Efficient filtering with the Co-Vector Processor

B.L. Dang[††], Nur Engin[†], G.N. Gaydadjiev[††]
[†] Philips Research Laboratories, Eindhoven,The Netherlands
[††] EEMCS Faculty, Delft University of Technology, The Netherlands
nur.engin@philips.com, georgi@ce.et.tudelft.nl

Abstract—**This paper describes the mapping of Finite Impulse Response (FIR) and Decimation filters on a new DSP architecture: the Co-Vector Processor (CVP) developed by Philips. This architecture is targeting the baseband signal processing algorithms for the third generation mobile communication (3G). CVP is a Very Long Instruction Word (VLIW) architecture with functional units supporting vector parallelism. To exploit efficiently the architecture, a large portion of the targeted DSP algorithms must be properly vectorized. In this paper, different vectorization strategies for FIR and Decimation filters for the CVP architecture are investigated. The approach used is to restructure the original sequential [1] algorithms into block forms [2] that are suitable for parallel processing. In addition, the vectorization should fully utilize the Multiply-Accumulate (MAC) structure. It is shown that for the targeted filters, several good vectorization strategies can be applied. The benchmark results obtained using the proposed strategies outperform results of other architectures previously reported.**

Keywords— **Vectorization; FIR; Decimation; Co-Vector Processor (CVP); vector parallelism.**

I. Introduction

The demand toward high speed, integrated wireless telecommunication services have been increasing rapidly in the last decades. Currently a number of third generation wireless communication standards have emerged, e.g. UMTS/TDD, UMTS/FDD, TD-SCDMA etc., and these standards are expected to co-exist with the 2G standards as well as their extensions (2.5G). Considering the above diversity, the need for a single flexible architecture that has the processing power to support different standards is arising.

Observing the trends as noted above, the so-called Co-Vector Processor (CVP) architecture is being developed. This architecture is to form part of a scalable, low-cost, low-power baseband processing system for 3G handsets. CVP is a vector processor that employs many advanced techniques from VLIW and vector processors. Furthermore, CVP is a co-processor, which is operating as a slave to a host processor. The host processor is responsible for executing the control and irregular tasks, whereas CVP can offer high processing power for the regular operations in the inner loops of 3G algorithms[6].

In order to exploit the CVP architecture efficiently, the targeted DSP algorithms must be properly vectorized, meaning converted to block form from sequential form. Up to now, the vectorization of a number of algorithms have been investigated, eg. RAKE, viterbi decoder [2], FFT. In this paper, two filtering algorithms, i.e. Finite Impulse Response (FIR) and Decimation FIR filter will be investigated. First they will be restructured into appropriate block forms and next mapped to CVP all of this with adequate processing performance.

This paper is organized as follows: Section II presents the CVP architectural description. Section III and IV shows the vectorization and mapping of FIR and Decimation filters to CVP. Section V concludes the paper.

II. The Co-Vector Processor

The details of CVP architecture have already been introduced in a previous publication [6]. Here, only a short overview will be given.

The CVP is a VLIW architecture, comprising seven Functional Units (see Figure 1). At each clock cycle, CVP can issue one very long instruction that contains seven execution slots. Instruction Distribution unit (IDU) is responsible of distributing VLIW instructions to all other units. ALU-MAC Unit (AMU) is the computational heart of CVP, where general purpose integer arithmetic and logic instructions exist alongside instructions tuned to specific applications (such as ACS for Viterbi decoding [2]). Vector Memory Unit (VMU) supplies the data to all units and can support up to 1 vector read/write and 1 scalar read/write each clock cycle. Code Generation Unit (CGU) has Galois field instructions supporting channelization and scrambling code generation for 3G standards. Shift Left, Shift Right and ShuFfLe Units (SLU, SRU and SFU) take care of the operations in which a vector should be converted to/from a sequence of scalars or where the vector elements need to be reordered (shuffled).

