Vectorized APL2: Design and Implementation

Nancy Wheeler

IBM
APL Development
General Products Division
Santa Teresa Laboratory
San Jose, California

Abstract

This report discusses some of the design factors effecting the implementation of support for the 3090 Vector Facility in APL2. The vectorization of the reduction operator is used as a detailed example of the design process.

Contents

Figures

Introduction

The implementation of support for the 3090 Vector Facility is different from the usual task of adding support for a new function or feature, because there is an almost infinite amount of work which could be done.

There are many places in the APL2 interpreter where vectorization techniques can be used. Some are obvious (add, subtract, etc.) and some less obvious. Some are simple to implement, others very complex. Some have not yet been discovered.

Given a limited development resource, therefore, a subset of the many vectorizable items were chosen for APL2 Release 3. This paper will discuss the factors involved in designing and implementing those items.

While performance is discussed in general terms here, it is not the purpose of this paper to present specific performance numbers for APL2 on the Vector Facility. See Appendix C for references to publications which address that issue.

Vector Performance Factors

When implementing support for the 3090 Vector Facility, the foremost consideration, of course, is performance. It does not make sense to vectorize if performance will not be improved. The following factors effect the ability to vectorize or the performance of vectorized routines:

- **Data Type**. Most of the vector instructions operate on floating point numbers, and many on fullword integers. There are also some Boolean operations. APL2 does represents numbers in those types, but it also uses other types such as byte integers, complex numbers and characters. Usability of the vector facility depends on the type of the data and the availability of vector operations for that data type and function.
- **Vector Length**. Even when vector operations exist, there is overhead involved in their use. If the vectors involved are very short, the overhead cost can be greater than the gain realized by using vector operations.
- **Stride**. The vector facility will operate on vectors whose items are contiguous, or non-contiguous vectors whose items are regularly spaced (as are the columns of an APL array). Stride is the distance between elements of a vector. In general, the larger the stride, the smaller the performance gain possible with the vector facility. In fact, with very large stride, degradation may even be experienced. When items are very far apart, new pages of memory need to be brought in more often, and paging can become the dominant factor in performance.
- **Algorithm**. As is the case with any problem to be solved on a computer, there is often more than one way to vectorize a given function. Since each vector instruction operates on an entire segment of data, choosing the most efficient algorithm is extremely important. Even one additional vector instruction can have a significant effect on performance.
- **Cache/Page Management** As has already been mentioned, stride has an effect on paging, which in turn has an effect on performance. Vector algorithms can be further optimized by taking into account the existence and characteristics of cache memory.

In different areas of the APL2 implementation of vector facility support, these performance factors take on different levels of importance. Data type, vector length, stride, and algorithm are all factors in our design. As

of now, we have not added special logic for cache management, although it is implicitly part of the picture wherever stride is considered.

Scalar Primitive Functions

Theory of Scalar Primitives

Scalar primitive functions (also called pervasive functions) are primitive functions that operate on array arguments to produce array results of the same shape. It is easy to think about these functions in terms of the vector facility. Figure 1 shows a pictorial representation of how one of these functions is executed by the APL2 interpreter on a scalar machine.

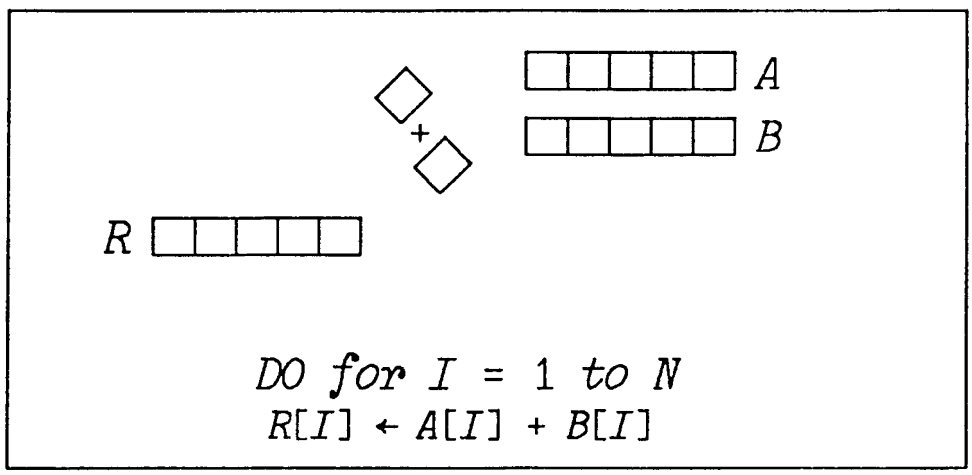

Figure 1. Scalar Primitive Functions on a Scalar Machine

Note that because the arrays are stored in the APL2 workspace in row major order, the shape (ρ) of the array is not an issue. The loop delimiter is the count of items in the array (×/ρ), and the result takes the same shape as the argument(s).

In order to execute this scalar primitive function on a machine which operates on vectors of arbitrary length, then, one would simply replace the scalar loop with an instruction which operates on the entire vector:

```
R ← A + B
```

(This notation should begin to look very familiar to APL programmers.)

In reality, the vector hardware operates on vectors of a fixed length called the **section size**, so the loop must be rewritten as shown in Figure 2 on page 4.

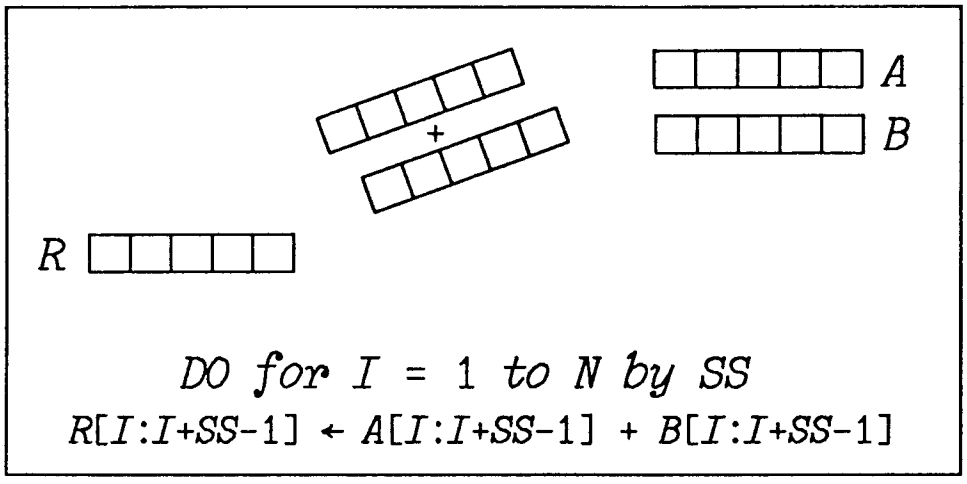

Figure 2. Scalar Primitive Functions on a Vector Machine

Implementation of Vectorized Scalar Primitives

For scalar primitive functions, then, the APL2 interpreter must do two things. First, the loops which control execution must be rewritten to operate on sections of an array rather than elements. Second, routines must be provided that will apply the appropriate functions to the array sections. For functions like add, subtract, multiply and divide, providing such a vector routine is a matter of executing a vector facility machine instruction. For others, such as logarithm and exponential, the logic can be very complicated.

