result.

5.5 Sun SPARCstation SLC

As a final example, let us look at the Sun SPARCstation SLC. With an entry-level list price of about $5000, it is one of the cheapest RISC-based workstations on the market in late 1990. It is purely a scalar machine, but the compiler does provide four different levels of optimization. What is of interest is to see whether the loop unrolling techniques that were so beneficial on the IBM RS/6000 are of any value on the Sun SPARC architecture.

Sun SPARCstation SLC
`DOUBLE PRECISION`
100×100 Matrix Multiplication

Loop type	`-O0` Mflops	`-O1` Mflops	`-O2` Mflops	`-O3` Mflops
		$\boldsymbol{A = BC}$		
normal	0.46	0.47	1.18	1.16
unrolled	0.55	0.63	1.07	1.07
2 x 2	0.50	0.51	1.13	1.12
3 x 3	0.59	0.59	1.12	1.10
4 x 4	0.64	0.65	1.19	1.19
5 x 5	0.68	0.67	1.21	1.21
		$\boldsymbol{A = B^T C}$		
normal	0.47	0.47	1.20	1.21
unrolled	0.64	0.62	1.03	1.01
2 x 2	0.51	0.50	1.44	1.36
3 x 3	0.59	0.58	1.33	1.30
4 x 4	0.64	0.64	1.37	1.30
5 x 5	0.67	0.67	1.33	1.37

Without optimization, unrolling helps, but with optimization, it reduces performance. $P \times P$ unrolling is always better than the normal loop, but the total improvement is very small, only about 10%. I expect that similar results would be found for most conventional scalar machines.

6 The future

On the basis of the case studies presented here, and the results of the long report [2], what can we conclude about the prospects of future performance increases in affordable machines for high-performance computing?

The MIPS R6000 CPU today uses the most advanced practical fabrication technology, bipolar ECL. Gallium arsenide technology is being used in the Cray 3, but development schedules have been delayed by difficulties with that technology. Indium phosphide lies beyond gallium arsenide, but neither are likely to be available for many years

yet in quantities and prices that make them usable in desktop workstations, and in any event, they may require substantial cooling systems.

The MIPS R6000 has a 66.7 MHz clock rate, 50% higher than the next fastest clocks on machines in this price range, those of the 41.7 MHz IBM RS/6000-550, and the 40 MHz Sun SPARCstation 2. The cycle time of the fastest supercomputers today is around 5 nsec, equivalent to a 200 MHz clock rate. We may see clock speeds in desktop systems reach 50 to 80 MHz in the next few years, but *decreased cycle time alone does not offer much hope for improved performance.*

I would like to offer several observations about the future of increased performance on affordable machines:

- Scalar performance is reaching its practical limit in current chip technologies. The bipolar ECL in the MIPS R6000 requires additional cooling, making it unsuitable for desktop workstation configurations.
- Additional performance will be seen primarily through vector and parallel architectures.
- Performance differences between CPUs at similar clock rates are going to arise mainly from cleverness in the instruction set, since that allows reducing the number of instructions needed. Reduced instruction set architectures were introduced to make better use of chip circuitry than complex instruction sets can. However, improving instruction sets usually means making them more complex, which conflicts with this goal. In the RS/6000, IBM seems to have been able to produce a superior RISC architecture, for at least these reasons:
  - Integer multiply and divide instructions are available; they are often needed for array indexing computations. The Sun SPARC architecture does not provide them.
  - Auto-incrementing load instructions make it possible to fetch an array element and point to the next one in a single clock cycle.
  - The floating-point multiply-add instruction gives two flops in one cycle, and is very often usable in the inner loops of scientific and engineering code. It also gives higher accuracy, because the multiply is exact; only one rounding, instead of two, is required to obtain the result.
  - Separate branch, integer, and floating-point functional units offer possibilities of speed-ups from instruction overlap. For example, the 18 Mflops obtained in the normal loop for the RS/6000-530 corresponds to 110 nsec per loop iteration. However, the four instructions in the loop should take 4 cycles, or 160 nsec. Overlap has therefore reduced the effective iteration time to $110/160 \times 4 = 2.8$ cycles.
- Other RISC architectures, including MIPS, Motorola 88000, and Sun SPARC, will *not* be able to compete against the IBM RS/6000 unless they evolve to incorporate auto-incrementing loads and stores, floating-point multiply-adds, and

increased functional unit overlap. Any clock speed increases made by one vendor can soon be matched by another; the winner will be the one with the fewest instructions in the inner loop, assuming comparable speeds of cache and central memory.

