HAL open science

Polyhedra at Work: Automatic Generation of VHDL Code for the Sherman-Morrison Formula

Michel Lemaire, Daniel Massicotte, Jeremy Poupart, Patrice Quinton, Sanjay Rajopadhye

▶ **To cite this version:**

Michel Lemaire, Daniel Massicotte, Jeremy Poupart, Patrice Quinton, Sanjay Rajopadhye. Polyhedra at Work: Automatic Generation of VHDL Code for the Sherman-Morrison Formula. Impact 2024 - 14th International Workshop on Polyhedral Compilation Techniques, ACM, Jan 2024, Munich, Germany. pp.1-10. hal-04401934

HAL Id: hal-04401934

https://inria.hal.science/hal-04401934v1

Submitted on 18 Jan 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Polyhedra at Work: Automatic Generation of VHDL Code for the Sherman-Morrison Formula

Michel Lemaire
Université du Québec à Trois-Rivières
Trois-Rivières, Canada
michel.lemaire1@uqtr.ca

Daniel Massicotte
Université du Québec à Trois-Rivières
Trois-Rivières, Canada
daniel.massicote@uqtr.ca

Jeremy Poupart
Université du Québec à Trois-Rivières
Trois-Rivières, Canada
jeremy.poupart@uqtr.ca

Patrice Quinton
École Normale Supérieure de Rennes
Rennes, France
patrice.quinton@ens-rennes.fr

Sanjay Rajopadhye
Colorado State University
Fort-Collins, USA
Sanjay.Rajopadhye@colostate.edu

Abstract

Simulation of electrical circuits in real-time requires very short latency implementations that can be achieved only on special-purpose hardware accelerators. The heart of such algorithms is a matrix-vector multiplication between the inverse of the admittance matrix of the circuit and of its current state vector, yielding the next state vector. The main difficulty is to obtain the inverse of the admittance matrix in real-time, as this matrix depends on the state–open or closed–of the switches of the circuit. We consider here the problem of obtaining a new inverse matrix, using the Sherman-Morrison update algorithm, and we explain how various VHDL implementations of this algorithm can be obtained from a high-level, polyhedral equational, description of the circuit.

Keywords: Simulation of Electrical Circuits, Polyhedral Model, Sherman-Morrison Formula, VHDL, FPGA

1 Introduction

The simulation of electrical circuits in real-time is used in many applications [3, 14]. Real-time simulations are often needed, for example, when some parts of the system under design are replaced by a simulator.

In most methods, electrical simulation amounts to solving a linear system $Ax = b$ where A is the *admittance matrix* of the circuit, b is the values of currents and voltages of the circuit at a given instant of time, and x is (the vector of) the current and voltages after a small time step. The values of the A matrix represent the admittance (reciprocal of the resistances, basically) of the edges of the circuit, represented as a graph. One step of simulation is to compute $x = A^{-1}b$, which requires to compute the inverse of A.

However, electrical circuits contain switches, the admittance of which may be modified from one step to the next one, if the status (open vs. closed) of the switch changes. Thus, matrix A^{-1} may change from one step to the next.

For real-time simulations, current electrical simulators are implemented on FPGA platforms which provide the low latency needed as well as resources to execute in parallel some calculations. In practice, when the size of the circuit is small enough, inverse matrices A^{-1} are pre-computed and stored on the memory of the FPGA, but this is of course limited by the size of the circuit, and by the number of switches that it contains.

An alternative is to update the inverse of the A matrix when the state of some switches changes. But, without optimization, the $O(N^3)$ complexity of matrix inversion prevents such a method to be done in real-time, on the current FPGA platforms.

Implementing such an algorithm on a FPGA is not an easy task, even with modern *high-level synthesis* tools that are more an more often used. This task requires the exploration of several solutions, to make sure that under the limited resources of the FPGA platform, the performance of the design meets the requirements of the application. Besides, the numerical stability of the chosen algorithm has to be ensured, since simulations may last several hours or even days.

In this paper, we explore the automatic generation of synthesizable VHDL code for the $O(N^2)$ Sherman-Morrison algorithm [25] for *incremental* matrix inversion, in the context of electrical circuit simulation, using an approach based on the Polyhedral Equational Model. In this model, calculations are expressed as recurrence equations that can be seen as single-assignment statements describing the problem to solve. We use tools based on the ALPHA language [10, 22], one of the long-standing vehicles for this approach. Our contribution is to present a fully automatic design flow that starts from an ALPHA description and produces in a few seconds, a VHDL program which can be synthesized for a FPGA.

