

Lecture 13: Vectorization

Professor: S. Lennart Johnsson

TA: Wei Ding

1 Vectorization

Vectorization is a set of techniques for making effective use of pipelined functional units, such as load/store units, adders and multipliers. The idea is to organize a set of operations such that a given operation, or sets of operations, carried out by any of the pipelined functional units can be performed for all elements in the set. For instance, the addition in the loop below can be viewed as the addition of two vectors.

```
FOR I=0 TO N DO
    Z(I)=Y(I)+X(I)
ENDFOR
```

We represent operations on vectors by using the notation $Z(1 : N)$ for a set of N elements. Thus, the vectorized version of the loop above is written

$$Z(1:N)=Y(1:N)+X(1:N)$$

In general, vectors need not consist of all elements of an array. Selecting a subset of elements with a stride is made by using a *triplet* notation, (start: end: stride). The *stride* defines the increment in the index between successive selected elements along the axis. Thus, $(1 : N : 3)$ selects every third element of an array, or elements 1, 4, 7, 10, etc. This triplet notation is used in, for instance, Fortran 90 [6].

Since the essence of vectorization is to arrange computations such that the same operation can be performed on sets of data, it is natural that most techniques for vectorization are focused on loops. Vectorizing compilers perform loop restructuring when necessary and possible to allow the operations in the loop to be vectorized. Due to the data dependencies that may exist, restructuring may or may not be possible. Common dependencies are:

- *True data dependencies* arise from a read-after-write (RAW) operation.
- *Antidependence* arises from a write-after-read (WAR) operation.
- *Output dependence* arises from a write-after-write (WAW) operation.
- A *loop-carried dependence* arises due to looping (i.e., the dependence occurs between iterations). For instance, if a variable is written in one iteration, then read in the next, a read-after-write, loop-carried dependence occurs.

Antidependencies and output dependencies are not true data dependencies. These are, in fact, name conflicts that can be resolved by renaming. The techniques used to restructure loops for vectorization, and increased performance, are:

- Variable renaming
- Statement reordering
- Loop distribution
- Loop reordering
- Scalar expansion
- Variable copying
- Index splitting
- Node splitting
- Loop unrolling
- Loop peeling
- Loop rerolling
- Loop collapsing

We will discuss these techniques next.

1.1 Variable renaming

Consider the following loop [2]:

Example 3.1

```
FOR I=1 TO 100 DO  
[1]   Y(I) = X(I) / S  
[2]   X(I) = X(I) + S  
[3]   Z(I) = Y(I) + S  
[4]   Y(I) = S - Y(I)  
ENDFOR
```

We observe the following:

- There is a RAW dependence from [1] to [3] and from [1] to [4] because of $Y(I)$. This dependence is not loop-carried, since the dependence occurs within the loop. However, in the absence of chaining, the execution of [3] and [4] will not commence until [1] is finished, even though different functional units are used.
- There is a WAR dependence from [1] to [2] because of $X(I)$.
- There is a WAW dependence from [1] to [4] because of $Y(I)$.
- There are no loop-carried dependencies.

The WAR and WAW dependencies can be eliminated by renaming as shown below:

```

FOR I=1 TO 100 DO
[1]   T(I) = X(I) / S
[2]   X1(I) = X(I) + S
[3]   Z(I) = T(I) + S
[4]   Y(I) = S - T(I)
ENDFOR

```

After the loop, the variable X has been renamed $X1$. Renaming does not require a copy. The compiler can simply use $X1$ instead of X in subsequent parts of the code.

1.2 Statement reordering

Consider the loop:

Example 3.2

```

FOR I=1 TO 100 DO
[1]   U(I) = X(I-1)*U(I)
[2]   X(I) = Y(I) + Z(I)
ENDFOR

```

There is a loop-carried dependence in that statement [1] depends upon the value of $X(I)$ computed in statement [2] in the previous iteration. This loop cannot be executed in a vector mode due to this dependence. The code

```

U(1:100) = X(0:99)*U(1:100)
X(1:100) = Y(1:100)+Z(1:100)

```

does **not** yield the same result. However, if the two statements are reordered to

```

FOR I=1 TO 100 DO
[2]   X(I) = Y(I) + Z(I)
[1]   U(I) = X(I-1)*U(I)
ENDFOR

```

Then, the dependence is said to be *forward*, and vectorization is possible. A dependence is *backward* if [1] precedes [2] and [1] depends upon [2], as in the first ordering of the statements. After the statement reordering, the loop can be vectorized as

```

X(1:100) = Y(1:100)+Z(1:100)
U(1:100) = X(0:99)*U(1:100)

```

1.3 Loop distribution

The example we just considered is indeed also an example in which loop distribution was used. The loop

Example 3.3

```

FOR I=1 TO 100 DO
[2]   X(I) = Y(I) + Z(I)
[1]   U(I) = X(I-1)*U(I)
ENDFOR

```

can be rewritten as two independent loops

```

FOR I=1 TO 100 DO
[2]   X(I) = Y(I) + Z(I)
ENDFOR

```

```

FOR I=1 TO 100 DO
[1]   U(I) = X(I-1)*U(I)
ENDFOR

```

which is indeed the basis for the vectorization shown above. Another more complex example of statement reordering and loop distribution is the following

Example 3.4

```

FOR I=1 TO 100 DO
[1]   Z(I) = X(I-2)*Y(I)
[2]   U(I) = Y(I) + Y(I-1)/2
[3]   X(I) = Z(I)+2
ENDFOR

```

Statement reordering and loop distribution yields

```
FOR I=1 TO 100 DO
[1]   Z(I) = X(I-2)*Y(I)
[3]   X(I) = Z(I)+2
ENDFOR
```

```
FOR I=1 TO 100 DO
[2]   U(I) = Y(I) + Y(I-1)/2
ENDFOR
```

The last loop can be vectorized:

```
FOR I=1 TO 100 DO
[1]   Z(I) = X(I-2)*Y(I)
[3]   X(I) = Z(I)+2
ENDFOR
```

$$U(1:100) = Y(1:100) + Y(0:99)/2$$

1.4 Loop reordering

Loop reordering or loop interchange is a frequently used technique for achieving vectorization, and for controlling memory access patterns.

Consider the nested loops

Example 3.5

```
FOR I=1 TO 100 DO
  FOR J=1 TO 100 DO
    X(I,J+1) = X(I,J)*Y(I,J)
  ENDFOR
ENDFOR
```

Clearly, there is a loop-carried dependence of the RAW type. This dependence is in the innermost loop. However, there is no dependence with respect to the index of the outer loop. Thus, interchanging the loop orders will still yield a correct result

```
FOR J=1 TO 100 DO
  FOR I=1 TO 100 DO
    X(I,J+1) = X(I,J)*Y(I,J)
  ENDFOR
ENDFOR
```

Now, the inner loop can be vectorized.

```
FOR J=1 TO 100 DO
  X(1:100,J+1) = X(1:100,J)*Y(1:100,J)
ENDFOR
```

A good example of loop reordering is matrix–vector multiplication, $y = Ax$, where A is a $P \times Q$ matrix, x is a Q –vector, and y is a P –vector. The product, y , is defined as follows:

$$y(i) = \sum_{j=1}^Q (A(i, j) \times x(j)), \quad \text{for all } i \in [1, P]$$

