Automatic Code Generation for SIMD DSP Architectures: An Algebraic Approach

J.P. Robelly, G. Cichon, H. Seidel and G. Fettweis
Vodafone Chair for Mobile Communications Systems
Dresden University of Technology
01062 Dresden, Germany
Email: robelly@ifn.et.tu-dresden.de

***Abstract*— Driven by the ever increasing algorithm complexity on the field of mobile communications systems, SIMD DSP architectures have emerged as an approach that offers the necessary processing power at reasonable levels of die size and power consumption. However, this kind of DSP architectures imposes new challenges for programmers, since algorithms have to be designed to exploit the available parallelism on the processor. Taking as a starting point an algebraic framework that captures the SIMD computational model, we report in this paper about our efforts to design and automatically generate object code for our family of DSP architectures independent of the available SIMD parallelism. We show how these algebraic structures can be used as a high level programming language that offers a unified approach to design and describe algorithms using SIMD parallelism. Moreover, we show how these algebraic structures offer concise rules for the automatic code generation.**

I. Introduction

SIMD (Single Instruction Multiple Data) parallelism offers the potential to speed up the computation of signal processing algorithms. This is of paramount importance in high end applications where the required processing power to compute the algorithms must be trade-off with power consumption and die size issues.

However, SIMD DSP processors are a major challenge for programmers, since algorithms have to be designed to exploit the available parallelism. In [1] we have presented a novel DSP architecture that offers SIMD parallelism. In fact, STA (Synchronous Triggered Architecture) based processor cores can be automatically generated by means of our integrated design flow. Thus, we are able to generate processor cores with different levels of SIMD parallelism. This imposes a new challenge to algorithm designers: scalability. Hence, the question here arises is not only how a programmer can design an algorithm that exploit the available parallelism, but how this algorithm can be designed in such a way that it can be easily scaled.

Our approach to deal with these challenges is based on a set of parameterizable algebraic structures that captures the SIMD computational model. This algebraic components reveal the kind of computations that our family of STA processors can support. Thus, the task of algorithm design reduces to find the way an algorithm can be expressed by means of these algebraic constructs. We use Matlab syntax as the language with which our algebraic structures are programmed. Thus, algebra can be regarded as a unified language used in order to design algorithms into SIMD DSP architectures. Moreover, it provides a set of rules that can be used to automatically generate object code.

II. Algebraic Framework

A fundamental role on the algebraic framework plays the *Kronecker* product. Davio [2] in his classical paper developed an algebraic framework, where he established the connection between the *Kronecker* product of matrices and stride permutations. In his paper he proved the important *commutation theorem* of Kronecker products

$$A \otimes B = P(mn, m)(B \otimes A)P(mn, n), \tag{1}$$

where $P(L, s)$ is a matrix that represents the permutation of L elements with stride s and $A \otimes B$ is a matrix known as the Kronecker product of the $m_A \times n_A$ matrix A and the $m_B \times n_B$ matrix B that is defined as follows:

$$A \otimes B = \begin{bmatrix} a_{0,0}B & \dots & a_{0,m_A-1}B \\ a_{1,0}B & \dots & a_{1,m_A-1}B \\ \vdots & \dots & \vdots \\ a_{n_A-1,0}B & \dots & a_{n_A-1,m_A-1}B \end{bmatrix} \tag{2}$$

This remarkable result enabled researchers to formulate different fast algorithms for signal transformations like discrete fourier transforms, walsh-hadamard transforms, discrete cosine, sine transform, etc. using the same algebraic framework. Tolimieri [3] uses Kronecker products in order to formulate FFT algorithms for different parallel architectures. Tolimiere makes also the observation that different models of parallel computation can be described by means of Kronecker products. This has originated a lot of research in order to automatically generate code of signal transformations like FFT, Cosine and Sine transform for different parallel architectures [4]. Although there has been a lot of research on the generation of signal transformations, there has been scarce research on extending these ideas in order to support a wider range of digital signal processing algorithms.