This paper is organized as follows. In Section 2, we present the context of this research, the simulation of electrical circuits. Section 3 describes the Sherman-Morrison formula. The Polyhedral Equational Model is presented in Section 4,

IMPACT 2024, January 17, 2024, Munich, Germany
2024.

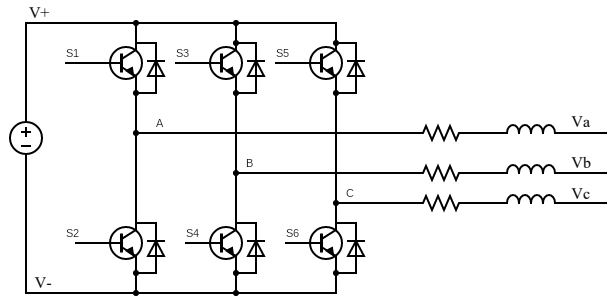

Figure 1. Electrical circuit of a power converter, containing a power source and 6 switches noted S_1 to S_6

together with the ALPHA language and associated tools. Section 5 describes a first implementation of the Sherman-Morri-son algorithm using ALPHA. The translation to VHDL is explained in Section 6. In Section 7, we present an optimized version of the Sherman-Morrison formula, in the context of the electrical simulation. Section 8 is devoted to results. In Section 10, we discuss some aspects of this work, and Section 11 concludes and presents some future research directions.

2 Simulation of Electrical Circuits

Fig. 1 shows the schematics of the electrical circuit of a power converter.

Simulating the behaviour of such a circuit amounts to solve the linear system $Ax = b$ to obtain, from the current voltage and current vector b, the new values x of these quantities.

When the electrical circuit contains switches–as is the case in this power-converter,–the A matrix may change between two simulation steps, since switches are modeled by a resistance, very high when the circuit is open, and very low otherwise. The matrix A, being dependant on the switches states, requires an inversion at every switch state change.

The real-time simulation of such a circuit can be done on a fast accelerator, for example, a FPGA platform, but to reach the low latency required by real-time, one has either to store all possible values of the A^{-1}, which can be done if the circuit is small enough, or to re-compute A^{-1} as needed.

In general, the complexity of solving a linear system of size N is $O(N^3)$. However, in the particular case of a circuit simulation, only a few elements of the A matrix change.

In the example shown by Fig. 2, the g_1 value corresponds to the admittance of switch S_1 of Fig. 1, and this value only appears in elements a_{11}, a_{31}, a_{13} and a_{33} of matrix A.

Therefore, it is interesting to consider methods to update the A matrix with a lower complexity, among which, the Sherman-Morrison approach.

3 The Sherman-Morrison Formula

Given a square, invertible matrix A, the Sherman-Morrison algorithm allows one to efficiently compute the inverse of an order-one perturbation of A, according to the formula:

$$(A + uv^T)^{-1} = A^{-1} - \sigma A^{-1} u v^T A^{-1} \quad , \tag{1}$$

where

$$\sigma = \frac{1}{1 + v^T A^{-1} u} \quad . \tag{2}$$

In this formula, the outer-product uv^T represents an order-one perturbation of the matrix A.

Computing this formula requires the calculation of two vector-products–$A^{-1}u$ and $v^T A^{-1}$–that require each $O(N^2)$ operations, an outer product, and a subtraction of matrices–each one also $O(N^2)$ operations–plus a dot product and a division.

This is better than a full-matrix inversion which needs $O(N^3)$ operations.

Note that higher-order perturbations of A can be dealt with by successive applications of the Sherman-Morrison method.

Several problems have to be faced, when implementing this algorithm in the context of electrical simulation, in practice:

- The latency of the algorithm, i.e., the number of steps between the moment that the changes of A are known, and the moment when the update of A^{-1} is computed, must be very small, which requires a highly parallel implementation and the use of a special-purpose platform such as a FPGA.
- The accuracy of the implementation must be such that the simulation results remain correct, even after long simulations.
- Many optimisations might have to be tried, necessitating rapid design-space exploration.

4 The Polyhedral Equational Model

The Polyhedral Equational Model is a formalism based on recurrence equations which has been developed over years to represent regular calculations and transform these representations in order to derive executable code for parallel architectures, or even, to produce hardware descriptions [18, 23]. The ALPHA language [10, 12, 22] that we shall consider in this paper is a functional language invented to express polyhedral equations. In this section, we first present briefly the ALPHA language (4.1), then its transformations (4.2), and the tools which are available (4.3).