For each primitive vectorized, of course, it is desirable to find the best possible algorithm. Even a simple primitive like addition can be optimized. The more general method,

```
LOAD REGISTER   (right argument)
LOAD REGISTER   (left argument)
ADD             (register/register)
STORE           (result)
```

can be improved:

```
LOAD REGISTER   (right argument)
ADD             (register/storage)
STORE           (result)
```

One can imagine that as the algorithms increase in complexity, the number of possible implementations increases.

For APL2 Release 3, all the scalar primitive functions and the dyadic circle functions (which can each be treated as an individual monadic scalar function) were evaluated for vectorization. Decisions on which to vectorize were made on the basis of the availability of existing vector routines (such as the VS FORTRAN mathematical routines), and the feasibility of writing new routines where none existed. It must be remembered also that routines can not always be provided for all data types due to the lack of vector instructions which operate on those data types. In some cases, arguments are converted to a different type (such as integer to floating point in many scientific routines), but in other cases the conversion would be more expensive than the gain from using the vector facility (Boolean arguments to the same routines).

A summary of the vector routines available for scalar primitives in APL2 Release 3 is found in Appendix A.

Performance Factors for Scalar Primitives

The performance factors considered in the scalar primitives are data type, vector length, and algorithm. Two of these have already been discussed. Data type helps determine whether a vector routine can be provided at all, and the algorithm should be chosen to get the best possible improvement from the vector facility.

Vector length is the other factor which was evaluated. As has been stated, if vectors are too short, the overhead of loading the vector registers and getting started can outweigh the benefits of operating on vectors. Measurements were made in APL2, and a **vector cutoff** of 20 elements was chosen as the best estimate of the length where most primitives start to see improvement. The APL2 interpreter will not use vector routines on arrays with a count ($\times/\rho$) of less than 20. The optimum improvement, of course, comes as vector length approaches section size, since the vector registers are then fully loaded and operating at their best. The section size on the IBM 3090 Vector Facility is 128 elements.

The storage arrangement of arrays in the APL2 workspace places the elements contiguously, in row major order. Even if an expression involves indexing, the index operation is performed first, creating a new contiguous array. In the case of scalar primitives, therefore, stride is not a factor in determining performance. Stride is always 1 (for floating point and integer arguments) or 2 (for complex numbers). The vector facility is optimized for both these values of stride.

How can APL2 application programmers effect their vector performance? For scalar primitives, they can make sure that their code can take advantage or the vector facility by simply using arrays whenever possible. The APL2 language is made for operating on arrays, and code that operates on arrays is made for the vector facility.

Strings of Scalar Functions

Theory of Scalar Function Strings

Another way in which APL2 Release 3 optimizes performance with the vector facility is by recognizing the situation where several scalar primitives are being executed, creating one result. For example,

```
R ← A + B × C - D
```

Each of these functions has a vector routine, so if *A*, *B*, *C* and *D* are all arrays, the vector facility can be used to compute the result *R*.

Normally, when interpreting that expression in APL, each function is executed independently, and a temporary result created at each step:

```
T1 ← C - D
T2 ← B × T1
R  ← A + T2
```

Even when using a scalar machine, the usage of the temporary results consumes time and storage. On a vector machine, the problem is of greater magnitude since loading and storing the vector registers is a relatively expensive operation. If the interpreter were able to eliminate the temporary result, it would appear that substantial performance gains could be realized. The APL2 Release 3 interpreter contains special code to recognize these strings of scalar functions if the 3090 Vector Facility is active.

Pitfalls of Scalar Function Strings

The objectives in implementing support for strings of scalar functions are twofold. The first, as has been discussed, is to optimize performance and reduce use of storage by eliminating temporary results. Second, and equally important, the recognition of the strings must be as efficient as possible to avoid impacting performance negatively with the recognition process. If both these objectives can be met, we should be able to achieve the theoretical performance gains.

In practice, some problems do stand in the way of our objectives. There is an overhead in performing the scan, which because of the interpretive nature of the language is done at execution time. Furthermore, there are many expressions that do not qualify for the optimization. For example, index brackets and parentheses stop the scan of the expression, and functions which are not scalar primitives cannot be executed in this manner because the result shape does not remain the same across the string.

As a result of the many non-qualifying cases, our ability to achieve an efficient scan is challenged. When a qualifying scalar function string is found and executed, the performance gain far outweighs the cost of the extra scanning. When no qualifying code can be found in an expression, and the scan is restarted many times due to brackets and parentheses, and the expression is repeated a large number of times, the cost of the scan can be measurable.

Just as APL2 application programmers can cause the vector facility to be used more often by replacing loops of operations on scalars with operations on arrays, they can also effect their performance in this area. Elimination of redundant parentheses is one way, and using indexing sparingly is another. For example, the expression

```
R ← A[I;] + B × A[I;] * 2
```

could be written in a better style for both the scalar and vector machines:

```
R ← A[I;]
R ← R + B × R * 2
```

Now the indexing is only done once, and the optimizing code for strings of scalar functions will find a qualifying case.

Note, however, that pre-executing all indexing operations is not advisable. In the case of the expression

```
R ← A[I;] + B × C[J;] * 2
```

rewriting it as

```
T1 ← A[I;]
T2 ← C[J;]
R ← T1 + B × T2 * 2
```

does not provide the advantage of eliminating duplicate work, and has the disadvantage of increasing the complexity of the code and the size of the name table.

Operators, Etc.

When progressing from the orderly realm of scalar primitives into non-scalar primitives and operators, the techniques used in vectorizing become more complex, and need to be individually developed, since these functions and operators all have their own unique rules for the formation of results.

In these cases, there are many more factors to be considered. The result is not necessarily the same shape as the argument. The shape of the argument is significant, where it was not for scalar primitives. Applying a function or operator along a different axis can change the approach taken entirely. It is here that stride becomes a factor in choosing a vector algorithm, because the data elements are not always contiguous.

In order to show the kinds of design decisions that are made in vectorization, the rest of this paper will be devoted to an example. The design theory behind the vector implementation for the reduction operator will be given in detail. It will be shown that data type, shape, and stride are all factors in the choice of algorithms.

A list of the operators which have been vectorized in APL2 Release 3 is given in Appendix B, along with a list of idioms and other special cases which are recognized when using the vector facility.

The Reduction Operator

The reduction operator, / or ⌿, operates on a dyadic function to cause that function to be applied between elements of an array. For example,

```
+/ 2 3 4
```

is the same as

```
2 + 3 + 4
```

and yields a result of 9.

When the argument has more than one dimension, the function is applied along one axis of the array.

```
+/ 2 3 4
   5 6 7
```

is the same as

```
(2 5) + (3 6) + (4 7)
```

and yields a result of (9 18).

Changing the shape of the argument changes the result:

```
+/ 2 3
   4 5
   6 7
```

is the same as

```
(2 4 6) + (3 5 7)
```

and yields a result of (5 9 13).

Now change the axis of reduction for yet a different result:

```
+⌿ 2 3
   4 5
   6 7
```

is the same as

```
(2 3) + (4 5) + (6 7)
```

and yields (12 15).

From these simple examples it can already be seen that the argument shape and axis of reduction effect the result value and shape. Finding one vector algorithm for reduction is not sufficient. More than one will be necessary, and the choice of an algorithm will depend on many different factors.

