IMPLEMENTATION OF SPARSE MATRIX ARITHMETIC ON A DSP PROCESSOR

Davorka Petrinovic, Ivan Lukacevic, Davor Petrinovic,

Faculty of Electrical Engineering and Computing
University of Zagreb,
Unska 3, Croatia

Abstract

The paper presents a method for sparse matrix multiplication on a DSP processor. Its high efficiency is a consequence of the proposed pseudo-random data memory access and parallelism of the multifunctional instructions of a DSP. Sparse matrix multiplication is implemented as linear expanded DSP code automatically generated by specially designed program. The method is applied to predictive vector quantization of Line Spectrum Frequencies vectors used in speech coding. It will be shown that the obtained reduction in computational complexity and fixed storage requirements is between two and three-fold.

Key Words

sparse matrix, random access, DSP, speech coding, LSF quantization, predictive vector quantization

1. Introduction

There are many applications that require operations with matrices. In this paper we will limit our interest primarily to multiplication of a matrix with a vector. If matrix dimensions are relatively large and this operation is repeated frequently, it represents a computationally intensive task for the processor. This task can be made less demanding if the employed matrices are sparse (i.e. a certain portion of matrix elements is equal to zero). Sparse matrices may have a regular structure (e.g. diagonal, multi-diagonal, etc.) or the structure may be completely arbitrary (determined by the particular application). The latter case will be analyzed in this paper. It represents the most demanding problem, since reduction of computational complexity and storage requirements can be achieved only by careful and efficient realization.

Architecture of a processor executing the algorithm significantly effects the actual implementation and its performance. Efficient implementation of the sparse matrix arithmetic requires random data memory access. This may not be a problem for a RISC-like processor architectures with large general purpose register files tightly coupled with ALU [1]. CPU registers can be used for temporary storage of vector components which can then be accessed randomly within the program. This is also not a significant problem for general purpose CISC architectures that have sophisticated offset-index addressing modes and can randomly access data memory buffer.

However, random memory access is not conveniently supported on a typical DSP architecture, what can significantly degrade the system performance for applications of interest. DSP processors were primarily designed for signal processing applications with sequential data access and have dedicated hardware support for indirect memory addressing. Typically, this is the only type of data access that can be used simultaneously with ALU or MAC operations in so-called multifunctional instructions. Nevertheless, it will be shown in this paper, that even on a DSP-like architecture it is possible to randomly access data within small memory buffers. This can be done by combining several adjacent circular buffers accessible through a set of address registers. The proposed method called Pseudo-Random Access (PRA) was successfully applied for sparse matrix multiplication and will be illustrated in the paper for a fixed-point implementation on a DSP from the Analog Devices ADSP21XX family.

In the second part of the paper, the proposed sparse matrix multiplication is used for efficient implementation of the switched-adaptive predictive Vector Quantization (VQ) of Line Spectrum Frequencies (LSF) vectors for applications in speech coding [2], [3]. Finally, we will demonstrate how design parameters of the coder affect the computational complexity, memory requirements and performance/quality of the LSF coder.

2. Sparse matrix multiplication

We will consider a particular problem of multiplying a sparse matrix $\mathbf{A}=\{a_{i,j}\}$ by a full column vector $\mathbf{X}=\{x_j\}$ as illustrated in the following equation:

$$\begin{bmatrix} 0 & a_{1,2} & 0 & a_{1,4} & a_{1,5} & 0 & 0 & 0 & 0 & a_{1,10} & 0 & 0 \\ 0 & 0 & a_{2,3} & 0 & 0 & 0 & a_{2,7} & a_{2,8} & 0 & 0 & 0 & 0 \\ a_{3,1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & a_{3,10} & 0 & 0 \\ \vdots & & & & & \vdots & & & & & & \vdots \\ 0 & 0 & 0 & a_{r,4} & 0 & a_{r,6} & a_{r,7} & 0 & 0 & 0 & a_{r,11} & 0 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{12} \end{bmatrix} \quad (1)$$

