# Vectorization of Multiple Small Matrix Problems

Nikos Pitsianis* Charles Van Loan

March 31, 1994

**Abstract**

Multiple independent matrix problems of very small size appear in a variety of different fields. In this work, we study the implementation of elementary linear algebra subroutines so as to best use vectorizing compilers and vector hardware for multiple small problem instances. We check the performance improvement over the single-instance optimized codes on different vector supercomputers. We also describe how to automate the transformation of a single-instance linear algebra solver into a multiple instance solver.

## 1 Introduction

We are interested in multiple-instance matrix problems where the matrices are typically small. The task of coding an elementary linear algebra algorithm for a matrix problem of small dimension is not particularly easier than writing code for large problems. Thus, a user is tempted to just make a call to the code from a standard library like LINPACK [DMBS79] instead of writing a multiple-instance version of the algorithm, even though such a code is optimized for large problems. If the number of instances of small problems is small too, then the performance penalty is negligible. But if the number of instances is large and/or such a routine is the critical operation of an iterative algorithm, then the performance degradation cannot be ignored. The standard approach of solving each instance independently / sequentially does not take good advantage of vector facilities because current production compilers do not discover much of the inherent parallelism of a multiple-instance problem. Instead, it is better to group all small independent instances into a single problem of higher dimension and modify the algorithm for solving a single-instance so that *each* step is applied on *all* instances. Not only the indexing calculations are amortized over all

*Partially supported by the Cornell Theory Center which receives funding from members of its Corporate Research Institute, the National Science Foundation (NSF), the Advanced Research Projects Agency (ARPA), the National Institutes of Health (NIH), New York State, and IBM Corporation.

instances, but also there is a big speedup from the use of vector hardware. This approach leads to very efficient use of the vector facilities with frequently dramatic improvement in performance.

## 1.1 Applications

There are many applications that have as a bottleneck the solution of a large number of small matrix problems. Examples come from computer vision, computer visualization and the solution of partial differential equations.

As an example we mention a problem from computer vision. A *feature* in an image is often based on small squares of pixels, *e.g.* 10-by-10. A feature-based vision system requires the tracking of a few hundred features, say 500, throughout a sequence of image frames at 30 frames/second. For each frame and feature, the tracking method proposed in [ST93] requires a Newton-Raphson style search method. Typically, ten iterations are sufficient for the convergence of the search. Each iteration requires the solution of a 2-by-2 linear system. This means that we need to solve 500, 2-by-2 linear systems per iteration for 10 iterations at a rate of 30 frames every second.

A much larger number of 2-by-2 independent symmetric eigenvalue problems arise in the calculation of texture content of an image. After finding the Gaussian of an image [Hor86], a measure of texture content is expressed by the actual and relative sizes of the eigenvalues of the smoothed Gaussian of the image; that means that we need to find the eigenvalues of a 2-by-2 real symmetric matrix for each picture element.

For visualizing second-order tensor fields in computer graphics for scientific visualization, T. Delmarcelle and L. Hesselink [DH93] need to solve a 3-by-3 symmetric eigenvalue problem at every point of the field.

In a numerical method developed by S. B. Pope [Pop92] to integrate the stochastic differential equations that arise in a particle method for turbulent flows, a 3-by-3 symmetric eigenvalue problem has to be solved at every grid point, at every iteration.

## 1.2 Related Work

The Connection Machine Scientific Software Library (CMSSL) provides subroutines that work with either a single or multiple-instances of a problem depending on the *rank* of the input arrays. That is, the user stores all instances of a problem in a multidimensional array and designates the dimensions for solving the problems at the CMSSL subroutine invocation [Thi93a]. From the subroutine point of view, there is an *instance axis*, in addition to the problem axes. For example, for inner products there is a problem axis and an instance axis. For the matrix multiplication, three axes are considered for each matrix, two to describe the matrix and the third to describe the instances. With the axis lengths known at run-time (because the layout of an array is not known at compile time) the loop structures are ordered accordingly, taking into consideration register file size and page faults [JO92].

