NUMERICAL LINEAR ALGEBRA

W. Kahan

Table of Contents

0. Introduction. The primordial problems of linear algebra are the solution of a system of linear equations

$$A\underline{x} = \underline{b} \quad (\text{i.e., } \Sigma_j a_{ij} x_j = b_i) ,$$

and the solution of the eigenvalue problem

$$A\underline{x}_k = \lambda_k \underline{x}_k$$

for the eigenvalues λ_k and corresponding eigenvectors $\underline{x}_k$ of a given matrix A. The numerical solution of these problems without the aid of an electronic computer is a project not to be undertaken lightly. For example, using a mechanical desk-calculator to solve five linear equations in five unknowns (and check them) takes me nearly an hour, and to calculate five eigenvalues and eigenvectors of a five-by-five matrix costs me at least an afternoon of drudgery. But any of today's electronic computers are capable of performing both calculations in less than a second.

* Sections 6 to 11 will appear in this Bulletin at a later date.

Canadian Mathematical Bulletin
V. 9, n. 6, 1966.

MATHEMATICS OF COMPUTATION
VOLUME 50, NUMBER 182
APRIL 1988, PAGES 449-480

The Probability That A Numerical Analysis Problem Is Difficult

By James W. Demmel

Abstract. Numerous problems in numerical analysis, including matrix inversion, eigenvalue calculations and polynomial zerofinding, share the following property: The difficulty of solving a given problem is large when the distance from that problem to the nearest "ill-posed" one is small. For example, the closer a matrix is to the set of non-invertible matrices, the larger its condition number with respect to inversion. We show that the sets of ill-posed problems for matrix inversion, eigenproblems, and polynomial zerofinding all have a common algebraic and geometric structure which lets us compute the probability distribution of the distance from a "random" problem to the set. From this probability distribution we derive, for example, the distribution of the condition number of a random matrix. We examine the relevance of this theory to the analysis and construction of numerical algorithms destined to be run in finite precision arithmetic.

1. Introduction. To investigate the probability that a numerical analysis problem is difficult, we need to do three things:

(1) Choose a measure of difficulty,

(2) Choose a probability distribution on the set of problems,

(3) Compute the distribution of the measure of difficulty induced by the distribution on the set of problems.

The measure of difficulty we shall use in this paper is the *condition number*, which measures the sensitivity of the solution to small changes in the problem. For the problems we consider in this paper (matrix inversion, polynomial zerofinding and eigenvalue calculation), there are well-known condition numbers in the literature of which we shall use slightly modified versions to be discussed more fully later. The condition number is an appropriate measure of difficulty because it can be used to measure the expected loss of accuracy in the computed solution, or even the number of iterations required for an iterative algorithm to converge to a solution.

The probability distribution on the set of problems for which we will attain most of our results will be the "uniform distribution" which we define as follows. We will identify each problem as a point in either $\mathbf{R}^N$ (if it is real) or $\mathbf{C}^N$ (if it is complex). For example, a real n by n matrix A will be considered to be a point in $\mathbf{R}^{n^2}$, where each entry of A forms a coordinate in $\mathbf{R}^{n^2}$ in the natural way. Similarly, a complex nth degree polynomial can be identified with a point in $\mathbf{C}^{n+1}$ by using its coefficients as coordinates. On the space $\mathbf{R}^N$ (or $\mathbf{C}^N$) we will take any spherically symmetric distribution, i.e., the induced distribution of the normalized problem $x/\|x\|$ ($\|\cdot\|$ is the Euclidean norm) must be uniform on the unit sphere in

Received May 4, 1987; revised August 18, 1987.

1980 *Mathematics Subject Classification* (1985 *Revision*). Primary 15A12, 53C65, 60D05.

Parts of this paper appeared previously under the title "The Geometry of Ill-Conditioning," in *The Journal of Complexity*, Vol. III, pgs. 201-229, 1987.

SPE
486

SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 18(00), 000–000 (1988)

Compiler Support for Floating-point Computation

CHARLES FARNUM
Computer Science Department, University of California, Berkeley, Berkeley, California 94720, U.S.A.

SUMMARY

Predictability is a basic requirement for compilers of floating-point code — it must be possible to determine the exact floating-point operations that will be executed for a particular source-level construction. Experience shows that many compilers fail to provide predictability, either because of an inadequate understanding of its importance or from an attempt to produce locally better code. Predictability can be attained through careful attention to code generation and a knowledge of the common pitfalls. Most language standards do not completely define the precision of floating-point operations, and so a good compiler must also make a good choice in assigning precisions of subexpression computation. Choosing the widest precision that will be used in the expression usually gives the best trade-off between efficiency and accuracy. Finally, certain optimizations are particularly useful for floating-point and should be included in a compiler aimed at scientific computation. But predictability is more important than efficiency; obtaining incorrect answers fast helps no one.