Matrix $\mathbf{A}$ has r rows and c columns and has an a-priori defined fixed sparse matrix structure (i.e. pattern of

non-zero elements). Since **A** can have an arbitrary structure, the positions (i,j) of its non-zero elements must be encoded and stored in memory together with their values $a_{i,j}$. However, the structure can also be encoded implicitly if the expanded matrix equations are used, as illustrated in (2) for the first three rows of **A**.

$$\begin{aligned} y_1 &= a_{1,2} \cdot x_2 + a_{1,4} \cdot x_4 + a_{1,5} \cdot x_5 + a_{1,10} \cdot x_{10} \\ y_2 &= a_{2,3} \cdot x_3 + a_{2,7} \cdot x_7 + a_{2,8} \cdot x_8 \\ y_3 &= a_{3,1} \cdot x_1 + a_{3,10} \cdot x_{10} \end{aligned} \quad (2)$$

In this case, a separate program must be designed to implement the multiplication for any given structure, thus eliminating the need for indexing. Due to irregular structure of each row, loops cannot be used so the program must be written as linear code. Although program memory utilization is increased due to expanded linear code, it is comparable to the size of memory that would otherwise be required for non-zero index storage. In this case, only non-zero values need to be stored in memory, in sequential order: $a[1]=a_{1,2}$, $a[2]=a_{1,4}$, ..., $a[9]=a_{3,10}$, ... The expanded matrix equations become:

$$\begin{aligned} y_1 &= a[1] \cdot x_2 + a[2] \cdot x_4 + a[3] \cdot x_5 + a[4] \cdot x_{10} \\ y_2 &= a[5] \cdot x_3 + a[6] \cdot x_7 + a[7] \cdot x_8 \\ y_3 &= a[8] \cdot x_1 + a[9] \cdot x_{10} \end{aligned} \quad (3)$$

Multiply-accumulate (MAC) unit of a DSP processor enables efficient implementation of the inner-product calculation. Accumulator holds intermediate results and only the final sum needs to be transferred to a memory location. Therefore, the ith component, y_i, of the output vector **Y** is computed by a sequence of MAC operations. Products of all non-zero elements of the ith matrix row by the appropriate elements of vector **X** are accumulated and finally stored in y_i.

The highest performance can be achieved by using multifunctional instructions in which MAC operations are combined with data fetches, thus preparing the arguments for MAC instructions that follow. To achieve high code density (24-bit instruction word), data access is usually based on indirect addressing by means of dedicated Data Address Generation units (DAG). Typically, DAGs are able to prepare two independent data pointers in each instruction. Although DAG units are very flexible, they were primarily designed for sequential data access. In the stated problem, the matrix elements $a[\ ]$ and the output vector elements y_i are indeed accessed sequentially, but components x_j have to be accessed in a completely random order, as can be seen from the equation (3). Direct implementation of random data access on a DSP processor is usually very inefficient, since the address register must be initialized in a separate instruction prior to each access. This may completely diminish complexity reduction offered by sparse matrices. To solve this problem, a simple and efficient procedure will be proposed for mimicking random access with DSP DAG units that does not need repeated address register initialization. By incorporating the proposed Pseudo-Random Access (PRA) procedure into sparse matrix multiplication, we will demonstrate that the total number of instructions for implementation is closely proportional to the number of non-zero elements of **A**.

3. Pseudo random access using DAG

To access any of c elements of the vector **X** in a random order, one would need c pre-initialized address registers, that could be referenced in data fetch instructions. The second possible implementation is by using a single address register that is post-modified for each consecutive data fetch by one of at least c different values held in c post-modify registers. Neither of these approaches is very good, since the number of address registers and post-modify registers in a DAG is limited. For example, in ADSP21XX family there are only 4 address registers (I) and 4 post-modify registers (M) for one data space **[4]**.

The method proposed in this paper, originally developed in [5**]**, is based on combination of two described approaches. By initializing the m post-modify registers to values M0=0, M1=1, M2=2, ..., Mm-1=m-1, it is possible to modify register I to any address within a circular buffer of the length m. All available address registers I0, I1, ... Ik-1 can be initialized to form k adjacent circular buffers, each of length m. Thus, by using k I registers and m M registers it is possible to randomly access $m \cdot k$ locations in the memory buffer.

