Design and Automatic Code Generation of the LMS Algorithm for SIMD Signal Processors.

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

***Abstract*— Taking as a starting point a collection of algebraic primitives that captures the SIMD computational model, we show in this paper our methodology for designing, mapping and implementing algorithms for SIMD-vector signal processors with scalable level of parallelism. Taking as an example the LMS, we show how an algorithm, which has been designed to exhibit a suitable level of data parallelism can be described by these algebraic primitives. In turn, these algebraic primitives are programmed in a matrix oriented language. A suitable compiler generates object code for SIMD processors with a scalable number of processing elements.**

I. Introduction

The advancement on VLSI technology has prompted renewed attention onto SIMD-vector signal processors, since it enables the application of the SIMD computational model for low-power, high-performance programmable devices. Thus, many ideas developed for supercomputers have reached the field of embedded signal processing upon the promise of delivering to programmable devices the required processing power at reasonable levels of power consumption and die size.

In [1] and [2], we have presented a novel compiler friendly microarchitecture called Synchronous Transfer Architecture STA, which offers scalable parallelism at the instruction and at the data level. In this paper we only address the problem of exploiting SIMD data level parallelism for our family of STA processor cores. Readers interested on instruction level parallelism are referred to [3].

The classical vectorization of algorithms for SIMD architectures is based on automatic loop vectorization. Hereby, algorithms are represented as sequential programs consisting of loops. Then, data dependence is established by searching on the space spanned by the loop indexes. Direct application of loop vectorization to algorithms, which are serial in nature might lead to modest speedup factors. This is especially true for the LMS, where data is processed in serial fashion due to direct data dependence in the computation of the coefficients update.

In this paper we present a methodology for the implementation of signal processing algorithms into our family of STA processor cores. Our methodology is based on three steps:

1) Algorithm Design
2) Algorithm Mapping
3) Algorithm Implementation and Code Generation

Taking as an example the LMS algorithm, we introduce a series of mathematical and software tools that enable this methodology.

II. SIMD Computational Model

In figure 1 we observe a block diagram that illustrates a high level model of a SIMD processor. The purpose of this model is to abstract many details of the architecture and it is not intended to be a detailed description of our STA processor cores.

The model mainly consists of a vector unit, a scalar unit and an interconnection network. The model can deal with three different types of data: address data type $\mathbb{Z}$, scalar data type $\mathbb{R}$ and vector data type $\mathbb{V}$. The vector data type consists of ν elements of $\mathbb{R}$. The scalar unit operates onto data of type address and data of type scalar. The vector unit deals with data of type vector exclusively. Both the scalar and the vector unit have their own register files and memory elements.

The scalar unit is intended to execute address computations and scalar computations of the algorithm. This unit is furnished with the usual operations like addition, multiplication, right and left shift, bitwise and, bitwise or, bitwise not, bitwise xor, modulo, etc. The vector unit contains enough processing elements to manipulate vectors of ν elements. This unit is furnished with the same operations of the scalar unit but it operates on vectors. Thus, this unit executes componentwise addition, multiplication, right and left shift, and, or, xor, etc. The interconnection network supports different vector data transfers and the transfer of data between the scalar and the vector unit. Among the vector data transfers we can mention vector right and left shift and stride permutations. Scalar data can be transferred to the vector unit either via broadcast transfer, or by forming a base vector on $\mathbb{V}$ weighted with a scalar value. Finally, an element of a vector residing on the interconnection network can be selected in order to be manipulated by the scalar unit.

A. Algebraic Primitives

Davio [4] in his classical paper established the connection between the *Kronecker* product of matrices and stride permutations. In his paper he proved the important *commutation theorem* of Kronecker products

$$A \otimes B = P\left(m_A m_B, m_A\right)\left(B \otimes A\right) P\left(n_A n_B, n_B\right),$$

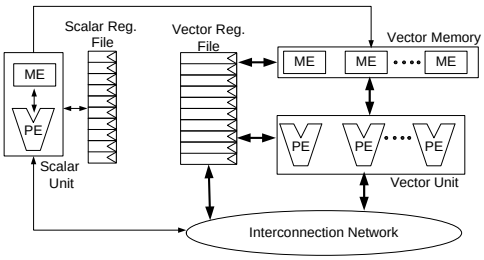

Fig. 1. SIMD organization

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:

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

Later, Tolimieri [5] makes the observation that different models of parallel computation can be described by means of Kronecker products. More recently, in [6] these ideas were extended to automate the generation of Fast Fourier Transforms for SIMD processors.