KEY WORDS Compilers Floating-point arithmetic Optimization

INTRODUCTION

Floating-point programs must be carefully compiled in order to produce accurate results. Data accessing primitives, control structures and integer arithmetic are clearly defined in most language standards, since most hardware is more or less equivalent in the support it offers for these tasks. But floating-point systems vary widely, so language standards cannot specify perfectly the semantics of source-level floating-point operations.[1] The implementor is left with the difficult task of deciding what machine-level operations will result from source-level code.

Unfortunately, most compiler writers are ill-equipped to handle this task. Compiler texts and classes rarely address the peculiar problems of floating-point computation, and research literature on the topic is generally confined to journals read by numerical analysts, not compiler writers. Many production-quality compilers that are excellent in other respects make basic mistakes in their compilation of floating-point, resulting in programs that produce patently absurd results or, worse, reasonable but inaccurate results.

An implementation of a large floating-point library is a job for specialists. But given such a library, little information is actually needed to produce good floating-point code

0038–0644/88/000000–00$05.00

Received 24 March 1987
Revised 21 December 1987

G.2 DISCRETE MATHEMATICS

G.2.1 Combinatorics

Combinatorial algorithms
See: 8804-0242 [E.3]

Permutations and combinations
See: 8804-0242 [E.3]

G.3 PROBABILITY AND STATISTICS
See: 8804-0260 [K.6.2—*Computing equipment management*]

Statistical computing
See also: 8804-0238 [D.2.9—*Software quality assurance (SQA)*]

JOHNSON, MARK E. (Los Alamos National Laboratory, Los Alamos, NM) 8804-0248
Multivariate statistical simulation.
John Wiley & Sons, Inc., New York, NY, 1987, 230 pp., $34.95, ISBN 0-471-82290-6. [Wiley series in probability and mathematical statistics.]

This monograph describes an approach to generating continuous multivariate distributions. Typically, the approach throughout is to use a transformation of one or two easily generated random variables to achieve a new distribution. A few parameters provide a mechanism for generating one of a family of possibilities. To help choose the correct parameter values, the reader is then presented with an array of three-dimensional and contour plots for the families.

A large number of distributions are described. These include the well known, the less well known, and the rare. The multivariate normal and multivariate uniforms are examples of the first type. The somewhat less well known or rare include the lognormal, logistic, Pareto, Burr, and Wishart, to name some of the remaining ones. This work is a serious contribution to the field and could be of use to those doing Monte-Carlo or discrete event simulations.

—*T. Brown,* Flushing, NY

GENERAL TERMS: ALGORITHMS, THEORY

G.4 MATHEMATICAL SOFTWARE

KEMPF, JAMES (Hewlett-Packard Company) 8804-0249
Numerical software tools in C.
Prentice-Hall, Inc., Englewood Cliffs, NJ, 1987, 261 pp., $28, ISBN 0-13-627274-6. [Prentice-Hall software series.]

This book is not a book about numerical software, using C as the vehicle for presenting examples; rather it is a book about C and UNIX, with the presentation tied together by the numerical slant of the examples. Twenty percent of the book is devoted to an initial chapter on the basics of C. This is followed by another 20 percent devoted to a chapter using C to write the dot product, matrix addition, and Gaussian elimination. Virtually nothing is said about numerical issues such as when to use the partial pivoting included in the program and what the other choices are. Instead, the emphasis is on program structure issues such as modularity, orderly development, specification, and realization of I/O interfaces.

Chapters 3 and 4, which comprise 30 percent of the book, are devoted to numerically related I/O issues: exchanging data via UNIX pipes and graphics. The slant of the book away from high volume numerical computation is established by the encouragement to convert double precision numbers to character strings and back again in order to pass them between programs. It is suggested that "a faster interface is probably more desirable" for large applications, but this is not pursued. The graphics chapter is fairly extensive and treats such things as clipping and windowing as well as curve plotting. Neither perspective projection nor contouring for presenting functions of two variables is covered.

The final 30 percent of the book is devoted to one chapter on optimization and one on differential equations. Again the emphasis is on program design and structure, and the discussion of numerical issues is abridged. You will find the word *stiff* mentioned, but techniques for handling stiff differential equations are beyond the scope of this book.

While this book handles modularity and interfaces fairly well, it has a number of peculiarities I tend to associate with C programming. In the main program for vector optimization, for example, the call to the routine that the text later identifies as "the heart of the vector optimization module" is buried in the condition of an *if* statement and identified only by the comment

```
\*
error in method occurred
*/.
```

—*H. F. Jordan,* Boulder, CO

GENERAL TERMS: ALGORITHMS, LANGUAGES

H. Information Systems

H.0 GENERAL

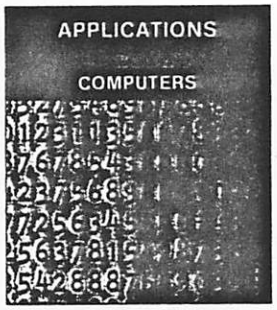
FLYNN, ROGER R. (Univ. of Pittsburgh, Pittsburgh, PA) 8804-0250
An introduction to information science.
Marcel Dekker, Inc., New York, NY, 1987, 793 pp., $39.75, ISBN 0-8247-7508-2. [Library and information science.]