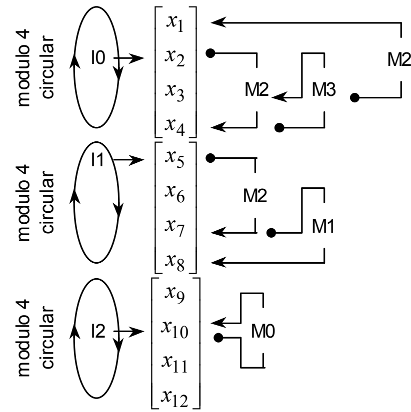

Figure 1 An example of Pseudo-Random Access for sparse matrix multiplication in equation (3)

To illustrate the proposed PRA method, implementation of the same example given by equations (1) to (3) will be analyzed. The **X** vector with 12 elements is split into three groups of four elements as shown in Figure 1. Register I0 is used to access components x_1 to x_4, register I1 for x_5 to x_8 and finally I2 for x_9 to x_{12}. Registers I0, I1 and I2 are initialized to point to the first element used in each group (x_2, x_5 and x_{10}) as shown on the left side of the figure. Registers I4 and I5 point to the beginning of the matrix buffer (&$a[1]$) and the beginning of the output buffer (&y_1) respectively, both residing in the Y data space (program memory, PM, in the case of ADSP21XX processors). Registers I4 and I5 are post-modified by M4=1 after each access.

The actual implementation of the example on DSP processor ADSP2181 is shown in Figure 2.

```
; initialization code
....
; calculate the product of the 1st matrix row and vector elem.
{1}                             MX0=DM(I0,M2),  MY0=PM(I4,M4);
{2} MR=MX0*MY0 (SS),            MX0=DM(I0,M3),  MY0=PM(I4,M4);
{3} MR=MR+MX0*MY0 (SS),   MX0=DM(I1,M2),  MY0=PM(I4,M4);
{4} MR=MR+MX0*MY0 (SS),   MX0=DM(I2,M0),  MY0=PM(I4,M4);
{5} MR=MR+MX0*MY0 (SS),   MX0=DM(I0,M2),  MY0=PM(I4,M4);
{6} PM(I5,M4)=MR1;              save the first result

; calculate the product of the 2nd matrix row and vector elem.
{7} MR=MX0*MY0 (SS),            MX0=DM(I1,M1),  MY0=PM(I4,M4);
{8} MR=MR+MX0*MY0 (SS),   MX0=DM(I1, ... ),   MY0=PM(I4,M4);
{9} MR=MR+MX0*MY0 (SS),   MX0=DM(I0, ... ),   MY0=PM(I4,M4);
{10} PM(I5,M4)=MR1;             save the second result

; calculate the product of the 3rd matrix row and vector elem.
{11} MR=MX0*MY0 (SS),           MX0=DM(I2, ... ),   MY0=PM(I4,M4);
{12} MR=MR+MX0*MY0 (SS),  MX0=DM(..., ... ),  MY0=PM(I4,M4);
{13} PM(I5,M4)=MR1;             save the third result
; ........
; continue for other rows
```

Figure 2 Implementation of the example on ADSP2181

After initialization, the first vector component to be used (x_2) as well as the first matrix element (a[1]=$a_{1,2}$) are fetched into MAC input registers (instruction {1} in Figure 2). Within the same instruction, the index register I0 is updated by M2 since the 4th component of the first group (x_4) will be used in the next instruction {2} (see Figure 1 and equation (3)).

After multiplication of the first pair, x_4 and $a[2]=a_{1,4}$ are fetched and address registers are post-modified. I0 is updated with M3 to address the component x_3 that will be fetched in instruction {5} and later used for multiplication with $a[5]=a_{2,3}$ in instruction {7}. In this case, register M3 is used since adding 3 to the register I0 has the same effect as decrementing it by 1 (circular addressing with modulo 4 is used).

