AN OPTIMIZING PRECOMPILER FOR FINITE-DIFFERENCE COMPUTATIONS ON A VECTOR COMPUTER

John Gary
Scientific Computing Division
National Bureau of Standards, Boulder CO

Lloyd Fosdick
Department of Computer Science
University of Colorado, Boulder CO

ABSTRACT

This paper is concerned with compiler optimization for vector computers which have a relatively long startup time for vector arithmetic. And it describes a precompiler for the Cyber 205[1], a machine which has a long startup time for vector arithmetic. This precompiler does not vectorize Fortran 77 DO loops, instead it vectorizes and optimizes programs written in the explicit array syntax of the proposed Fortran standard (Fortran 8X). This optimization is intended to be most effective for algorithms commonly used in finite difference approximations of partial differential equations. Finite difference schemes frequently evaluate the same expression over the interior of a multidimensional rectangular array. In Fortran 77 the computation is done with a nested set of DO loops, one for each array subscript. Existing compilers for the Cyber 205 only vectorize the innermost DO loop. Our precompiler is able to vectorize the computation over the entire array by generating bit vectors and using conditional evaluation. We describe other optimization techniques as well, but this one yields the greatest benefit.

[1] Certain commercial equipment is identified in this paper in order to adequately describe our results. This identification does not imply recommendation or endorsement by the National Bureau of Standards.

1. Introduction.

In practice the effective speed of vector machines often falls far short of the peak speed. Effective speeds that are 10% of peak speeds, or less, are not uncommon. While this speed loss may be due to properties of an algorithm which prevent vectorization, it is often due to failure of the compiler to vectorize the code optimally, or at all. It is the latter problem, for finite-difference programs written for the Cyber 205, that we address here.

The usual approach to solving this problem is to build a preprocessor that is smarter than the compiler in vectorizing Fortran DO loops. This, for example, is the approach taken by Kuck's group in the Parafrase system [KLW84], and by Brode in the Vast system [Bro81]. Arnold [Arn82] has compared the performance of KAP, a commercial product based on Parafrase, and Vast. Paul and Wilson [Wil78] adopted a different approach that is related to ours. They developed a language called Vectran, close to Fortran but with array constructs, and built a prototype compiler. However, their attention was focussed primarily on the development of a language, and not on the problem of translation to achieve optimal code on vector machines which is our main theme.

In our approach we use a source language that has array operands and operations, thus we entirely avoid the problem of trying to vectorize DO loops. The advantage of this approach in simplifying the vectorization problem has been stressed by Hockney and Jesshope [Jes81, pp 212-215]. The main point is that better optimization can be achieved if the program does not explicitly prescribe an ordering of the computation beyond that which is inherent in the algorithm. Advocates of functional programming languages [WoM84] cite them as the best language for this purpose, but it is unlikely that these languages will be accepted soon by scientists who use vector machines. Since we are interested in providing an efficient tool for improving the effective speed

of the Cyber 205, one that is likely to be accepted now as a practical tool, we have adopted a middle ground on the language turf, namely Fortran 8X [84], and we have restricted our attention to finite-difference calculations.

On the Cyber 205 near optimal code can be obtained if the programmer is willing to devote the effort required to write programs that use the explicit vector operations in Fortran 200, the Fortran dialect used on this machine. Unfortunately, this task is far from easy because low level details of the data structures and the cost of operations on them must be taken into account. Indeed, the programmer is working at a level that is not far removed from assembly language programming. In this paper we describe a precompiler that allows the programmer to avoid these low level details without sacrificing efficiency. The source language for the precompiler is Fortran 77, extended to include a subset of the array expressions proposed for Fortran 8X. Finite-difference computations can be expressed easily and compactly in this source language; and, because array operations are explicit, the precompiler generates efficient Fortran 200 code without great difficulty.

The key to achieving efficiency on the Cyber 205 is the use of long vectors. If the points on a row or column of the mesh in a finite-difference computation are treated as vectors, the vectors are usually not long enough. Moreover, elements of vectors must be in contiguous locations, so the points on a row would have to be moved into contiguous locations, assuming standard Fortran storage order. If, on the other hand, all of the points of the mesh, or all of the points on a plane of a three-dimensional mesh, are used to form a vector then generally the vectors are long enough to assure efficient use of the machine. However, the fact that boundary points are treated differently from interior points, and the requirement that elements of a vector must be stored contiguously make this collective treatment of the mesh points difficult. The precompiler puts the entire mesh, or plane of the mesh, into a vector, automatically taking

care of the problems just mentioned. Thus efficiency is achieved, and the programmer need not be concerned with the details of the data manipulations required to achieve it.

In the following sections we describe some essential features of Fortran 200, F200 for short. Then, with a small example, we discuss the cost of two ways to organize an SOR computation. Besides comparing costs, this example illustrates some of the complexities of the programming effort required when one programs directly in F200. Next, some features of Fortran 8X used in the precompiler source language are described. Then we discuss methods for optimizing finite difference computations. Finally we come to the main subject, the design and operation of the precompiler. We close with a discussion of some results we have obtained and problems that require further attention.

2. Vector operations in F200.

Features of F200 that will be used later are briefly described here. These include vector notation, bit vectors, the WHERE statement, and the built-in procedures for gather, scatter, and compress. Details can be found in the reference manual [CDC85]

An F200 vector is a sequence of contiguously stored elements within an array. The array is declared in the usual way. The vector is specified by giving the name of the array, the position in the array of the first element of the vector, and the length of the vector. For example, given the declaration

DIMENSION A1(1:50), A2(1:50, 1:100)

then

A1(3; 40), A2(41, 2; 60)

are vectors. The position of the first element of the vector is on the left of the semicolon, the length is on the right of the semicolon. In particular

A2(41, 2; 60) = (A2(41, 2), A2(42, 2), ..., A2(50, 3))

Notice that the usual column order is understood. A binary operation on a vector yields a result vector whose elements are obtained by doing the operation pairwise on corresponding elements of the operand vectors.