What topics should be included in a course entitled "Introduction to Information Science?" This is not a question that has a single answer, for people view information science from different perspectives, and the discipline itself is changing rapidly.

Roger Flynn, the book's author, is a professor in the Department of Information Science at the University of Pittsburgh. He developed the course materials through "a lengthy process of selection, trial, and revision." The text is designed for use in an introductory course at an undergraduate level. Information science is equated with answering questions, that is, with the activities involved in seeking, processing, and using information to solve problems and to make decisions. That these activities are complex and that

Supercomputing for one

Interactive, high-resolution graphics and vector processing combine for the first time in these desk-side units



In the culture of engineering—where a project's importance is often gauged by the computational power it requires—supercomputers have something of a mystique. Virtually all electrical engineers know what supercomputers can do, who makes them, and how much they cost. But because of its price tag, most EEs will never see a supercomputer, let alone use one—there are only 300 or so worldwide.

But supercomputing no longer necessarily means multimillion dollar machines produced by a few companies for a tiny technical and scientific elite. Minisupercomputers have been steadily gaining in popularity since they were first introduced a few years ago. Minisuper processing rates range from roughly 3 million to 20 million floating-point operations per second (megaflops) on the standard Linpack benchmark. They offer about one-third, some as much as one-half, the peak performance of a typical full-size supercomputer.

Last month, supercomputing made another great stride toward egalitarianism. Two U.S. manufacturers introduced their versions of a graphics supercomputer, a new class of machine that integrates a portion of the computational power of a supercomputer and the interactive, three-dimensional visual capability of a state-of-the-art workstation.

Graphics supercomputers are parallel, multiprocessor systems. They have high-speed integer processors and 64-bit vector processors like those used in supercomputers and minisupers to handle calculations that simulate complex physical events. They use the Unix operating system and have compilers that automatically transform and optimize code written in Fortran or C to exploit vector and parallel hardware, with no need for machine-specific extensions or assembly language. Supported by high processor-to-memory bus bandwidth and highly interleaved memory, graphics supercomputers can sustain more than 6 megaflops on a 100-by-100 compiled Linpack benchmark, and can peak at 64 megaflops. (The best technical workstations, like the Sun-4 from Sun Microsystems Inc. of Mountain View, Calif., run at no more than 1.1 megaflops on the Linpack.)

Priced between $80 000 and $150 000, these new machines offer about one-fourth the performance of a Cray X-MP for no more than one-twentieth the price. An EE, for example, could use one to shrink the design cycle by simulating such complex circuitry as a floating-point chip, or by precisely modeling the emission pattern of a new antenna. Further, graphics supercomputers make it possible to simulate circuits too large—and therefore too expensive—to be handled by existing computers.

As their name suggests, graphics supercomputers also provide integral graphics processing so that engineers can express computations visually, modify a design interactively, and see the results immediately. An engineer might alter some element of the circuit's design and get instant, visual feedback on how that change affects the chip's function or output. Likewise, modifying the antenna would produce an immediate change in the emission pattern shown on the computer's monitor.

Unlike a supercomputer or minisuper, which is usually accessed by several users at once, a graphics supercomputer can be dedicated to a single user. Designed to be interactive, it lets a scientist or engineer close in on an optimal design or solution through step-by-step refinements. By executing computations under the direct control of a single user, it may actually provide higher throughput and productivity for the one application than would a faster machine that typically runs multiple jobs in a noninteractive, or batch, environment.

C. Gordon Bell, Glen S. Miranker, and Jonathan J. Rubinstein
Ardent Computer Corp.

Start-ups carry the flag

The first two full-fledged graphics supercomputers both come from start-up companies: Ardent Computer Corp. of Sunnyvale, Calif., and Stellar Computer Inc. of Newton, Mass. But they are unlikely to have the field to themselves for long.

On March 1, when Ardent was introducing its machine in San Francisco, workstation manufacturer Apollo Computer Inc. was introducing its Series 10000 Personal Supercomputers in Boston. The Ardent and Apollo machines both incorporate from one to four reduced instruction-set processors that are said to offer, correspondingly, integer processing capabilities from 16 million to 64 million instructions per second (MIPS). But unlike the Ardent and Stellar machines, the Apollo uses proprietary floating-point hardware rather than vector processors, and it will not have full three-dimensional graphics capabilities until after it reaches market.

Other workstation manufacturers, like Silicon Graphics Inc. and Sun Microsystems Corp., both of Mountain View, Calif., are working on machines similar to Apollo's. Hewlett-Packard Co.,

Defining terms

Backplane: a hardware system for transferring data at very high speeds between a computer's circuit boards.
Linpack: a package of linear algebra subroutines, widely used as a performance benchmark for floating-point performance; as a benchmark, the 100-by-100 Linpack solves a system of 100 equations with 100 unknowns.
Port: a read or write channel to a memory or register file.
Vector: an ordered sequence of numbers often used to represent physical characteristics or quantities in a simulation.
Vector processor, vector unit: a high-speed processing unit designed to perform simultaneous operations on vectors.
Virtual memory: a technique using both hardware and software that permits storage of programs and data outside a computer's main memory. In a multiuser machine, virtual memory also protects data and code when several programs are running at once.