The 5th vector component x_5 (first of the second group) is fetched in instruction {3} by using I1 register and is later used in instruction {4}. There are no other matrix elements in the 1st row corresponding to the second group, so I1 register is updated with M2 to address the 7th component x_7 used with the 2nd matrix row element $a[6]=a_{2,7}$ (fetched in {7} and used in {8}).

In instruction {4}, the 10th component, x_{10}, (the only one used in the third group) is fetched with I2 and used for calculation in {5}. There is no need to update I2, since it will be used in the 3rd row with the same component x_{10}. This is achieved by adding M0=0 to it.

Last calculation for the first matrix row is performed in {5} together with the fetches of the 3rd component x_3 as well as the first nonzero element of the 2nd matrix row a[5]=$a_{2,3}$ that will be used in instruction {7}.

Accumulated value of the first output vector component y_1 is saved in {6} to the output buffer in PM, and I5 register is incremented by M4=1 to point to y_2. The rest of the code can be interpreted analogously.

3.1 Computational complexity

It is obvious from this example that total number of instructions for multiplication of the whole sparse matrix **A** using the proposed PRA based data access is:

$$K_s = C(\mathbf{A}) + r + 1 \tag{4}$$

Number $C(\mathbf{A})$ denotes the number of non-zero elements of **A** and r is the number of its rows. The absolute minimum would be $K_s = C(\mathbf{A})$, i.e. one MAC for each non-zero element of **A**. However, for each of r rows one additional instruction is required for storage of the output component. Also a single additional instruction is required for the first fetch (instruction {1}). Initialization of all used I and M registers was excluded from the above analysis, requiring additional k+m+3 instructions.

The minimal number of instructions to implement full-matrix multiplication K_f can also be determined using equation (4), where $C(\mathbf{A}) = r \cdot c$. Therefore, complexity reduction offered by this particular implementation of sparse matrix multiplication can be determined as:

$$R = \frac{K_f}{K_s} = \frac{r \cdot c + r + 1}{C(\mathbf{A}) + r + 1} \tag{5}$$

3.2 Implementation for large matrices

As was previously discussed, the maximum buffer size that can be randomly accessed using the proposed PRA method is $m \cdot k$, thus limiting the maximum number of columns in **A** to $c \le m \cdot k$. For a typical DSP, m=k=4, so the maximum size of **A** is $r \times 16$. Number of rows r is limited only by the available program memory size needed to implement the expanded multiplication code.

However, the PRA method can also be applied to larger matrices ($c > m \cdot k$) if matrix **A** is split into vertical stripes of width $m \cdot k$, e.g.: $\mathbf{A}_1$, $\mathbf{A}_2$ and $\mathbf{A}_3$. Vector **X** is also split into three sub-vectors $\mathbf{X}_1$, $\mathbf{X}_2$ and $\mathbf{X}_3$ as illustrated in Figure 3.

$$\underbrace{\left[\mathbf{A}_1 \;\; \mathbf{A}_2 \;\; \mathbf{A}_3\right]}_{c=3m \cdot k} \cdot \begin{bmatrix}\mathbf{X}_1\\ \mathbf{X}_2\\ \mathbf{X}_3\end{bmatrix} = \mathbf{Y}_1 + \mathbf{Y}_2 + \mathbf{Y}_3 \quad (r \text{ rows})$$

Figure 3 Implementation of PRA method for multiplication of large sparse matrices

Thus, matrix multiplication can be performed by adding the results of three sub-matrix multiplication:

$$\mathbf{Y} = \mathbf{Y}_1 + \mathbf{Y}_2 + \mathbf{Y}_3 = \mathbf{A}_1 \cdot \mathbf{X}_1 + \mathbf{A}_2 \cdot \mathbf{X}_2 + \mathbf{A}_3 \cdot \mathbf{X}_3 \tag{6}$$

Multiplication of the sub-matrix $\mathbf{A}_i$ and the sub-vector $\mathbf{X}_i$ is performed as already described, since all elements of the sub-vector are accessible randomly through the PRA method. Before proceeding to the next sub-matrix, the address pointers I0 to Ik-1 must be re-initialized to point within k groups of the next sub-vector $\mathbf{X}_{i+1}$. This

re-initialization requires only k additional instructions for each sub-matrix.