Reduction of a Vector

Reduction of a vector (a one-dimensional array) is the simplest case, but will vectorize least often. Because reduction causes a function to be applied between elements of one array rather than two, the vector routines we have written for scalar primitives, which take two arrays as arguments, cannot be used. We can vectorize one-dimensional reduction only if a machine-level solution is available which does exactly what we want.

Fortunately, machine-level solutions are available for three of the most common reductions: plus (+), maximum (⌈) and minimum (⌊). Unfortunately, the vector facility instructions that implement these solutions operate only on floating point numbers, so the vectorization is limited to that type.

Reduction of a Matrix

Increasing the dimensions of the array to two opens up new possibilities for vectorization.

Reduction Along Last Axis

Reduction of a two-dimensional matrix along the last axis requires that each row of the matrix be reduced to one element by applying the function between the elements of the row. Figure 3 shows that there are two different ways that could be accomplished.

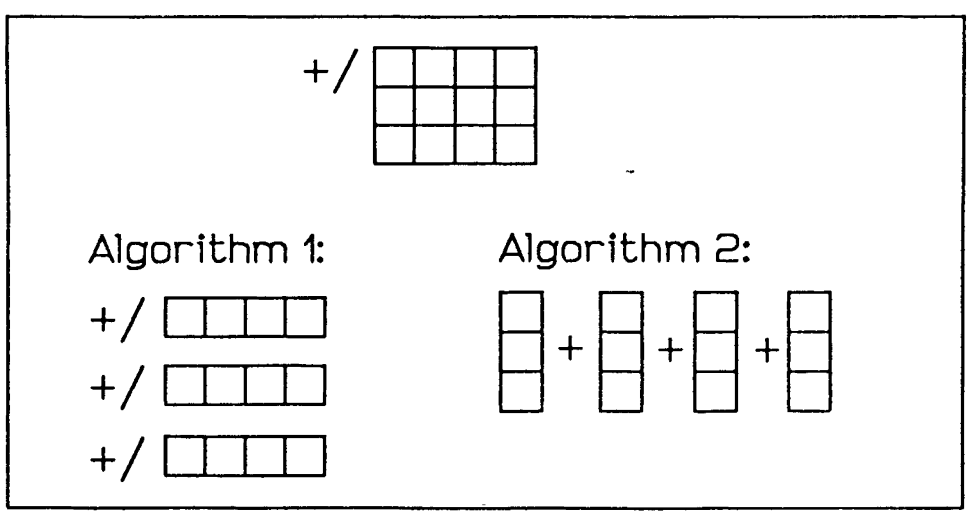

Figure 3. Reduction of a Matrix (Along Last Axis)

The first algorithm takes each row of the matrix and reduces it as a vector. This will only work, of course, if the function is +, ⌈, or ⌊, and the argument is floating point. The second algorithm takes each column

of the matrix as a vector, and uses the standard vector routine for the function. This algorithm will work for any scalar primitive function which has a vector routine for the argument and argument type we have.

Which algorithm is better? For the case in Figure 3 on page 10, it is hard to tell. It looks like each would involve about the same number of machine-level instructions. Algorithm 1, however, operates on rows, which are contiguous (stride one), and Algorithm 2 operates on columns, which are not (this case has stride 4).

Figure 4 shows a different example.

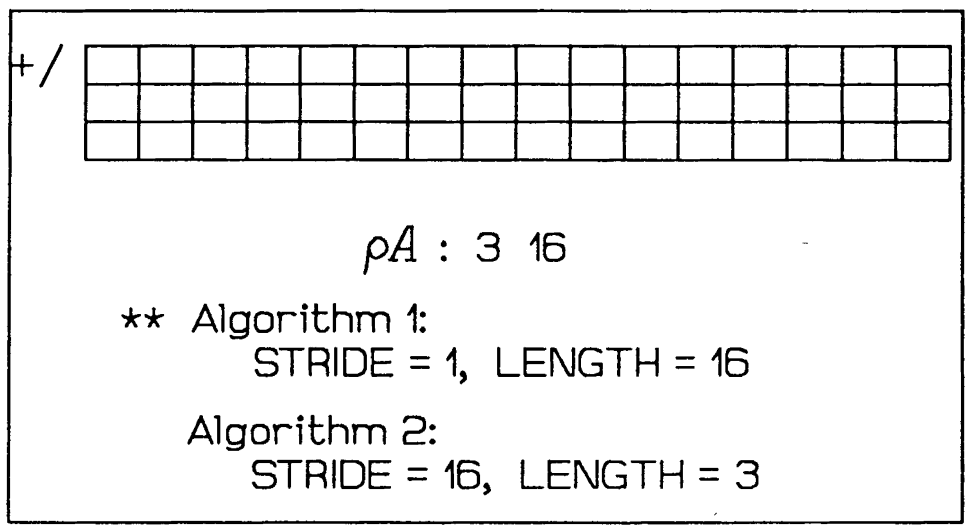

Figure 4. Reduction of a Matrix with Long Last Dimension

In that case, the last dimension is significantly longer, and the choice of an algorithm becomes clearer. Algorithm 1 has a short stride and a long length, while Algorithm 2 has a long stride and short length. Since our goal is always longer lengths and shorter strides, Algorithm 1 is a clear winner.

Figure 5 on page 12, however, shows the other extreme. In that case, the first dimension is longer. While Algorithm 2 still has a slightly longer stride, its length is much longer and causes it to win out.

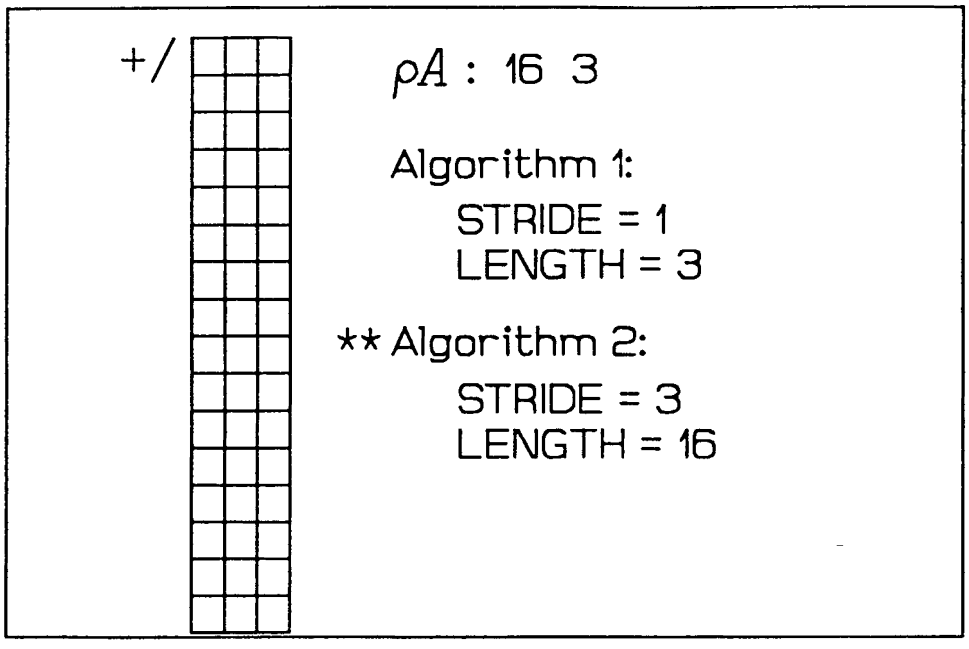

Figure 5. Reduction of a Matrix with Long First Dimension

Reduction Along First Axis

Reduction of a two-dimensional matrix along the first axis requires that each column of the matrix be reduced to one element by applying the function between the elements of the column. Figure 6 shows that there are also two different ways to accomplish that. The two algorithms are similar to, but not exactly the same as, those for reducing along the last axis.