0018-9235/88/0400-0046$1.00©1988 IEEE

image information in a standard format, regardless of the characteristics of the actual scanner. Brotz did not think it would work and "Warnock promptly labeled it 'Andy's Stupid Input Device.'"

Still, Brotz thought it might be helpful for generating test patterns, and when he implemented it, "it turned out that Andy's Stupid Input Device was the lowest common denominator and all the special-case code could disappear." Problems arise only when the image data has been compressed for transmission or storage; the programmer then has to insert a routine to decompress the data before it is handed to the image algorithm.

Another improvement involved performance profiling—running various tests to see what frequently used functions slowed down operation. Floating-point routines were the chief culprits because they are computationally intensive. So the team took some of the algorithms for the common operations, such as breaking curves into vectors and drawing outlines, and rewrote them in less flexible fixed-point arithmetic. Now only when fixed-point arithmetic would be too imprecise does the interpreter call the floating-point routine.

"So with no loss of generality," says Edward Taft, Adobe senior computer scientist, "we were handling 99 percent of the cases five times faster than we were before."

To improve the other 1 percent, Belleville sent one of his engineers over from Apple—Jerome Coonen, a recognized expert in floating point. He optimized the algorithms so, Taft says, "whereas formerly an algorithm required six multiplies, four divides, and three square roots, now it only required three multiplies, four divides, and some approximation of a square root."

Throughout the design of PostScript, speed was regularly traded off to ensure that any image would print. The group reasoned that if they built in all this functionality, they could eventually improve the performance; but if they left out functions, they might never be able to add them back in.

However, says Putman, sometimes they had doubts. So they designed a version of PostScript that spat out information as fast as the laser moved across the page. The expense of the frame buffer was eliminated—along with the ability to print pages too complicated for the software to process in time.

Adobe called this implementation Subscript, but dropped it after six months. As Taft says, "If you're trying to promote a standard, there is nothing worse than issuing a subset of the standard. It means that all of the applications are going to be targeted to the lowest common denominator."

Debugging throughout the project was strenuous because the Adobe team was "terrified of putting all this code out on ROMs," Brotz says "We came from the school of thought that software is soft. So if you have problems, you just have another release. But Apple was telling us, 'Hey, we always ship our system in ROM, why can't you?'"

In January of 1985 the Apple LaserWriter was introduced, virtually bug-free. In 1984, Adobe signed licensing agreements with QMS Inc., Linotype, and Dataproducts Corp. Today, even Hewlett-Packard Co., whose PCL page description language was one of PostScript's earliest competitors, is among the 23 companies offering PostScript interpreters for their printers.

Cheap pays off

Although the Adobe group made some key technical breakthroughs, three other components were necessary to make PostScript a runaway success not just in low-volume professional publishing but in the high-volume office environment.

As noted earlier, one was a cheap laser printer. When Adobe was founded, the cheapest cost around $10 000. It also weighed as much as a desk, so that it had to be serviced on site and sold through a distributor, not on a cash-and-carry basis. Then Canon Inc., of Tokyo, Japan, introduced the Canon LBP-CX desktop laser printer, which, moreover, printed beautifully. "If it had been poor xerography," says Paxton, "it wouldn't have mattered how good our technology was."

Also on the horizon was a bit-map–based personal computer—the Apple Macintosh. All previous low-cost personal computers had used character graphics, for which daisy-wheel printers made more sense.

The third piece of luck was the decline in the price of memory chips. "We started this development on an uneconomic basis," Warnock says. "The LaserWriter's first controller needed forty-eight 256K DRAM chips, which up to December of 1984 cost about $30 each. That meant Apple would have had to sell that machine for about $10 000—but its computer cost $2400."

But, with Belleville's and Jobs's strong support, the Adobe team bet that the memory prices would drop. "Sure," says Paxton, "the projections were that the RAM prices were going to drop, but you had to have a very strong stomach to be able to go up to the wall and pray that the door was going to open."

Warnock comments, "Most companies will only deal with present-day technology and known costs. The brilliance of Steve Jobs is that he will say, 'There will be this chip coming out at that price point at that time, and I will design my product to use it.'" And indeed, when the LaserWriter was announced in January of 1985, 256K RAMs cost about $4 each and the printer could be priced at $6995.

Today, some 40 companies have announced their equipment is compatible with PostScript and that their interpreters run faster and cost less than Adobe's version. They cannot offer the same font library, but they say they have fonts and font algorithms as good as Adobe's. At this writing, however, none of these companies had apparently shipped a PostScript clone to a customer, and they reportedly have found it harder to replicate Adobe's work than they had anticipated.

When they do finally ship, and if they can interpret 80 or 90 percent of PostScript programs, Adobe is resigned to facing "good old-fashioned American competition," says Geschke. The company has no patents to defend, only copyrights and trade secrets, so if other companies can reproduce Adobe's technology, it has no legal recourse. "The most we can do is to continue to improve our technology," Geschke says.

What's NeXT?