In order to characterize the SIMD computational model let A be an $m \times m$ matrix, $\underline{x}$ a $Nm \times 1$ vector, I_N the $N \times N$ identity matrix and consider the following important

expression:

$$\begin{aligned}\underline{y} &= (A \otimes I_N)\underline{x} \qquad (3)\\ &= \begin{bmatrix} a_{0,0}I_N & \dots & a_{0,m-1}I_N \\ a_{1,0}I_N & \dots & a_{1,m-1}I_N \\ \vdots & \dots & \vdots \\ a_{m-1,0}I_N & \dots & a_{m-1,m-1}I_N \end{bmatrix} \begin{bmatrix} \underline{x}_0 \\ \underline{x}_1 \\ \vdots \\ \underline{x}_{m-1} \end{bmatrix},\end{aligned}$$

where using Matlab syntax we define the vector of N elements $\underline{x}_i = \underline{x}\big(iN : (iN+N-1)\big)$ for $i = 0, 1, \dots, m-1$. Equation (3) is known as a Kronecker vector factor [3], since it expresses the computation of m vectors $\underline{y}_i$ of size $N \times 1$ as vector operations that deal with vectors of size N. This is better illustrated if we rewrite equation (3) as follows:

$$\underline{y}_i = \sum_{q=0}^{m-1} (a_{i,q} \otimes I_N)\underline{x}_q, \qquad (4)$$

for $i = 0, 1, 2, \dots, m-1$ and using Matlab syntax again $\underline{y}_i = \underline{y}\big(iN : (iN+N-1)\big)$. Equations (3) and (4) captures the SIMD computation model. Further we assume for the level of SIMD parallelism

$$N = 2^n \quad \text{for } n = 1, 2, 3, \dots$$

III. Application Example 1: Tensor Product Representation of FIR Filters

In this section we show by means of a simple example how the algebraic notation presented in section II can be used in order to describe the SIMD computation of an algorithm.

Let $\underline{x}$ and $\underline{y}$ be input and output vectors defined as

$$\begin{aligned}\underline{x} &= [x(k) \quad x(k+1) \quad x(k+2) \quad x(k+3)]^T,\\ \underline{y} &= [y(k) \quad y(k+1) \quad y(k+2) \quad y(k+3)]^T,\end{aligned}$$

and a vector of coefficients defined as

$$\underline{h} = [h_0 \quad h_1 \quad h_2 \quad h_3]^T.$$

We consider the following computation

$$\underline{y} = H\underline{x}, \qquad (5)$$

where the transformation matrix is formed taking the coefficients of the vector $\underline{h}$ and has a Toeplitz structure as follows:

$$H = \begin{bmatrix} h_0 & 0 & 0 & 0 \\ h_1 & h_0 & 0 & 0 \\ h_2 & h_1 & h_0 & 0 \\ h_3 & h_2 & h_1 & h_0 \end{bmatrix}$$

Assuming that two results can be computed in parallel, we can write for the computation of (5) the following equations

$$\begin{aligned}\underline{y}_0 = &\begin{bmatrix} h_0 & 0 \\ 0 & h_0 \end{bmatrix}\begin{bmatrix} x(k) \\ x(k+1) \end{bmatrix} + \begin{bmatrix} h_1 & 0 \\ 0 & h_1 \end{bmatrix}\begin{bmatrix} 0 \\ x(k) \end{bmatrix}\\ &+ \begin{bmatrix} h_2 & 0 \\ 0 & h_2 \end{bmatrix}\begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} h_3 & 0 \\ 0 & h_3 \end{bmatrix}\begin{bmatrix} 0 \\ 0 \end{bmatrix},\end{aligned}$$

$$\begin{aligned}\underline{y}_1 = &\begin{bmatrix} h_0 & 0 \\ 0 & h_0 \end{bmatrix}\begin{bmatrix} x(k+2) \\ x(k+3) \end{bmatrix} + \begin{bmatrix} h_1 & 0 \\ 0 & h_1 \end{bmatrix}\begin{bmatrix} x(k+1) \\ x(k+2) \end{bmatrix}\\ &+ \begin{bmatrix} h_2 & 0 \\ 0 & h_2 \end{bmatrix}\begin{bmatrix} x(k) \\ x(k+1) \end{bmatrix} + \begin{bmatrix} h_3 & 0 \\ 0 & h_3 \end{bmatrix}\begin{bmatrix} 0 \\ x(k) \end{bmatrix},\end{aligned}$$

where

$$\begin{aligned}\underline{y}_0 &= [y(k) \quad y(k+1)]^T\\ \underline{y}_1 &= [y(k+2) \quad y(k+3)]^T\end{aligned}$$