The matrix–vector multiplication can be viewed as the computation of P inner–products, exactly as defined above. Each such inner–product can be computed analogously with the summation of the elements of an array, i.e., $y(1) = 0$, $y(i) = A(i, j) \times x(j) + y(i)$, $i = 1, 1, \dots, P$, which recurses on $y(i)$ (compare with $SUM = SUM + X(I)$, $I = 1, 1, \dots, N$). The inner computation can be expressed as

Example 3.6

```
FOR I=1 TO P DO
  Y(I)=0.
  FOR J=1 TO Q DO
    Y(I)=Y(I)+A(I,J)*X(J)
  ENDFOR
ENDFOR
```

Vectorizing the matrix–vector computation can be made using the same techniques as in summing an array of elements, or computing inner–products. However, matrix–vector products have an additional loop which allows for an additional degree of freedom in vectorization. The matrix–vector multiplication can be viewed as a collection of inner–products with one operand being shared between all inner–products. This latter view is commonly used in parallelizing matrix–vector multiplication. Here, we will use the same fact, but express it in vectorized form

$$y(1 : P) = \sum_{j=1}^Q (A(1 : P, j) \times x(j)),$$

Operations of the form $y(1 : P) = A(1 : P, j) \times x(j) + y(:)$ are known as SAXPY (or DAXPY), which stands for **S**ingle (or **D**ouble) **P**recision **A** \times **X** + **Y**. Using this formulation, the matrix–vector multiplication can be expressed as:

Example 3.7

```

FOR I=1 TO P DO
  Y(I)=0.
ENDFOR
FOR J=1 TO Q DO
  FOR I=1 TO P DO
    Y(I)=Y(I)+A(I,J)*X(J)
  ENDFOR
ENDFOR

```

Note that there is no dependence between successive iterations of the inner loop, and that this loop can be vectorized. The initialization of Y can clearly also be vectorized.

$Y(1:P)=0.$

```

FOR J=1 TO Q DO
  Y(1:P)=Y(1:P)+A(1:P,J)*X(J)
ENDFOR

```

Each iteration in the loop on J involves one scalar, $X(J)$, two input vectors, $A(1 : P, J)$ and $Y(1 : P)$, and one result vector $Y(1 : P)$. The vector operation inside the loop is often programmed very carefully in a high level language, or assembly code due to its importance in many scientific and engineering codes. The `_AXPY` family of routines for different floating-point data types are part of the Basic Linear Algebra Subroutines known as the BLAS [4]. The BLAS was originally conceived as a Fortran 77 library. Strides and vector length are given explicitly, as shown below. In the subroutine call the first argument is the vector length, the second the scalar, the third argument is the vector to be scaled, the fourth the stride for this vector, the fifth argument is the accumulation vector and the last argument its stride.

$Y(1:P)=0.$

```

FOR J=1 TO Q DO
  CALL DAXPY(P, X(J), A(1,J), 1, Y(1), 1)
ENDFOR

```

Note that the `_AXPY` formulation corresponds to an interchange of the loop orders. This is apparent in considering the vector expression in loop form.

$Y(1:P)=0.$

```

FOR J=1 TO Q DO
  FOR I=1 TO P DO
    Y(I)=Y(I)+A(I,J)*X(J)
  ENDFOR
ENDFOR

```

Column major: 111, 211, 311, 411, 121, 221, 321, 421, 131, 231, 331, 431, 112, 212, 312, 412,
122, 222, 322, 422, 132, 232, 332, 432

Row major: 111, 112, 121, 122, 131, 132, 211, 212, 221, 222, 231, 232, 311, 312, 321, 322,
331, 332, 411, 412, 421, 422, 431, 432

Figure 1: Column and row major ordering of an array.

In the inner-product formulation, the loop for the Q -axis forms the inner loop, while the loop on the P -axis forms the outer loop. In the `_AXPY` formulation the loop on the P -axis is the inner loop, while the loop for the Q -axis is the outer loop.

Remark: The subroutine call in the example above exposed the memory mapping of arrays in Fortran 77 (in specifying strides), also known as storage and sequence association. Traditional languages, such as Fortran 77 and C, are based on the notion of a Random Access Memory, i.e., any memory location can be accessed within the same time. Based on this model, a *linearized address space* is sensible. Higher dimensional arrays are converted to one-dimensional arrays using some rule. Fortran 77 uses *column major ordering* of multi-dimensional arrays in converting the array index to a linear sequence. In column major ordering, multi-dimensional arrays are linearized with the leftmost axis having stride one and the rightmost axis having the highest stride. Thus, for an array $X(K, L, M)$ linearized in column major order, element $X(k, l, m)$ is assigned location $k + (l - 1)K + (m - 1)KL$. In *row major ordering* the rightmost axis has stride one and the leftmost axis has the highest stride $m + (l - 1)M + (k - 1)LM$. The two rules for linearizing the index space is illustrated in Figure 1 for an array $X(4, 3, 2)$.

We will discuss the issue of storage and sequence association further in the context of memory systems.

1.5 Scalar expansion

Scalar expansion is a transformation in which a scalar variable inside a loop is promoted to a vector in order to facilitate vectorization. The replacement of the scalar by a vector is made in order to remove dependencies in a way that preserves the semantics of the program. Consider the example

Example 3.8

```
FOR I=1 TO N DO
[1]  X = Y(I)*Z(I)
[2]  U(I) = X+1
[3]  V(I) = X*(U(I)-2)
ENDFOR
```

Statements [2] and [3] both depend upon statement [1], but there are no loop-carried dependencies. By introducing a temporary array $TX(1 : 100)$, the loop can be rewritten as

```

FOR I=1 TO N DO
[1] TX(I) = Y(I)*Z(I)
[2] U(I) = TX(I)+1
[3] V(I) = TX(I)*(U(I)-2)
ENDFOR

```

which can be vectorized as

```

TX(1:100) = Y(1:100)*Z(1:100)
U(1:100) = TX(1:100)+1
V(1:100) = TX(1:100)*(U(1:100)-2)

```

1.6 Variable copy

Variable copy serves the same purpose as scalar expansion. The following example shows how variable copying can allow for vectorization.

Example 3.9

```

FOR I=1 TO N DO
[1] X(I) = Y(I)*Z(I)
[2] U(I) = X(I)+X(I+1)
ENDFOR

```

The antidependence can be removed by introducing a temporary variable $TX(1 : 100)$.

```

TX(1:100) = X(2:101)

```

```

FOR I=1 TO N DO
[1] X(I) = Y(I)*Z(I)
[2] U(I) = X(I)+TX(I)
ENDFOR

```

which can be vectorized as

```

TX(1:100) = X(2:101)
X(1:100) = Y(1:100)*Z(1:100)
U(1:100) = X(1:100)+TX(1:100)

```

1.7 Index splitting

Index set splitting is used when the iteration space for a loop can be split into parts each of which can be vectorized. For instance, it may be the case that one part of the iteration space

results in a true dependence, while the other results in an antidependence, as in the following example.