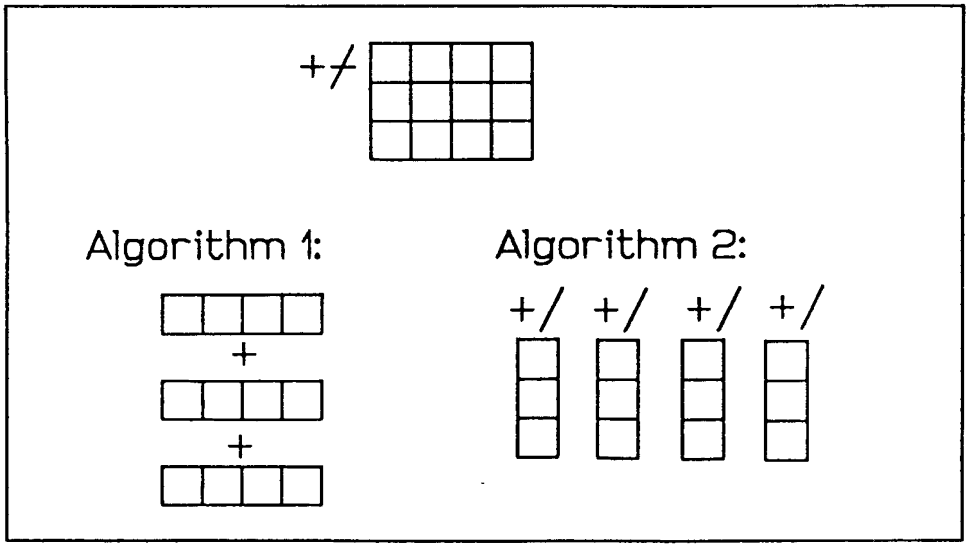

Figure 6. Reduction of a Matrix (Along First Axis)

The first algorithm takes each row of the matrix and uses the standard vector routine for the function. This algorithm will work for any scalar primitive function which has a vector routine. The second algorithm takes each column of the matrix and reduces it as a vector. This will only work if the function is +, ⌈, or ⌊. Again, it is difficult to see an advantage for either algorithm unless the cases are more extreme.

Figure 7 shows the type of case that will prove Algorithm 1 to be superior, and Figure 8 the type that Algorithm 2 will do best at.

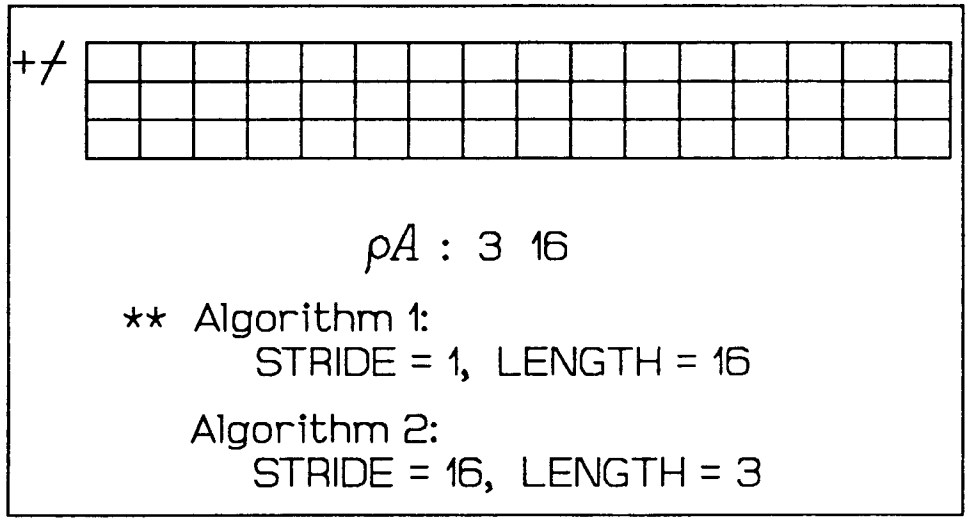

Figure 7. Reduction of a Matrix with Long Last Dimension

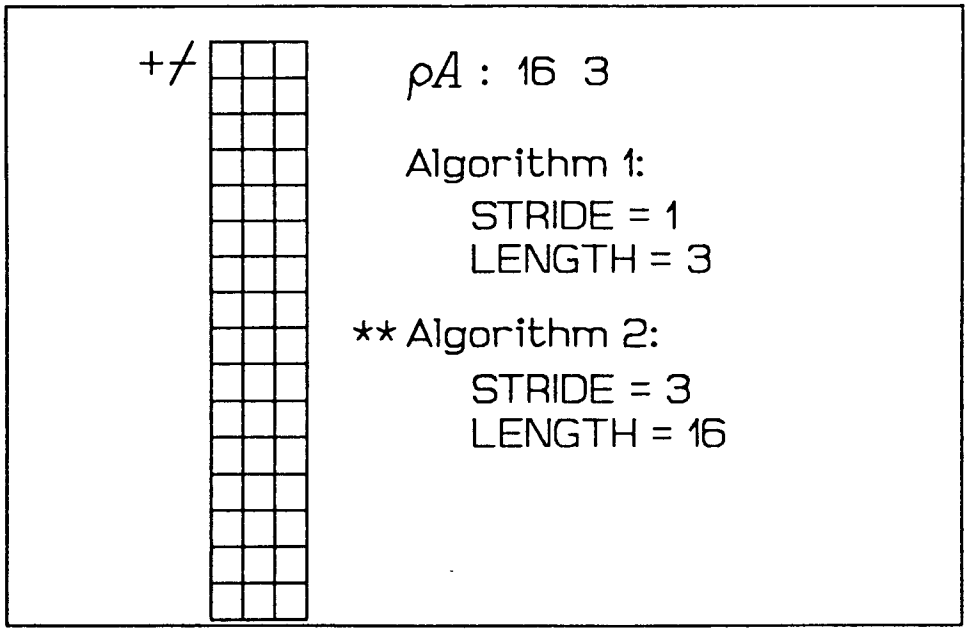

Figure 8. Reduction of a Matrix with Long First Dimension

Reduction in Three Dimensions

Let us go one step further into three dimensions to look at even more ways to vectorize.

Reduction Along First Axis

Reduction of a three-dimensional array along the first axis requires that the function be applied between elements of the planes of the argument to create a two-dimensional result. Figure 9 shows two different ways to accomplish that goal.

