International Journal of Information Technology and Knowledge Management
July-December 2010, Volume 2, No. 2, pp. 375-378

VECTOR ARCHITECTURES FOR MATRIX OPERATIONS

Satya Prakash Singh

Numerical algorithms involve not only vector operations but also matrix operations. Today's vector processors only support vector operations, and the execute matrix operations in terms of vector operations, in one instruction. This will lead to poor sustained performances of vector machines. In this paper we will discuss how to support both vector operations and matrix operations in vector architectures. At first sub array patterns for vector and matrix operations are introduced. Then we present a set of accessing modes which can make vector architectures to access bother sector and matrix operands. Finally the performance imprudent for matrix multiplication and the FFT is demonstrated.

1. INTRODUCTION

Matrix multiplication can be executed in the outer product method as follows :

```
FORK := 1 TO n DO
 FORI : = 1 TO n DO
   FORJ := 1 TO n DO
    C [I,J] := C[I,J] + A [I,K] * B [K,J];
   END:
 END:
END:
```

The two inner loops over I and J can be evaluated in parallel for all n*n elements of C, where the multiplication operation is an element-by element multiplication of an*n matrix formed by duplicating the kith column of A and an n*n matrix formed by duplicating the kith row of B. That is, this algorithm could be arranged in n pipelined operations.

With the length n*n However today's vector architectures have to execute this algorithm in n*n pipelined operations with the length of n, because they can not access the matrices required as above in one instruction.

The same situation occurs in many elementary numerical algorithms. From elementary numerical algorithms such as matrix multiplication, matrix transposition, LR decomposition, Gaussian Elimination, Matrix inversion, symmetric matrix operations, fast Fourier transformation and solution of partial differential equations we can abstract the following sub array patters for both vector operations and matrix operations [2], In the following

Sr. Lecturer, Computer Science & Engg., Birla Institute of Technology, Mesra, Noida Campus

Email: spsinghbit@yahoo.co.in, spsingh11in@yahoo.co.in

A is declared as "ARRAY [I1..U1], [I2..U2]….[In…un] of type

1. A subvector from an n-dimensional array can be represented by Pattern A.

   FOR I := TO J BY s DO.

   access(A[a_1+b_1*I,a_2+b_2,a_3+b_3*1,..., a_n+b_n*1]):

   END:

2. Submatrices from an n-dimensional array can be represented by Pattern B and Pattern C.

   Pattern B represents a submartix accessed in the row major order:

   FOR I_k := a TO b BY s_1DO

   FOR I_j = c*I_k +f BY s_2 DO

   Access(A[I_1,I_2,…,I_j,…,I_k,…,I_n]);

   END;

   END:

   Pattern C represents a submatrix accessed in the column major order

   FOR I_k := a TO b BY s_1DO

   FOR I_j = c*I_k +d TO e*I_k f BY s_2 DO

   Access(A[I_1,I_2,…,I_j,…,I_k,…,I_n]);

   END;

   END:

   If e = c a rectangular or parallelogram submatrix is expressed. If l (e-c)*s1/s2 l=1, triangular submartix is expressed.

3. Submatrices expanded from a subsector in an n-dimensional array can be represented by Pattern D and Pattern E.

Pattern D represents a submartix expanded from a subvector as a column.

FOR I_k := a TO b BY s_1DO

FOR J = c*I +d TO e*I f BY s_2 DO

Access(A[I_1,I_2,...,I_{k-1},...,I_{k+2},...,I_n]);

END;

END:

If e = c, a rectangular or parallelogram sub matrix is expressed. If I (e-c) *s1/s2 I=1, a triangular sub matrix is expressed.

Executing matrix operations in terms of vector operation will lead to poor sustained performances of vector machines, because the sustained performance of a vector architecture is directly proportional to the length of pipelined operations. In this paper we will discuss how to support both vector opera operation and matrix operation in vector architectures. Section 2 introduces a set of accessing modes which can make vector architectures to access bother vector and matrix operands. Section 3 demonstrates the performance improvement by our approach.

2. SUBARRAY ACCESSING

There are three kinds of memory accessing in vector architectures: sampling, indexing and masking [4]. Indexing is used indirect accessing and masking is used in conditional accessing. The typical sampling modes found in the today's vector architectures are "Sequence Sampling" and "Stride Sampling".

(1) Sequence Sampling: "Sequence Sampling" extracts a contiguous subsequence according to a specified start index and a length. Sequence (vec: VECTOR; start, length: INTEGER) =

<vec[start], vec[start +1], vec[start +2],...., vec[start +length-1]>

Where start and length are Positive integers.

(2) Stride Sampling: "Stride Sampling" selects a specified number of elements. It starts at a specified start index and chooses other elements equidistantly.

Stride (vec: VECTOR; start, step, length: INTEGER) =

<vec[start], vec[start+step], vec [start+2*step],...., vec[start+(length-1)*step]>

Where start and length are positive integers, skip is an integer.

We find that the above sampling modes are not sufficient in subarray accessing for matrix operations. Therefore other sampling modes are needed.

(3) Multiple Stride Sampling: The sequence selected by "Stride Sampling" is called a stride sequence (a sequence with strides). "Multiple Stride Sampling" selects several stride sequences.