Another architectural paradigm employed by CVP is

[1] A sequential algorithm is defined as an algorithm for calculating single output at a time

[2] Inversely, a block algorithm is defined as an algorithm capable of calculating multiple outputs at a time

vector processing. Each functional unit operates on vectors of standard length (256 bits). Most vector instructions for CVP units support scalable precision, so a vector can contain 32 elements of 8-bit single-word, 16 elements of 16-bit double-word or 8 elements of 32-bit quad-word. CVP supports both real and complex fixed-point arithmetics

The vector processing nature of CVP functional units is based on the observation that a large part of the algorithms are vectorizable. However, in many algorithms a small amount of operations exist which are inherently scalar in nature. This can seriously limit the speedup achieved through vectorization even if the non-vectorizable parts form a very small percentage of the total amount of operations in an algorithm (a special case of Amdahl's Law, see [3]). Some of the non-vectorizable operations are related to looping and address calculation. These are dealt by means of loop-control units in the IDU and address calculation units in the VMU [6], so these operations are executed in parallel with the vector operations, and there is no speed limitation caused by them. Furthermore, there are some irregular scalar operations in many algorithms. To overcome the slowdown due to these operations, scalar processing hardware has been included in the CVP functional unit alongside the hardware for vector processing, so that a scalar instruction can be executed in parallel with a vector instruction. We call the combination of these scalar processing elements the "scalar path". Examples of scalar functionality are scalar send and receive instructions at the VMU and instructions between a scalar and vector (by means of scalar broadcasting) in the AMU.

Unlike traditional vector processors, CVP supports only unit-stride vector accesses. This poses a problem on data-packing when exploiting Instruction Level Parallelism exposed in many algorithms [4], [3]. For instance, data gathering is very important for Decimation filters. This can be done easily in traditional vector processors where non-unit stride accesses are supported. In the case of CVP, this problem is addressed by using the SHUFFLE (SFU) unit (Figure 1) that is able to rearrange elements of a vector in an arbitrary order.

III. Vectorization of FIR filters

In this section, we describe the mapping of an N-tap FIR filter with 16-bit fixed point coefficients and inputs to CVP [3]. It is supposed that the number of inputs are multiple of 16 as it matches the architecture of CVP (256 wide data path). However, this assumption would not reduce the generality of the problem since normally, when the number of inputs is not a multiple-of-16 number and much larger than the number of filter coefficients , we can always use zero padding to extend the length of the input vector to a proper number. To exploit the parallelism offered by CVP first a proper parallelization strategy need to be chosen given an application. In other words, the investigated problem must be vectorized or algorithmically restructured.

[3]Because of the precision granularity of CVP is 8 bits, this is also the case for filters with coefficients and inputs encoded by 9-bit to 16-bit fixed-point representation

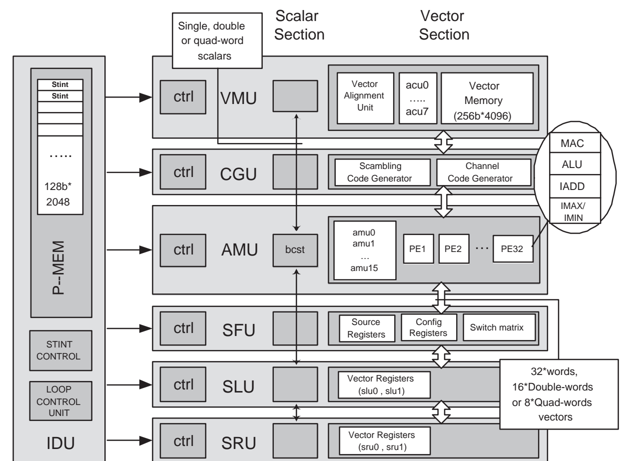

Fig. 1. Block diagram of CVP

A. The vectorization strategies

This section describes two vectorization strategies. We show that the second strategy exploits efficiently the architecture. This strategy uses both the vector and the scalar data paths available in CVP.