Elements of a bit array have type BIT in F200. Bit vectors are used to mask the storage of the elements of a vector in assignment statements. Bit vectors also can be used as operands in logical expressions with the logical operators .OR., .AND., and .NOT.; and the value of a vector relational expression may be treated as a bit vector. For example,

.NOT.(A1(1;25) .LT. A1(26; 50)),

which normally would be type LOGICAL (values .TRUE. AND .FALSE.), may be used in expressions as a bit vector.

The WHERE statement is used to control vector assignment statements. Its form is:

```
WHERE (bit_vector)
    sequence_of_vector_assignment_statements
OTHERWISE
    sequence_of_vector_assignment_statements
ENDWHERE
```

In the assignment statements before OTHERWISE the assignment is made for the k-th element only if the k-th bit in bit_vector is 1; after OTHERWISE, only if the k-th bit in bit_vector is 0. The OTHERWISE part may be omitted.

F200 has many procedures, named Q8..., for common vector operations. The ones of particular interest here are "gather", "scatter", and "compress", with names Q8VGATHR, Q8VGATHP, Q8VSCATR, Q8VSCTAP, and Q8VCMPRS. The gather operation collects noncontiguous elements from a vector and puts them into a contiguous sequence; the location of the values gathered is specified by an index list

(Q8VGATHR), or by giving a constant spacing, k, if every k-th element is to be gathered (Q8VGATHP). The scatter operation acts like the inverse of gather. The compress operation deletes elements from a vector; a bit vector specifies elements to be deleted.

3. The cost of vector computations.

Here, with a simple SOR example, we compute the cost of alternate vectorizations of the computation. This example illustrates factors, found in many finite-difference computations, that have affected the design of the precompiler. It also demonstrates the rather tricky nature of writing explicit vector operations in F200.

We consider an SOR computation on an (N+1)-by-(N+1), red-black mesh, with Dirichlet boundary conditions. The mesh, with N = 6, is illustrated in Fig. 3.1. We assume that N is even because the algorithm for the computation is simpler in this case. When N is even the sequence of mesh points, in column storage order, is an alternating sequence of red and black points and so it may be easily treated in a uniform way; when N is odd the sequence does not have this property. We also assume that the point with coordinates (0,0) is red. Since the problem of vectorizing the computation on the black points is the same as for the red points, we confine attention to the red points. The basic computation in Fortran 77 is illustrated in Fig. 3.2.

There are two problems for the vectorization. The red points are not in contiguous locations, and the boundary points must be treated differently than the interior points. These problems can be solved in two ways: by doing the basic SOR update on all of the mesh points at once, treating the entire mesh as a single vector, but storing updated values only at the interior red points; or, by making two vectors, one out of the red mesh points, the other out of the black mesh points, doing the computation with these vectors, and scattering the final results back into the mesh. With the first approach, approximately 50% of the computations are wasted; with the second

approach there are almost no wasted computations (some waste is caused by including boundary points in the vectors), but there is the additional cost of gathering and scattering. The size of the mesh, the number of iterations, and the costs of the vector operations will determine which solution has the minimum cost.

The F200 segment in Fig. 3.3 shows the updating of the red points with a WHERE statement. Here BV is a bit vector defined by the logical expression

BV = (101010...) AND (111...[N-1]...100111...[N-1]...100...)

where the notation in the term on the right indicates a sequence of N-1 1's followed by two 0's followed by N-1 1's and so forth. The effect of the first term in BV is to control the stores so only values on red points are stored, and the effect of the second term is to assure that only values on interior points are stored. Finally, the value of LEN is (N*N-3), the length of the vector extending from the first red point updated to the last red point updated. The cost, in cycles (20 nsec), of this computation is

COST_1 = 6*(50 + (N*N - 3)/2) + 50

where we assume two pipes, the usual configuration on the Cyber 205. The first term is the cost of the six arithmetic operations; the second term is the cost of starting the WHERE. The cost of making BV is negligible.

Now we consider another solution to this problem. The values on the red points are gathered into a vector, R, and the values on the black points are gathered into a vector, B. In particular

R = (A(0,0), A(2,0), ..., A(N,0), A(1,1),..., A(N-2,N)

B = (A(1,0), A(3,0), ..., A(N-1,0), A(0,1), ..., A(N-1,N))

These vectors have length N*(N/2+1). The cost of the gather for R is about [Arn83]

COST_GATH = 42 + 1.4*N*(N/2 + 1)

and for B it is the same. The update of the values on the red points is done with the F200 segment shown in Fig. 3.4. Here the length of the vector operands is LEN1 =

N*(N/2+1) but the length of the vectors actually used in doing the arithmetic is LEN2 = N*N/2-1. The bit vector is given by

BV = 111...[N/2]...10111...[N/2 - 1]...10111...[N/2]...10111...

This bit vector assures that only values on red interior points are updated. The cost of doing the arithmetic in this way is given by

COST_ARITH = 6*(50 + (N*N/2 - 1)/2)

In the final step of this computation the red and black values are scattered into A. Only a segment of R, of length N*N/2 - 1, needs to be scattered and the cost is approximately

COST_SCAT = 75 + 1.2*(N*N/2 - 1).

The cost of scattering B is the same.