Example 3.10

```
FOR I=1 TO 200 DO
[1]  Y(I) = X(201-I)+Z(I)
[2]  X(I) = Z(I-1)+2
ENDFOR
```

In this case the dependence is such that the values $Y(1 : 100)$ are computed based on the values of $X(101 : 200)$ prior to the loop, while $Y(101 : 200)$ are computed based on values of $X(1 : 100)$ computed in previous iterations of the loop. Thus, splitting the index set for the loop as follows

```
FOR I=1 TO 100 DO
[1]  Y(I) = X(201-I)+Z(I)
[2]  X(I) = Z(I-1)+2
ENDFOR
```

```
FOR I=101 TO 200 DO
[1]  Y(I) = X(201-I)+Z(I)
[2]  X(I) = Z(I-1)+2
ENDFOR
```

yields two loops each of which can be vectorized.

$$Y(1:100) = X(200:101:-1)+Z(1:100)$$

$$X(1:100) = Z(0:99)+2$$

$$Y(101:200) = X(100:1:-1)+Z(101:200)$$

$$X(101:200) = Z(100:199)+2$$

1.8 Node splitting

In all previous examples we treated each statement as an atomic unit. All transformations aimed at removing or altering dependences while preserving the semantics of the program were applied to whole statements. However, if the dependence preventing vectorization is localized to part of a statement it may be possible to split the statement into two or more statements, and either directly remove the inhibiting dependence, or apply another transformation for vectorization after the splitting of the statement. The statement splitting is referred to as node splitting.

We illustrate node splitting by the following example

Example 3.11

```

FOR I=1 TO 100 DO
[1]  Y(I) = X(I)+Z(I)*U(I)
[2]  X(I+1) = Y(I)*(U(I)-Z(I))
ENDFOR

```

This loop has a true dependence and cannot be vectorized. However, through node splitting some of the operations can be vectorized.

```

FOR I=1 TO 100 DO
    T1(I) = Z(I)*U(I)
    T2(I) = U(I)-Z(I)
[1]  Y(I) = X(I)+T1(I)
[2]  X(I+1) = Y(I)*T2(I)
ENDFOR

```

Through loop distribution this loop can be rewritten as

```

T1(1:100) = Z(1:100)*U(1:100)
T2(1:100) = U(1:100)-Z(1:100)

```

```

FOR I=1 TO 100 DO
[1]  Y(I) = X(I)+T1(I)
[2]  X(I+1) = Y(I)*T2(I)
ENDFOR

```

1.9 Loop Unrolling

Loop unrolling is a technique used to reduce the looping overhead and to reduce memory requests by keeping intermediate results in registers between successive iterations in a loop. In unrolling a loop, copies of the loop body is made. The number of copies of the loop body is equal to the *depth* of the unrolling. Thus, a loop unrolled to depth four will have four copies of the loop body as in-lined code.

For instance, unrolling the loop

Example 3.12

```

FOR I=1 TO N DO
    SUM=SUM+X(I)
ENDFOR

```

to depth four yields the following code:

```

FOR I=1 TO N-3 STEP 4 DO
    SUM=SUM+X(I)+X(I+1)+X(I+2)+X(I+3)
ENDFOR

FOR J=I TO N DO
    SUM=SUM+X(J)
ENDFOR

```

The last loop handles the remainder for the cases when N is not a multiple of four. Thus, the number of iterations in the last loop is one, two, or three in our example, since the unrolling depth is four.

These loops still do not vectorize well, but they illustrate the savings in the number of memory requests. For the main loop, the variable SUM is stored to and loaded from memory once for every four elements being summed, while in the original loop a load/store operation was required for every element of the array X . Thus, ignoring the remainder, there is a savings in the number of load/store operations by a factor of four for intermediate results. Thus, the unrolling reduces the number of memory requests from $3N$ to $N + 2 * N/4 = 1.5N$.

The number of load/store operations in handling the remainder can also be reduced. Though for an unrolling depth of four it may not be of much significance, the time for handling the remainder can be quite significant for large unrolling depths, like 16, for which the number of iterations in the main loop may be few. Unrolling can be applied to the loop remainder as well. The difficulty is that the number of iterations in the remainder is not known until, most likely, run-time. Unrolling the remainder can be made through the use of a case-statement. The number of cases for fully unrolling all cases of the remainder is one less than the unrolling depth. In our example, fully unrolling the remainder results in

```

CASE(N-1-N4)
    CASE(1)
        SUM=SUM+X(N-1)
    CASE(2)
        SUM=SUM+X(N-2)+X(N-1)
    CASE(3)
        SUM=SUM+X(N-3)+X(N-2)+X(N-1)
ENDCASE

```

The following algorithm for summing the elements of an array vectorizes easily, except for the final stages. It is a version of Algorithm 3 in Lecture 2. In order to vectorize the summation we first accumulate every MVL th element of the array X into a vector of length MVL , then sum the values of this vector.

Example 3.13

```

M = N MOD MVL
T(1:M) = X(1:M)
T(M+1:MVL) = 0.
K = N/MVL
FOR I=1 TO K DO
    T(1:MVL)=T(1:MVL)+X(M+1+(I-1)MVL:M+1+I*MVL)
ENDFOR

```

C Add the elements of T

```

SUM=0.
FOR I=1 TO MVL DO
    SUM=SUM+T(I)
ENDFOR

```

The remainder may be treated beneficially by an algorithm such as Algorithm 2. Executed exactly as written, a compiler will generate the following code for the inner loop:

```

Load T(1) – T(MVL) into V0
Load X(M+1+(J-1)*MVL) – X(M+1+J*MVL) into V1
Add V1 to V0
Store V0 into T(0) – T(MVL-1)

```

Every iteration of the inner loop requires two vector loads, one vector add, and one vector store. The number of vector loads and vector stores can be reduced by unrolling the inner loop of the main loop as follows:

```

FOR I=1 TO K-7 STEP 8 DO
    T(1:MVL) = T(1:MVL) + X(M+1+(I-1)*MVL:M+1+I*MVL)
    + X(M+1+I*MVL:M+1+(I+1)*MVL)
    + X(M+1+(I+1)*MVL:M+1+(I+2)*MVL)
    + X(M+1+(I+2)*MVL:M+1+(I+3)*MVL)
    + X(M+1+(I+3)*MVL:M+1+(I+4)*MVL)
    + X(M+1+(I+4)*MVL:M+1+(I+5)*MVL)
    + X(M+1+(I+5)*MVL:M+1+(I+6)*MVL)
    + X(M+1+(I+6)*MVL:M+1+(I+7)*MVL)
ENDFOR

FOR J=I TO K DO
    T(1:MVL)=T(1:MVL)+X(M+1+(J-1)*MVL:M+1+J*MVL)
ENDFOR

```

In all previous examples there was only one loop to which unrolling could be applied. Depending on the data reference pattern, loop unrolling may be applied to any loop, or loops, in a set of nested loops. We illustrate this case for matrix–matrix multiplication.

Example 3.14 [5]

```

FOR J=1 TO R DO
  FOR I=1 TO P DO
    A(I,J)=0.
  ENDFOR
ENDFOR

FOR K=1 TO Q DO
  FOR J=1 TO R DO
    FOR I=1 TO P DO
      A(I,J)=A(I,J)+B(I,K)*C(K,J)
    ENDFOR
  ENDFOR
ENDFOR

```

Note that the loops are ordered such that the product matrix is computed as a sum of outer-products. The loop on the “middle” axis, the Q -axis is outermost. The inner loop forms a $_AXPY$ operation. We first rewrite the loops using the vector notation.

```

FOR J=1 TO R DO
  A(1:P,J)=0.
ENDFOR

FOR K=1 TO Q DO
  FOR J=1 TO R DO
    A(1:P,J)=A(1:P,J)+B(1:P,K)*C(K,J)
  ENDFOR
ENDFOR

```

Note that for the initialization the loop order is such that for a column major ordering of the data, the vectors have elements with stride one. Interchanging the loop order and vectorizing over the R -axis yields a stride of P .