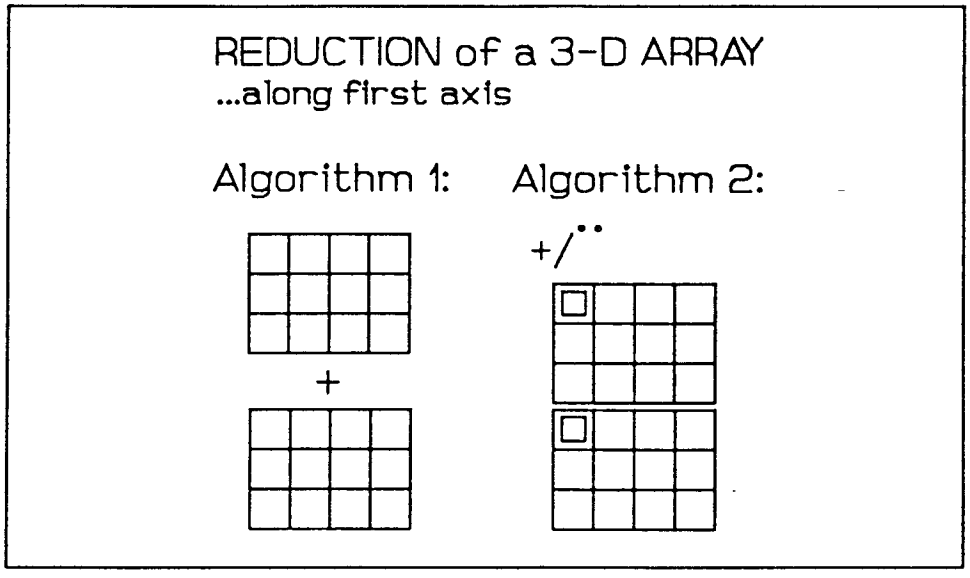

Figure 9. Reduction of a 3-D Array (Along First Axis)

Algorithm 1 treats each plane as a vector (remember that the scalar primitive function routines ignore shape) and applies the standard vector routine between the planes. Algorithm 2 treats as a vector corresponding items in each plane, and uses the machine-level reduction on the resulting vectors.

At first glance it would seem that Algorithm 1 is clearly superior, and for arrays with a long last or middle dimension that is the case. Figure 10 on page 15 shows the case with long first dimension. Since stride is one for Algorithm 1, and the length is long, it will perform much better.

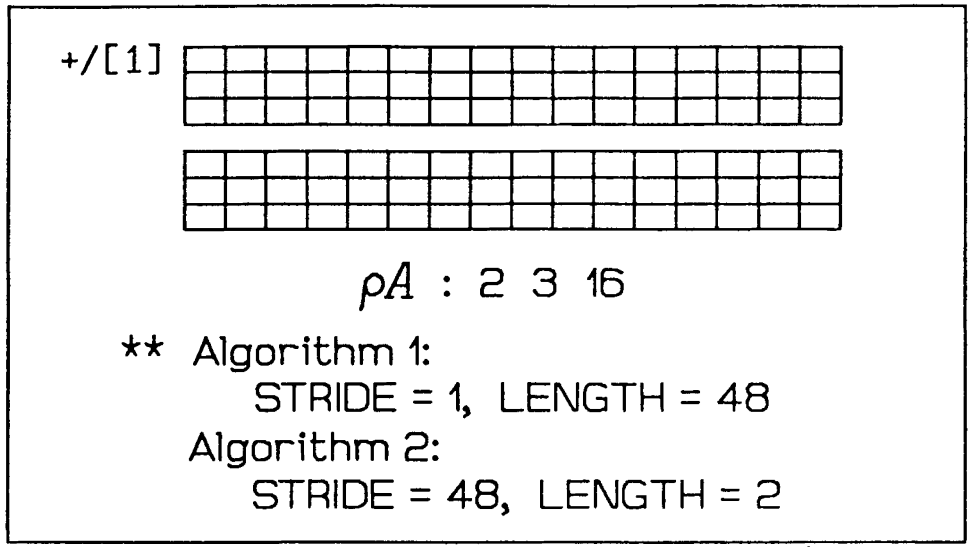

Figure 10. Reduction of a 3-D Array with Long Last Dimension

There is a case, however, where Algorithm 2 can be beneficial. Figure 11 shows a three-dimensional array with a long first dimension and short last and middle dimensions. In that case, even though stride is not one, the long length gives an advantage to Algorithm 2.

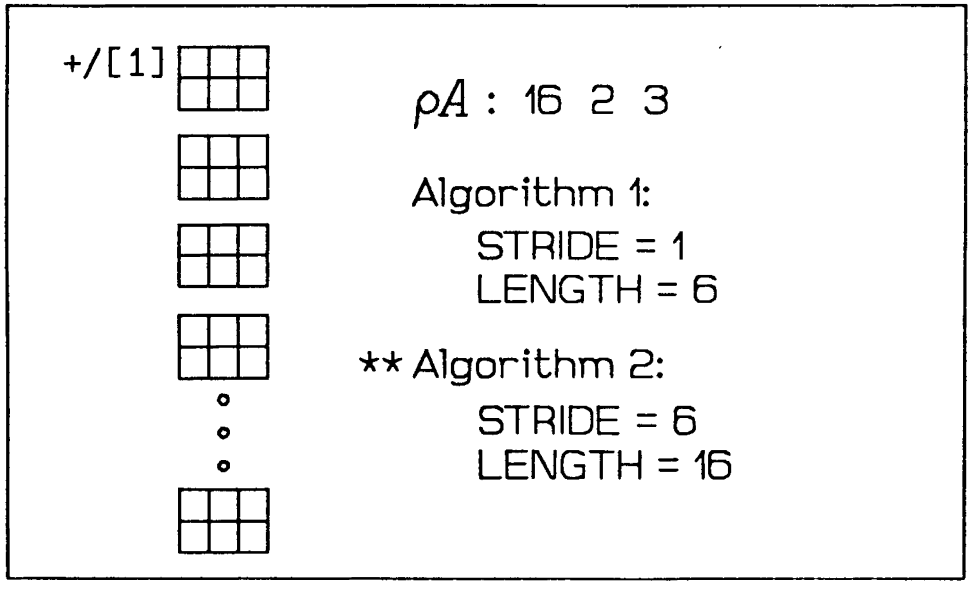

Figure 11. Reduction of a 3-D Array with Long First Dimension

Reduction Along Last Axis

Reduction of a three-dimensional array along the last axis requires that the function be applied between elements of the rows of the argument. Figure 12 on page 16 shows two different algorithms for this case.