B. The horizontal strategy

A FIR filter is presented by

$$y_n = \sum_{k=0}^{N-1} h_k x_{n-k} \tag{1}$$

where $h_k's$ $(0 \leq k \leq N-1)$ are the filter coefficients and $x_k's$ are the inputs. This equation can also be represented in matrix form (for the case 16 outputs are calculated from an N-tap FIR filter)
or using compact notation:

$$Y = \mathbf{X} * H \tag{2}$$

where Y and H are the output and coefficient vectors respectively and $\mathbf{X}$ is the input matrix. An obvious way to parallelize this algorithm for CVP's vector processing is to perform the multiplications horizontally and then add the matrix elements together (called "intra-add") to get a single output at a time.

$$\begin{bmatrix} y_k \\ y_{k+1} \\ \vdots \\ \vdots \\ \vdots \\ y_{k+15} \end{bmatrix} =$$

$$\begin{bmatrix} x_k & x_{k-1} & \cdots & x_{k-N+1} \\ x_{k+1} & x_k & \ddots & x_{k-N+2} \\ \vdots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ x_{k+15} & x_{k+14} & \cdots & x_{k-N+16} \end{bmatrix} \begin{bmatrix} h_0 \\ h_1 \\ \vdots \\ \vdots \\ \vdots \\ h_{N-1} \end{bmatrix}$$

Fig. 2. The matrix representation of the N-tap FIR filter

$$y_k = h_0 x_k + h_1 x_{k-1} + \cdots + h_{N-1} x_{k-N+1} \tag{3}$$

In this paper, we define an intra-add operation as follows:

Definition 1: The intra-add of every L elements in a vector of length N: $X = [x_0,\ x_1, \cdots,\ x_{N-1}]$ is also a vector of length N:
$Intra_Add_L(X) = R\ \ (R = [r_0,\ r_1, \cdots,\ r_{N-1}])$ where

$$r_k = \begin{cases} \sum_{i=k}^{k+L-1} x_i & \text{if} \quad (k\, mod\, L) = 0 \\ 0 & \text{if} \quad (k\, mod\, L) \neq 0 \end{cases}$$

Intra-add operation is available as a CVP instruction. Using the above definition, Equation 3 can be re-written in a compact form

$$y_k = Intra_Add_N(X_k * H) \tag{4}$$

where $X_k = [x_k, x_{k-1}, \cdots, x_{k-N+1}]$ and N is the filter length. The vector multiplication in the above equation is element-wise.

The advantage of such implementation is that the program will contain only one loop and thus is simple and small in code size. However, this implementation would result in significant inefficiency since we have to scale the results of the multiplication before intra-adding. This increases the latency due to the operations inside the loop, therefore the performance of this strategy will not be high.

C. The Vertical Strategy

An alternative approach is to vectorize the algorithm vertically or partition the input matrix **X** column-wise (or vertically). The FIR equation in Equation 1 can be rewritten as follows:

$$Y = X_0 * h_0 + X_1 * h_1 + \cdots + X_{N-1} * h_{N-1} \tag{5}$$

where

$$\begin{cases} X_0 = [x_k,\ x_{k+1},\ x_{k+2}, \cdots,\ x_{k+15}] \\ X_1 = [x_{k-1},\ x_k,\ x_{k+1}, \cdots,\ x_{k+14}] \\ X_2 = [x_{k-2},\ x_{k-1},\ x_k, \cdots,\ x_{k+13}] \\ \cdots\cdots \\ \cdots\cdots \\ X_{15} = [x_{k-15},\ x_{k-14},\ x_{k-13}, \cdots,\ x_k].\ ^{1} \end{cases}$$

In the vectorized expression 5, instead of calculating a single output at a time, 16 outputs will be calculated at once. This scheme needs N vector Multiply and Accumulate (MAC) operations to get those 16 outputs. The operands of a MAC operation are an input vector (X_i) and a coefficient scalar (h_k). By the application of the broadcast register in ALU and MAC Unit (AMU), a scalar can be received from other functional units. It is then broadcast or replicated across an entire vector and hence can be used as an operand in the MAC operation. We have:

$$Y = MAC(X_0,\ h_0) + MAC(X_1,\ h_1) + \cdots + MAC(X_{N-1},\ h_{N-1}) \tag{6}$$