In order to characterize the SIMD computational model let A be an $m \times n$ matrix, $\underline{x}$ a $\nu n \times 1$ vector, $\underline{y}$ a $\nu m \times 1$ vector, I_ν the $\nu \times \nu$ identity matrix and consider the following important expression:

$$\underline{y} \quad = \quad (A \otimes I_\nu)\underline{x}. \tag{1}$$

Equation (1) is known as a Kronecker vector factor, since for its computation only vectors in $\mathbb{V}$ are manipulated. This is better illustrated if we partition the input and output vectors and using Matlab syntax we define $\underline{x}_q = \underline{x}\big(q\nu : (q\nu+\nu-1)\big) \in \mathbb{V}$ for $q = 0, 1, \dots, n-1$ and $\underline{y}_i = \underline{y}\big(i\nu : (i\nu+\nu-1)\big) \in \mathbb{V}$ for $i = 0, 1, 2, \dots, m-1$. Then, we can write for equation (1) the following:

$$\underline{y}_i = \sum_{q=0}^{n-1} (a_{i,q} \otimes I_\nu)\underline{x}_q. \tag{2}$$

Equations (1) and (2) capture the SIMD computation model. Further we assume for the level of SIMD parallelism

$$\nu = 2^\gamma \quad \text{for } \gamma = 1, 2, 3, \dots$$

Recalling our model of figure 1, we observe that equation (2) can be efficiently implemented if the coefficients $a_{i,q} \in \mathbb{R}$ are stored in scalar memory, then these coefficients are broadcasted. The resulting vector is componentwise multiplied by the vector $\underline{x}_q$ and accumulated. After n iterations we obtain $\underline{y}_i$.

In order to support a larger family of algorithms, we have introduced other data transfer operators. For instance, $(Z_\nu)^i$ describes a downwards vector shift by i positions and $\left(Z_\nu^T\right)^i$ describes an upwards vector shift by i positions. Other important operations are the selection of a scalar from a vector and the construction of a vector from scalars. In our framework this is supported by e_ν^i, which represents a base vector on $\mathbb{V}$. The most used algebraic primitives are summarized in table I.

SCALAR UNIT	
scalar addition	$c = a + b$
scalar subtraction	$c = a - b$
scalar multiplication	$c = a * b$
bitwise shift	$c =$ bitshift(a,b)
VECTOR UNIT	
vector addition	$\underline{v}_1 = \underline{v}_2 + \underline{v}_3$
vector substraction	$\underline{v}_1 = \underline{v}_2 - \underline{v}_3$
vector componentwise multiplication	$\underline{v}_1 = \underline{v}_2 . * \underline{v}_3$
INTERCONNECTION	
broadcast	$\underline{v} = a \otimes I_\nu$
vector shift upwards	$\underline{v}_1 = (Z_\nu)^i \underline{v}_2$
vector shift downwards	$\underline{v}_1 = \left(Z_\nu^T\right)^i \underline{v}_2$
vector construction from scalar	$\underline{v} = a.e_\nu^i$
scalar selection from vector	$a = \left(e_\nu^i\right)^T .\underline{v}$
Stride Permutation	$\underline{v}_1 = P(\nu, s)\underline{v}_2$

TABLE I
ALGEBRAIC PRIMITIVES

III. PARALLEL LMS ALGORITHM DESIGN

In this section we present a parallel formulation of the LMS algorithm as derived in [7]. The starting point is the serial representation, which is given by:

$$e(k) \quad = \quad d(k) - \underline{u}^T(k)\underline{w}(k), \tag{3}$$

$$\underline{w}(k+1) \quad = \quad \underline{w}(k) + \mu e(k)\underline{u}(k), \tag{4}$$

where μ is the adaption factor, $d(k)$ is the desired filter output, $e(k)$ is the error signal, the $p \times 1$ vector of filter coefficients $\underline{w}^T(k) = [w_1(k), w_2(k), \dots, w_p(k)]$ and the delay chain of the filter formed by the input samples is $\underline{u}^T(k) = [u(k), u(k-1), \dots, u(k-p+1)]$. In the following we consider $p/\nu = M$ and $M = 1, 2, \dots$. The LMS algorithm expressed as in equations (3)(4) processes the input data in serial fashion and therefore, it is not suitable for the implementation into SIMD-vector processors. Our purpose is to derive a formulation of the algorithm that process the input data as vectors of ν elements. Such a vector processing formulation of recursive algorithms can be derived via lookahead technique [8]. In fact, applying ν times lookahead to equations (3)(4) and after some algebraic manipulations we can derive the following equations for the LMS computation [7]:

$$\underline{e}(k) \quad = \quad \mathbf{G}(k)\Big[\underline{d}(k) - \mathbf{U}(k)\underline{w}(k-\nu+1)\Big], \tag{5}$$

$$\underline{w}(k+1) \quad = \quad \underline{w}(k-\nu+1) + \mu\mathbf{U}^T(k)\underline{e}(k), \tag{6}$$

where $\underline{e}^T(k) = [e(k-v+1), \dots, e(k-1), e(k)] \in \mathbb{V}$ is the error vector, $\underline{d}^T(k) = [d(k-v+1), \dots, d(k-1), d(k)] \in \mathbb{V}$ is the desired vector at the output of the block filter. $\mathbf{U}^T(k) = [\underline{u}(k-\nu+1), \dots, \underline{u}(k-1), \underline{u}(k)]$ is the $p \times \nu$ matrix formed by ν delay chains. The computation of the block filter is given by $\mathbf{U}(k)\underline{w}(k-\nu+1)$. $\mathbf{G}(k)$ is the $\nu \times \nu$ error correcting matrix and can be regarded as the overhead introduced by

the lookahead transformation. The overhead matrix can be computed as $\mathbf{G}(k) = \big(I_\nu + \mathbf{S}(k)\big)^{-1}$, where

$$\mathbf{S}(k) = \begin{pmatrix} 0 & 0 & \dots & 0 \\ s_1(k-\nu+2) & 0 & \dots & 0 \\ s_2(k-\nu+3) & s_1(k-\nu+3) & & \vdots \\ \vdots & \vdots & & \vdots \\ \vdots & \vdots & & \vdots \\ s_{\nu-1}(k) & s_{\nu-2}(k) & \dots\ s_1(k) & 0 \end{pmatrix} \tag{7}$$

The entries of this matrix are $s_i(k) = \mu \underline{u}^T(k)\underline{u}(k-i)$. Thus, equations (5)(6) describe a block formulation of the LMS algorithm that process input data in a vector fashion.

A. Interim Discussion

The overhead matrix $\mathbf{G}(k)$ is a lower triangular matrix with ones on its main diagonal. This matrix has to be computed every time a new input vector is processed. The matrix entries of the lower subdiagonals of this matrix are computed using the entries $s_i(k)$ of $\mathbf{S}(k)$. Hence, the number of scalar operations increases with the level of parallelism ν. Moreover, the number of scalar operations for computing the entries $s_i(k)$ increases with the number of filter coefficients p. Thus, for large values of ν and p the algorithm spends a lot of time computing scalar operations in order to calculate the overhead matrix. However, the number of scalar operations can be dramatically decreased by means of two approaches.

On the one hand, it is important to note that the elements of the first column of $\mathbf{S}(k)$ in terms of the elements of the last row that were computed during the previous block can be recursively computed as follows:

$$\begin{aligned} & s_i(k-\nu+i+1) \\ & \quad = s_i(k-\nu) + \mu\Bigg[\sum_{j=0}^{i} u(k-\nu+i-j+1) \\ & \qquad \cdot u(k-\nu-j+1) \\ & \qquad - \sum_{j=0}^{i} u(k-p-\nu+i-j+1) \\ & \qquad \cdot u(k-p-\nu-j+1)\Bigg], \end{aligned} \tag{8}$$

for $i = 1, \dots, \nu-1$. For the entries of the subdiagonals we can write:

$$\begin{aligned} s_i(k+1) = & s_i(k) + \mu\big[u(k+1)u(k-i+1) \\ & - u(k-p+1)u(k-i-p+1)\big] \end{aligned} \tag{9}$$

Thus, the number of scalar operations for computing the entries of $\mathbf{S}(k)$ becomes independent of the number of filter coefficients p.

On the other hand, the number of scalar operations can be alleviated if the error correcting matrix is relaxed. This influences the performance of the algorithm according to the autocorrelation characteristics of the input signal. In figure 2, we can compare the error curves for different approximations of the overhead matrix with the error curve of the original algorithm. For the formulation of the algorithm with $\nu = 4$ we have used only three diagonals of the overhead matrix instead of four. For the case $\nu = 16$, we considered only four diagonals. For the error curves shown we have passed linearly independent symbols through an ISI channel. As we can observe from the figure, the number of scalar operations can be dramatically reduced at expenses of marginal changes on the performance of the algorithm.