4.1 The ALPHA language

Fig. 3 shows the ALPHA description of a matrix-vector algorithm that will be used in the following.

$$A = \begin{bmatrix}
g1+g3+g5 & 0 & -g1 & -g3 & -g5 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & g2+g4+g6 & -g2 & -g4 & -g6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
-g1 & -g2 & g1+g2+gR1 & 0 & 0 & -gR1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-g3 & -g4 & 0 & g3+g4+gR2 & 0 & 0 & -gR2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-g5 & -g6 & 0 & 0 & g5+g6+gR3 & 0 & 0 & -gR3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & -gR1 & 0 & 0 & gL1+gR1 & 0 & 0 & -gL1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & -gR2 & 0 & 0 & gL2+gR2 & 0 & 0 & -gL2 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -gR3 & 0 & 0 & gL3+gR3 & 0 & 0 & -gL3 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -gL1 & 0 & 0 & gL1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & -gL2 & 0 & 0 & gL2 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -gL3 & 0 & 0 & gL3 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}$$

Figure 2. The admittance matrix for the circuit of Fig. 1. The g_1 to g_6 values correspond to the admittance of the 6 switches S_1 to S_6. Only these values are subject to a modification, when the status–open or closed–of the switches changes.

```
1   system matVect: {N | 2<=N}
2     (a : {i,j | 1<=i<=N; 1<=j<=N} of integer;
3      v : {i | 1<=i<=N} of integer)
4   returns (c : {i | 1<=i<=N} of integer);
5   var
6     X : {i,j | 1<=i<=N; 0<=j<=N} of integer;
7   let
8     X[i,j] =
9         case
10          { | j=0} : 0;
11          { | 1<=j} : X[i,j-1] + a[i,j] * v[j];
12        esac;
13    c[i] = X[i,N];
14  tel;
```

Figure 3. Matrix-vector algorithm in ALPHA

After the key-word `system`, the name of the program is given, followed by the definition of a size parameter N with its domain.

The inputs of the program (lines 2 and 3) are a matrix a and a vector v (for the sake of simplicity, of integers). The output (line 4) is a vector c. The domains of these variables are given as polyhedra, in the obvious way.

The equations that define the calculations make use of an intermediate–local–variable X that is defined using a simple induction (line 8 to 12), and the definition of the output c (line 13) as the last element of this recurrence.

This program is shown in its so-called *array form*, that allows classical numerical calculations to be expressed in a conventional way. However, the definition of ALPHA is functional, and the basic constructs of the language allow formal manipulations of code suitable for parallel expression to be performed [12].

The ALPHA language contains the notion of *subsystems* that allows a system to be reused in a hierarchy of definitions, while maintaining the properties of the core language. For example, the matrix-vector ALPHA model shown in Fig. 3 can be included inside another system by:

```
use matVect[N] (A, x) returns (Y)
```

provided that the inputs A, x and the output Y are declared with the same domains and scalar types as the formal parameters A, v and c in the `matVect` model. The `matvect[N]` part of the statement indicates that the model is used with the same parameter N. In general [5], the form of such a statement includes a mapping domain, where the model can be instanciated, and the parameter of the instanciated system can be any affine expression of the size parameters of the calling system.

4.2 Main Polyhedral Transformations

The following semantics-preserving transformations of code will be extensively used during the synthesis:

- **Substitution** aka., **inlining**: a variable can be replaced by its definition, anywhere, by simple cut and paste. Conversely, by **anti-substitution**, one can replace any expression by a new variable, whose definition domain can be computed easily from the constituents of the expression.
- **Normalization**: any expression can be rewritten in the simple *case-restriction-dependence* form, much as shown in the definition of X in the program of Fig. 3.
- **Change-of-basis**: the domain of definition of a local variable can be replaced by its image under a linear, unimodular mapping, by a simple, syntactic rewriting. This allows *space-time mapping* to be done: provided the mapping $(z \longrightarrow t, p)$ that transforms the indexes z of some variable V into a pair of time and processor coordinates, the new program becomes executable, by evaluating a calculation $V(z)$ on processing unit p at time t.
- **Pipelining**: by identification of values that are common to several expressions, reuse of these common

values can be performed through the rephrasing of some equations using linear inductions.

- **Simplification of reductions**: the ALPHA language allows the direct expression of reductions such as Σ, Π, etc. For example, we could have written the definition of c in the matrix-vector program as

```
c[i] = reduce( +, j, a[i,j] * v[k])
```

that reads obviously $c[i] = \sum_j a[i,j] \times v[j]$.