In a typical computation a number of iterations of the vector arithmetic would be done for each gather-scatter pair. The total cost of a computation involving a gather, followed by ITER iterations of the vector arithmetic, followed by a scatter is

COST_2 = COST_GATH + COST_SCAT + ITER*COST_ARITH.

We include here only the cost of gathering R, not gathering B even though B is used in the update of the red points. The justification is that the cost of gathering B can be charged to the cost of updating values on the black points. Figure 3.5 shows the ratio (COST_2)/(ITER*COST_1) as a function of N for different values of ITER. The asymptotic approach to 1/2 as N and ITER become large is expected because the length of the vectors involved in the arithmetic for the second algorithm is one-half the length of the vectors involved in the arithmetic for the first algorithm.

A third method is to use a compress operation instead of a gather to form the R and B vectors. There is an advantage in using it if a relatively small number of elements need to be deleted from a long vector. A scatter is still the best way to restore the values into A. An expand followed by a controlled store could be used but would

cost more.

Let us now look at an abstraction of the problem above, and we again consider two forms of the computation, represented by Algorithm A and Algorithm B below:

(Algorithm A)
Repeat the following statement ITER times.
WHERE (bit vector)
Perform N vector arithmetic operations
on operands of length SIZE.
END WHERE

and

(Algorithm B)
Gather NOP vector operands of length F*SIZE from containing array.
Repeat the following statement ITER times.
WHERE (bit vector)
Perform NARITH vector arithmetic operations
on operands of length F*SIZE.
END WHERE
Scatter NOP vector operands of length F*SIZE into containing array.

Here SIZE represents the number of elements in the containing array, and F denotes the fraction of them used in Algorithm B. In the SOR computation F = 1/2. We define

ALPHA = (2*NOP)/(ITER*NARITH)

thus ALPHA is just the ratio of scatter-gather operations to vector arithmetic operations. In the SOR example ALPHA = 1/(3*ITER).

We are interested in the ratio of the cost of Algorithm B to the cost of Algorithm A and how it depends on the parameters SIZE, F, and ALPHA. We assume that the cost of a gather or a scatter is about

COST_GORS = 75 + 2.7*F*SIZE

The value of 2.7 used as the cost per element is the median of the highest cost, 4.05, and the lowest cost, 1.35, as given by Arnold [Arn83]. The reason for the range in costs is memory bank conflicts. In the previous calculation we assumed the minimum cost because there were no bank conflicts in that case. We assume the cost of an arithmetic operation is

COST_ARITH = 50 + (P*F + (1-P))*SIZE/2

where P=0 for Algorithm A, and P=1 for Algorithm B. A small constant term representing the cost of a WHERE is neglected here. Thus the ratio of interest is

RATIO = (ALPHA*COST_GORS + COST_ARITH(P=1)) / COST_ARITH(P=0).

Figure 3.6 shows RATIO as a function of F for different values of ALPHA, and SIZE.

Ideally, an optimizing precompiler would use results of this kind to guide it in generating optimal code. While our precompiler does optimization in generating F200 code and, in particular, does make the choice of when to use scatter-gather operations, it is not now so sophisticated as to take into account the kind of analysis suggested here. But, the important point is that programmers should not have to spend their time trying to take them into account. It is something that can and should be done automatically.

4. Fortran 8X.

Features of Fortran 8X used later are described briefly here. These include array sections, and the statements WHERE and FORALL.

An array section is a subarray of an array, say A, composed of the elements in one or more contiguous rows, columns, etc. of F. Thus, given the declaration

DIMENSION A1(1:50), A2(1:50, 1:100)

then

A1(3:42), A2(48:50, 97:100)

are sections; the first is a one-dimensional, 40-element, subarray of A1, and the second is a 3-by-4 subarray of A2. A binary operation on array sections yields a result array, whose elements are obtained by performing the operation pairwise on corresponding elements of the operands. The operands must be "conformable"; i.e. they must have the same number of elements in corresponding dimensions. It is important to recognize that an array section does not, in general, consist of contiguously stored elements. Therefore, the translation of expressions using array sections as operands into the vector expressions of F200 is more than a trivial change in notation.

The WHERE statement is the same as in F200 with a minor syntactic difference: ELSEWHERE is used instead of OTHERWISE. The FORALL statement is a controlled assignment statement. The form of this statement is

FORALL(subscript_range, bit_vector) assignment_statement

The subscript_range specifies the subscript values used in executing the assignment_statement. The bit_vector controls the assignment_statement. For example,

FORALL(I = 1:50, J = 1:50, I .LT. J) A2(I, J) = A2(J,I)

makes the 50-by-50 subarray of A2, in the upper left corner, symmetric. The bit_vector is optional. If it were omitted in this example, then the subarray would be transposed.

5. Optimization of difference schemes for the Cyber 205

The primary objective is to arrange the computation so that the vectors are as long as possible. The second major objective is the avoidance of unnecessary gathers and scatters. In addition we will describe some techniques which are equivalent to

moving loop invariants from loops in a scalar environment.

5.1. Optimization for an explicit scheme

The technique that seems to yield the best return is the extension of a computation from the interior of a multidimensional array to the entire array by the use of bit vector controlled evaluation. Generally, finite difference computations will involve several variables defined over the same mesh. Therefore these arrays will all be declared with the same size (that is, the same dimensions). The use of bit vectors will not be efficient unless this condition is satisfied. Finite difference computations are typically different in the interior than at the boundary. To obtain a long multidimensional vector, it is necessary to include the boundary points in the vector. Then the interior computation has to skip over these boundary points. Of course, the computation of the boundary operator will be taken over lower dimensional vectors. However, the improvement in the interior computation can be considerable.