Adobe's latest technical breakthrough, demonstrated in San Francisco in January, is a version of PostScript that controls images on a computer screen as well as on a printed page. Called Display PostScript, this product is the first to provide device-independent graphics for computer screens.

Display PostScript, like the original PostScript printer protocol, had a nudge from Jobs. His new company, NeXT Inc., Palo Alto, Calif., worked with Adobe to develop it, and it will be the graphics standard for all NeXT's computers. Digital Equipment has already licensed Display PostScript for its DEC Windows workstation architecture. If other major companies follow, Adobe could be well on the way to setting its second standard.

To probe further

Everything a programmer or user might want to know about the PostScript language is provided in "PostScript Language Tutorial and Cookbook" and "PostScript Language Reference Manual," both written by Adobe Systems Inc. and published by Addison Wesley Publishing Co. (New York, 1985). In addition, Adobe periodically publishes a newsletter, "Colophon," with programming tips and news about PostScript products. For a free subscription, write to Colophon, Adobe Systems Inc., 1870 Embarcadero Rd., Palo Alto, Calif. 94303.

Interpress, the page description language from Xerox Corp.'s Palo Alto Research Center (PARC) that preceded PostScript in the laboratory but followed it in the marketplace, is described in the June 1986 issue of IEEE's magazine, *Computer* (pp.72–77). For more information on Xerox PARC, see "Inside the PARC: the 'information architects,'" *Spectrum*, October 1985, p.62.

"Window on PostScript" in *MacWeek*, Feb. 2, 1988, pp.28–29, contains a discussion of competitors' attempts to clone the language. ◆

The IBM System/370 Vector Architecture: Design Considerations

ANDRIS PADEGS, SENIOR MEMBER, IEEE, BRIAN B. MOORE, RONALD M. SMITH, AND WERNER BUCHHOLZ, FELLOW, IEEE

Abstract—**This paper reviews the considerations that shaped the architecture of the IBM System/370 Vector Facility. It summarizes the architectural requirements, decisions, and innovations, and it gives the rationale for the choices that were made. Issues related to vector function, performance, compatibility, migration, and integration with the rest of the System/370 architecture are covered.**

Index Terms—**Arithmetic, array processors, chaining, computer architecture, engineering/scientific applications, IBM 3090, processor design, supercomputers, System/370, vectors.**

I. Introduction

SUPERCOMPUTERS and attached processors provide familiar forms of vector computing (Fig. 1). Early supercomputers like the CDC Cyber 200 Model 205 [1] and the Cray-1 [2] combined vector instructions with a powerful scalar CPU, using a specialized, high-bandwidth main-storage interface to raise the performance of the vector unit. Newer supercomputers (the Cray X-MP [3], Fujitsu VP200 [4], Hitachi S810/20 [5], and NEC SX-2 [6]) have continued on this path. Attached processors like the FPS 164 [7] and the IBM 3838 [8] offer an optional vector capability for scalar systems.[1] They are special-purpose "number crunchers" which typically receive programs and data from the host CPU, compute independently of other host operations, and return the results to the host. Transfers between host main storage and the attached processors occur at channel speeds. Supercomputers deliver extremely high system performance at prices in the $10–22 million range, while individual attached processors have more modest prices. An intermediate approach to vector computing, that of providing an optional, integrated vector capability with a general-purpose scalar system, was adopted for the IBM 3090 Vector Facility (Fig. 2).

This paper describes the considerations that shaped the architecture of the 3090 Vector Facility. The term architecture is used here to denote the attributes of a system as seen by the programmer, that is, the conceptual structure and functional behavior, as distinct from the organization of the data flow and controls, the logical design, and the physical implementation. The vector architecture continues the tradition, started with System/360, that architecture and implementation should be separated, and that one need not imply the other. Thus, although the immediate motivation was to satisfy the requirements of the 3090 Vector Facility, the vector architecture was designed with the possibility in mind of implementation in machines of both higher and lower cost and performance.

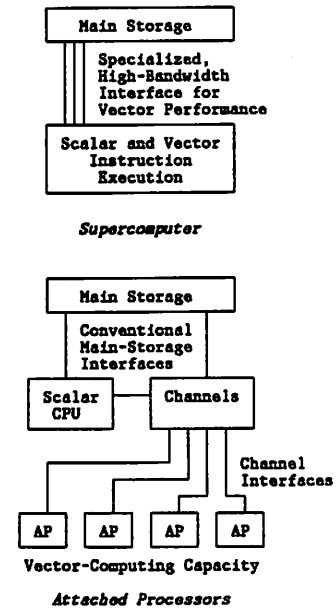


Fig. 1. Two familiar forms of vector computing.

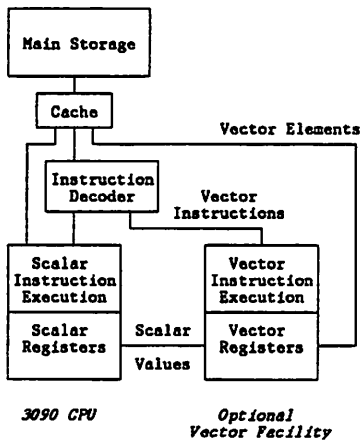


Fig. 2. IBM 3090 CPU with vector facility.