Multiple-Stride(vec: VECTOR, start, step, count, skip length: INTEGER)=

IF length = 0 THEN NULL;

ELSE ;step and skip are integers, and @ represents a concatenation.

(4) Incremental Stride Sampling: "Incremental Stride Sampling" selects several stride sequences with the increment by "step" in skips between sequences.

Incremental-Stride(vec: VECTOR, start, step, count, skip length INTERGER)=

IF length = 0 THEN NULL;

ELSE

Stride (vec, start, step, count,) @Multiple-incremental-start+(count-1)*step+skip, step, count+1, skip-step, length-1);

END;

Where start, length, and count are positive integers, step and skip are integers, and @ represents a concatenation.

(5) Decremental Stride Sampling: "Decremental Stride Sampling" selects several stride sequences with the decrement by 1 in sequence lengths and the increment by "step" in skips between sequences.

Decremental-Stride(vec: VECTOR, start, step, count, skip length INTERGER)=

IF length = 0 THEN NULL;

ELSE

Stride (vec, start, step, count,) @ Decremental - Stride start + count-1)*step + skip, step, count-1, skip+step, length-1);

END;

Where start, length, and count are positive integers, step and skip are integers, and @represents a concatenation.

[Example]

V=< 1 2 3 4

5 6 7 8

9 10 11 12

13 14 15 16>

Sequence (V, start = 5, length =4) = <5 6 7 8>

Stride(V, start =2, step =4, length = 4) = <2 6 10 14>

Multiple-Stride (V, start =1, step=3, count=2, skip = 5, length =2) = < 1 4 9 12>

Incremental-Stride (V, start=1, step=1, count=1++, skip =4—(1), length =4) = < 1 5 6 9 10 11 13 14 15 16>

Decremental-stride (V, start =1, step=4, count =4 —, skip= -7++ (4), length =4) = <1 5 9 13 6 10 11 1314 15 16>

The formal analysis of hw these sampling modes can be used to access sub arrays can be found in [2]. We can prove that "Multiple Stride Sampling" can be used to access Pattern A, Patterns B,C,D, and E in the case of e=c; "Incremental Stride Sampling" can be used to access Patterns B,C,D, and E in the case of (e-c)* s_1/s_2 = -1.

3. PERFORMANCE IMPROVEMENT

Support of matrix operations, a simple approach is to support matrix operations, a simple approach is to introduce "Multiple Stride Sampling". "Incremental Stride Sampling" and "Decremental Stride Sampling" in hardware.

If a matrix operation is not supported by a vector architecture, it can be execute as several vector operations. The former is implemented by one pipelined operations. Note that in the two implementations the amount of the element operations is the same, But the number of the Pipelined operations is different. Therefore the performance improvement for the matrix-oriented implementation can be obtained by reducing the number of the star-up's of pipelined operations. It is understandable that the more start-up time is,(or the shorter vector operations are,) the more performance improvement is expected for the matrix oriented implementation. Such performance improvement has been proved in our simulation. We have done a simulation to evaluate the performance of matrix multiplication and the FFT on a basic vector architecture based on Chapter 7 of [9] in which $T_{element}$ = I clock, T_{add} = 6 clocks, T_{mult}= 7 clocks, and $T_{load/store}$ = 12 clocks.

The performance is calculated by the time for all element operations /the total execution time.

The preferment of matrix multiplication (size =4*4, 8*8, 16*16, 32*32, 64*64, 128*128, 256*256, 512 *512, 1024*1024) on register-to-register vector architectures and on memory-to-memory vector architectures is shown in Table 1.1 for the vector-oriented implementation (1-D) and the matrix-oriented implementation (2-D).

The performance evaluation of the FFT is made for n =

Table 1.1
Performance of Matrix Multiplication

Size	R-To-R		M-to-M	
	1-D	2-D	1-D	2-D
4*4,	16.2%	43.6%	11.7%	33.3%
8*8, 16*16,	26.3%	74.1%	19.5%	65.2%
32*32, 64*64,	40.4%	91.6%	31.5%	87.8%
128*128,	56.9%	97.7%	47.1%	96.6%
256*256,	72.2%	99.4%	63.7%	99.1%
512*512,	83.8%	99.9%	77.7%	99.8%
1024*1024	91.1%	99.9%	87.4%	99.9%
	95.4%	99.9%	93.2%	99.9%
	97.6%	99.9%	96.6%	99.9%

Table 1.2
Performance of the FFT

Size	R-To-R		M-to-M	
	1-D	2-D	1-D	2-D
8	7.1%	15.2%	5.3%	11.7%
16	8.7%	26.4%	6.6%	20.9%
32	10.4%	41.4%	7.8%	34.5%
64	12.0%	58.9%	9.1%	51.3%
128	13.6%	74.1%	10.4%	67.8%
256	15.2%	85.1%	11.7%	80.8%
512	16.8%	92.0%	12.9%	89.4%
1024	18.3%	95.8%	14.2%	94.4%

8,16, 32, 64, 128, 256, 512, 1024. The performance of the FFT on register-to-register vector architectures and on memory-to-memory vector architectures is shown in table 1.2 for the vector-oriented implementation (I-D). The performance improvement in the FFT is larger than that in matrix multiplication, because in the former shorter vector-oriented implementation. It the FFT, there are N/2 loops in which the ith loop will be executed 2i times with the vector length of N/2 i+1; in matrix multiplication there are N loops in length of N.