Using the notation introduced in section II, we can write for the computation of equation (5) the following

$$\underline{y}_i = \sum_{q=0}^{3} (h_q \otimes I_2) \cdot \Big[(Z_2)^{(q) \bmod 2}\underline{x}_{i-\lfloor \frac{q}{2} \rfloor} + (Z_2^T)^{(2-q) \bmod 2}\underline{x}_{i-1-\lfloor \frac{q}{2} \rfloor}\Big],$$

where $0 \le i < 2$, $\lfloor \cdot \rfloor$ is the floor operator, Z_2 is a 2×2 shift matrix defined as

$$Z_N = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix},$$

and the vector $\underline{x}_j$ is defined as

$$\begin{aligned}\underline{x}_j &= \big[x\big(k+2j\big) \quad x\big(k+2j+1\big)\big]^T \quad \text{for } 0 \le j < 2,\\ \underline{x}_j &= [0 \quad 0]^T \qquad\qquad\qquad\qquad \text{otherwise.}\end{aligned}$$

The equations we developed above can be generalized for an arbitrary level of SIMD parallelism N and for a vector of coefficients:

$$\underline{h} = [h_0 \quad h_1 \quad \dots \quad h_{p-1}],$$

where p/N is a positive integer greater that one. For the input and output vectors we can write

$$\begin{aligned}\underline{x} &= [x(k) \quad x(k+1) \quad \dots x(k+L-1)]^T,\\ \underline{y} &= [y(k) \quad y(k+1) \quad \dots y(k+L-1)]^T.\end{aligned}$$

Further, we assume that the number of input samples to be processed L is also a multiple of N, thus the following holds:

$$\frac{L}{N} = J \qquad \text{for } J = 1, 2, 3, \dots$$

In order to formulate the scalable computation of equation (5), we partition the input and output vectors in sub-vectors of size N. Thus, for the input vectors we can write

$$\underline{x}_j = [x(k+Nj) \quad x(k+Nj+1) \dots x(k+Nj+N-1)]^T,$$

for $0 \le j < J$. Otherwise, $\underline{x}_j$ is a vector with N zero elements. For the output vectors, we can write

$$\underline{y}_i = [y(k+Nj) \quad y(k+Nj+1) \dots y(k+Nj+N-1)]^T,$$

for $0 \leq i < J$. Finally the computation of (5) for an arbitrary level of SIMD parallelism can be expressed by means of the following expression:

$$\underline{y}_i = \sum_{q=0}^{p-1} (h_q \otimes I_N) \cdot \Big[(Z_N)^{(q) \bmod N} \underline{x}_{i-\lfloor \frac{q}{N} \rfloor} + (Z_N^T)^{(N-q) \bmod N} \underline{x}_{i-1-\lfloor \frac{q}{N} \rfloor} \Big] \tag{6}$$

In this equation I_N and Z_N are the $N \times N$ identity and shift matrix respectively. Thus, equation (6) describes an algorithm for the implementation of the core computations of an FIR filter with an scalable level of SIMD parallelism. This algorithm can be extended by any suitable partitioning technique like for example *overlapping save* in order to obtain the time-continuous computation of the filter.

IV. Application Example 2:Tensor Product Representation of Recursive Filters

In [5], we presented a block formulation of pure recursive filters using Kronecker vector factors. The block formulation was derived using a lifting-isomorphism [6] with a raising factor N applied to the state-space equations of the serial formulation of the recursive filter. Our starting point for the derivation of the raised algorithm is the serial formulation of a recursive filter of order p, namely

$$y(k) = u(k) + \sum_{i=1}^{p} a_i y(k-i).$$

The state-space representation of this algorithm is given by

$$\begin{aligned} \underline{x}(k+1) &= A\underline{x}(k) + Bu(k) \\ y(k) &= C\underline{x}(k) + Du(k), \end{aligned}$$

where the state vector $\underline{x}(k) = [x_1(k) \quad x_2(k) \ldots x_p(k)]^T$ contains past computed samples. Thus,