The structure of the following Fortran 8X code segment is typical of explicit finite difference schemes. The array U2 contains the mesh values at the new time level. These are obtained from finite difference operators on the array U1 at the old time level.

```
(1) REAL U1(0:NX,0:NY), U2(0:NX,0:NY), A, B
    FORALL(I=1:NX,J=1:NY)  U2(I,J)=U1(I,J) +   &
          A*(U1(I,J)-U1(I-1,J)) + B*(U1(I,J)-U1(I,J-1))
```

(The ampersand denotes line continuation.) Note that the left boundary points where I=0 are not updated in this computation. Therefore the computation is not done over a contiguous set of memory elements. If the boundary points are not included, then the computation uses NY vectors of length NX. This means that the vector startup

cost will be incurred NY times. For a vector of length 20, the startup cost is five times the computational cost. Instead, we can compute over the entire two dimensional array and suppress the computation at the boundary by using a bit vector to control the computation. The bit vector that is needed for this example must have sequences of NX ones separated by a single zero. If BV is the bit vector and LEN contains the length of the vector, then the F200 code for equation (1) is the following.

```
WHERE ( BV(1,1;LEN) )
  U2(1,1;LEN) = U1(1,1;LEN) + A*( U1(1,1;LEN) - U1(0,1;LEN)) &
                B*( U1(1,1;LEN) - U1(1,0;LEN) )
END WHERE
```

This usage of the bit vector may not be efficient if the bit vector must be recomputed each time it is used. Unless an adaptive mesh is used, the mesh will remain constant throughout the calculation. Therefore the bit vector need be computed only once.

5.2. The conjugate gradient iteration

This algorithm requires the multiplication of a matrix by a vector. In the case of a three dimensional problem, each element in the vector corresponds to a mesh point in a rectangular array. For a second order scheme, the matrix will have seven "diagonals". It is convenient to index the matrix and vector by the same subscripts (I,J,K) used to index the mesh. Then the following code segment will multiply one diagonal of the matrix, stored in D(I,J,K,1) by the vector.

```
REAL D(1:NX,1:NY,1:NZ,1:7),U(1:NX,1:NY,1:NZ),W(1:NX,1:NY,1:NZ)
FORALL(I=1:NX-1,1:NY,1:NZ) W(I,J,K) = D(I,J,K,1)*U(I+1,J,K)
```

The efficient implementation of this segment on the vector machine will also require a bit vector, since the boundary point (I=NX) must be included in order to obtain a

contiguous vector.

5.3. Gathers to obtain contiguous vector elements

The elements in the vector U(I,2:31) are not contiguous. Therefore a gather is required in order to use the vector hardware. As discussed in section 2 this gather is rather expensive. If both the vectors U(I,1:30,2:31) and U(I,3:32,2:31) are required, then it is more efficient to gather U(I,1:32,1:32) and use a bit vector to suppress the computation at the boundary points in the gathered vector. This technique is used in the precompiler, and is described in the next section.

5.4. Gathers to reorder arrays in an implicit scheme.

A splitting scheme for the solution of a partial differential equation in three dimensions will involve the parallel solution of tridiagonal systems in each of the three coordinate directions. Gathers can be avoided by a proper ordering of the sweep directions. If the mesh indices are (I,J,K), then the first sweep should be in the K direction. Then the calculation will use vectors with contiguous elements in the (I,J) indices. On the first sweep, the recursion will be in the K direction. At the end of the first sweep, the arrays are reordered (K,I,J) by using scatters. Then the second sweep can be in the J direction with the results scattered back in the (J,K,I) order. Then the last sweep is in the I direction. Then the vectors are always formed from the first two subscripts, so that the vectors can be accessed without gathers. The scatters are likely to be required in any case, so using the sweeps in the proper order and reordering the operands between sweeps can require less computer time.

6. The precompiler.

We call the source language for the precompiler Veclan 205. The precompiler translates programs written in Veclan 205 into Fortran 200. In the next section we illustrate some of the features of the precompiler with a series of small examples. A

detailed explanation of its use can be found in the report "User's Manual for Veclan 205" [GaF85].

6.1. Examples of precompiler input and output.

The first example is shown in Fig. 6.1a. There are six lines prefixed with a mark, the pound sign (#). These marked lines are the only lines that really affect the actions of the precompiler; the unmarked lines pass through it without change and are ignored. There are two classes of marked lines, directives and Fortran 8X lines. This example has three directives (#D, #B, and #F) and three Fortran 8X lines. The directive #D indicates the position where the precompiler is to put declarations for work space; #B indicates where a data statement that initializes run-time stack pointers is to be put; #F indicates the position of the first executable statement. There are some additional directives not illustrated here. Two of the marked lines that are Fortran 8X statements, are also Fortran 77 statements: the declaration "real ...", and "end". All declarations for variables appearing in marked executable statements must be marked, and the end statement must be marked.

The output from the precompiler for this example is shown in Fig. 6.1b. Notice that marked lines in the source become comment lines in the output, excepting the marked "end" which simply has its mark stripped; related statements generated by the precompiler follow the marked line. The precompiler uses three run-time stacks named "ISK000" (for integers), "ESK000" (for reals), "HSK000" (for half precision), and "BSK000" (for bit vectors). All of them are declared, but only one, ESK000, is actually used in this example. The initialization of the stack pointers can be seen in the data statements following the comment line "C#B". (F200 permits initialization of values in labeled common blocks with data statements.) Let us now look at the three lines generated by the precompiler following the marked assignment statement. Since vector division on the Cyber 205 is relatively slow, the vector division by 2.0 has been

changed to vector multiplication by (1.0/2.0). The scalar (1.0/2.0) is put in the stack after verification that there is space for it: this is done in the first two lines. The third line is an F200 array assignment statement, equivalent to the marked Fortran 8X array assignment statement.