Although CMSSL offers flexibility in structuring loops according to the input object shape, if the data (object shape) have not been aligned correctly in the first place by the user, utilization of the vector hardware is poor. Another drawback in the CMSSL implementation is the lack of special handling for cases of small dimension. Again it is incumbent upon the user to lay out the data in an effective way. To our knowledge, the only work that specifically addresses multiple small problems is Dubrulle and Shieh [DS88]. They examine different ways to solve small linear systems of equations of identical sizes.

To illustrate the potential of multiple-problem solvers, we modified a number of elementary linear algebra algorithms to handle multiple instances and we present the results of experiments conducted on an IBM 3090VF, IBM ES-9000, CM-5 and Cray-C90. Our goal is to study the effective stacking of small problems in various high-performance environments.

We begin in §2 with a brief introduction in vector hardware, vectorizing compilers and how we make use of them. In §3, we present our approach to multiple problems with the modification of a single-instance triangular backsolver. Next we give the performance comparisons between the single and multiple approaches on the architectures mentioned above. In §5 we discuss a precompiler that can translate general single-instance code to multiple-instance code. Some conclusions are offered in the final section.

## 2 High Performance Computing using Vector Hardware

Vector units perform vector operations such as dot products, saxpy's and elementwise operations significantly faster because of special hardware that exploits the regular sequence of scalar operations and *pipelining*. Vector performance depends upon the length of the vector operands, the amount of data movement, the location of the data in the memory hierarchy, *stride*[1], the number of vector loads-stores and the vector re-use an algorithm makes.

In general, a programmer does not have direct access to vector hardware from a high-level language. It is necessary to write code in such a way that the vectorizing compiler will find the structure and produce code that uses the vector hardware. A vectorizing compiler analyzes only DO loops and certain IF-GOTO loops [IBM90]. Unfortunately, there are a lot of problems that interfere with vectorization:

- recurrence dependencies between statements of a loop,
- backward and exit branches,

[1] Stride is the distance in memory between consecutive vector components.

- use of external functions, nonintrinsic functions and subroutine calls in the body of the loop,
- limitations imposed by the compiler, like use of certain datatypes, or depth of nested loops.

We reduce the complexity of dependencies in loops by expressing our algorithms in terms of matrix and vector operations. In what follows we show how to make these matrix and vector operations suitably long so as to exploit the vector hardware capabilities.

# 3 Stacking Problems

Suppose we have a large number of small linear equation problems to solve of the same dimension. If we solve the problem by solving the small subproblems one by one, then the vector hardware cannot be fully exploited due to their small size. The vector overhead is greater than the speed gain.

Instead, we can stack the small instances together and execute the solver on all instances simultaneously. For example, suppose we have $p$ upper triangular systems of size $n$. Using the backsolver from [GVL89] we can solve our problem with a program[2] that looks like:

```
do k = 1, p
   call backsolve(U(1,1,k), r(1,k), n)
end do

   subroutine backsolve(A, b, n)
   dimension A(n,n), b(n)
   do j = n, 2, -1
      b(j) = b(j) / A(j,j)
      do i = 1, j-1
         b(i) = b(i) - b(j) * A(i,j)
      end do
   end do
   b(1) = b(1) / A(1,1)
   return
```

where `U(1:n,1:n,k)` is the $k^{\text{th}}$ instance of an upper triangular matrix and `r(1:n,k)` the corresponding right hand side $n$-vector. The only part of code that can be vectorized is the inner loop:

```
b(1:j-1) = b(1:j-1) - b(j) * A(1:j-1,j).
```

[2] Throughout this paper, we discuss vectorization in terms of the FORTRAN programming language, although the techniques mentioned can be applied to other languages too.

If $n$ is small, then there is no performance gain from vectorization. To rectify this, we

- Expand the dimension of all the variables except the loop variables (`i, j` and `k`) and the problem-size "constants" (`p` and `n`) making the problem-instance index to be first. That is, local scalars become vectors, vectors become matrices and matrices become tensors.
- Distribute the problem-index loop over all problem steps (loop splitting), and change the order of loops (loop interchange) to make the problem-index loop the innermost.