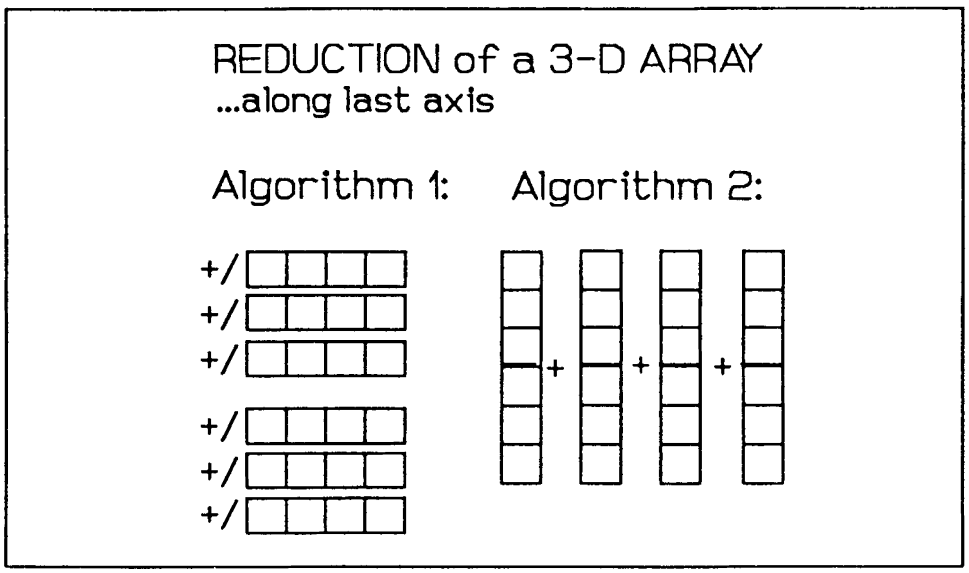

Figure 12. Reduction of a 3-D Array (Along Last Axis)

Algorithm 1 treats each row as a vector and uses the machine-level reduction on them. Algorithm 2 treats each column as a vector and applies the standard vector routine between those vectors, across each row. It can ignore the plane boundaries and make one vector for all the corresponding columns in each plane,

Here again the algorithm choice depends on the shape of the argument. With a long last axis, as shown in Figure 13, Algorithm 1, with stride of one and long length, is superior. With a long middle or first axis, however, Algorithm 2 is better, with short stride and long length. Figure 14 on page 17 shows an example.

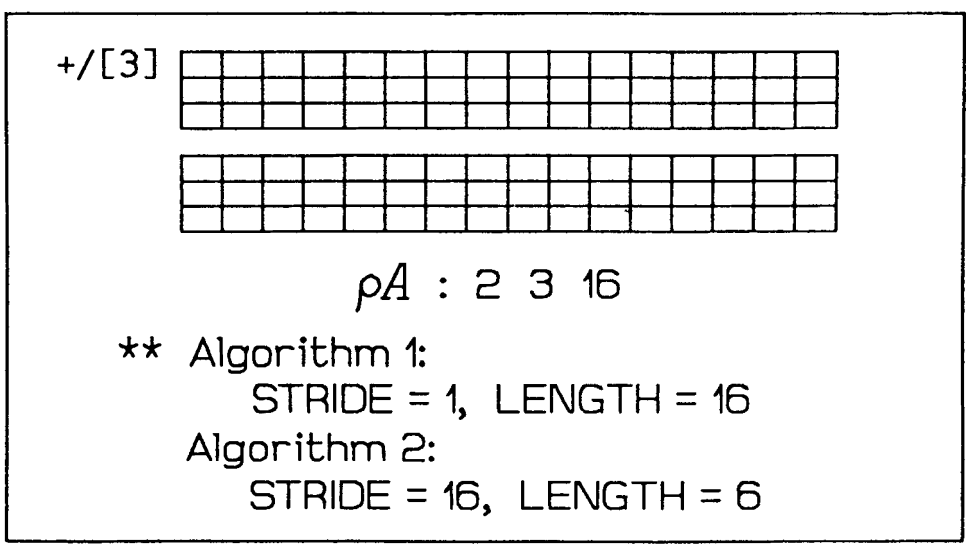

Figure 13. Reduction of a 3-D Array with Long Last Dimension

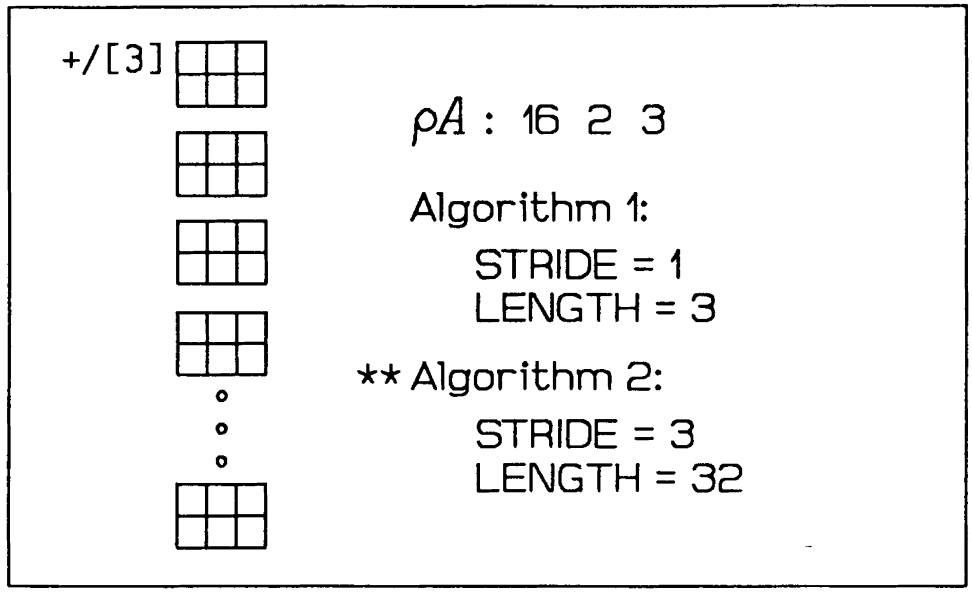

Figure 14. Reduction of a 3-D Array with Long First Dimension

Reduction Along Middle Axis

Reduction of a three-dimensional array along the middle axis requires that the function be applied between elements of the columns of the argument. Figure 15 shows three different algorithms for this case.

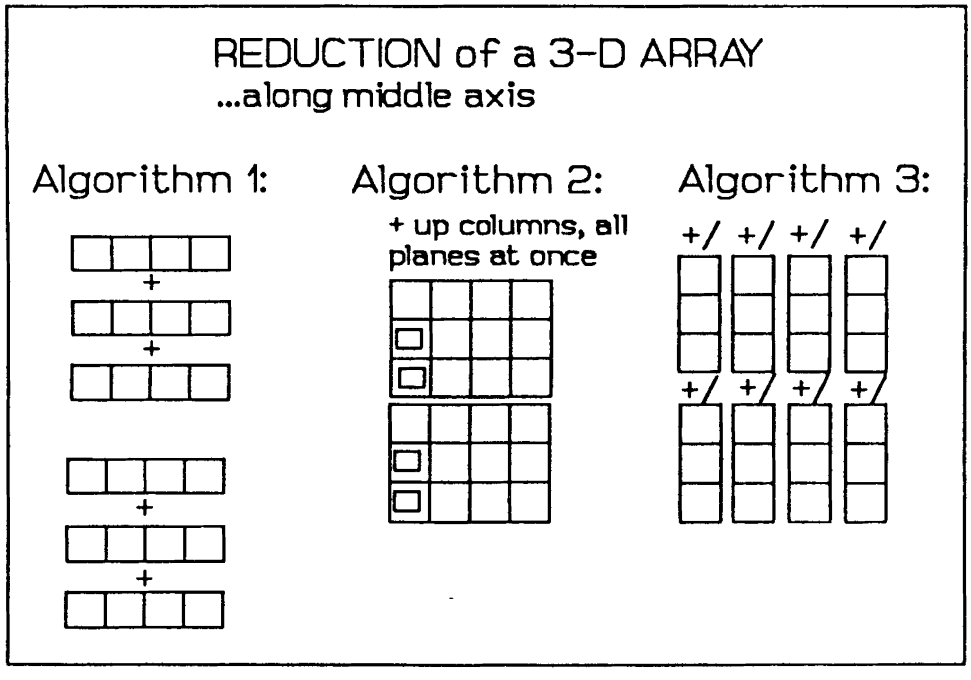

Figure 15. Reduction of a 3-D Array (Along Middle Axis)

Algorithm 1 treats each row as a vector and applies the vector routine between them. Algorithm 2 treats corresponding items of the planes as vectors and applies the vector routine between them. Algorithm 3 applies the machine-level solution to each column.