Computation of the final result $\mathbf{Y}$ can be performed at the end by adding contributions $\mathbf{Y}_1$, $\mathbf{Y}_2$ and $\mathbf{Y}_3$. Alternatively, it can be done within the main program itself if the accumulated contributions of the previous sub-matrix multiplication are read into MAC before accumulating the new products in each of the rows. A single additional instruction for each of the rows of each of the sub-matrices is sufficient for this task.

4. Application of the method to predictive quantization of speech

The actual problem that inspired the development of the proposed method is related to quantization of Linear Predictive Coding (LPC) parameters of speech. Most contemporary speech coders use the LPC parameters or some equivalent representation such as Line Spectral Frequencies (LSF) (as it is done in this paper) for modeling the transfer function of speech production system. In the encoder, these parameters are frequently grouped into vectors and are quantized by means of Vector Quantization (VQ) before transmission or storage. It has been shown [6] that exploiting dependencies between consecutive vectors enables reduction of the overall bit-rate. One solution for achieving this is by means of Predictive Vector Quantization (PVQ).

A simplified PVQ encoder is illustrated in Figure 4. Prediction $\tilde{\mathbf{x}}(n)$ of the input LSF vector $\mathbf{x}(n)$ is calculated using typically first order vector linear predictor (a matrix) and a previous quantized vector $\hat{\mathbf{x}}(n-1)$. The difference of the original and the predicted vector, $\mathbf{e}(n)$ is then vector quantized by means of split or multistage VQ [7].

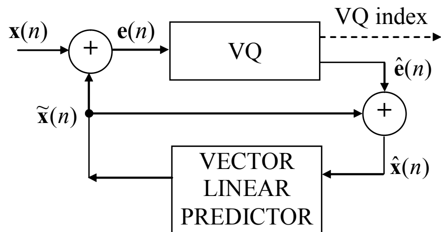

Figure 4 PVQ quantization scheme

To enable adaptation of the predictor to the LSF vector process the predictor is frequently realized as a bank of N switched matrices obtained in the design process, [8], [9]. So called switched-prediction (SP) is therefore performed as in (7), by multiplying each of the matrices $\mathbf{A}_l$ with a vector $\hat{\mathbf{x}}(n-1)$.

$$\tilde{\mathbf{x}}_l(n) = \mathbf{A}_l \cdot \hat{\mathbf{x}}(n-1) \quad , \quad l = 1, \cdots, N \qquad (7)$$

The matrix yielding prediction vector $\tilde{\mathbf{x}}_l(n)$ closest to the input vector $\mathbf{x}(n)$ is then selected for prediction.

Decoder incorporates the same set of prediction matrices and the VQ codebooks as the encoder. The information describing which matrix has actually been used represents side information and has to be sent to the decoder together with the index of the selected VQ codeword. Bit allocation between switched predictor and VQ determines the trade-off between quality and complexity of the quantization scheme. It has been shown [9] that for a fixed bit-rate, the SP-VQ scheme with predictor consisting of only two matrices results with the best performance. On the other hand, this case is also the most computationally demanding since VQ codebooks have to be rather large if perceptually transparent quantization is required.

Recently a research has been performed [2], [3] in order to investigate possible reduction of complexity of the SP-VQ quantization scheme on the account of increased number of switched prediction matrices. It was shown that 16, 32 or 64 matrices instead of 2 can save quite a lot of operations. The approach is most effective for higher bit-rates and lower analysis frame-rates.

To further simplify prediction yet with higher number of matrices (of dimension 10×10 in our aplication), the use of sparse matrices is introduced instead of regular full matrices. Therefore a new method for optimal sparse matrix design was proposed that insures minimal loss of prediction gain due to introduced simplification [3]. The obtained sparse prediction matrices actually reflect only the most significant correlations between components of the input vector process and therefore have irregular structures as can be seen in an example in Figure 5.