Now we turn attention to another example, shown in Fig. 6.2a. It appears to differ very little from the first example but quite different output is given by the precompiler, as shown in Fig. 6.2b. The reason why the output is so different can be seen by comparing the two array assignment statements. In the first example the elements involved in the computation are contiguous in memory, in the second example they are not. The declaration part is the same as in Fig. 6.1b. The statements in the first group of executable statements generated by the precompiler, make an index vector for a gather. Since the index vector only needs to be made once the statements that make it are embedded in a block IF that distinguishes the first entry to this program unit from subsequent entries. This distinction would be relevant if this unit were a function or subroutine, but in this example it is irrelevant since the unit is a main program. In the next group of statements generated by the precompiler we see the two gathers (the F200 function "Q8VGATHR" does a gather operation). The third group is concerned with the optimization discussed in the first example. Finally, in the last group we see the F200 array assignment, followed by a scatter operation (the F200 function "q8vscatr" does a scatter operation).

The precompiler will generate scalar F200, using DO loops instead of array assignment statements, for aid in debugging. The example shown in Figs. 6.3a (the input) and 6.3b (the output) illustrate this. The directive on the first line of Fig. 6.3a signals the precompiler that scalar F200 output is desired.

Our last example is a small segment, typical in form of the kind of code that is used in finite-difference computations. The Veclan 205 segment is shown in Fig. 6.4a,

the F200 output is shown in Fig. 6.4b. We direct our attention to the F200 output. The first group of executable statements generated by the precompiler contains the code for making index vectors for the gather and scatter operations and a compress operation. As in Fig. 6.2b this code is embedded in a block IF. The next group contains the code for gathering the elements of U1 and V1. The third group is where the arithmetic is actually done. It is embedded in a WHERE. A point of particular interest here is an optimization that reduces the number of vector operations. Notice that a scalar (DT/(2.*DX)) is used in this block. Looking back at the Veclan 205 source in Fig. 6.4a it is easily seen that this comes from moving the scalar factor DT to the right. In the fourth group of statements generated by the precompiler the result vector from the arithmetic done in group 3 is compressed and the results scattered into the U2 array. The compression reduces the number of scatters that must be done.

6.2. The design of the precompiler.

Each block of code, terminated by a "# END" statement is processed separately. The input is read and the unmarked lines are passed directly to the output and also to a "listing" file. The latter contains the input with line numbers and diagnostic messages added. The marked lines are processed by a finite state lexical analyzer to produce a string of tokens along with a symbol table. Then an operator precedence parser builds a parse tree for the declarations and vector replacement statements within the program unit delimited by the END statement. Then a single pass over the parse tree is made to generate code for the marked statements. This generated code is written to a file. This output is then merged with the unmarked lines which were copied to the first output file. The result is the final code for the Cyber 205. A listing file and a file containing diagnostic messages is also produced.

6.3. Attributes in the parse tree.

Code for each replacement statement is generated independently of the remaining replacement statements. First the subtree for the replacement statement is traversed to set attributes needed at each node for the code generation. Among these attributes are the rank of the vector at the node, a flag to indicate that the node can be evaluated with bit vector conditioned operations, a flag to indicate that all the vector subscripts have constant upper and lower limits, and several other parameters. Information is passed up the tree in a postfix traverse, so that when the traverse ends, the information is available for the code generation. The information is used to decide how the code is to be generated.

6.4. Code generation for a replacement statement.

The Fortran 8x input statement is translated to a single extended Fortran statement. In addition several other output statements may be generated. If bit vector conditional evaluation is used, then a WHERE block containing the output replacement is generated. The bit vector used in the WHERE block must also be generated. In addition, if any operands in the replacement statement must be gathered, the statements for these gathers must be generated immediately ahead of the replacement statement. If the vector on the left side of the replacement is not contiguous, then the result of the expression on the right must be compressed and then scattered into the vector on the left side. One last complication occurs when the vector is too long (more than 64535 elements). Then a DO loop is generated for at least one of the subscripts. The output replacement statement is contained within this loop. The gathers and the scatter require the generation of gather index lists. The bit vectors and gather index lists are generated in a separate block of code which is placed at the beginning of the subroutine containing the replacement statement.

6.5. The case where gathers are required.

Consider the translation of the following statement.

```
#     REAL A(5,10,10), B(5,0:11,0:11)
#     INTEGER I
#     A(I,1:10,1:10) = B(I,2:11,2:11) - B(I,0:9,0:9)
```

In this case the outer size of the array A on the left is not the same as the outer size of the array references on the right, that is the array dimensions are not the same. However, the vector size of all the terms is the same, as they must be for a correct statement. Since the outer size is not uniform, a bit vector conditional evaluation will not be used. In these cases the precompiler should use the bit vector to compute the right side expression and then scatter the result into the array on the left. However, it not yet able to do that. In this case the two references to the array B will be gathered into two temporary vectors. We name these T1 and T2. An additional temporary, T3, is needed to store the result of the expression on the right so that it can be scattered back into the A array. All three vectors have length 100. The creation of the gather index vector GB1 and the scatter index SA will be given later. The code generated for this replacement is similar to the following.

```
REAL T1(100), T2(100), T3(100)
INTEGER SA(100)
T1(1;100) = Q8VGATHR(B(I,2,2;100),GB1(1;100);T1(1;100))
T2(1;100) = Q8VGATHR(B(I,0,0;100),GB1(1;100);T2(1;100))
T3(1;100) = T1(1;100) - T2(1;100)
A(I,1,1;100) = Q8VSCATR(T3(1;100),SA(1;100);A(I,1,1;100) )
```