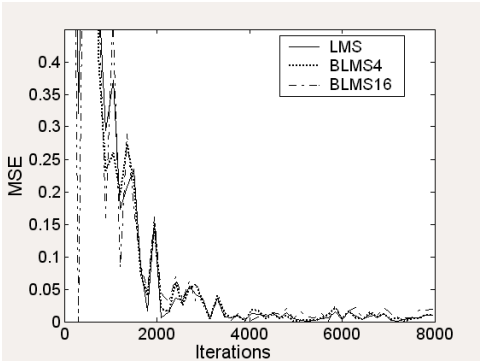

Fig. 2. Error curves for an adaptive filter with 128 coefficients and adaption factor $\mu = 0.01$ for the LMS, BLMS4 and BLMS16.

IV. Mapping the Parallel LMS into Algebraic Primitives

In this section we express the parallel computation of the LMS algorithm in terms of the algebraic primitives presented in section II. The algorithm as presented in equations (5)(6) has three stages: block filtering and error computation, computation of the overhead matrix and error correction, and coefficients update.

In the following we assume that the length of the training sequence satisfies $L/\nu = J$ and $J = 1, 2, 3, \dots$. Thus, the input samples to be processed by the algorithm are collected in a vector $\underline{u} = [u(k), u(k+1), \dots, u(k+L-1)]^T$. Likewise, we can define the vector of filter output samples $\underline{y}$ and the vector collecting the training sequence $\underline{d}$. In order to process the input data in vector fashion, we partition $\underline{u}$ and construct $\underline{u}_j = [u(k+\nu j), u(k+\nu j+1), \dots, u(k+\nu j+\nu-1)]^T \in \mathbb{V}$ for $0 \le j < J$, otherwise this is a vector with ν zero elements. Likewise, we can partition $\underline{y}$ and $\underline{d}$ and we form $\underline{y}_j$ and $\underline{d}_j$. The block filter can be formulated by means of the algebraic primitives as follows:

$$\begin{aligned} \underline{y}_j = \sum_{q=0}^{p-1} (w_q \otimes I_\nu) \cdot \Big[& (Z_\nu)^{(q) \bmod \nu} \underline{u}_{j-\lfloor \frac{q}{\nu} \rfloor} + \\ & (Z_\nu^T)^{(\nu-q) \bmod \nu} \underline{u}_{j-1-\lfloor \frac{q}{\nu} \rfloor} \Big], \end{aligned} \tag{10}$$

where mod is the modulo operator and $\lfloor \cdot \rfloor$ is the floor operator. The error vector can be computed as $\underline{e} = \underline{y}_j - \underline{d}_j \in \mathbb{V}$.

For calculating the error correction matrix we use equations (8)(9) for computing the scalar values of the matrix

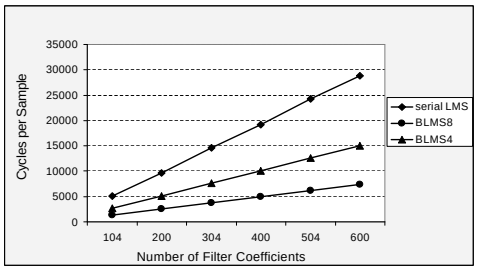

Fig. 3. Number of cycles per sample for the LMS algorithm computed serially, computed in a SIMD processor with $\nu = 4$ and computed in a SIMD processor with $\nu = 8$.

entries. We can use these scalar values to form the vectors that build the $\nu \times \nu'$ relaxed correction matrix $\mathbf{G}'$, where ν' is the number of diagonals taken from the original matrix. Moreover, the chosen diagonals of $\mathbf{G}(k)$ are stored in $\mathbf{G}'$ as columns completed with zeros if necessary. Thus, for the computation of the error correction and using Matlab syntax we can write:

$$\underline{e}' = \sum_{q=0}^{\nu'-1} \mathbf{G}'(:,q). * \underline{e}, \tag{11}$$

for $\underline{e}' \in \mathbb{V}$. This completes the computation of a relaxed version of equation (5). For the computation of equation (6) we need the following result:

$$\begin{aligned} \underline{e}'' = \Big[&(Z_\nu)^{(q) \bmod \nu} \underline{u}_{j-\lfloor \frac{q}{\nu} \rfloor} + \\ &(Z_\nu^T)^{(\nu-q) \bmod \nu} \underline{u}_{j-1-\lfloor \frac{q}{\nu} \rfloor} \Big] \cdot \underline{e}', \end{aligned} \tag{12}$$

for $0 \leq q < p-1$. Adding the ν elements of the resulting vector we obtain:

$$\underline{x}(q) = \sum_{i=0}^{\nu-1} \left(e_\nu^i\right)^T \underline{e}''. \tag{13}$$

It is important to note that $\underline{x}$ is a vector of p elements. However, loading ν elements of this vector $\underline{x}\big(m\nu : (m+1)(\nu-1)\big)$, for $0 \leq m < M$ results in vectors in $\mathbb{V}$ and we can write for the coefficients update the following equation

$$\begin{aligned} \underline{w}\big(m\nu : (m+1)(\nu-1)\big) = &\underline{w}\big(m\nu : (m+1)(\nu-1)\big) \\ &+ (\mu \otimes I_\nu) \\ &\underline{x}\big(m\nu : (m+1)(\nu-1)\big) \end{aligned} \tag{14}$$

This completes the computation of the algorithm using the algebraic primitives that describes the SIMD computational model of section II.

V. Code Generation and Results

Algorithms expressed as in equations (10)-(14) can be easily programmed using a matrix oriented language like Matlab or Octave. In [2][3] we presented a compiler infrastructure, which process programs written in Matlab or Octave. One stage of the compiler features pattern matching and uses the algebraic primitives in order to establish the rules for the generation of a medium intermediate representation that contains information about data level parallelism. Further stages of this compiler exploits instruction level parallelism and deals with architecture dependent issues like for example register allocation.

The LMS algorithm was programmed in Octave once and code was generated for STA cores with different levels of parallelism ν. The code was run in cycle accurate processor models at the register transfer level. If figure 3 we can observe the number of cycles per sample for the case $\nu = 4$, $\nu' = 3$ (BLMS4) and $\nu = 8$, $\nu' = 4$ (BLMS8). For comparison purposes a serial implementation of the algorithm, which was entirely computed on the scalar unit of the processor is presented. The code for this implementation was also generated using our compiler, yet the Octave code was written using the original serial mathematical formulation of the algorithm. As we can observe, important speed up factors are achieved.

VI. Conclusion and Future Work

Many state of the art code optimization techniques are yet to be implemented in our compiler infrastructure. We expect that these techniques will improve the code quality and we will reduce the number of cycles per sample in at least one order of magnitude. In future publications we will report about these efforts.

The model presented in section A provides an abstraction system that describes the functionality of a SIMD-vector processor. In order to find a mapping between an algorithm and the architecture, we have introduced primitive algebraic constructions, which enables a mathematical formulation taking into account architectural features of the processor. The algebraic constructions are primitive in the sense that they are general enough to enable the formulation of a large family of algorithms. Moreover, the algebraic primitives determine transformation rules for the synthesis of code that exploit the features they describe. In this paper we have presented this approach taking as example the LMS algorithm.

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] P. Robelly, G. Cichon, H. Seidel, M. Bronzel, and G. Fettweis, "A hw/sw design methodology for embedded simd vector signal processors," *International Journal of Embedded Systems IJES*, January 2005, to be published.

[3] G. Cichon, P. Robelly, H. Seidel, M. Bronzel, and G. Fettweis, "Compiler scheduling for STA processors," In Proc. International Conference on Parallel Computing in Electrical Engineering PARELEC 2004. Dresden, Germany, pp. 45–50, Sept. 2004.

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

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

[6] F. Franchetti and M. Pueschel, "A simd vectorizing compiler for digital signal processing algorithms," In Proc. International Parallel and Distributed Processing Symposium (IPDPS), pp. 20–26, 2002.

[7] J. Benesty and P. Duhamel, "A fast exact least means square adaptive algorithm," *IEEE trans. on SP.*, vol. 40, no. 12, pp. 2904–2990, Dec. 1992.

[8] G. Fettweis and L. Thiele, "Algebraic recurrence transformations for massive parallelism," in *Proceedings of the IEEE Workshop on VLSI Signal Processing*, Napa, California, October 1992, pp. 332–341.