The program will now contain two loops - an inner loop to calculate 16 outputs and an outer loop that evolves vertically along the input matrix (**X**). This will result in a longer program compared to the first strategy and is shown in Figure 3.

```
initialize_address_pointers();
LOOP(16 times);
    {
    load(data_vector x);
    load(coefficient_scalar b);
    multiply_accumulate(x, b);
    update_address_pointers();
    }
```

Fig. 3. Pseudo code for the algorithm

D. The algorithm implementation for CVP

In this section, the algorithm presented in Figure 3 will be mapped into CVP. The inner loop contains only four tasks and can be packed into single CVP's VLIW instruction. The most important part is to get data ready for MAC

[4]Notice that the definition of X_k in this strategy is different from the previous one

instructions; this includes initializing, updating pointers and loading pointed data into the registers.

Figure 4 shows the organization of the inputs and coefficients in the CVP's memory. Two pointer registers are needed for addressing an input vector and a coefficient scalar. At the initialization stage, the first pointer (e.g. acu0) must point to the first data vector $\underline{x}_0$ and the second (e.g. acu1) points to the beginning of the filter's coefficient vector (b_0). This coefficient will be sent directly from Vector Memory Unit (VMU) using "scalar-send" instruction (SSND).

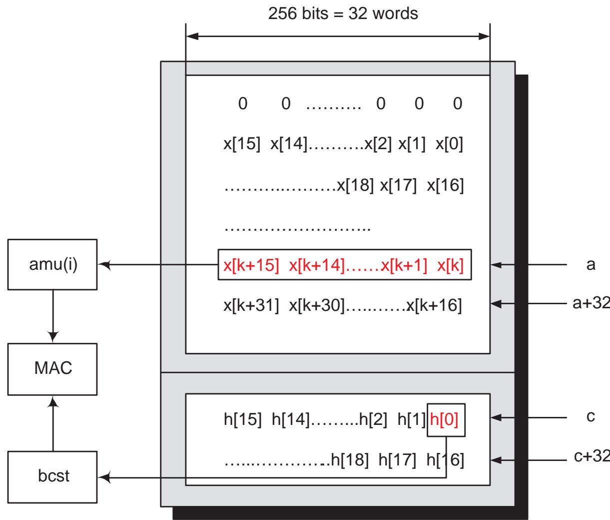

Fig. 4. Organization of filter coefficients and input in CVP's vector memory

E. Performance

The benchmark results of various DSPs for calculating 1024 outputs with a 50-tap FIR filter are presented in Table I. As can be seen from the table, the performance of CVP is better than its counterparts at equal or lower clock frequency. This has been achieved by the vector parallel architecture and the efficient algorithm mapping presented in this paper.

	Clock-rate (Mhz)	Clock cycles	Execution time (μs)
CVP	300	3728	12.4
ALTIVEC	600	9334	15.6
TIGERSHARC	300	7200	24
TMS320C64x	600	16243	27

TABLE I
PERFORMANCE OF FIR FILTER ON CVP

IV. Vectorization of Decimation filters

Figure 5 shows the block diagram of a decimation filter with decimation factor M.

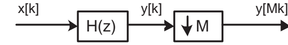

Fig. 5. Decimation and Interpolation filters

or in matrix representation as shown in Figure 6.

$$\begin{bmatrix} y_k \\ y_{k+2} \\ \vdots \\ \vdots \\ y_{k+14} \\ \vdots \end{bmatrix} = \begin{bmatrix} x_k & x_{k-1} & \cdots & x_{k-N+1} \\ x_{k+2} & x_{k+1} & \ddots & x_{k-N+3} \\ \vdots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ x_{k+14} & x_{k+13} & \cdots & x_{k-N+15} \\ \vdots & \ddots & \ddots & \vdots \end{bmatrix} \begin{bmatrix} h_0 \\ h_1 \\ \vdots \\ \vdots \\ \vdots \\ h_{N-2} \\ h_{N-1} \end{bmatrix}$$

Fig. 6. The matrix representation of a decimation FIR filter with factor of 2

The most straight-forward strategy to implement this algorithm is to calculate a single output at a time directly from the equation presented in Figure 6.

$$y_{2n} = \sum_{k=0}^{N-1} h_k x_{2n-k} \tag{7}$$

This strategy, however, is exactly the same as the Horizontal strategy for FIR, hence not efficient. In this section, we propose two new strategies that exploit better the MAC structure of decimation filter and the CVP architecture.