If all of the arrays have the same size (in the vector subscripts), then a bit vector conditional evaluation is used. This case is illustrated by the following input.

```
#     REAL A(5,0:11,0:11), B(5,0:11,0:11)
#     INTEGER I
#     A(I,1:10,1:10) = B(I,2:11,1:10) - B(I,0:9,1:10)
```

The full extent of the last two subscripts of B can be gathered into a temporary. Then the two operands on the right side of the replacement can be extracted from this temporary by using bit vector evaluation in the usual way. This means that only a single gather is required instead of the two gathers used in the example above. However, a compress into another temporary T3 is required in order to scatter into the A array. If the compress into T3 is not done, then the boundary elements in T2 would have to be scattered into a "null array" since they can not be placed in A. This could be done to avoid the compress, but the precompiler currently does the compress. We assume that the gather index list is contained in GB. This vector has length 144. The bit vector BV has length 118, and the scatter index vector is SA of length 100. The generated code is

```
REAL T1(0:11,0:11), T2(0:11,0:11), T3(100)
INTEGER GB(144),SA(100)
BIT BV(118)
T1(0,0;144) = Q8VGATHR(B(I,0,0;144),GB(1;144);T1(0,0;144))
WHERE( BV )
T2(1,1;118) = T1(2,2;118) - T1(0,0;118)
END IF
T3(1;100) = Q8VCMPRS(T2(1,1;118), BV(1;118);T3(1;100))
A(I,1,1;100) = Q8VSCATR(T3(1;100),SA(1;100); A(I,1,1;100))
```

sh 2 "Generation of the bit vectors."

The array dimensions and subscript ranges are all constants, known at compile time. We take advantage of this by placing all the code used to generate these vectors for a subroutine in an initialization block at the beginning of the subroutine. This block is executed on the first call of the subroutine and the results saved so that the block is not executed on subsequent calls of the subroutine. The code which is input to the precompiler must contain a line with the "#F" mark which locates the position of this initialization block.

The following example will illustrate the algorithm used to generate the bit vectors. Assume that an array U is declared

```
REAL U(NX,NY,NZ)
```

where NX,NY, and NZ are constants and the reference to the array is

```
U(2:LX+1,2:LY+1,1:LZ)
```

where LX,LY, and LZ are constants. Then (NX,NY,NZ) is the array (or outer) size and (LX,LY,LZ) is the vector (or inner) size.

```
BIT BT(NX,NY,NZ)
BT=B'0'
DO 20 K=1,LZ
DO 20 J=1,LY
  BT(1,J,K;LX) = Q8VMKO(LX,LX;BT(1,J,K;LX))
20    CONTINUE
```

The call to Q8VMKO inserts LX ones into the vector starting at location BT(1,J,K). Note that the lower subscript limits for U do not effect the bit vector. The same bit vector can be used for

U(2:LX+1,2:LY+1,1:LZ) and U(1:LX,1:LY,1:LZ)

6.6. Generation of gather index vectors.

These vectors are also generated within the initialization block. Given the restrictions which apply to the the precompiler input, these vectors are also constant. Consider the following example.

```
REAL U(5,5,10,20,20,2)
INTEGER I,J,K,L
```

The following vector reference will require a gather index vector.

```
... U(I,J,2:9,K,1:18,L) ...
```

This is a two dimensional vector section with 8 elements per column and 18 columns. Successive elements in the column are located 25 memory units apart in the U array. The distance between corresponding elements in successive columns is 5*5*10*20=5000. Thus 25 and 5000 are the two numbers required to generate the gather index vector. The first element to be gathered is U(I,J,2,K,1,L). Successive elements in the first column are obtained by incrementing the location by 25. The first element in each successive column is obtained with an increment of 5000. Note that the gather index does not depend on the values of the I,J,K, or L subscripts. It depends only on the array dimensions and the position of the vector subscripts in the subscript list. The following DO loop can be used to generate the gather index.

```
      INTEGER GU(8,18)
      DO 20 J=1,18
       GU(1,J;8) = Q8VINTL(1+5000*(J-1), 25; GU(1,J;8))
20    CONTINUE
```

The call of the Q8VINTL routine generates a vector of length 8. The first element in this vector is the first argument of Q8VINTL and the increment between elements is the second argument. Therefore the elements in this vector are

1,26,51,76,101,126,151,176,5001,5026, ...

To gather the vector section of U we can use the following Q8 call.

```
TP(1;144) = Q8VGATHR(U(I,J,2,K,1,L;144), GU(1,1;144); TP(1;144))
```

6.7. Vectors which are too long for the vector operations.

If the length of a vector exceeds 65535, then it can not be used as an operand to the vector arithmetic unit. For example, the following reference can not be translated into a single vector operation.

```
REAL U(10,100,80)
U(1;10, 1:100, 1:80) = 0.
```

Instead, the precompiler will start at the leftmost subscript and vectorize as many subscripts as possible without exceeding the 65535 limit.

6.8. Folding out scalar operations.

This is the vector equivalent of the removal of invariant subexpressions from scalar DO loops. For example, a programmer may prefer to write

```
U(2:N-1,2) = U(2:N-1,1) +                                    &
  DELT*U(2,N-1,1)*(U(3:N,1)-U(1:N-2,1))/(2.*DELX)
```

rather than the more efficient

```
CT = DELT/(2.*DELX)
U(2:N-1,2) = U(2:N-1,1) + U(2:N-1,1)*(U(3:N,1)-U(1:N-2,1))*CT
```

The precompiler will perform elementary transformations on the parse tree which are intended to fold out the scalar operations and move the scalars up the tree and to the right. This will transform the first replacement statement above into the more efficient form.