The above code becomes the multiple backsolve:

```
call m_backsolve(U, r, n, p)

   subroutine m_backsolve(A, b, n, p)
   dimension A(p,n,n), b(p,n)
   do j = n, 2, -1
      do k = 1, p
         b(k,j) = b(k,j) / A(k,j,j)
      end do
      do i = 1, j-1
         do k = 1, p
            b(k,i) = b(k,i) - b(k,j) * A(k,i,j)
         end do
      end do
   end do
   do k = 1, p
      b(k,1) = b(k,1) / A(k,1,1)
   end do
   return
```

To emphasize the vector operations, let us express the inner loops as vectors.

```
call m_backsolve(U, r, n, p)

   subroutine m_backsolve(A, b, n, p)
   dimension A(p,n,n), b(p,n)
   do j = n, 2, -1
      b(1:p,j) = b(1:p,j) / A(1:p,j,j)
      do i = 1, j-1
         b(1:p,i) = b(1:p,i) - b(1:p,j) * A(1:p,i,j)
      end do
   end do
   b(1:p,1) = b(1:p,1) / A(1:p,1,1)
   return
```

By making the problem index the first index of our variables, we achieve unit stride for the vector operations. By distributing the problem-instance loop on each algorithmic step, we achieve rich vector operations.

# 4 Test Cases - Performance Results

Several elementary linear algebra algorithms were modified in order to work on multiple-instances. The transformed algorithms are:

- back substitution (row version),
- back substitution (column version),
- Gauss elimination with partial pivoting,
- QR decomposition with Givens rotations,
- QR decomposition with Householder rotations,
- least squares,
- singular value decomposition with Jacobi rotations,
- symmetric eigenvalue problem.

The benchmarks were executed on different architectures with the following characteristics[3]:

| Computer | Compiler | CPU's used | MFLOPS per CPU | Vector Length |
|---|---|---|---|---|
| IBM 3090-200J VF | VSF 2.5 | 1 | 138 | 128 |
| IBM ES/9000 | VSF 2.5 | 1 | 444 | 128 |
| TMC CM-5 | CMF 2.1 B0.1 | 128 | 32 | 16 |
| Cray Y-MP C90 | CF77 6.0 (6.39) | 16 | 1000 | 128 |

In each case, 1024 instances of problems are processed for instance sizes 2 to 12. For each problem size, 10 executions are done with random data of uniform distribution. The executions are timed, whereby the average time for each case is calculated and the ratio of total time for solving the multiple problems one at a time, over the total time for solving them together is reported (*speedup* or *Performance Gain*). No attempt was made to make the runs on stand alone systems or to optimize our code using special compiler directives or other methods of interactive code compilation-optimization. The 64-bit arithmetic is used in all programs.

We observe in general that the performance gain is maximum when the size of the instances is very small and drops as the size of the instances gets larger. Figures 1 and 2 display the speedup for the different algorithms.

[3]The CM-5 we used has 32 processing nodes (SPARC's) and four vector units per processor. In VU mode the machine reports 32*4 = 128 processors.