Unrolling the main loop to depth six yields

```

FOR J=1 TO R DO
  A(1:P,J)=0.
ENDFOR

FOR K=1 TO Q-5 STEP 6 DO
  FOR J=1 TO R DO
    A(1:P,J)=A(1:P,J)+B(1:P,K)*C(K,J)+B(1:P,K+1)*C(K+1,J)
    +B(1:P,K+2)*C(K+2,J)+B(1:P,K+3)*C(K+3,J)
  ENDFOR
ENDFOR

```

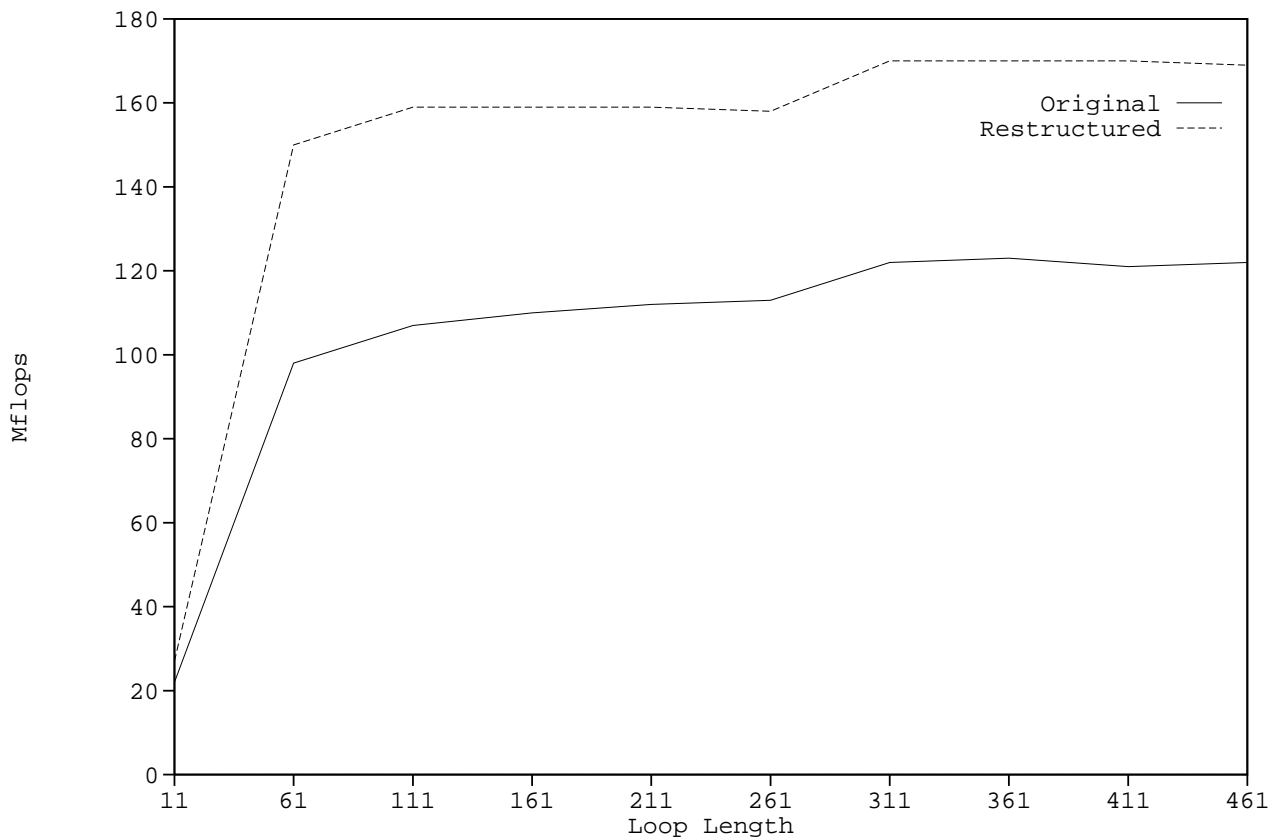


Figure 2: Performance for the matrix multiplication algorithm in Example 3.14 with and without loop unrolling.

```

+B(1:P,K+4)*C(K+4,J)+B(1:P,K+5)*C(K+5,J)
  ENDFOR
ENDFOR

FOR I=KTO Q DO
  FOR J=1 TO R DO
    A(1:P,J)=A(1:P,J)+B(1:P,K)*C(K,J)
  ENDFOR
ENDFOR

```

Unrolling the outer loop reduces the number of load/store operations for the array A by a factor of six in the main loop. In [5] performance data for the original and restructured loops are reported for the Cray X-MP. The performance for the two cases as a function of matrix size with $P = Q = R$ is shown in Figure 2.

Example 3.15

Our final example of loop unrolling is matrix–vector multiplication based on `_AXPY` compu-

tations. After having shown how unrolling can be applied, we will analyze the performance benefits in detail. The vectorized expression is

$Y(1:P)=0.$

```
FOR J=1 TO Q DO
  Y(1:P)=Y(1:P)+A(1:P,J)*X(J)
ENDFOR
```

Unrolling the inner loop to depth four yields

$Y(1:P)=0.$

```
FOR I=1 TO Q-3 STEP 4 DO
  Y(1:P)=Y(1:P)+A(1:P,J)*X(J)+A(1:P,J+1)*X(J+1)
  +A(1:P,J+2)*X(J+2)+A(1:P,J+3)*X(J+3)
ENDFOR
```

```
FOR J=I TO Q DO
  Y(1:P)=Y(1:P)+A(1:P,J)*X(J)
ENDFOR
```

As mentioned previously, the last loop nest can be replaced by a **CASE** statement for complete unrolling.

It is interesting to consider how the index space is traversed for the various loop orderings, vectorization and unrolling. The order in which the index space is traversed is of critical importance with respect to performance in almost any memory system for high performance computers. We will review this issue later in detail.

In the SAXPY (DAXPY) formulation the inner loop is strip mined before execution. Thus, with no unrolling the index space of the matrix is traversed as illustrated in Figure 3a, whereas the unrolled and strip mined code imply a traversal of the index space as shown in Figure 3b.

The loop unrolling in fact introduces a *blocking* of the matrix–vector multiplication. The optimal block size is a function of the number of registers. Maximizing the block size typically yields the maximum performance, but this is not always true.

The blocks are also referred to as *tiles*. The *shape* of the tile is a function of both the number of registers available, and the length of the vector registers. In our example, the tile shape for the majority of the computations is $MVL \times 4$. For an unrolling depth of d the tile shape in the main loop is $MVL \times d$.

When P is not a multiple of MVL , and Q is not a multiple of 4, then some of the *boundary tiles* have a different shape than the *internal tiles*. Strip mining handles the boundary tiles at the top (or bottom) of the matrix, while the case statement (the last loop nest in our examples) handles the tiles on the right boundary.