Again, there are cases where each of the three algorithms is best. Figure 16, Figure 17, and Figure 18 on page 19 show examples of a case for each algorithm.

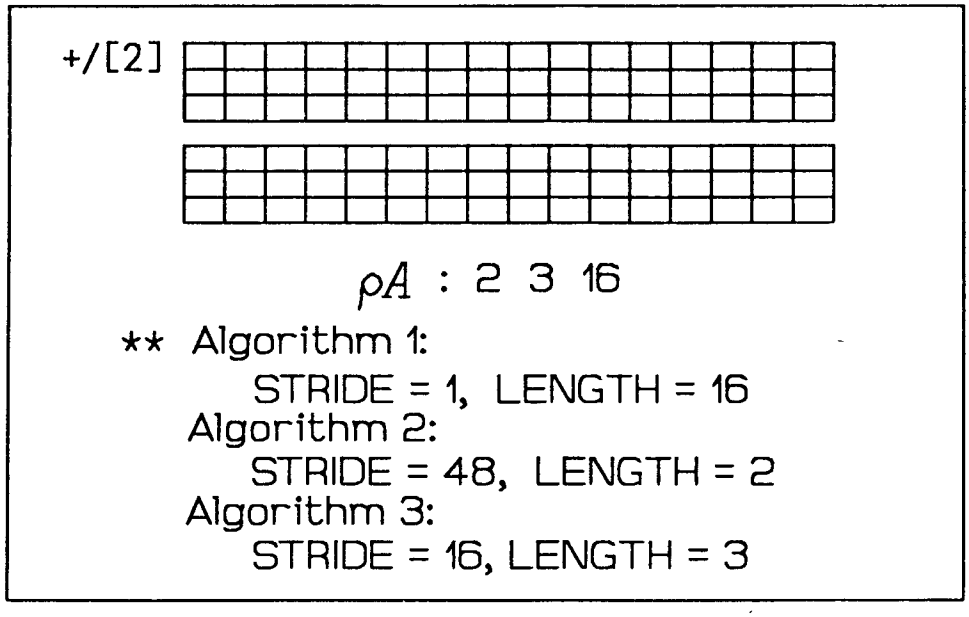

Figure 16. Reduction of a 3-D Array with Long Last Dimension

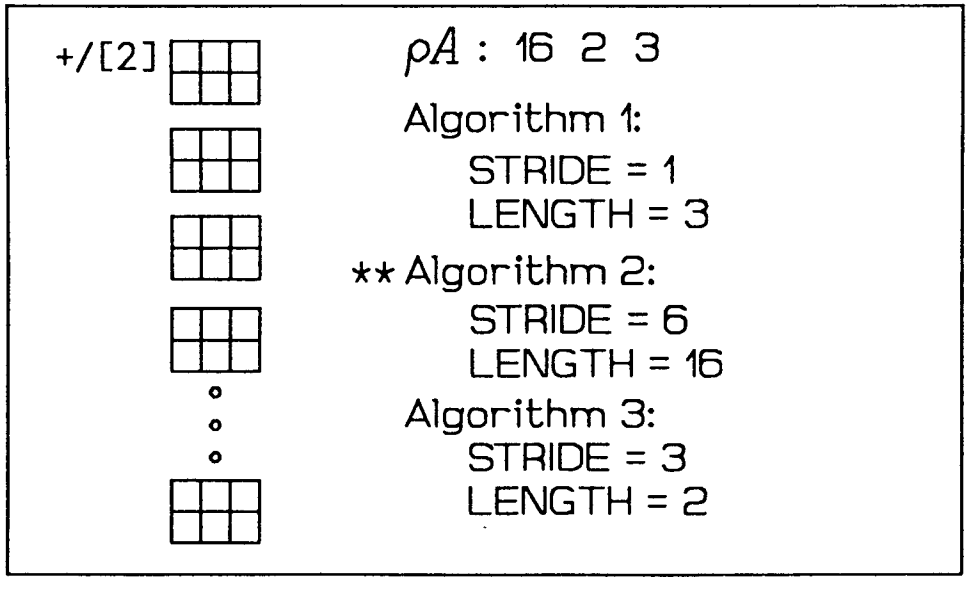

Figure 17. Reduction of a 3-D Array with Long First Dimension

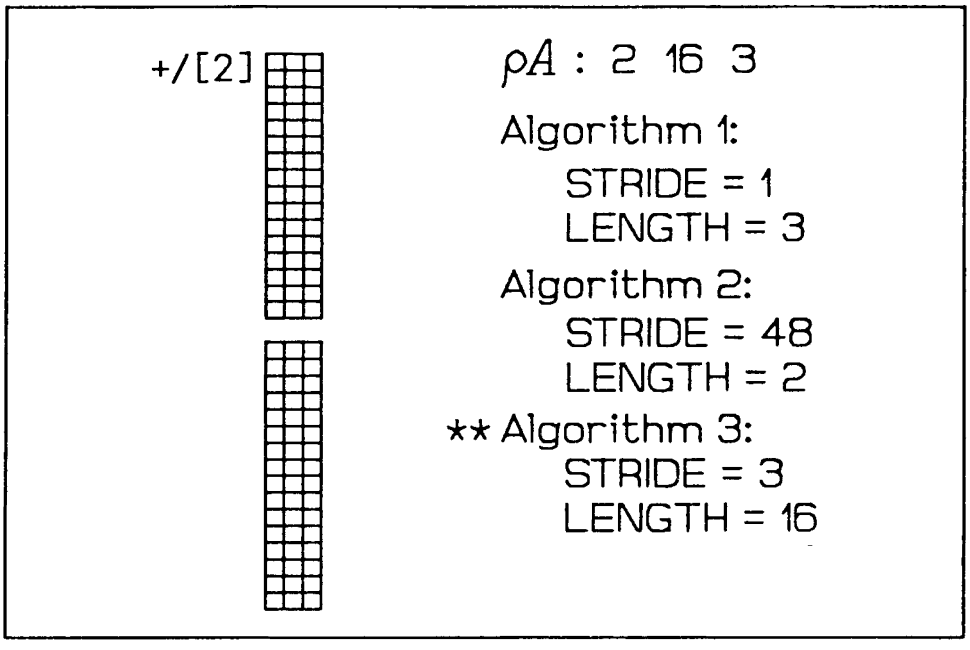

Figure 18. Reduction of a 3-D Array with Long Middle Dimension

The Generalization of Reduction

We have demonstrated the many cases of reduction in different dimensions, and shown how vector length, stride, function and data type are involved in choosing an algorithm.