A. Strategy 1

Another way to reduce the computation requirement is to use polyphase representation of the decimation filters [5]. Figure 7 shows the block diagram of the 2-component polyphase representations of the decimation shown in Figure 5 ($M = L = 2$).

In Figure 7, the input sequence (x_k) is divided into two shorter sequences. The first sequence contains the even-numbered input samples and the second contains the odd-numbered ones. As a result, the lengths of both input sequences (x_{2k} and x_{2k+1}) are half of the original (x_k).

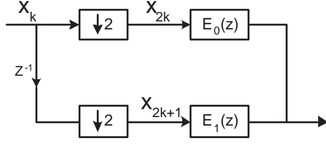

Fig. 7. Polyphase representations of Decimation and Interpolation filters

These two shorter sequences are then filtered by $E_0(z)$ and $E_1(z)$ whose lengths are again half of the original filter $H(z)$. The filters $H(z)$, $E_0(z)$, $E_1(z)$, $R_0(z)$ and $R_1(z)$ are related by the following formulas:

$$\begin{cases} H(z) = \sum_{n=-\infty}^{\infty} h(n) z^{-n} \\ E_0(z) \equiv R_0(z) = \sum_{n=-\infty}^{\infty} h(2n) z^{-n} \\ E_1(z) \equiv R_1(z) = \sum_{n=-\infty}^{\infty} h(2n+1) z^{-n} \end{cases}$$

For a N-tap FIR filter, single output requires N multiplications. Hence for M outputs, we need MxN multiplications. By using the polyphase representations, both the number of input samples and the length of each component FIR filters are reduced by the factor of two. As a consequence, each decomposed filter - $E_0(z)$ or $E_1(z)$ - will be used to calculate only $M/2$ outputs. The total number of multiplications required will be $N/2M/2 + N/2M/2 = MN/2$ which is half of the original computation requirement.

B. Strategy 2

The previous strategy has improved the efficiency of the decimation filter but still suffers from the shuffling operations required. In this section, we derive another algorithm which is able to exploit the MAC structure of the algorithm and thus increase the efficiency when implemented in DSP architectures.

The matrix equation of a decimation filter (with the decimation factor of 2) shown in Figure 6 can be rewritten as follows:

$$Y = X_0 * H_0 + X_1 * H_1 + \cdots + X_{N-1} * H_{N-1} \quad (8)$$

where

$$\begin{cases} X_0 = [x_k,\ x_{k+1},\ x_{k+2}, \cdots,\ x_{k+15}] \\ X_1 = [x_{k-1},\ x_k,\ x_{k+1}, \cdots,\ x_{k+14}] \\ \cdots\cdots \\ \cdots\cdots \\ X_{N-1} = [x_{k-N+1},\ x_{k-N+2},\ x_{k-N+3}, \cdots,\ x_{k-N+16}]. \end{cases}$$

and

$$\begin{cases} H_0 = [h_0,\ 0,\ h_0,\ 0, \cdots,\ h_0,\ 0] \\ H_1 = [h_1,\ 0,\ h_1,\ 0, \cdots,\ h_1,\ 0] \\ H_2 = [h_2,\ 0,\ h_2,\ 0, \cdots,\ h_2,\ 0] \\ \cdots\cdots \\ \cdots\cdots \\ H_{N-1} = [h_{N-1},\ 0,\ h_{N-1},\ 0, \cdots,\ h_{N-1},\ 0]. \end{cases}$$