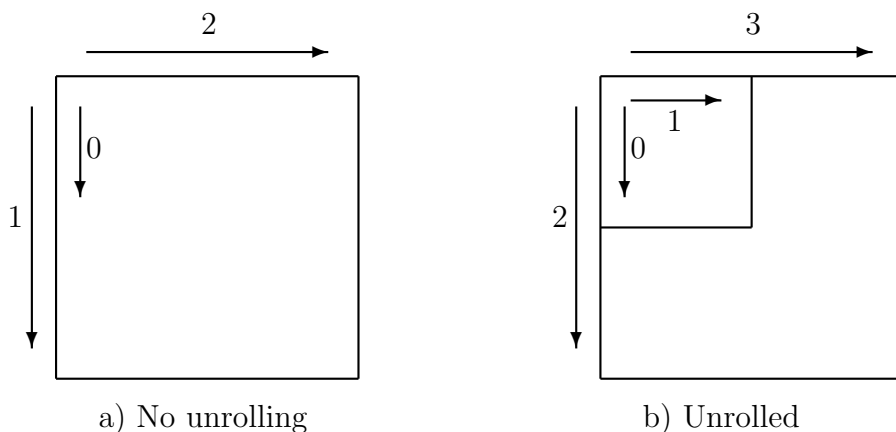


Figure 3: Index space traversal for matrix–vector multiplication.

The optimal loop ordering within the tile, as well as the optimal order in which tiles are processed depends upon both architectural characteristics and the matrix shape [3].

Performance estimates for unrolled matrix–vector multiplication

We will now make a careful estimate of the performance gain due to loop unrolling for the matrix–vector multiplication algorithm in Example 3.15. Unrolling the loops can yield a substantial performance improvement on some architectures. For VP–1, the timing analysis for the inner loop before unrolling (assuming $X(J)$ is already loaded) is

Operation	Cycles	
Load $A(:,J)$	$12+VL$	
Multiply $A(:,J) \times X(J)$	7	<i>(chained with 1st load, overlapped by 2nd load)</i>
Load $Y(:)$	$12+VL$	
Add $A(:,J) \times X(J)$ to $Y(:)$	6	<i>(chained with 2nd load)</i>
Store $Y(:)$	$12+VL$	
Stall for store	4	<i>(store depends on add)</i>
Total	$46+3VL$	

Note that in the above example, the latency of the multiplier is overlapped by the memory latency for the second load. Hence, the start-up time is only $12+12+6+4+12=46$, and the timing expression for the inner loop before unrolling is

$$T_N = 10 + \left\lceil \frac{N}{64} \right\rceil (15 + 46) + 3N.$$

The asymptotic rate, r_∞ , is 0.253×2 operations per cycle. The peak rate is two operations per cycle. The *efficiency* is $r_\infty/2 = 0.253$.

For VP–1, the depth four unrolled loop yields:

Operation	Cycles
Load Y(:)	12+VL
Load A(:,J)	12+VL
Multiply A(:,J)×X(J)	7 (chained with load)
Add Y(:)+A(:,J)×X(J)	6 (chained with multiply)
Load A(:,J+1)	12+VL (overlapped with previous multiply-add)
Multiply A(:,J+1)×X(J+1)	7 (chained with load)
Add Y(:)+A(:,J+1)×X(J+1)	6 (chained with multiply)
Stall for load-mult-add-chain	4 ((J+1)th result depends on Jth result)
Load A(:,J+2)	12+VL (overlapped with previous multiply-add)
Multiply A(:,J+2)×X(J+2)	7 (chained with load)
Add Y(:)+A(:,J+2)×X(J+2)	6 (chained with multiply)
Stall for load-mult-add chain	4 ((J+2)th result depends on (J+1)th result)
Load A(:,J+3)	12+VL (overlapped with previous multiply-add)
Multiply A(:,J+3)×X(J+3)	7 (chained with load)
Add Y(:)+A(:,J+3)×X(J+3)	6 (chained with multiply)
Stall for load-mult-add chain	4 ((J+3)th result depends on (J+2)th result)
Store the result	12+VL (overlapped with previous multiply-add)
Stall for load-mult-add chain	4 (store depends on final result)
Total	104+6VL

Since the result of a load–multiply–add pipeline (i.e., $Y(\cdot)$) is used for the next load–multiply–add pipeline, stalls occur between each such pair of pipelines. Since the multiply–add pipeline is 6+7 cycles, the latency in the load pipeline is fully overlapped. Hence, the start–up time $T_{start} = 12 + 12 + (13 + 4)4 + 12 = 104$. The unrolled loop corresponds to four iterations in the original loop; the timing formula is

$$T_N = 10 + \left\lceil \frac{N/4}{64} \right\rceil (15 + 104) + 6(N/4).$$

The asymptotic rate r_∞ is $2 \times 0.509 = 1.018$ operations per cycle. Unrolling the loop to depth four improves performance by a factor of about two.

A performance increase by a factor of 2 – 4 is common in practice, depending upon architecture and the depth of the unrolling [1, 5]. An unrolling of more than eight is rarely used. For extreme matrix shapes, loop reordering and unrolling may yield considerably larger gains than a factor of 2 – 4.

We conclude by two examples of the performance improvement due to unrolling in matrix–matrix multiplication [5]. The first is a `_AXPY` based matrix–matrix multiplication algorithm for the multiplication of a 4×4 matrix by a $4 \times R$ to produce a $4 \times R$ matrix. The second example is the multiplication of a $P \times 4$ matrix by a 4×4 matrix to form a $P \times 4$ matrix.

Example 3.16

The first code before unrolling is

```
FOR J=1 TO 4 DO
```

```

C(J,1:R)=0.
FOR K=1 TO 4 DO
  C(J,1:R)=C(J,1:R)+A(J,K)*B(K,1:R)
ENDFOR
ENDFOR

```

After unrolling both the K and J loops we have

```

FOR J=1 TO 4 DO
  C(J,1:R)=0.
ENDFOR

```

$$\begin{aligned}
C(1,1:R) &= C(1,1:R) + A(1,1)*B(1,1:R) + A(1,2)*B(2,1:R) \\
&\quad + A(1,3)*B(3,1:R) + A(1,4)*B(4,1:R) \\
C(2,1:R) &= C(2,1:R) + A(2,1)*B(1,1:R) + A(2,2)*B(2,1:R) \\
&\quad + A(2,3)*B(3,1:R) + A(2,4)*B(4,1:R) \\
C(3,1:R) &= C(3,1:R) + A(3,1)*B(1,1:R) + A(3,2)*B(2,1:R) \\
&\quad + A(3,3)*B(3,1:R) + A(3,4)*B(4,1:R) \\
C(4,1:R) &= C(4,1:R) + A(4,1)*B(1,1:R) + A(4,2)*B(2,1:R) \\
&\quad + A(4,3)*B(3,1:R) + A(4,4)*B(4,1:R)
\end{aligned}$$

The performance measured on Cray X-MP is shown in Figure 4.