$$x_i(k) = y(k-i) \quad \text{for } i = 1, 2, \ldots, p$$

For the system matrices we have

$$A = \begin{bmatrix} a_1 & a_2 & \ldots & a_p \\ 1 & 0 & \ldots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \ldots & 1 & 0 \end{bmatrix},$$

$$B = [1 \quad 0 \quad \ldots \quad 0]^T,$$

$$C = [a_1 \quad a_2 \quad \ldots \quad a_p]^T,$$

$$D = 1$$

We define the input and output vectors of the raised algorithm as follows:

$$\begin{aligned} \underline{u}(k) &= [u(Nk) \quad u(Nk+1) \quad \ldots \quad u(Nk+N-1)]^T \\ \underline{y}(k) &= [y(Nk) \quad y(Nk+1) \quad \ldots \quad y(Nk+N-1)]^T, \end{aligned}$$

where N is the raising factor. Thus, and according to [5], we can write for the Kronecker vector formulation of the algorithm the following

$$\underline{y}(k) = \sum_{i=1}^{p} \big(y(Nk-i) \otimes I_N\big)\underline{c}_i + (D \otimes I_N)\,\underline{u}(k) + \sum_{q=1}^{N-1} \big(CA^{q-1}B \otimes I_N\big)\, Z_N^q \underline{u}(k), \tag{7}$$

where $\underline{c}_i$ are vectors formed by the columns of a raised system matrix

$$C^{[R]} = \begin{bmatrix} C \\ CA \\ \vdots \\ CA^{N-1} \end{bmatrix},$$

and Z_N is the $N \times N$ shift matrix.

The importance of equation (7) resides on the matter that it expresses the computation of the algorithm in terms of Kronecker vector factors. The algorithm deals with input vectors of size N and produces output vectors of size N and thus, it is independent of the available level of SIMD parallelism.

V. Kronecker Compiler

The automatic code synthesis is based on the matter that algorithms expressed as in equations (6) and (7) uses certain algebraic structures, which have a direct interpretation in processor instructions. For example, let us take the following expression of equation (7)

$$\big(y(Nk-i) \otimes I_N\big)$$

This expression means that the scalar $y(Nk-i)$ has to be broadcasted to all the data paths of the parallel DSP architecture. Thus, this algebraic structure will be translated into a broadcast data transfer instruction. Let us take a new example

$$\big(y(Nk-i) \otimes I_N\big)\underline{c}_i$$

This expression contains two actions that have to be carried out by the processor: 1.- broadcast of $y(NK-i)$, 2.- componentwise vector multiplication. As we can see from this example, the algorithm is expressed in such a way that enables scalability, since all the components involved in the equation have as a parameter the number of data paths of the processor N. Finally, we consider the expression

$$\sum_{i=1}^{p} \big(y(Nk-i) \otimes I_N\big)\underline{c}_i,$$

where p vector MAC operations are described. In the example above, we have presented a simple data transfer, namely broadcasting. However, we have extended the number of elementary algebraic structures in order to support a wider range of algorithms. This was the case of the $N \times N$ shift matrix Z_N, which resembles the *Zurich-Zip* data transfer.

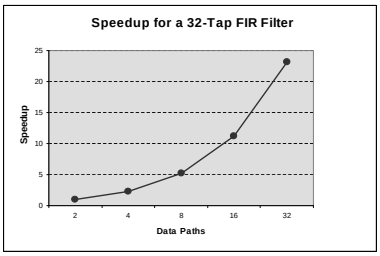

Fig. 1. Speedup Factor vs. Level of Parallelism for a 32-taps FIR Filter

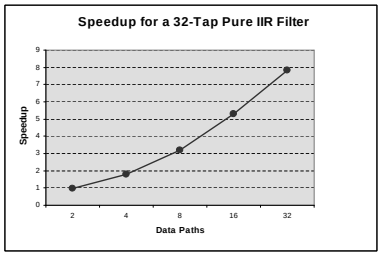

Fig. 2. Speedup Factor vs. Level of Parallelism for a 32-taps pure IIR Filter

More complex transfers that are supported by our processor like stride permutations $P(L, s)$ are also part of our algebraic repertoire.