Our performance improvement evaluation is a conservative one, because (1) only the pipeline latency caused by the pipeline depth is considered in the start-up time and other overhead of a pipelined operation is neglected; (2) memory bank conflicts are not considered in our simulation.

If other overhead of a pipeline operation is considered, more start-up time is needed in the vector-oriented implementation than in the matrix-oriented implementation. Therefore more performance improvement is expected.

We consider memory bank conflicts the memory throughput in the matrix-oriented implementation will be higher than that in the vector-oriented implementation. This is proved by a simulation [2] based on an interleaved memory systems with buffers introduced by [8].

The performance improvement in memory-to-memory vector architecture is larger than that in a register-to-register vector architecture, because the depth of chinned pipelines in a memory-to-memory vector architecture is always greater than that in a register-to register vector architecture. (The load and the store must be performed in every operation in a memory-to-memory vector architecture.)

On a register-to-register vector architecture, supporting matrix operations requires that a vector register must be long enough. However it does not mean that a vector register must be impractically long, because supporting matrix operations is effective within a range of problem sizes. For example, for matrix multiplication it is suitable that the length of vector register is chosen to be 1024, because a 32*32 matrix multiplication by the outer-product method has already reached 97% of its peak performance

It is known that many vector architectures have the attractive peak performance, but the sustained performance of them is always much lower than the peak performance. Although introducing more sampling modes will have no influence on the peak performance, our study shows that by direct support of matrix operations, the sustained performance will be improved significantly.

4. CONCLUSIONS

Improvement of performance is obtained only within the certain problem size. That is, if vector operation is long enough to reach the maximum performance of a vector architecture, supporting matrix operations is not much attractive. Our evaluation shows the performance improvement for the problem size between 4*4 and 1024*1024. However, this result is significant for the following reasons:

1. The range between 4*4 and 1024*1024 is typical for the problem solving in the today's vector architectures.
2. Supporting matrix operations is especially useful for a minisupercomputer, because on a minisupercomputer problem of relatively small sizes are more often to be solved.
3. Supporting matrix operations is especially useful for MIMD/SIMD architectures. Supercomputers such as a ETA-10, Cray-2, Cray-3, and Cary X/Y-MP are the multiple processors with vector processors as their node processors. On such architectures a problem of large sizes must be divided into problems of small sizes.

As indicated by J. Backus in his ACM Turing Award Lecture[1], a large part of the traffic in the von Baumann bottleneck is not useful data but merely names of data, as well as operations and data used only to compute such names. Our work highlights the address calculation.

The newly introduced sampling modes can be implemented by the dedicated hard ware called "Address Generator Units" as in the "Data Structure Architecture STARLET" [3, 5, 6, 7]. On this architecture, how to develop programs in a high-level language and how to compile such programs are discussed in [2]

ACKNOWLEDGMENTS

I would like to thank Prof. P. C. Saxena for the guidance in this paper.

REFERENCES

[1] J. Backus. "Can Computing be Liberated From the von Neumann Style? A Functional Style and Its Algebra of Programs". CACM 21, 8, August, 1987, pp. 613-641.

[2] H. Bi. Exploiting Two-dimensional Explicit Parallelism on Vector Architecture. Doctor Thesis of Technical University of Berlin, April, 1991.

[3] A. Bottcher. "STARLET-II System Description & Programming Reference", Presented at the GMD-First Seminar "Pope and STARLET-II; Two Innovative Parallel Architecture", Oct. 2, 1990, Berlin.

[4] M.D. Ercegovac and T. Lang. "Vector Processing", Supercomputers, Class VI Systems, Hardware and Software, S. Fernbach (Editor), Elsevier Science Publishers B.V. (North- Holland), 1986, pp. 29-57.

[5] W.K. Giloi and H.K. Berg. "Introducing the Concept of Data Structure Architecture", Proc. 1977 Int. Conference on Parallel Processing, IEEE Catelog No. 77CH1253-4C, pp. 44-51.

[6] W.K. Giloi and R. Guth. "Concepts and Realization of a High Performance Data Type Architecture". International Journal of Computer and Information Sciences, 11. Jan. 1982. Pp. 25-54.

[7] W.K. Giloi. "Data Structure Architectures and Its Application", in "Berichte der Deutsch-Chinesischen Elektronik Woche in Peking 1987", Teil 4, Data Processing. VDE Verlag, Offenbach, 1987, pp. 15-29.

[8] D.T. Harper and J.R. Jump. " Vector Access Performance in Parallel Memories using a Skewing Storage Schema", IEEE Trans. on Computers, C-36, No. 12. Dec. 1987, pp.. 1440-1449.

[9] J.L. Hennessy and D.A. Patterson. Computer Architecture: Quantitative Approach, Morgan Kaufmann Publishers Inc., 1990.