Manuscript received May 15, 1986; revised November 25, 1986.

The authors are with the Data Systems Division, IBM Corporation, Poughkeepsie, NY 12602.

IEEE Log Number 8717220.

[1] Common vector computer terminology is used here. A *scalar* is a single floating-point or binary value. A *vector* is an ordered collection of scalars. *Scalar instructions* (e.g., the System/370 floating-point instructions) are executed by a *scalar CPU*. A *vector unit* executes vector instructions. Vector computer concepts are presented in [25].

Systolic Super Summation

PETER R. CAPPELLO, MEMBER, IEEE, AND WILLARD L. MIRANKER

Abstract—**A principal limitation in accuracy for scientific computation performed with floating-point arithmetic may be traced to the computation of repeated sums, such as those which arise in inner products. We propose the design of a systolic super summer, a cellular piece of hardware for the high throughput performance of repeated sums of floating-point numbers. The apparatus receives pipelined inputs of streams of summands from one or many sources (say as a coprocessor unit in a supercomputer). The floating-point summands are converted into a fixed-point form by a sieve-like pipelined cellular packet-switching device with signal combining. The emerging fixed-point numbers are then summed in a corresponding network of extremely long acumulators (i.e., super accumulators). At the cell level, the design uses a synchronous model of VLSI. The amount of time the apparatus needs to compute an entire sum depends on the values of the summands; at this architectural level, the design is asynchronous. The throughput per unit area of hardware approaches that of a tree network, but without the long wire and signal propagation delay that are intrinsic to tree networks.**

Index Terms—**Floating-point arithmetic, inner product, scientific computation, systolic array, VLSI.**

I. Introduction

A. The Inner Product

FLOATING-POINT arithmetic is fundamental to scientific computation. The four basic floating-point operations $\boxplus$, $\boxminus$, $\boxtimes$, $\boxslash$ have been part of the arithmetic units of digital computers since the 1950's. Today no processor intended for scientific or engineering applications can fail to offer high-performance floating-point arithmetic in some form (intrinsic hardware, coprocessor hardware, microcoded implementation, etc.).

As an approximation to the exact arithmetic operations $+$, $-$, $\times$, $/$ performed on pairs of real numbers, the floating-point operations $\boxplus$, $\boxminus$, $\boxtimes$, $\boxslash$ when performed on pairs of floating-point numbers deliver results which are accurate to the last figure of precision in the computer. Of course, this is predicated on a proper implementation of both a rounding operator $\square$ and of floating-point arithmetic on the computer [1], [2]. However, when floating point operations are combined, even for a computation as elementary as $a \boxplus b \boxplus c$, the relative error of the result may be as large in magnitude as the greatest floating-point number representable in the computer [3]. In addition to the reals, scientific computation employs the so-called higher data types of computation such as complex (numbers), vectors and matrices of real and complex, as well as intervals over all of these. The basic operations for such data types involve expressions such as $a \boxplus b \boxplus c$. Indeed these basic operations involve inner products $\boxdot$ of n-tuples of floating-point numbers. Thus, even with the best possible implementation of the floating-point operations $\boxplus$, $\boxminus$, $\boxtimes$, $\boxslash$, a computer will frequently deliver poor results for the basic operations of scientific computation.

A new theory of floating-point arithmetic [1] has shown one way out of this limitation to accuracy in scientific computation. Basically by including a fifth floating-point operation $\boxdot$, called the inner product, to the conventional four floating-point operations, the full computer accuracy available in floating-point operations on pairs of reals can be provided for all floating-point operations on all of the higher data types of scientific computation. In this sense, the result of such operations are the data types closest to the ideal full precision results.

Given two vectors $x = (x_1, \cdots, x_N)$ and $y = (y_1, \cdots, y_N)$ of floating-point numbers, the operation $\boxdot$ is defined by $x \boxdot y = \square(\Sigma_{i=1}^{N} x_i \times y_i)$. That is, $x \boxdot y$ is that floating-point number which would be obtained by first computing $\Sigma_{i=1}^{N} x_i \times y_i$ in exact arithmetic and then rounding the sum once. We may say that $x \boxdot y$ is that floating-point number which represents the exact inner product $\Sigma_{i=1}^{N} x_i \times y_i$ with an accurracy equivalent to the loss of information represented by a single rounding operation.

Such an operation can be simulated by iterative algorithms [1, ch. 6]. Parallel versions of these algorithms also have been studied [4]. A hardware unit has also been devised for a higher performance implementation [5]. This particular hardware implementation involves a so-called long accumulator. This latter approach has been more or less implemented, by means of microcoded assists, in a commerically available processor (the IBM 4361). The IBM 4361 offers the fifth floating-point operation $\boxdot$.