Fortunately, the implementation of vectorized reduction is not as complicated as these examples might indicate. Reduction can be generalized. In fact, all cases of reduction can be described in terms of the three-dimensional, middle-axis case, the last case described here, and all of the vector algorithms we saw can be equated to one of the three algorithms for that case.

Figure 19 shows how we describe reduction in terms of the three dimensional case. The axis of reduction always becomes the middle dimension. Axes on the left of the axis of reduction are combined to form the first dimension. Axes on the right of the axis of reduction are combined to form the last dimension. If there is nothing to the left or right, that dimension takes length one.

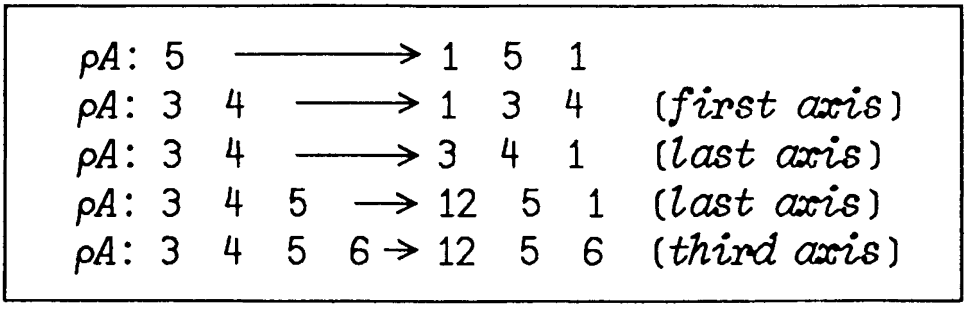

Figure 19. Generalization of Reduction

The three basic reduction algorithms are seen in the three- dimensional middle-axis case. These algorithms have all been implemented and tested for performance.

Algorithm 1 involves treating rows as vectors. This algorithm gives the best results if length is reasonably long (the vector cutoff of 20 is used) because stride is always one.

Algorithm 2 involves treating corresponding items of each plane as vectors. It gives good results if the first dimension is long, and stride (which will be the product of the other two dimensions) is not too long. The cutoff used in that case is one also arrived at by empirical measurements, and may change as different vector machines are developed.

Algorithm 3 involves the use of the machine-level instructions for +, ⌈, and ⌊. It gives the least improvement, but is useful if the other two are eliminated due to short lengths or long strides.

Figure 20 summarizes the rules for generalized reduction in the APL2 interpreter. It should also be noted that implied in all these decisions is the existence of a vector routine for the function operand and argument data type of the reduction. Without that routine, no vectorization is possible.

```
IF the last dimension is large,
    USE Algorithm 1
ELSE
    IF the first dimension is large,
    AND the product of the other two
    dimensions (STRIDE) is not too large,
    USE Algorithm 2
ELSE
    IF the middle dimension is large,
    and we have +, ⌈, or ⌊
    USE Algorithm 3
```

Figure 20. Rules of Vectorized Reduction

Conclusion

Implementing support for the 3090 Vector Facility is an almost infinite task whose individual parts range from the extremely simple to the very complex. It has been both challenging and exciting to complete a part of that task in APL2 Release 3.

Of course, there are many other areas of APL2 where vectorization can be achieved. More scalar primitives can be vectorized. Some computational non-scalar primitives are candidates for vectorization. In addition, it is possible to move beyond strictly computational functions and use the vector facility to move data as well. The next part of the task should be just as challenging and exciting.

Appendix A. Tables of Vectorized Scalar Primitives

I indicates an identity function (result equal to argument)

D indicates domain error (function not defined for the type)

V indicates existence of vector routine

Monadic Scalar Functions

FUNCTION	BOOLEAN	INTEGER	REAL	COMPLEX
Conjugate (+)	I	I	I	V
Negative (-)		V	V	V
Direction (×)	I	V	V	V
Reciprocal (÷)		V	V	V
Magnitude (|)	I	V	V	V
Floor (⌊)	I	I		
Ceiling (⌈)	I	I		
Exponential (*)		V	V	V
Natural Logarithm (⍟)		V	V	
Pi times (○)		V	V	V
Factorial (!)				
Not (~)		D	D	D
Roll (?)				

Dyadic Scalar Functions

FUNCTION	BOOLEAN	INTEGER	REAL	COMPLEX
Addition (+)		V	V	V
Subtraction (-)		V	V	V
Multiplication (×)		V	V	V
Division (÷)			V	V
Residue (|)				
Minimum (⌊)		V	V	D
Maximum (⌈)		V	V	D
Power (*)			V	
Logarithm (⍟)				
Binomial (!)				
And (∧)		D	D	D
Or (∨)		D	D	D
Nand (⍲)		D	D	D
Nor (⍱)		D	D	D
Less (<)		V	V	D
Not Greater (≤)		V	V	D
Equal (=)		V	V	
Not Less (≥)		V	V	D
Greater (>)		V	V	D
Not Equal (≠)		V	V	

The Circular Functions

FUNCTION	BOOLEAN	INTEGER	REAL	COMPLEX
*0J1×R (¯12o)		V	V	V
0J1×R (¯11o)		V	V	V
Conjugate (¯10o)	I	I	I	V
Identity (¯9o)	I	I	I	I
-(¯1-R*2)*.5 (¯8o)				
Arctanh (¯7o)				
Arccosh (¯6o)		V	V	
Arcsinh (¯5o)				
(¯1+R*2)*.5 (¯4o)		V	V	
Arctan (¯3o)		V	V	
Arccos (¯2o)				
Arcsin (¯1o)				
(1-R*2)*.5 (0o)		V	V	
Sine (1o)		V	V	
Cosine (2o)		V	V	
Tangent (3o)		V	V	
(1+R*2)*.5 (4o)		V	V	
Sinh (5o)				
Cosh (6o)				
Tanh (7o)				
(¯1-R*2)*.5 (8o)				
Real Part (9o)	I	I	I	V
Magnitude (10o)	I	V	V	V
Imaginary Part (11o)		V	V	V
Phase (12o)		V	V	

Appendix B. Other Vectorization in APL2

Operators

- Expand (\)
- Reduction (*f*/)
- Inner Product (*f*.*g*)
- Outer Product (*f*∘.*g*)

Note that the vectorization of reduction, inner product and outer product will occur only if, at interpretation time, the arguments are determined to be of appropriate dimensions for vectorization, and if vectorized routines exist for the function operands and argument types.

Idioms

The idioms implemented are implemented for both the vector and the scalar machine. Where a type is indicated, the vectorization will only occur for that type.

- *V*⍳⌈/*V* (Real)
- *V*⍳⌊/*V* (Real)
- +/⍤1
- ⌈/⍤1
- ⌊/⍤1

Special Cases

Some additional cases have special vector routines:

- +/ (Boolean vectors)
- Square (*2)
- Square Root (*.5)
- Base 10 Logarithm (Integer and Real)

Appendix C. References

IBM System/370
Vector Operations
SA22-7125

Understanding the IBM 3090 Vector Facility
Dr. James A. Brown
APL Chief Architect
SEAS AM87

APL2 Release 3 Performance
Henry Altaras
International Technical Support Center
ZZ81-0196

APL2 Version 1 Release 3
Primitive Function Performance on the
IBM 3090 Vector Facility
Michael Van Der Meulen
Mario V. Morreale
IBM Kingston
TR 21.1078