- **Inlining and exlining**: the use of a model can be replaced by its core definition, provided some simple composition rules are applied to the equations. This inlining process is most often used prior to the transformation of a design. However, a fully structured [20] design path allows a design to be done in a hierarchical way, but this will not be used in this paper examples. Conversely, *exlining* allows some calculations to be separated from a given system, and this operation is actually used in the MMALPHA design trajectory to separate the control and datapath parts of the program, since they are translated in a different way in VHDL.

4.3 Equational Polyhedral Tools

There exists currently two sets of tools allowing ALPHA programs to be transformed. Both are based on the same abstract syntax, although, for historical reasons, their concrete syntax slightly differs.

MMALPHA [4] was developed at Irisa over years, and it targets, currently, the generation of fine-grain architectures. It includes the following transformations:

1. Inlining of sub-systems.
2. Scheduling (using affine by variable schedules).
3. Apply a schedule to operate a time-space mapping of the program.
4. Find pipelines.
5. Rewrite a scheduled and allocated program into a RTL-like code suitable for hardware generation.
6. Generation of naive, single-assignment C code, for simulation and verification purpose.
7. Generation of synthesizable VHDL code.
8. Generation of stimuli files for the hardware description and simulation.

The ALPHAZ tool [28] is merely a transformation exploration system, where the user can provide a script that allows code to be generated for a larger class of parallel architectures. An important difference with MMALPHA is that ALPHAZ allows:

1. Tiling, i.e., organizing the calculations in blocks of the same size called *tiles*, where the executions follows a sequential order. This is essential to obtain good tradeoffs between time and space, in order to optimize the resource utilization.
2. Optimized code generation, including memory mapping of variables [2, 17].
3. Simplification of the reductions. In high dimensional designs, it may happen that sharing the calculation of sub-expressions allows the complexity of the whole program to be reduced by several orders of magnitude [8].

It is possible to combine both MMALPHA and ALPHAZ in order to organize the design path of an application, since at any time, the design can be expressed as an abstract ALPHA program combined with some annotations expressing the current state of transformations.

5 Expression and Synthesis of the Simulation Algorithm

In this section, we present the ALPHA code for an implementation of the Sherman-Morrison formula (5.1), then we describe the synthesis flow of this program (5.2).

5.1 ALPHA Code

A naive description of the Sherman-Morrison algorithm using the ALPHA language is shown in Fig. 4. This encoding maps almost directly the equations of the definition into the language.

Lines 1 to 4 are include statements that allow some other, basic functions to be used. The `outProd` function computes the outer products of two vectors x and y of size N into the square matrix $(x_i y_j)_{1 \le i,j \le N}$.

The inputs are the matrix B that contains the current version of A^{-1}, and the vectors u and v. The output newB is the updated matrix.

Remark: In this version, as well as throughout this paper, the σ coefficient is set to 1, in order to avoid a division. Indeed, the use of a division can be avoided by a precalculation of all its possible values, and then, memorized in a look-up table. This part of the design is not considered here, in order to simplify the presentation.

5.2 The Synthesis Process

In this section, we illustrate the use of MMALPHA to generate VHDL code, for various situations resulting from several optimization choices.

The goal of the synthesis is to derive a VHDL block that can be eventually included in a larger schematics, targeted to the Mathworks System Generator tool [27].

5.2.1 Principles of the Synthesis.

The synthesis process is *fully automatic*. It consists of a series of well-identified steps which require no a priori information others than the ALPHA program.

Until the last, VHDL generation step, the synthesis results in a transformation of the initial ALPHA code that is semantically equivalent to the initial description.

```
include outProd.alpha
include matVect.alpha
include dot.alpha
include transpose.alpha
system shermanMorrison:{N | 2<=N}
(
  B : {i,j | 1<=i<=N; 1<=j<=N} of integer;
  u : {i | 1<=i<=N} of integer;
  v : {i | 1<=i<=N} of integer
)
returns
(
  newB : {i,j | 1<=i<=N; 1<=j<=N} of integer
);
var
  Btrans: {i,j | 1<=i<=N; 1<=j<=N}
    of integer;
  oprv, incrA : {i,j | 1<=i<=N; 1<=j<=N}
      of integer;
  sigma: integer;
  r: {i | 1<=i<=N} of integer;
  l: {i | 1<=i<=N} of integer;
  d: integer;
let
  -- B represents A-1
  -- Transpose of B
  use transpose[N] ( B ) returns (Btrans);
  -- r = B.u
  use matVect[N] ( B, u ) returns (r);
  -- l = B^T. v
  use matVect[N] ( Btrans, v ) returns (l);
  -- d = l.u
  use dot[N] (l,u) returns (d);
  -- Sigma = 1/(1+d), but forced to 1
  sigma[] = 1[];
  -- oprv = r outer l
  use outProd[N] (r,l) returns (oprv);
  -- Increment of A
  incrA[i,j] = sigma[]*oprv[i,j];
  -- New result
  newB = B - incrA;
tel;
```