Speedup versus Instance Size on the IBM ES/9000

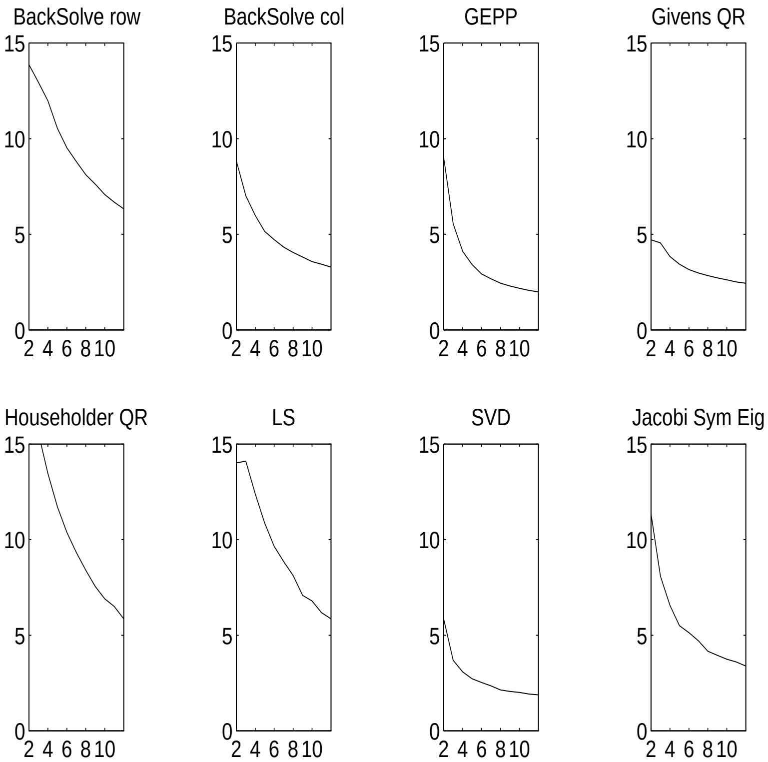


Figure 1: Speedup of some elementary linear algebra algorithms on the IBM ES/9000.

## 4.1 IBM and CRAY results

Figure 1 displays the speedup on the IBM ES/9000. The speedup on the IBM 3090 is similar. The biggest speedup is shown in the Householder QR and the row version of the triangular backsolver. The lowest speedup is for the SVD and the Givens QR. There is no vectorized version of the intrinsic function `SIGN` in the FORTRAN compiler we used, and this was the reason for some poor performance. After exchanging the function call with a conditional assignment, the loops became vectorized and the performance improved.

Figure 2 displays the speedup on the CRAY C-90, the results are consistently good.

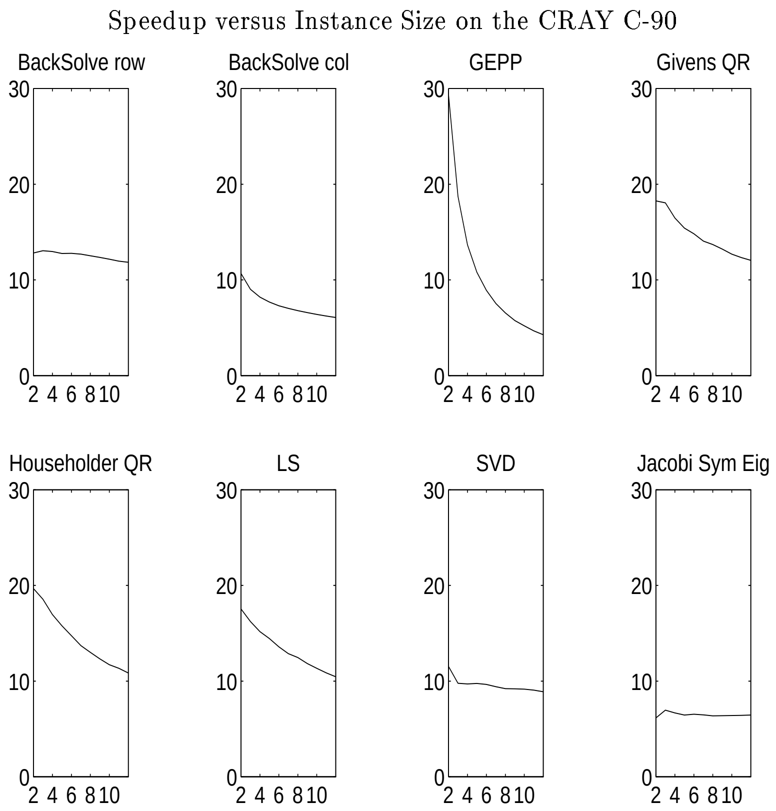


Figure 2: Speedup of some elementary linear algebra algorithms on the Cray C-90.