A normalized floating-point number x (in sign-magnitude representation) is a real number x in the form $x = \sigma m b^e$. Here $\sigma \in \{+, -\}$ is the sign of the number (sign (x)), m is the mantissa (mant (x)), b is the base of the number system in use, and e is the exponent (exp (x)). b is an integer greater than unity. The exponent is an integer between two fixed integer bounds $e1$, $e2$, and usually, $e1 \leq 0 \leq e2$. The mantissa m is of the form $m = \Sigma_{i=1}^{l} d[i] b^{-i}$. The $d[i]$ are the digits of the mantissa numbered in decreasing order of significance. They have the properties $d[i] \in \{0, 1, \cdots, b-1\}$ for all $i = 1(1)l$ and $d[1] \neq 0$. Without the condition, $d[1] \neq 0$, floating-point numbers are called denormalized. The set

Manuscript received May 14, 1986; revised April 8, 1987. This work was supported in part by the National Science Foundation under Grant ECS-8307955.

P. R. Cappello is with the Department of Computer Science, University of California, Santa Barbara, 93106.

W. L. Miranker is with IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598.

IEEE Log Number 8717693.

Programming in VS Fortran on the IBM 3090 for Maximum Vector Performance

Bowen Liu and Nelson Strother

IBM Research Division

The IBM 3090 — a high-performance, general-purpose computer—when enhanced by the Vector Facility and the VS Fortran Compiler Version 2 with vectorization capabilities becomes a powerful tool for large-scale scientific and engineering computations. This article illustrates programming techniques necessary for high performance on the 3090 and demonstrates that VS Fortran programs can achieve near maximum execution rates. The ideas behind these techniques apply to other vector processors as well. Implementation, however, may differ significantly depending on machine organization.

The IBM Engineering and Scientific Subroutine Library (ESSL)[1] is a collection of high-performance mathematical subroutines coded primarily in assembly language using state-of-the-art algorithms tailored to the 3090 Vector Facility. Execution rates delivered by most ESSL routines can be nearly equalled by programming similarly efficient algorithms in Fortran and compiling with the VS Fortran Version 2 compiler.[2]

Fortran program efficiency has practical importance. When Fortran programs perform inefficiently, programmers must resort to special subroutine libraries or assembly language programming for high performance. These solutions are not completely satisfactory. Efficient subroutine libraries, although useful, lack sufficient flexibility; efficient subroutines to perform the desired computation may not exist. Assembly language programs require too much effort to develop, maintain, and modify. Thus, the extent to which the execution power of a computer is realized for scientific and engineering applications often depends on Fortran program efficiency and hence the ability of the Fortran compiler to generate optimal object codes.

General programming techniques for hierarchical storage management can improve 3090 CPU performance up to three times and elapsed time performance up to twenty times for some vector codes.

Any attempt to achieve high performance on a computer must consider its architecture. We review relevant features of the 3090 architecture in the next section. An optimal program on the 3090 Vector Facility must make efficient use of a hierarchical storage system and take advantage of the compound vector instructions. The key programming techniques for managing the storage hierarchy are loop sectioning, loop distribution, and data compaction. The sections "Vector register reuse," "Cache reuse," and "Virtual memory, storage format, and page reuse" show how these techniques can lead to efficient use of the vector registers, the high-speed cache, and the virtual memory system, respectively. The compound vector instructions are discussed in the section "The Multiply-And-Add compound instruction."

Previous work has developed[3-7] and implemented[8] some of these programming techniques and demonstrated their

0018-9162/88/0600-0065$01.00©1988 IEEE

Simulation Modeling and Statistical Computing

Richard E. Nance Editor

Efficient and Portable Combined Random Number Generators

PIERRE L'ECUYER

ABSTRACT: In this paper we present an efficient way to combine two or more Multiplicative Linear Congruential Generators (MLCGs) and propose several new generators. The individual MLCGs, making up the proposed combined generators, satisfy stringent theoretical criteria for the quality of the sequence they produce (based on the Spectral Test) and are easy to implement in a portable way. The proposed simple combination method is new and produces a generator whose period is the least common multiple of the individual periods. Each proposed generator has been submitted to a comprehensive battery of statistical tests. We also describe portable implementations, using 16-bit or 32-bit integer arithmetic. The proposed generators have most of the beneficial properties of MLCGs. For example, each generator can be split into many independent generators and it is easy to skip a long subsequence of numbers without doing the work of generating them all.

1. INTRODUCTION

Random number generators are used in many areas including computer simulation, Monte-Carlo techniques in numerical analysis, test problem generation for the performance evaluation of computer algorithms, statistical sampling, and so on. Despite the large amount of theoretical research already done on this subject, many of the generators currently in use, especially those on the microcomputers, are seriously flawed [15]. Even some recently proposed [3, 20] or evaluated [6, 7] generators have a very weak theoretical justification. The aim of this paper is to propose an efficient way to combine two or more *random number generators* to obtain a new, hopefully better one.