Such architectural changes would introduce binary incompatibility in the software market, unless users are willing to pay a penalty of having run-time software emulation of new instructions on older hardware. Each vendor will have to weigh the commercial consequences of architectural changes against sales losses to the faster competition.

Because UNIX has become a *de facto* operating system in scientific and engineering computing, and the X Window System is available on virtually all workstations at very low cost, customers' software investments no longer need tie them to particular computer vendors, as was the case only a few years ago.

This situation, combined with the relatively low cost of desktop systems, makes it likely that users will switch vendors more often, in response to performance differences in machines on the current market. This trend to the commodity market of consumer goods, like automobiles, laundry detergent, and televisions, has already been evident in the IBM PC clone market. Workstations are clearly next, and the impact on the computing world will be substantial. Competition and increased sales volumes will drive down prices, and smaller vendors will disappear from the low-margin marketplace. Technological innovations from small groups, such as happened initially with personal computers and Sun workstations, will be increasingly unlikely to succeed.

- Specialized RISC architectures, like the Intel i860, may offer temporary performance boosts with third-party add-in boards for conventional workstations, but they require additional, and probably expensive, compiler support. So far, the only vendor to incorporate the Intel i860 in a desktop system has been Stardent, which uses it to implement the vector unit of the recently-announced desktop model 500.

- The wide performance ranges seen on the IBM RS/6000 and Stardent 3040 clearly demonstrate the *importance of fast memory and large caches*, and suggest that these newer machines are not well-balanced between CPU and memory speeds. While memory prices continue to fall, and chip capacities increase, memory speeds have regrettably not improved very much in the last decade. At present, it is economically infeasible to produce desktop systems with several tens of megabytes of cache memory.

  Programmers on these machines will therefore have to pay much more attention to code tuning than they did in the past.

- Benchmark results for parallel machines, like those from Silicon Graphics and Stardent, demonstrate that *compiler-generated automatic parallelization* for computations similar to that of the benchmark code is indeed feasible, and worthwhile. Convex and Stardent offer four-processor systems, IBM six (on the 3090

only), and Silicon Graphics eight. IBM has not yet announced a multi-processor version of the RS/6000, but such a product must certainly be in development.

On the basis of these benchmarks, and several published studies that I have read in recent years, the useful limit for parallelization of standard programs may be reached with somewhere between 8 and 32 processors. Beyond that, new algorithms will likely be necessary. The extreme parallel system today is the 64K-processor Connection Machine, for which quite different programming techniques are required to obtain substantial speed-ups.

7 Conclusions

We can express a hope that compilers will eventually be able to do strip mining and loop expansions automatically. In the meantime, it is evident that routines like the level-2 and level-3 Basic Linear Algebra Subroutines (BLAS) [5, 6, 8, 11] make a great deal of sense.

They can be written for a particular architecture with the ugly details of the optimizations described above hidden from the user, yet the user code remains portable across architectures.

With a good optimizing compiler, there is likely little to be gained from programming such primitives in assembly code; they can be kept in a higher-level language, like Fortran.

Ideally, optimized versions of the BLAS should be part of every computer vendor's standard software. Just as Fortran programmers have been accustomed to having elementary scalar functions (*exp(), log(), sin(), sqrt(),* …) available as a standard part of the language, it is clearly time to provide matrix and vector functions in a standard fashion. I am certainly not calling here for changes to programming languages, in the directions taken by the Fortran 90 committee; I believe that such actions are premature, and detrimental to code portability. However, making a set of highly-tuned standard matrix and vector functions available would be very beneficial. Since simple untuned implementations can be easily written, code portability to architectures that lack optimized versions is not compromised.

The various techniques of loop unrolling here are critical for pushing towards the limits of high-performance vector and parallel, or highly pipelined, machines, like the Stardent, IBM 3090, and IBM RS/6000; they offer only modest performance increases on conventional machines like the Sun SPARCstation SLC.