### 4.2 CM-5 results

The current version of the CMF compiler for the CM-5 did not permit local (nodal) programming in CMFortran and in extension, it did not permit access to vector hardware. Instead we used CDPEAC [Thi93b] and programmed by hand the vector units. The results we obtained are very similar to the results for the other architectures, but since we really had to code in assembly language to achieve them, we violated our principle to avoid machine depending optimizations. As we have been informed, the upcoming version of CMF (version 2.1 or 2.1.1) will allow nodal programming.

## 5 Precompiler Issues

The process of transforming general dense linear algebra code that solves a single-instance problem into a multiple-instance solver can be automated. It is an application of loop transformations [AK87] [PW86], though without the need of extensive data dependency analysis since the loop transformations are known in advance.

Such a precompiler consists of the following modules:

- the parser that reads the source code and creates an Abstract Syntax Tree (AST) [ASU88],
- the unscrambler that eliminates complex and tangled control structure, such as the uses of computed and uncomputed GOTOs, multiple entry points and exchanges them with structured branching constructs such as block IF-THEN-ELSE, CASE and WHILE statements,
- the transformer that changes the single-instance AST to multiple-instance AST,
- the code generator that produces the new source code from the multiple-instance AST.

The most critical and interesting part is the code transformation and we outline a few of the details. We call a variable *global* if and only if it is independent in all the instances, otherwise we call the variable *local*. For example, the variables holding the instance size or the number of instances are global. Any variable that only depends on global variables is global too. A loop-index variable is global iff the initial, final and step values are expressions of global variables and constants. A loop with global index is a *global loop*. Non-global loops require local indices and masking operations.

To transform a single instance code to multiple instance code, we must

- Introduce the *problem-index loop* that enumerates all problem instances.

- Expand the dimension of all the local variables by one. Thus, local scalars become vectors, vectors become matrices and matrices become tensors.
- Distribute the problem-index loop over all problem steps (loop splitting), and change the order of loops (loop interchange) to make the problem-index loop the innermost of the global loops.

What do we do when we have a local loop? Suppose we have the following:

```
anrm = norm(A,n)
do while (anrm > 1e-14)
  ...
  A(p,j) = c*A(p,j) - s*A(q,j)
  ...
  anrm = norm(A,n)
end do
```

where `norm` is a function that calculates some norm of a matrix `A` of size `n`. This loop is local because its termination criterion depends on the value of an instance variable. Following the semantics of the single-instance code we generate the multiple-instance code that looks like

```
call m_norm(A, n, n_prob, anrm)
not_done = any(anrm, gt, 1e-14, n_prob)
do while (not_done)
  ...
  do k = 1, n_prob
    if (anrm(k)>1e-14) then
      A(k,p,j) = c(k)*A(k,p,j) - s(k)*A(k,q,j)
    endif
  enddo
  ...
  call m_norm(A, n, p, anrm)
  not_done = any(anrm, gt, 1e-14, n_prob)
end do
```

where `m_norm` is now a subroutine that calculates the norms of the multiple instances of `A`, `n_prob` stands for the number of problems, `not_done` is a scalar boolean that receives the value of the function `any`.

Vector hardware offers special facilities to *conditionalize* vector operations like the one shown in the body of the loop above. For example, every vector unit on the CM5 has a special register called *vector mask register*, that is used to mask individual Arithmetic and Logic Unit (ALU) and memory operations. At each step of a vector operation, a *context bit* is shifted out of the vector mask register. This bit can be used to prevent the ALU operation, the memory operation, both or neither of them [Thi93b]. In the situation that most but a few elements of a vector are masked out of the operations, the use of vector hardware becomes very expensive. At this point the programmer can use scatter and gather operations that permit the indirect loading/storing of the vector elements that keep the vector pipeline full with no additional time penalty [Wat86].

However, for small sizes of instances, there is no big variation in the number of iterations needed to terminate the loops found in elementary linear algebra algorithms. Moreover, it is not "harmful" to keep iterating on an instance that has already converged. This fact allows us to skip the masking of the instructions inside the body of the local loops like the one on the example code above. That is, we iterate on all instances, until all of them converge, and thus gain in performance, because of the regularity of operations. By doing that though, we violate the semantics of the single instance code.