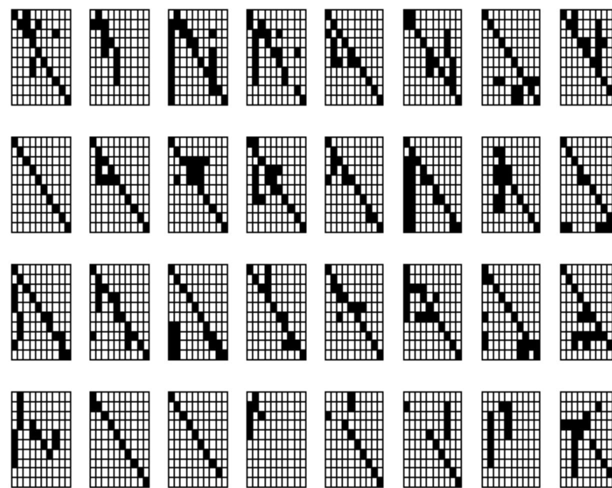

Figure 5 An example of sparse switched prediction matrices with $N = 32$ and $\eta = 6$

Non-zero elements are marked with black rectangles while all the rest are zeros. A design parameter η called the element reduction factor was introduced as the ratio of the total number of matrix elements of the full and the sparse case. In this example factor $\eta = 6$ was used, what means that only one sixth of the total number of full matrix elements is present in sparse matrices. It should be stressed that sparse matrix elements are not equal to those from the full matrices but are specially determined to be optimal for the particular sparse structure. The proposed sparse switched prediction was combined with split VQ with 4-6 split for ten-dimensional vectors $\mathbf{e}(n)$ (i.e.

$c = 10$). A number of LSF quantizers was designed by varying the following parameters: number of switched prediction matrices N, their sparse structure η and resolution of the VQ codebooks (b_1 for the first, 4-dimensional codebook and b_2 for the second, 6-dimensional). Details concerning design and evaluation of the proposed quantizers in Matlab can be found in [2], [3]. On the other hand, their actual realization for the fixed point DSP processor ADSP2181 will be explained in sequel.

4.1 Implementation of SP-VQ with sparse matrices

It is obvious from discussion in the previous section that effectiveness of the proposed SP-VQ method with sparse matrices directly depends on its implementation. For different values of parameters N and η, prediction matrices $\mathbf{A}_l$ have different structures so manual assembly coding would be rather tedious. Therefore, to reduce the programming effort as well as possible errors, an automatic DSP source code generator has been developed. It is a specialized tool realized in Matlab that starts from any number of designed sparse matrices, analyzes them and finally generates an optimal expanded prediction DSP code based on the proposed implementation described in section 3. The rest of the LSF quantizer has been hand-coded in the assembly language and that includes the following blocks: computation of the error vector and its energy, selection of the prediction matrix, split VQ, reconstruction and stabilization of the encoded LSF vector.

The results presented in this paper include PVQ quantizers with $N = 2$ and $N = 32$ switched matrices both designed at the analysis frame-rate of 100 frames/sec. Performance of the quantizers with total bit-rate B between 15 and 19 bits/frame was analyzed as a function of element reduction factor η. Computational complexity, K, expressed as the number of CPU cycles [cyc] for encoding one LSF vector and fixed memory storage requirements, M, were determined for all quantizers as measures of implementation effectiveness. To demonstrate the influence of each part of the algorithm, the results are also presented individually for prediction (denoted with K_p and M_p) and vector quantization (K_{VQ} and M_{VQ}).