The precompiler will also convert the division of a vector by a scalar to a scalar inversion followed by a multiplication of a vector by a scalar. This is done because vector division is considerably slower than vector multiplication on the Cyber 205.

7. Desirable extensions to the present precompiler.

Perhaps the most desirable addition would be the implementation of all of Fortran 8x to remove the extra precompilation step. However, short of this we list a few extensions which should not require major changes in the structure of the precompiler.

7.1. The FORALL statement.

We have given several examples of the use of this statement. Perhaps the most interesting is (1), below, which models a calculation involving a decomposition of the mesh into "red-black" points. We suspect that most programmers will prefer the FORALL form to the explicit vector notation in those situations where either can be used. The FORALL allows the order of the elements in the vector to be changed fairly easily. Furthermore, the code to do this is easy to understand. The FORALL represents a more gradual departure from Fortran 77. For example, consider the following code to compute the symmetric part of a matrix.

```
(1)        FORALL( I=1:N, J+1:N ) B(I,J) = .5*(A(I,J) + A(J,I))
```

Note that this has a different (and more useful) meaning than

```
PARAMETER( I=1:N, J=1:N)
B(I,J) = .5*(A(I,J)+A(J,I))
```

The subscripts in the body of a FORALL can be general linear combinations of the FORALL indices. For example

```
FORALL( I=2:N-1, J=2:M-1 ) U(I,J) = A(K1*I-K2*J,K3*J-K4*I)
```

This can result in a nonunique definition of elements on the left side which would be an error. In many, perhaps most finite difference cases, the FORALL statements can be optimized in the same way that the explicit vector notation is optimized. The following statement is an exception, as is (1).

```
FORALL( I=2:N ) D(I) = U(I,I-1)
```

A considerable expansion of the precompiler will be required to compile efficient code for the FORALL statement. It will be necessary to recognize those FORALL statements which are equivalent to explicit vector replacement statements and can therefore be compiled using the methods in the present precompiler. The remaining cases will require a more complex algorithm to generate the scatter/gather index vectors.

The FORALL computation can be conditioned by a vector Boolean expression. For example, consider the following SOR iteration over the "black" points in a three dimensional mesh (see section 2).

```
(2)   FORALL( I=1:NX, J=1:NY, K=1:NZ, MOD(I+J+K,2) .EQ. 1 )  &
       U(I,J,K) =  R(I,J,K)                                  &
        -D(I,J,K,1)*U(I+1,J,K) - D(I,J,K,2)*U(I-1,J,K)       &
        -D(I,J,K,3)*U(I,J+1,K) - D(I,J,K,4)*U(I,J-1,K)       &
        -D(I,J,K,5)*U(I,J,K+1) - D(I,J,K,6)*U(I,J,K-1) )/    &
          D(I,J,K,7)
```

The MOD function in the Boolean expression picks out the black points. This FORALL statement would be implemented by use of a bit vector for the Boolean expression. As pointed out in section 2, it is more efficient to compress out the red and

black points and recast the entire SOR algorithm in terms of these vectors. However, the entire SOR method would have to be rewritten. This is probably more than any compiler optimizer is capable of doing.

7.2. Optimization of a block of replacement statements.

The current precompiler generates code for single statements. That is, the optimizer looks only at single statements. If gathers are required, then, in many cases, a major improvement could be obtained by eliminating redundant gathers over the block of statements. For example, consider the following block of three statements.

```
DO(J=2:N)
FORALL( I=2:NX-1, K=2:NZ-1 )                              &
 U(I,J,K,2) = U(I,J,K,1) +  V(I+1,J,K,1) - V(I-1,J,K,1)   &
            + T(I,J,K,1)
 V(I,J,K,2) = V(I,J,K,1) + U(I,J,K+1,1) - U(I,J,K-1,1)    &
            + T(I,J,K,1)
 T(I,J,K,2) = T(I,J,K,1) + U(I-1,J,K-1,1)*V(I+1,J,K+1,1)
END FORALL
REPEAT
```

The above code segment assumes a block FORALL statement which does not exist in Fortran 8x. But the meaning of the statement should be clear. If the statements are treated independently by the precompiler, then each statement would require the gather of the three vectors U, V, and T over the vector indices I and K. If the three statements are optimized together, then the three vectors can be gathered once and used in all three statements. In many finite difference schemes this could be a very effective optimization.

7.3. Elimination of gathers within a recurrence relation.

This optimization would require the inclusion of the DO statement within the group of marked statements. Consider the example below.

```
DO(J=2,NY-1)
 FORALL( I=2:NX-1, K=2:NZ-1 )              &
  W(I,J,K) = W(I,J-1,K) +                  &
   A*(U(I,J-1,K) - 2.*U(I,J,K) + U(I,J+1,K))
REPEAT
```

The current precompiler will gather all three vectors
U(I,J-1,K), U(I,J,K), and U(I,J+1,K)
on each pass through the loop in J. It is necessary to gather only the last vector, the vectors on the J-1 and J levels have already been gathered and used on the previous pass through the J loop.