Example 3.17

The second example from [5] is an inner-product algorithm for matrix multiplication.

```

FOR I=1 TO P DO
  FOR J=1 TO 4 DO
    C(I,J)=0.
    FOR K=1 TO 4 DO
      C(I,J)=C(I,J)+A(I,K)*B(K,J)
    ENDFOR
  ENDFOR
ENDFOR

```

After unrolling both the K and J loops we have

```

FOR J=1 TO 4 DO
  C(1:P,J)=0.
ENDFOR

```

$$C(1:P,1) = A(1:P,1)*B(1,1) + A(1:P,2)*B(2,1)$$

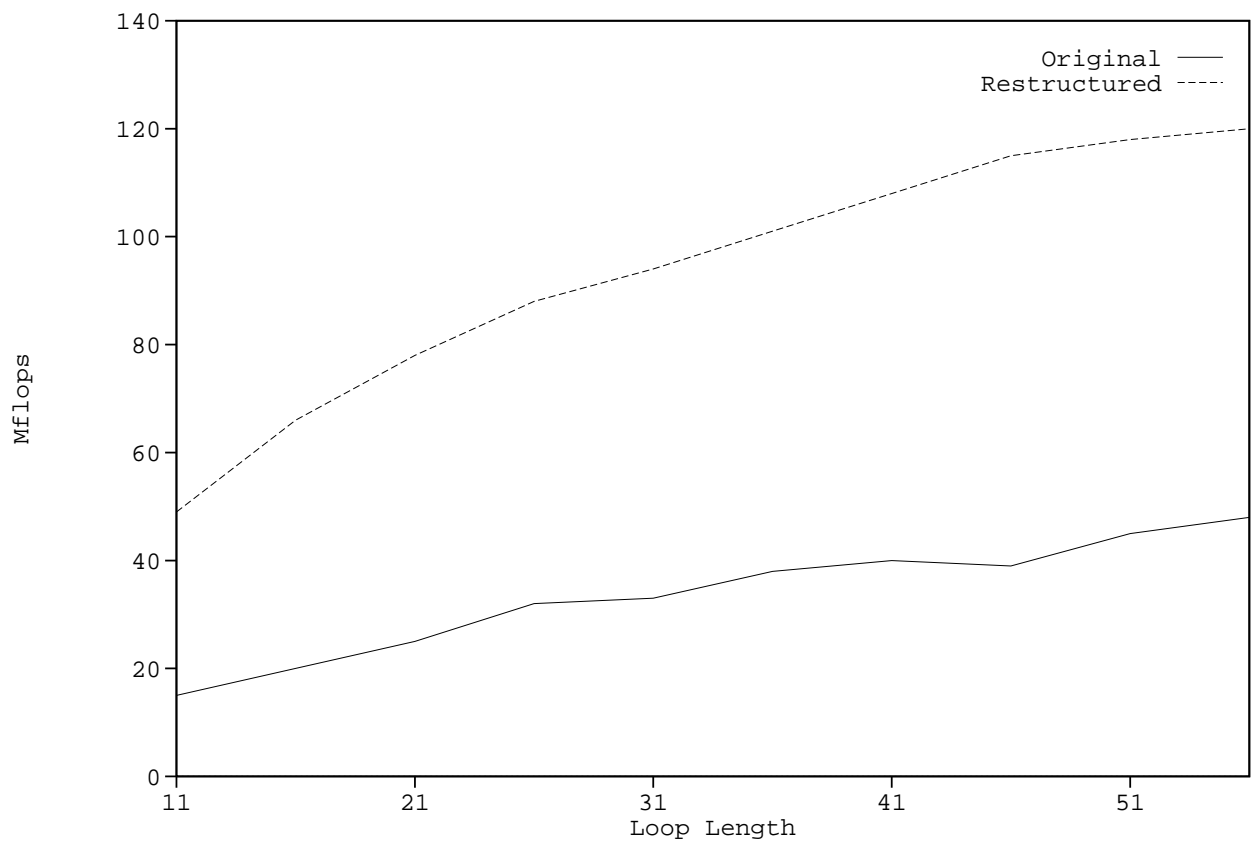


Figure 4: Performance for the matrix multiplication algorithm in Example 3.16 with and without loop unrolling.

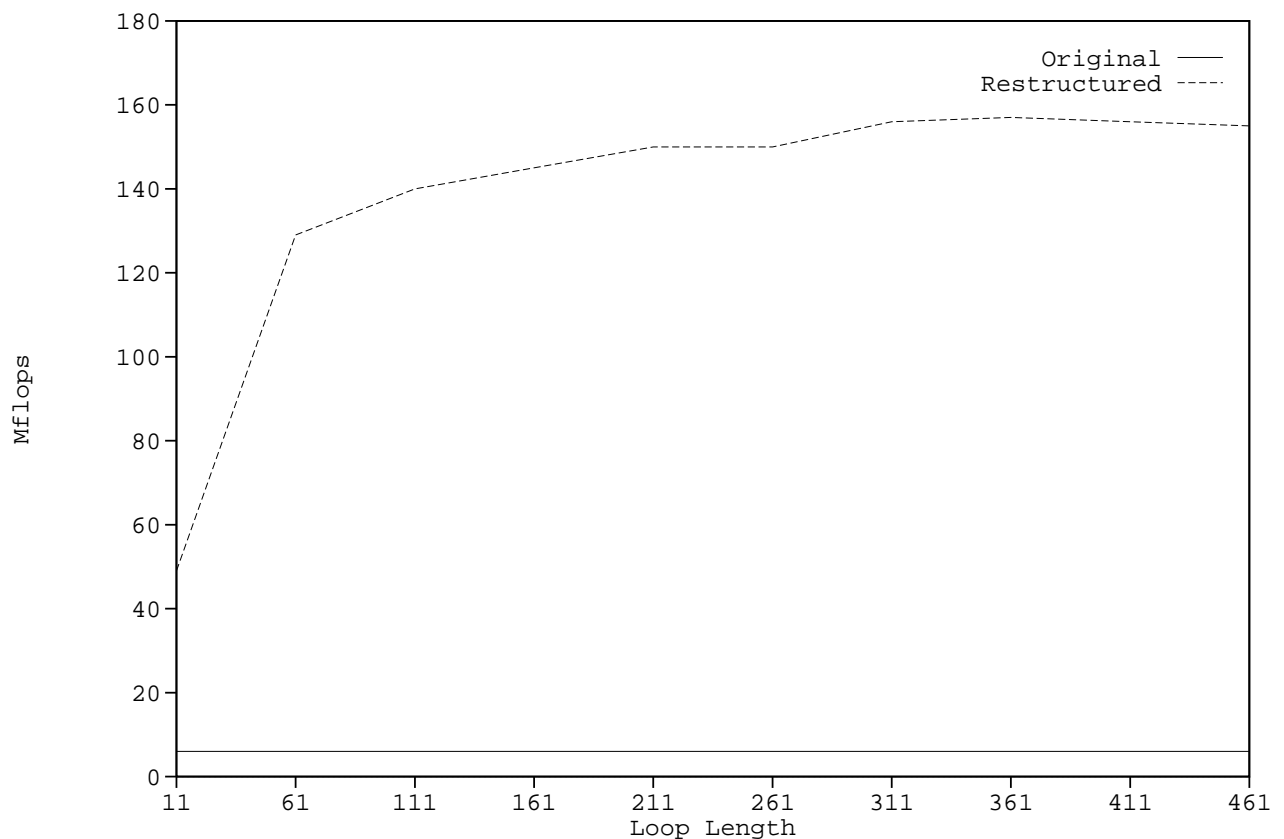


Figure 5: Performance for the matrix multiplication algorithm in Example 3.17 with and without loop unrolling.

$$\begin{aligned}
 &+A(1:P,3)*B(3,1)+A(1:P,4)*B(4,1) \\
 C(1:P,2) &=A(1:P,1)*B(1,2)+A(1:P,2)*B(2,2) \\
 &+A(1:P,3)*B(3,2)+A(1:P,4)*B(4,2) \\
 C(1:P,3) &=A(1:P,1)*B(1,3)+A(1:P,2)*B(2,3) \\
 &+A(1:P,3)*B(3,3)+A(1:P,4)*B(4,3) \\
 C(1:P,4) &=A(1:P,1)*B(1,4)+A(1:P,2)*B(2,4) \\
 &+A(1:P,3)*B(3,4)+A(1:P,4)*B(4,4)
 \end{aligned}$$

The performance measured on Cray X-MP is shown in Figure 5.