As it has been reported on a recent study of the performance of automatic vectorizing compilers by D. Levine et al. [LCD91], some compilers did not do particularly well in variable expansion, while it was difficult for many compilers to do loop interchange and only one or two compilers did any vectorization at all in call statements. Thus, by explicitly doing the described changes, it is very difficult for a compiler to fail in vectorizing a multiple-instance problem.

# 6 Conclusions

With this work we verify the intuition that it is faster to solve many small matrix problems together rather than one-by-one. Actually, our multiple small-problem approach works extremely well on supercomputers whose performance depends on vector hardware. This emphasizes the need for general multiple-instance problem solver libraries and for the automatic code transformation from single to multiple instance solvers.

Problem stacking did not give impressive results on RISC workstations because RISC compilers do a very good job in pipelining different arithmetic operations on superscalar architectures. The advantage of amortizing the cost of index calculations among all the instances is superseded by the heavy time penalties resulting by the poor data reuse that is a consequence of stacking the small problems together.

A significant point that needs further analysis is the way to handle algorithms that require a non deterministic number of steps, like the iterative methods for solving linear systems, SVD etc. What should we do when some instances converge before others? We can either keep iterating on the converged instances or use masking and gather-and-scatter operations to keep full vector pipelines, but on how many levels? How often?

We are also considering how to apply these ideas on multiple instance problems of different small sizes, through some form of masking. These problems come up in discretizations on non-regular lattices.

# 7 Acknowledgments

We wish to thank Cornell Theory Center, Northeast Parallel Architecture Center and Pittsburgh Supercomputing Center for providing the access to the IBM ES/9000, CM-5 and Cray C-90.

# References

[AK87] R. Allen and K. Kennedy. Automatic translation of fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4), 1987.

[ASU88] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques and Tools.* Addison Wesley, 1988.

[DH93] T. Delmarcelle and L. Hesselink. Visualizing second-order tensor fields with hyperstreamlines. *IEEE Computer Graphics & Applications*, 17(16):25–33, July 1993.

[DMBS79] J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. W. Stewart. *LINPACK Users Guide.* SIAM, 1979.

[DS88] A. A. Dubrulle and L. J. Shieh. On the solution of many small systems of linear equation with a vector computer. Technical Report G320-3512, IBM Corporation, Palo Alto Scientific Center, 1530 Page Mill Road, Palo Alto, California 94304, 1988.

[GVL89] Gene H. Golub and Charles F Van Loan. *Matrix Computations.* The Johns Hopkins University Press, 1989.

[Hor86] B. K. P. Horn. *Robot Vision.* The MIT Press, 1986.

[IBM90] IBM Corporation. *VS FORTRAN Version 2, Programming Guide, Release 5*, 1990.

[JO92] S. Lennart Johnsson and Luis F. Ortiz. Local Basic Linear Algebra Subroutines (LBLAS) for distributed memory architectures and languages with an array syntax. *The International Journal of Supercomputer Applications*, 6(4):322–350, 1992.

[LCD91] David Levine, David Callaahan, and Jack Dongarra. A comparative study of automatic vectorizing compilers. *Parallel Computing*, 17:1223–1244, 1991.

[Pop92] S. B. Pope. Particle method for turbulent flows: integration of stochastic differential equations. Technical Report FDA 92-03, Cornell University, February 1992.

[PW86] D. Padua and M. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, 1986.

[ST93] J. Shi and C. Tomasi. Good features to track. Technical Report TR 93-1399, Cornell University, November 1993.

[Thi93a] Thinking Machines Corporation. *CMSSL for CM Fortran: CM-5 Edition, Volumes I and II*, June 1993.

[Thi93b] Thinking Machines Corporation. *VU Programmer's Handbook*, cmost version 7.2 edition, August 1993.

[Wat86] T. Watanabe. Design concept for high speed vector and scalar processing: Architecture of the nec supercomputer sx system. *IEEE International Conference on Computer Design*, 2348(1):38–41, 1986.