7.4. Optimization of expressions involving variable dimensioned arrays.

The present version of the precompiler will only optimize expressions if the arrays involved have constant dimensions and the subscript limits of the vectors are also constant. This is a serious restriction. However, we feel that it will be difficult to eliminate this restriction without degrading the efficiency of the generated code. In most cases the mesh size in a finite difference model does not change during a calculation. Therefore the bit vectors and gather indices needed for efficient code on the Cyber 205 need be generated only once (provided sufficient memory is available to store them during the calculation). In order to avoid recomputing these vectors, run time tests would be required and a heap storage scheme instead of a stack scheme might be needed for these vectors. A more difficult problem is the limitation on the vector length. The same code must generated for vectors of size (40,50,50), (10,1000,10), or

(5,100,200). And this code should not be too complex, or contain too many run time tests.

7.5. Inclusion of intrinsic vector functions.

The system library on the Cyber 205 contains vector versions of many of the intrinsic functions such as SQRT, EXP, MOD, etc. Code such as the following should use the vector versions of these functions

```
FORALL( I=1:N ) U(I) = SQRT(X(I)) .
```

There are also functions or operators in Fortran 8x which change the rank of size of vectors. For example the multiplication of a matrix and a vector, or the summation of the elements of a vector.

```
S = SUM(X(1:N))
Y(1:N) = MATMUL(A(1:N,1:N), X(1:N)) .
Y = MATMUL(A, X)
```

Perhaps the APL notation could be included as an alternate to the Fortran 8x, that is

```
S = \+ X
Y = A \+* X .
```

7.6. Inclusion of a subscript increment in the explicit vector notation.

Fortran 8x permits an increment in the vector subscripts. Thus the Fortran 77 code

```
      DO 20 I=1,N,2
20    X(I) = C*Y(I)
```

can be written

```
X(1:N:2) = C*Y(1:N:2) .
```

It will not be difficult to add such increments to the vector syntax used in the precom-

piler.

8. Conclusion.

We have described a precompiler for certain vector algorithms which are found in codes using finite difference approximations to partial differential equations. If these codes involve spatial domains of more than one dimension, then it is difficult to design the codes for efficient operation on a vector computer which needs long vectors.

We have analyzed the cost of some typical finite difference algorithms on the Cyber 205 by parameterizing the algorithm according to storage allocation, the number of floating point operations, and the vector length. This shows the importance of performing the computation so that long vectors are obtained and gathers of data into vectors are avoided. In order to obtain the long vectors in a multidimensional finite difference code, it is necessary to include the boundary points in the vector. However, the finite difference algorithm is different at the boundary than in the interior of the mesh. The Cyber 205 allows the interior calculation to include the boundary points so that a long vector can be used. Then, by using bit vector conditioned evaluation, the incorrect values computed at the boundary are simply ignored. However, this extension to longer vectors through the use of bit vectors will not occur if the algorithm is expressed in standard Fortran 77 and compiled with the current compilers available on the Cyber 205.

We describe a precompiler which will take arithmetic expressions for these multidimensional difference schemes, and generate code for the Cyber 205 which performs this vector extension using bit vectors. These expressions are written in the proposed new Fortran, Fortran 8x, whose syntax allows vectors to be expressed directly, that is DO loops are not required. The precompiler does not vectorize DO loops, instead the algorithms to be translated for the Cyber 205 must be written in vector notation. The current version of the precompiler requires that the array dimensions and vector

lengths be constant. In finite difference terms, this means that the mesh size is constant throughout the computation. This can greatly improve the efficiency of the precompiled code, since the bit vectors and gather indices used to obtain long vectors need be computed only once. We have briefly described the design of the precompiler and given some examples to show the code which it produces. There is much more to be done before finite difference schemes written in Fortran 8x can be efficiently transformed into operations using long vectors on machines like the Cyber 205. For example, the FORALL, WHERE, and IDENTIFY statements in Fortran 8x must be compiled efficiently. Vector valued functions must be efficiently compiled. The techniques used to optimize register assignment for scalar variables can probably be used to avoid multiple gathers of the same vector from an array. We have briefly discussed some of these extensions.

[Arn82] C. N. Arnold, "Performance Evaluation of Three Automatic Vectorizer Packages", *Proc. 1982 International Conference on Parallel Processing*, 1982, 235-242.

[Arn83] C. N. Arnold, "Vector Optimization on the Cyber 205", *Proc. 1983 International Conference on Parallel Processing*, 1983, 530-536. Reprinted in Hwang [1984].

[Bro81] B. Brode, "Precompilation of Fortran Programs to Facilitate Array Processing", *IEEE Computer*, Sept. 1981, 46-51.

[CDC85] CDC, *CDC Cyber 200 Fortran Version 2*, Control Data Corporation, 1985.

[GaF85] J. Gary and L. Fosdick, "VECLAN User's Guide", 308, Dept. of Computer Science, University of Colorado, Sept. 1985.

[Jes81] C. R. Jesshope, *Parallel Computers*, Adam Hilger Ltd., Bristol, 1981.

[KLW84] R. H. Kuhn, B. Leasure and M. Wolfe, "The Structure of an Advanced Retargetable Vectorizer", in *Supercomputers: Design and Applications*, K. Hwang (editor), IEEE Computer Society, 1984, 163-178.

[Wil78] M. W. Wilson, "An Introduction to VECTRAN and its use in Scientific Applications Programming", RC 7287 (#31383), IBM T.J. Watson Research Center, Sept 1978.

[WoM84] M. Wolfe and J. R. McGraw, "A Debate: Retire FORTRAN?", *Physics Today*, May 1984, 66-75. Two articles in one; Kuck and Wolfe on one side of debate, McGraw on the other side..

[84] "Status of Work Toward Revision of Programming Language Fortran.", *ACM SIGNUM Newsletter 19*, 3 (July 1984), 1-42.