1.10 Loop collapsing

Loop collapsing is an operation that is used to both reduce loop overhead and to increase vector length. For instance, in the initialization loop nest in our matrix–matrix multiplication example,

Example 3.18

```

FOR J=1 TO R DO
  FOR I=1 TO P DO
    A(I,J)=0.
  ENDFOR
ENDFOR

```

the two loops can be collapsed into one loop of length PR , which in vectorized form is $A(1:PR)=0$.

1.11 Loop peeling

We conclude our discussion of loop restructuring techniques by illustrating loop peeling. For some loops, removing some of the iterations from the loop may allow vectorization of otherwise nonvectorizable loops. This removal of some iterations from the loop is known as loop peeling. Consider,

Example 3.19

```

FOR J=1 TO R DO
  FOR I=1 TO P DO
S1    X(I,J) = Y(I,J)+3*Y(I-1,J)
      IF J<2 THEN GOTO S4
      IF J<N THEN GOTO S3
S2    Y(I,J) = X(I,J)
S3    Y(I-1,J) = Z(I,J)
S4    Z(I,J) = X(I,J)
      ENDFOR
ENDFOR

```

By peeling off the first and last iteration for J , we get

```

FOR I=1 TO P DO
  X(I,1) = Y(I,1)+3*Y(I-1,1)
  Z(I,1) = X(I,1)
ENDFOR
ENDFOR

```

```

FOR J=2 TO R-1 DO
  FOR I=1 TO P DO

```

```

S1      X(I,J) = Y(I,J)+3*Y(I-1,J)
S3      Y(I-1,J) = Z(I,J)
S4      Z(I,J) = X(I,J)
      ENDFOR
ENDFOR

```

```

FOR I=1 TO P DO
S1  X(I,R) = Y(I,R)+3*Y(I-1,R)
S2  Y(I,R) = X(I,R)
S3  Y(I-1,R) = Z(I,R)
S4  Z(I,R) = X(I,R)
ENDFOR

```

Loop distribution followed by vectorization yields the final program

```

X(1:P,1) = Y(1:P,1)+3*Y(0:P-1,1)
Z(1:P,1) = Y(1:P,1)

```

```

X(1:P,2:R-1) = Y(1:P,2:R-1)+3*Y(0:P-1,2:R-1)
Y(1:P,2:R-1) = Z(1:P,2:R-1)
Z(1:P,2:R-1) = Y(1:P,2:R-1)

```

```

FOR I=1 TO P DO
S1  X(I,R) = Y(I,R)+3*Y(I-1,R)
S2  Y(I,R) = X(I,R)
ENDFOR
Y(0:P-1,R) = Z(1:P,R)
Z(1:P,R) = X(1:P,R)

```

2 Indirect Addressing

Irregular accesses to data structures are required in many computations. Sparse matrix computations are good examples. Irregular accesses in such cases is often supported through an index array, i.e., an array that contains the addresses of the elements to be accessed in another array. Index arrays may be shared between several data arrays. Indirect loads are often referred to as *gather* operations, while indirect stores are known as *scatter* operations.

Gather: $Y(I)=X(K(I))$

Scatter: $Y(K(I))=X(I)$

For a gather operation, the element to be fetched from X is determined by an element of K . Conversely, in a scatter operation, an element of K determines where an element of X is stored.

K is an *index array*. *Gather* and *scatter* operations are supported by machine instructions in some computers:

Example 3.20

Consider the following loop:

```
FOR I=1 TO N DO
    A(IA(I))=B(IB(I))+C(IC(I))
ENDFOR
```

With vectorized indirect addressing, a possible instruction sequence is as follows:

Operation	Cycles
Load IB into V0	12+VL
Load IC into V1	12+VL
Load B indirect into V2 using V0	12+VL
Load C indirect into V3 using V1	12+VL
Add A and B,(V4←V2+V3)	6+VL
Stall	4
<i>(add must wait for load)</i>	
Load IA into V5	12+VL
Store A indirect using V5	12+VL
Total	82+7VL

The timing for VP–1 without chaining is

$$T_N = 10 + \left\lceil \frac{N}{64} \right\rceil (15 + 82) + 7N.$$

The asymptotic rate, r_∞ , is 0.118 operations per cycle.

Note that due to the indirect addressing there are five loads and one store. With direct loads and stores, two loads and one store suffice. Thus, with indirection vectors stored in memory, the number of memory references has doubled because of the indirection. For an architecture with a single path to memory, such as the VP–1, indirection can degrade the performance by as much as a factor of two. If the indirect vector is shared between all arrays in our example, i.e.,

Example 3.21

```
FOR I=1 TO N DO
    A(K(I))=B(K(I))+C(K(I))
ENDFOR
```

then only a single extra load is required due to indirection. The number of memory references due to indirection is increased from three to four.

In practice, the performance degradation due to indirection is often much greater than this simple analysis predicts. The reason is the memory system and the mapping of the index space to the memory system. Most memory systems are designed to support regular accesses at close to their peak bandwidth, while irregular accesses often severely degrades the achieved memory bandwidth. Thus, indirection with index arrays stored in memory not only increases the number of memory references, but the memory references to the data array may also be substantially slower. We will examine this issue further in a later lecture.

For VP-3, we can take advantage of the increased load/store bandwidth, and if we allow chaining, the instruction sequence for Example 3.20 would be as follows:

Operation	Cycles
Load IB into V0 and IC into V1	12+VL
Load IA into V5	12+VL
Load B indirect into V2 using V0	12+VL
Load C indirect into V3 using V1	
Add B and C (V4←V2+V3)	6
<i>chained with load</i>	
Store A indirect using V5	12
<i>(chained with load/add)</i>	
Total	54+3VL

The timing for VP-3 with chaining is

$$T_N = 10 + \left\lceil \frac{N}{64} \right\rceil (15 + 54) + 3N.$$

The asymptotic rate r_∞ is 0.245 operations per cycle. The asymptotic rate of VP-3 with chaining is 2.1 times that of VP-1 without chaining. Since there are five loads required but only two load pipes, a total of three sequential loads are required. However, if two index arrays were required, then two parallel loads would suffice for a total of four loads, and the VP-3 architecture would be more efficiently used.

The VP-3 architecture is actually more capable than many existing computers in that most of them only allow one index vector for a load. Thus, unless all concurrent loads use the same index vector, the loads are made one at a time.

3 Vectorizing Conditionals

So far, all of the loops analyzed were unconditional loops. However, in practice, the bodies of loops often contain conditional expressions. Consider the following loop:

Example 3.22

```
FOR I=1 TO N DO
  IF (A(I).NE.0) THEN
```

```

        A(I)=A(I)-B(I)
    ENDIF
ENDFOR

```

As the loop is written, it is necessary to inspect each element of A to determine whether the subtractions should be performed. Clearly, this conditional execution will break a pipeline, preventing vectorization.

In order to allow for vectorization of loops of this type all vector architectures allow at least arithmetic units, but often all functional units to operate in *Masked* mode. In such a mode the operation may either not be performed at all, or more commonly, performed but the result is ignored, depending upon the value of the mask for the current set of operands. In the latter case, the time to perform the operation is the same as if the operation was performed on all elements of the vector.