Figure 4. Alpha code of the Sherman-Morrison algorithm

5.2.2 Synthesis Steps.

The synthesis proceeds as follows.

1. The initial program is parsed, and calls to subsystems are *inlined*. Another possibility is to do the synthesis in a structured way, without inlining subsystems, but this facility is not used in this example.
2. The parsing includes a thorough type-checking of the ALPHA code, in particular, the verification that domains of expressions are consistent, and that calls to subsystems are coherent.
3. The ALPHA program is then scheduled, using the so-called *vertex method* [13, 23, 24]. Each variable X is scheduled by means of an affine-by-variable integer schedule of the form $t_X(z) = \alpha_X(z) + \beta_X$. Causality is enforced by building an Integer Linear Program. Schedule may be multi-dimensional, but in our examples, this possibility is not used.
4. The ALPHA program is then transformed by applying to each variable a time-space change of basis. Each schedule is completed automatically by an affine allocation function in order to obtain a unimodular transformation (most often, this amounts to complete the schedule with a projection on $n-1$ indexes of the variable). At the end of this step, all variables of the program are indexed by t and p (where p represents a list of *processor* coordinates).
5. The time-spaced ALPHA program is massaged into a multi-dimensional RTL description, by means of simple transformations. After this step, equations are separated into:
   - *Combinational equations* of the form
     `X[t,p] = f(Y[t,p], ...)`
     where f is a function involving only combinational operators.
   - *Simple connections*, of the form `X[t,p] = Y[t,p]`.
   - *Register equations*, `X[t,p] = Y[f(t,p),g(t,p)]`.
   - *Multiplexers*, expressed using `if` and `case` expressions.
   - *Control equations*, involving boolean values.

   In addition, the program is separated into three parts: a *wrapper*, having the same signature that the initial ALPHA program, a *hardware module* comprising the data-path equations, and a *controller*, containing the equation that define boolean control signals. This representation is again, semantically equivalent to the initial ALPHA program.
6. All transformations are done independently on the size parameters. Before the generation of the VHDL code, the parameters are set to the value required for the design.
7. Finally, the VHDL code is generated. We detail this last step in section 6.

5.2.3 Simulations.

The synthesis process contains several ways to simulate a design under synthesis, at various levels. These simulations are intended to help verifying the initial ALPHA program, to generate stimuli for the VHDL code, and also, to test the MMALPHA tool.

A first simulation is based on a strict application of the denotational semantics of ALPHA, as defined by Mauras [12] and De Dinechin [5]. It allows an early verification of the

design, as well as generation of random inputs for creating automatically stimuli files.

Another simulation is done by a translation of the ALPHA program into a C program using Quillere's translator [17].

Finally, another simulation is based on repeated substitutions and normalization of equations. It is very slow, but can be applied to the final output of the synthesis, and is very helpful to verify the MMALPHA tool.

6 Translation to VHDL

The translation to VHDL targets a simple synthesizable form. At the end of the synthesis, the ALPHA program is separated into a control subsystem, a data-path subsystem, and a wrapper subsystem. The control subsystem is translated into a VHDL finite state machine. The wrapper subsystem is used to describe the detailed input/output of the system, and to generate a simulation script. In this section, we concentrate on the generation of the VHDL code for the data-path part. In 6.1, we explain how ALPHA variables are translated into VHDL arrays, and in 6.2, we describe the translation of the data-path equations. It should be noticed that the translation process that we describe here does not consider the input/output and the memorization of data. We report the discussion of this important limitation to Section 11.

6.1 Array Types of Variables

Variables are mapped to VHDL arrays. Any variable of the hardware module has a domain that is time-space mapped, of the form $\{t, p \mid c\}$ where c is a set of linear constraints. The bounding box of such a domain can be easily found, and it is used to define the VHDL array type of this variable.