Careful tuning of matrix primitives is essential on high-performance RISC and vector machines; naive code could easily run more than twenty times slower.

8 Acknowledgements

I want first to thank the many people, too numerous to name here, who have generously provided access to their machines for my benchmarking activities.

Michael Pernice of the Utah Supercomputing Institute provided several helpful suggestions about the content of this report.

Brian Bone and Jane Cullum of IBM Corporation directed me to IBM reports [3, 13] which gave valuable insights about the significant implications of the memory hierarchy on the performance of the IBM RS/6000 machines.

The Utah Supercomputing Institute workshop in early November 1990 provided a useful forum for the dissemination and exchange of ideas and experience in high-performance computing from experts in the field.

References

[1] Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. *The AWK Programming Language*. Prentice-Hall, 1988.

[2] Nelson H. F. Beebe. Matrix multiply benchmarks. Technical report, Center for Scientific Computing, Department of Mathematics, University of Utah, Salt Lake City, UT 84112, USA, November 1990. This report is updated regularly. An Internet e-mail list is maintained for announcements of new releases; to have your name added, send a request to `beebe@math.utah.edu`. The most recent version is available for anonymous `ftp` from the directory `~ftp/pub/benchmarks`. It is also accessible from the `tuglib` e-mail server. To get it, send a request with the texts `help` and `send Index from benchmarks` to `tuglib@math.utah.edu`.

[3] Ron Bell. IBM RISC System/6000 performance tuning for numerically intensive Fortran and C programs. Technical Report GG24-3611-00, IBM Corporation, August 1990.

[4] J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. W. Stewart. *LINPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, 1979.

[5] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. Algorithm 679: A set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16(1):18–28, March 1990.

[6] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. Algorithm 656: An extended set of Basic Linear Algebra Subprograms: Model implementation and test programs. *ACM Transactions on Mathematical Software*, 14(1):18–32, March 1988.

[7] B. S. Garbow, J. M. Boyle, J. J. Dongarra, and C. B. Moler. *Matrix Eigensystem Routines—EISPACK Guide Extension*, volume 51 of *Lecture Notes in Computer Science, Editors: G. Goos and J. Hartmanis*. Springer-Verlag, 1977.

[8] R. J. Hanson and F. T. Krogh. Translation of Algorithm 539: PC-BLAS Basic Linear Algebra Subprograms for Fortran usage with the Intel 8087, 80287 Numeric Data Processor. *ACM Transactions on Mathematical Software*, 13(3):311–317, September 1987.

[9] L. Kronsjö. *Computational Complexity of Sequential and Parallel Algorithms.* Wiley-Interscience, 1985.

[10] C. L. Lawson and J. A. Flynn. SFTRAN3 Programmer's Reference Manual. Technical Report 1846-98, Jet Propulsion Laboratory, Pasadena, CA, December 1978.

[11] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Algorithm 539: Basic Linear Algebra Subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5(3):324–325, September 1979.

[12] Bowen Liu and Nelson Strother. Programming in VS Fortran on the IBM 3090 for maximum vector performance. *Computer*, 21:65–76, 1988.

[13] Mamata Misra, editor. *IBM RISC System/6000 Technology, publication SA23-2619-00*. IBM Corporation, 1990.

[14] Michael Pernice. The performance of LU factorization algorithms on the IBM 3090/600S. Technical Report 1, Utah Supercomputing Institute, University of Utah, Salt Lake City, UT 84112, USA, 1990.

[15] B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler. *Matrix Eigensystem Routines—EISPACK Guide*, volume 6 of *Lecture Notes in Computer Science, Editors: G. Goos and J. Hartmanis*. Springer-Verlag, 1976.

[16] Herbert S. Wilf. *Algorithms and Complexity.* Prentice-Hall, 1986.

[17] James H. Wilkinson and Christian Reinsch. *Linear Algebra*, volume II of *Handbook for Automatic Computation, Editors: F. L. Bauer, A. S. Householder, F. W. J. Olver, H. Rutishauser, K. Samelson and E. Stiefel*. Springer-Verlag, 1971.

[18] S. Winograd. A new algorithm for inner product. *IEEE Trans. Comp.*, 17:693–694, 1968.

Index