Some vector architectures also have a *compressed-index* mode. In this mode, a vector register is generated containing the indices for which the condition is true. The register is then used to perform indirect address operations on the arrays subject to the conditional expression used to generate the vector register content. The operations are performed only on the indices for which the condition is true. Load/store operations are indirect, while arithmetic and logic operations can be performed without indirection on vectors of a length equal to the number of elements for which the condition is true.

3.1 Mask mode

In mask mode, the first step is to compute the mask. It consists of one bit for each element of a vector register. If the condition is true for a location in the vector registers, then the corresponding mask bit is 1, otherwise 0. For masked vector operations the mask is loaded into the *Vector Mask Register*, VMR. The functional units that can operate in masked mode reads this register.

Using a mask register, the loop in Example 3.22 can be carried out as follows:

```

Load Vector A into V1
Load Vector B into V2
Set F0 to 0.0
Set VMR to 1 if V1(i).ne.F0, 0 otherwise
Subtract V2 from V1 under mask
Set the vector mask to all 1's
Store the result in A

```

The timing analysis for a segment of the inner loop is:

Operation	Cycles
Load A	12+VL
Load B	12+VL
Set F0 to 0.0	1
Set VMR	VL
Subtract B from A under VMR	6+VL
Store the result	12+VL
Stall	4
<i>(store must wait for subtract)</i>	
Total	47+5VL

For the code segment above, vectorization is achieved. However, it is assumed that the operation is carried out even when the mask is zero (i.e., a correct result is obtained, but there is no savings in time). The total number of cycles for VP-1 and a vector of length N is $10 + \left\lceil \frac{N}{64} \right\rceil (15+47) + 5N$. The asymptotic rate is 0.168 operations per cycle. Though this rate is not high, it may still be significantly faster than the time for the scalar evaluation of the expression. (In scalar mode, each load and store would require 12+1 cycles.)

One implicit assumption made in the instruction sequence above was that loads and stores were unconditional. This is apparent in preparing for the final store in which all mask bits are set to true. Conditional stores may be quite expensive in this case, since in fact what might be required is a read, modify, write operation on the array into which data is stored. This is apparent in the following example.

Example 3.23

```

FOR I=1 TO N DO
    IF (X(I).GE.0) THEN
        Y(I)=A(I)+B(I)
    ENDIF
ENDFOR

```

The corresponding instruction sequence may be

```

Load Vector X into V1
Set F0 to 0.0
Set VMR(i) to 1 if V1(i).ge.F0, 0 otherwise
Load Vector A into V2
Load Vector B into V3
Add V2 and V3 and store in V4 (no mask)
Load Vector Y into V5
Copy V4 under mask into V5
Store the result in Y (no mask)

```

Another drawback with masked operations performed on the entire vector is that exceptions that apply to masked elements must be handled with great care. For instance, a conditional that is intended to prevent a divide by zero will be carried out if only stores are masked out but the divide instruction is executed, as in Example 3.24.

Example 3.24

```

FOR I=1 TO N DO
  IF (X(I).NE.0) THEN
    Y(I)=A(I)/X(I)
  ENDIF
ENDFOR

```

3.2 Compressed-index mode

In the compressed index mode, operations are only performed on elements for which the condition is true. Returning to Example 3.22, only elements of A for which $A(i).ne.0$ are operated upon, and only the required elements of B are loaded. To accomplish this, we will make use of both the vector mask register (VMR) and the vector length register (VLR) as follows:

```

Load Vector A into V1
Set F0 to 0.0
Set VMR to 1 if V1(i).ne.0, 0 otherwise
Create index vector under mask in V2
Count number of non-zeroes in VMR & set VLR
Set VMR to all 1's
Load A into V3 indirect using V2 for indirection
Load B into V4 indirect using V2 for indirection
Subtract V4 from V3 (no mask)
Store V3 indirect using V2 for indirection

```

The timing analysis for a segment of the inner loop is:

Operation	Cycles
Load A	12+VL
Set F0 to 0.0	1
Set VMR	VL
Create index vector	VL
Count non-zeroes and set VLR	1
Reset vector mask register	1
Load A indirect	12+f×VL
Load B indirect	12+f×VL
Subtract B from A	6+f×VL
Stall	4
<i>(subtract must wait for load)</i>	
Store the result indirect	12+f×VL
Stall	4
<i>(store must wait for subtract)</i>	
<hr/> Total	<hr/> 65+(3+4f)VL

The value f is the fraction of the elements of A which are non-zero. The total time is $10 +$

$\lceil \frac{N}{64} \rceil (15 + 65) + (3 + 4f)N$. The asymptotic rate in this case is $1/(4.266 + 4f)$ operations per cycle. Compared to the previous case, this latter approach is faster when $4f + 4.266 < 5.97$ or $f < 0.426$. Hence, because of the overhead in operating only on nonzero elements, a net gain occurs only when fewer than 43% of the elements are non-zero. When $f = 1.0$, the asymptotic performance is 0.121 operations per cycles, which is only 72% of the performance of the first approach. The two approaches are compared in Table 1 below.

f	Mask Operation (Method 1)	Select Elements (Method 2)	Relative Performance (Method 2 vs. Method 1)
0.0	0.168	0.234	1.40
0.1	0.168	0.214	1.28
0.2	0.168	0.197	1.18
0.3	0.168	0.183	1.09
0.4	0.168	0.170	1.02
0.5	0.168	0.160	0.95
0.6	0.168	0.150	0.90
0.7	0.168	0.142	0.84
0.8	0.168	0.134	0.80
0.9	0.168	0.127	0.76
1.0	0.168	0.121	0.72

Table 1: Comparison of the two methods for vectorizing conditionals.

3.3 Summary of conditional evaluation

- Vector mask operations does not incur a time penalty compared to operations under no mask (mask with all ones).
- Since masked operations take place on the entire vector, chaining is possible.
- On some systems, masked operations could result in an error condition due to the fact that the operation is performed on data for which the condition intended prevent an operation.
- Compressed-index operations use indirect addressing, which is inherently slower than direct addressing.
- Compressed-index mode may rule out chaining.
- Compressed-index mode is preferable when the condition is mostly false, while masked mode is preferable when the condition is mostly true.
- Compressed-index mode only operates on elements for which the condition is true, as intended in the program.

- For some cases scalar mode may actually be faster than either masked mode or compressed-index mode.

4 Vectorizing compiler technology

Vectorizing compilers attempts to carry out as many of the transformations discussed here as possible. For additional reading see for instance [5, 7].

References

- [1] Jack. J. Dongarra and Stanley C Eisenstat. Squeezing the most out of an algorithm in Cray Fortran. *ACM Trans. Math. Softw.*, 10(3):219–230, 1984.
- [2] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantative Approach*. Morgan Kaufmann Publishers, Inc, 1990.
- [3] 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.
- [4] C.L. Lawson, R.J. Hanson, D.R. Kincaid, and F.T. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM TOMS*, 5(3):308–323, September 1979.
- [5] John M Levesque and Joel W. Williamson. *A Guidebook to Fortran on Supercomputers*. Academic Press, 1989.
- [6] Michael Metcalf and John Reid. *Fortran 90 Explained*. Oxford Scientific Publications, 1991.
- [7] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press, 1991.