Notice that X_1 is produced by shifting X_0 right one element and half of the elements in H_k are 0's, so Equation 8 can rewritten as follows:

$$Y = Intra_Add_2(X'_1 * H'_1 + X'_3 * H'_3 + \cdots + X'_{N-1} * H'_{N-1}) \quad (9)$$

where

$$\begin{cases} X'_1 = [x_{k-1},\ x_k,\ x_{k+1}, \cdots,\ x_{k+14}] \\ X'_3 = [x_{k-3},\ x_{k-2},\ x_{k-1}, \cdots,\ x_{k+12}] \\ \cdots\cdots \\ \cdots\cdots \\ X'_{N-1} = [x_{k-N+1},\ x_{k-N+2},\ x_{k-N+3}, \cdots,\ x_{k-N+16}]. \end{cases}$$

and

$$\begin{cases} H'_1 = [h_1,\ h_0,\ h_1,\ h_0, \cdots,\ h_1,\ h_0] \\ H'_3 = [h_3,\ h_2,\ h_3,\ h_2, \cdots,\ h_3,\ h_2] \\ H'_5 = [h_5,\ h_4,\ h_5,\ h_4, \cdots,\ h_5,\ h_4] \\ \cdots\cdots \\ \cdots\cdots \\ H'_{N-1} = [h_{N-1},\ h_{N-2},\ h_{N-1},\ h_{N-2}, \cdots,\ h_{N-1},\ h_{N-2}]. \end{cases}$$

C. Performance

Table II shows the performance and the code sizes of the implementations of the two vectorization strategies on a 32-tap FIR decimation filter with 96 input samples. We have shown that it is possible to escape the unit-stride limitation in CVP. Additional investigation shows that the performance of Strategy 2 increases when the lengths of FIR filters used grow.

	Clock cycles	Code size (bytes)
Strategy 1	224	6,232
Strategy 2	148	4,482

TABLE II
PERFORMANCE OF VARIOUS PARALLELIZATION STRATEGIES ON CVP

The first strategy's performance is moderate since we have to spend additional cycles on the SHUFFLING operation (about 80 cycles for shuffling 6 16*1 input vectors). However, this strategy can be applied to a wide range of

decimation filters with different decimation factors. Furthermore, this strategy can also be applied for interpolation filers (simply by reversing the processing order). Unfortunately, the performance of this strategy decreases linearly as the decimation factors increase due to the shuffling operation.

V. Conclusions

We explored and studied two of the most important digital filtering algorithms in details. Different approaches to vectorize the algorithms have been investigated and implemented for CVP without loss of generality. The following issues were highlighted: The Vertical vectorization strategy for FIR filters not only guarantees the efficiency of its implementations but it can also be applied for FIR filters of arbitrary lengths. Although the number of inputs (or outputs) for the strategy should be a multiple-of-16 number, it does not affect the generality since zero-padding can be applied. Two vectorization approaches for Decimation filters with small factors were investigated . It has been shown how decimation filters with arbitrary large factors can be decomposed into multiple stages. In addition the proposed strategies where implemented on CVP. The performance of the presented implementations outperforms previously reported results.

References

[1] Ronald E. Crochiere and Lawrence R. Rabiner, *Multirate Digital Signal Processing*, Prentice Hall, 1983.

[2] N. Engin and K. van Berkel, *Viterbi decoding on a co-processor architecture with vector parallelism*, Proceedings of IEEE Workshop on Signal Processing Systems (2003), 334–339.

[3] John L. Hennessy and David A. Patterson, *Computer Architecture – A Quantitative Approach*, third ed., Morgan Kaufmann Publishers, 2002.

[4] Samuel Larsen and Saman Amarasinghe, *Exploiting Superword Level Parallelism with Multimedia Instruction Sets*, Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation, 2000, pp. 145–156.

[5] P.P. Vaidyanathan, *Multirate Digital Filters, Filter Banks, Polyphase Networks and Applications: A Tutorial*, vol. 78, IEEE, Jan 1990.

[6] K. van Berkel, P. Meuwissen, N. Engin, and S. Balakrishnan, *CVP: A programmable co vector processor for 3G mobile baseband processing*, Proceedings of World Wireless Congress (on CD-ROM) (2003).