The number of DSP cycles for prediction is shown in Table 1. It is divided into two parts. The first part, K_{pp}, is for the actual prediction and it is a function of all three parameters (N, c, η). The second part, K_{ps}, includes computation of the error vectors, their energies and final selection of the prediction matrix so it does not depend on η. Complexity reduction offered by sparse matrices for $N = 32$ can be observed by comparing the first row of the table to the other three with $\eta = 3$, 6 and 10. These values are close to the estimated value given by equation (4) which has to be multiplied with number of matrices N. Indeed, the reduction is almost 10-fold for $\eta = 10$. Unfortunately, the overall complexity reduction due to sparse matrices is lower since the matrix selection code must also be included with a fixed complexity of $K_{ps} = 1601$ cyc. For quantizers with $N = 2$ only the full matrix case with $K_p = 476$ cyc. was considered because the reduction offered by sparse structure is negligible compared to the VQ complexity K_{VQ} as will be shown.

Table 1 Number of CPU cycles for prediction

K_p [cyc]	$N = 2$		$N = 32$	
	K_{pp}	K_{ps}	K_{pp}	K_{ps}
full	379	97	6996	1601
$\eta = 3$	-	-	1460	
$\eta = 6$	-	-	948	
$\eta = 10$	-	-	735	

Fixed memory requirement for prediction part M_p, given in Table 2 includes storage of non-zero elements of all matrices as well as the number of program memory words for the respective implementation (expanded linear code for sparse case). Although the quantizers with 32 sparse matrices may seem inferior compared to the full case with $N = 2$, the overall benefit will be obvious after analyzing the VQ complexity as well.

Table 2 Number of fixed memory locations M_p used for prediction matrices and prediction code

M_p [words]	$N = 2$	$N = 32$
full	215	3215
$\eta = 3$	-	2527
$\eta = 6$	-	1481
$\eta = 10$	-	1055

Depending on the overall bit-rate B, one of the split VQ codebook pairs b_1 - b_2 listed in Table 3 with resolutions b_1 and b_2 is used in conjunction with 1 or 5 bits for the switched predictor. The first five pairs (with total VQ rate $B_{VQ} = b_1 + b_2$ from 10 to 14) are used with $N = 32$, while the last five pairs (B_{VQ} from 14 to 18) are used with $N = 32$. It can be observed that VQ complexity K_{VQ} and table sizes M_{VQ} increase 16-fold as the total rate B_{VQ} is increased from 10 to 18.

Table 3 Complexity K_{VQ} and memory storage requirement M_{VQ} for split VQ

B_{VQ}	b_1 - b_2	K_{VQ} [cyc]	M_{VQ} [words]
10	4-6	1731	448
11	5-6	1992	512
12	5-7	3405	896
13	5-8	6232	1664
14	6-8	6752	1792
15	6-9	12449	3328
16	7-9	13500	3584
17	7-10	24840	6656
18	8-10	26942	7168

Finally, by combining the results for prediction and VQ, the overall complexity K and fixed memory storage requirements M were determined for all combinations of B, N and η. These are shown in Table 4 and Table 5. The required MIPS throughput of a DSP for the whole task can be found by multiplying the values of K in Table 4 by the frame-rate of 100 frames/sec. By comparing the results for $N = 2$ and $N = 32$ reduction in both K and M is obvious, with cca. 3-fold reduction of complexity and 2-fold reduction of storage obtained at higher bit-rates.

Table 4 Total complexity K for encoding one LSF vector

K [cyc]	$N = 2$	$N = 32$			
B	full	full	$\eta = 3$	$\eta = 6$	$\eta = 10$
15	7228	8727	4792	4280	4067
16	12925	8988	5053	4541	4328
17	13976	10401	6466	5954	5741
18	25316	13228	9293	8781	8568
19	27418	13748	9813	9301	9088

Table 5 Total memory size M for tables and predict. code

M [words]	$N = 2$	$N = 32$			
B	full	full	$\eta = 3$	$\eta = 6$	$\eta = 10$
15	2007	3663	2975	1929	1503
16	3543	3727	3039	1993	1567
17	3799	4111	3423	2377	1951
18	6871	4879	4191	3145	2719
19	7383	5007	4319	3273	2847

These results would be misleading without considering the quality of the designed quantizers. Their performance was evaluated by the average log spectral distortion, *SD* between the unquantized and quantized LPC filters and is given in Table 6. More detailed discussion on simulation setup and evaluation can be found in [3]. As can be observed from the table, the increase of distortion compared to the baseline quantizers with $N = 2$ is a consequence of two factors: increased number of matrices and increased η. It is obvious that performance of quantizers with $\eta = 3$ is practically the same as that of full predictor quantizers with $N = 32$. On the other hand, degradation of quality for quantizers with $\eta = 10$ can hardly be justified by relatively small additional reduction of K and M. Therefore, a good compromise is offered by quantizers with η between 3 and 6.