In the algorithm of equation (6), we find the operation $\lfloor \frac{q}{N} \rfloor$. This can be easily implemented in a processor as a right shift by n positions, where $N = 2^n$ holds. Moreover, this operation determines that a new vector of input samples $\underline{x}_{i-\lfloor \frac{q}{N} \rfloor}$ is loaded from memory only when the index q becomes a multiple of N. This makes clear the advantageous property of FIR filters implemented into SIMD processors by which the number of expensive memory accesses are decreased with an increasing level of SIMD parallelism.

Matlab is a very popular language among digital signal processing designers. The matrix oriented capability of the language allows for easily programming algorithms using the elementary algebraic structures of equations (6) and (7). For example, the first part of equation (7) can be written using matlab syntax as follows:

```
mac=zeros(N,1) for i=1:p,
 mac=mac+kron(y(Nk-i),eye(N))*C_r(1:N,i)
end
```

In this example `kron()` and `eye()` are matlab functions that implement the Kronecker product and the identity matrix respectively. In this example is also important to note that the index `k` runs over the number of input vectors that have to be processed by the algorithm.

VI. Results

Using equation (6), we have implemented an FIR filter in Matlab and processor code was automatically generated for different levels of parallelism. In figure 1, we can observe the resulting speedup factors for a 32-taps FIR filter. The speedup is defined as the relation between the time that is required to compute a certain number of input samples with a processor architecture with one data path and the time required to compute the same number of input samples on the parallel processor.

Likewise, using equation (7) we have implemented a pure IIR filter in Matlab and processor code was automatically for different levels of parallelism. In figure 2, we can observe the resulting speedup factors for a 32-taps pure IIR filter.

A. Discussion

As we can observe from figures 1 and 2, the achievable speedup factor depends not only on the available processing power but it also depends on the data dependencies characteristics of the algorithm. In fact, due to the direct data dependencies of the pure IIR filter, the speedup factors we obtain for this algorithm are moderate in comparison with the speedup factors we obtain for the FIR filter. Moreover, in [5] we showed that the achievable speedup factor for pure recursive filters is upper bounded and once this upper bound has been reached no more gains on the speedup factor can be obtained even if the level of parallelism is incremented. These results illustrate the fact that in determining the optimal level of SIMD parallelism for a certain application not only power and area constraints have to be taken into account, but the characteristics of the algorithms play a paramount role. This stresses the necessity for a HW/SW design strategy, where this exploration can be supported.

VII. Conclusion

In this paper we have presented our approach for the code generation for our family of SIMD DSP architectures. We have presented the mathematical background of our compiler technique and taking an FIR and a recursive filter as examples, we have explained how our compiler generates a sequential instruction list from a set of basic algebraic structures. We believe that is approach offer an interesting way to close the gap between real-time signal processing, compiler technology and processor architecture.

References

[1] G. Cichon, P. Robelly, H. Seidel, M. Bronzel, and G. Fettweis, "Synchronous transfer architecture (STA)," in *Lecture Notes on Computer Science*, S. Vassiliadis, Ed. Berlin, Germany: Springer-Verlag, July 2004, to be published.

[2] M. Davio, "Kronecker products and shuffle algebra," *IEEE Trans. on Computers*, vol. C-30, no. 2, pp. 116–125, Feb. 1981.

[3] R. Tolimieri, M. An, and C. Lu, *Algorithms for discrete Fourier transform and convolution*. New York: Springer Verlag, 1997.

[4] M. Pueschel, B. Singer, M. Veloso, and J. Moura, "Fast automatic generation of DSP algorithms," in *Lecture Notes on Computer Science 2073*. Berlin, Germany: Springer-Verlag, 2001, pp. 97–106.

[5] J. Robelly, G.Cichon, H. Seidel, and G. Fettweis, "Implementation of recursive digital filters into vector SIMD DSP architectures," In Proc. International Conference on Acoustics, Speech and Signal Processing. ICASSP 2004. Montreal, Canada, May 2004, to be published.

[6] A. Feuer and G. C. Goodwin, *Sampling in Digital Signal Processing and Control*. Boston,Basel,Berlin: Birkhaeuser, 1996.