All practical "random number" generators on computers are actually simple deterministic computer programs producing a periodic sequence of numbers that should look "apparently random." A generator is defined by a finite *state space* S, a function $f: S \to S$ and an initial state s_0 called the *seed*. The state of the generator evolves according to the recursion

$$s_i := f(s_{i-1}), \qquad i = 1, 2, 3, \ldots \tag{1}$$

and the *current state* s_i at stage i is usually transformed into a real value between 0 and 1, according to

$$U_i := g(s_i) \tag{2}$$

where $g: S \to (0, 1)$. The *period* of the generator is the smallest positive integer p such that

$$s_{i+p} = s_i \qquad \text{for all } i > \nu \tag{3}$$

for some integer $\nu \geq 0$.

It is well accepted [2, 11] that to obtain a good generator, the choice of f and g should be based on a firm theoretical ground, and before being used for practical applications, the generator should be submitted to a comprehensive set of statistical tests. A good implementation of the generator should be reasonably fast, portable, and use few computer memory words [2, 19].

The most commonly employed generator today is the Lehmer *linear congruential generator* (LCG), for which

$$f(s) = (as + c) \text{ MOD } m; \qquad g(s) = s/m; \tag{4}$$

where the *modulus* m and the *multiplier* $a < m$ are positive integers; and the constant $c < m$ is a nonnegative integer. One usually chooses $c = 0$, in which case the generator is called *multiplicative linear congruential generator* (MLCG) and its *state space* is $S = \{1, 2, \ldots, m - 1\}$.

Integer Multiplication and Division on the HP Precision Architecture

DANIEL J. MAGENHEIMER, MEMBER, IEEE, LIZ PETERS, KARL W. PETTIS, AND DAN ZURAS

Abstract—**In recent years, many architectural design efforts have focused on maximizing performance for frequently executed, simple instructions. Although these efforts have resulted in machines with better average price/performance ratios, certain complex operations and, thus, certain classes of programs which heavily depend on these operations may suffer by comparison. Integer multiplication and division are such complex operations. This paper describes how a small set of primitive instructions combined with careful frequency analysis and clever programming allows the Hewlett-Packard Precision Architecture integer multiplication and division implementation to provide adequate performance at little or no hardware cost.**

Index Terms—**Addition chains, Booth encoding, code generation and optimization, computer architecture, division algorithms, HP Precision Architecture, multiplication algorithms, RISC (reduced instruction set computers).**

I. Introduction

MANY recent general purpose machine architectures (e.g., [19], [16]) have been designed around one fundamental tenet: by concentrating effort on a few frequently executed, simple instructions, average performance can be increased and at the same time hardware costs can be reduced. Many published papers [7], [15] contain instruction distributions ordered by frequency. The literature largely agrees that well-designed memory access instructions and low-overhead branches (both conditional and unconditional) are crucial to any machine design. Arithmetic, Boolean, and procedure call operations are also important.

Further down the list, near the bottom, are the more complex instruction classes: floating point, decimal, large block moves, and integer multiplication and division. Does the relative infrequency of these instructions imply that their implementation is unimportant? Hardly. Machine architects must avoid the tendency to either overdesign these—which results in costly additional (and largely unnecessary) hardware or increased cycle time; or to underdesign them, in which case the instructions become weak points awaiting exercise and abuse by programs and benchmarks which depend on reasonable performance for these functions. The analysis and work which allowed these tendencies to be avoided for the implementation of integer multiplication and division in the Hewlett-Packard Precision Architecture are the subject of this paper.

Manuscript received October 15, 1987; revised March 18, 1988.
The authors are with Hewlett-Packard Company, Cupertino, CA 95014.
IEEE Log Number 8821806.

II. Overview

Uses of Multiplication and Division[1]

Most programs use multiplication and/or division many times, either directly or indirectly. Almost all high-level languages directly support these operations with an explicit operator (e.g., "*" and "/") and almost all support constructs that implicitly require multiplication or division. For example, in C, accessing a two-dimensional array of structures

$$a = \text{structure}A[x][y].b$$

requires two implicit multiplications, namely

$$((x * y_{\max}) + y) * \text{sizeof}(\text{structure}A)$$

(where $y_{\max}$ is the declared upper bound of the second dimension) while

$$\text{diff} = \text{structure}B\_p1 - \text{structure}B\_p2$$

requires a division for the implied operation

$$(\text{structure}B\_p1 - \text{structure}B\_p2)/\text{sizeof}(\text{structure}B).$$

Languages such as Fortran, where matrix ranks can be passed as parameters, may have large numbers of implicit multiplications by variables.

Clever compilers can reduce the number of multiplications in a program by using a technique called "strength reduction." Strength reduction is the practice of replacing multiplications by additions and additions by increments wherever possible, since they are less costly than multiplications. For example,

```
for (i=0; i<10; i=i+1)
    j=j+i * 15.
```

In this simple example, the multiplication by 15 can be replaced by an addition of 15, since the multiplication results form an arithmetic progression.

In many cases, primarily if the induction variable is used in both a subscript expression and a nonsubscript expression, this optimization is difficult or impossible to perform. Furthermore, optimizations may be inadvertently defeated by the use of a global variable as a loop counter or by careless use of goto's. Since programmers are not always aware of these

[1] For the remainder of the text, the references to multiplication and division are of the integer variety.

0018-9340/88/0800-0980$01.00