To gain the best insight into the advantages of the proposed sparse matrix approach it is fair to compare the baseline ($N = 2$) and sparse quantizers of the same quality. It leads to comparison of sparse quantizers having one bit more of the total bit-rate then the baseline quantizers. For example, sparse quantizer with $N = 32$, $\eta = 6$ and $B = 18$ has the same quality as the full quantizer with $N = 2$ and $B = 18$ while the complexity and storage are reduced by factors 2.7 and 2.1, respectively.

Table 6 Average spectral distortion values, *SD*

SD [dB]	$N = 2$	$N = 32$			
B	full	full	$\eta = 3$	$\eta = 6$	$\eta = 10$
15	1.09	1.15	1.15	1.17	1.20
16	1.02	1.07	1.08	1.09	1.12
17	0.94	0.98	0.99	1.00	1.04
18	0.87	0.91	0.92	0.93	0.96
19	0.81	0.85	0.86	0.87	0.89

5. Conclusion

It has been demonstrated that even on a DSP-like architecture it is possible to access data memory locations randomly within multifunctional ALU/MAC instructions. With the proposed Pseudo-Random Access (PRA), multiplication of a sparse matrix with a column vector can be performed on a DSP with a minimal number of instructions. A method for predictive vector quantization of LSF vectors with sparse matrices was implemented using the PRA-based approach. Transparent quantization of LSF vectors (*SD* = 1 dB) can be obtained with sparse quantizer with 32 matrices, 1/6th of matrix elements and total bit-rate of 17 bits/vector. Such quantizer requires only 2377 words of table storage and 0.6 MIPS for execution on a DSP.

References

[1] J. L. Hennessy, D. A. Patterson, *Computer architecture: a quantitative approach* (San Francisco, Ca: Morgan Kaufmann Publishers, 1996)

[2] D. Petrinovic, D. Petrinovic, Sparse vector linear prediction with near-optimal matrix structures, *Proc. of International. Workshop on Image and Signal Processing and Analysis*, Pula, Croatia, 2000, pp. 235-240.

[3] D. Petrinovic, D. Petrinovic, Reduced complexity LSF vector quantization with switched-adaptive prediction, *to be presented at International Symposium on Image and Signal Processing and Analysis*, Rome, Italy, September 18-20, 2003

[4] *ADSP-2100 family user's manual* (Analog Devices, Inc., 1995)

[5] I. Lukacevic, *Predictive vector quantization of line spectrum frequencies of the speech signal*, Diploma Thesis (in Croatian), University of Zagreb, Faculty of Electrical Engineering and Computing, 2000

[6] T. Eriksson, J. Linden, J. Skoglund, Interframe LSF quantization for noisy channels, *IEEE Transaction on Speech And Audio Processing*, vol. 7, no. 5, pp. 495-509, Sept. 1999.

[7] A. Gersho, R. M. Gray, *Vector quantization and signal compression*, (Kluwer Academic Publishers, Boston/Dordrecht/London, 1991)

[8] M. Yong, G. Davidson, A. Gersho, Encoding of LPC spectral parameters using switched-adaptive interframe vector prediction, *Proc. ICASSP*, vol.1, 1988, pp. 402-405.

[9] S. Wang, E. Paksoy, A. Gersho, Product code vector quantization of LPC parameters, in *Speech and audio coding for wireless and network applications*, (editors: B. S. Atal, V. Cuperman, A. Gersho, Kluwer Acad. Pub., pp. 251-258, 1993.)