This process is sufficient for most of the variables, provided their *life-time* is bounded.

The notion of life-time was introduced in Feautrier and Lefebvre [11], and Quilleré and Rajopahdye [17], to cope with the memory-mapping of variables before the generation of parallel code.

The life-time of a variable instance, say $X[t, p]$ is the longest amount of time that this instance is required to be memorized to be consumed by the calculation of any other instance $Y[t', p']$.

Finding out the life-time of variables amounts to compute the maximum value of $t' - t$ over the set of all these points, which can be done thanks to parameter integer programming, for example, using the Pip software [7]. This maximum, in general, depends on p, and on the size parameters (here, N). However, as we do the generation of VHDL code after setting the value of the parameters, and since p is bounded, the life-time is a constant[1].

Several situations may occur.

- If the life-time is 0, the instance $X[t, p]$ is always used at the same instant of time when it is produced, and no memorization is required.
- If the life-time is 1, we implement this as a simple register.
- If the life-time is strictly higher than 1, we implement the memorisation of X by adding an additional variable, implementing a registered version of X, with an extra dimension to the declaration array of X, the size of which is defined by the life-time value.

6.2 Translation of the equations

The equation defining a variable is unique, and has the form `X[t,p] = ...`, where p is a set of indexes representing the space dimensions of X. This equation is translated into a nest of VHDL `For generate` statements, whose bounds are computed from the domain of p thanks to a domain to loop computation algorithm [1, 17].

The body of this nest depends on the type of equation. For a combinatorial equation, the translation is almost direct, by mapping the ALPHA operators to those of VHDL.

For a multiplexer, the translation is also very simple, adding a `When` condition to represent the condition.

The translation of a simple connection is also very simple.

More difficult is the translation of so-called register equations, which raise actually both memorization and communication issues.

Consider such an equation

```
X[t,p] = Y[f(t,p),g(t,p)]
```

where $f(t, p)$ is the expression of the time in Y and $g(t, p)$ is the expression of the space. Here, t and p are vectors of indexes, and f and g are multi-dimensional affine functions.

When f is the identity, such an equation represents a communication between the processors of X and those of Y. In the current implementation, we translate directly the g function in VHDL code, letting the VHDL hardware synthesis tools cope with the communication implementation. In some *extreme* situations, this may result in bad hardware implementations, and further optimizations based on the analysis of polyhedral equations is required (see for example Van Dongen and Quinton [21]).

When f is not the identity, we also separate the situations where $f(t, p)$ depends or not on p. In the most general case, we compute the maximum value of the difference $t - f(t, p)$, say m and we implement the access to the proper value $Y[f(t, p), g(t, p)]$ through the use of a shift-register that stores the m most recent values of Y.

Although this implementation may not be optimal, it is guaranteed to produce correct code.

[1]When the time is multidimensional, which is not considered here, the notion of life-time is still applicable, with possibly several dimensions.

7 Optimized Code for the Electrical Simulation

Although the description of the Sherman-Morrison formula of Fig. 3 could be used to solve the updating of the admittance matrix, some simplifications can be made to this program by taking into account the structure of this problem.

The A matrix elements that are changed, when the status of a switch changes, are the 4 elements a_{ii}, a_{ij}, a_{ji} and a_{jj}, and the corresponding modification is given by the matrix

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & -d_i & 0 & d_j & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & d_j & 0 & -d_i & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad , \tag{3}$$

which can be represented by the product uv^T where $u = (0, d_i, 0, -d_j, 0)$ and $v = (0, -1, 0, 1, 0)$.

The modification of the A^{-1} matrix, given by

$$A^{-1} \leftarrow A^{-1} + \sigma A^{-1} u v^T A^{-1} \quad , \tag{4}$$

can be done in the following way:

1. Select the columns c_i and c_j of A^{-1}, corresponding to the numbers i and j of the non-zero elements of u and v.
2. The product $l = A^{-1}u$ is then equal to $d_i \times c_i - d_j \times c_j$.
3. The product $r = v^T A^{-1}$ is equal to $c_i + c_j$.
4. $A^{-1}uv^TA^{-1}$ is then $\sigma(A^{-1} - lr^T)$.

8 Results

We have generated the VHDL code for two versions of the Sherman-Morrison formula, first, the full version of Fig. 3, then the optimized version of Section 7[2].

The produced VHDL code was simulated and then transformed into a schematics using the VIVADO System Generator software [27]. Table 1 displays the results of these experiments.

In this table, the number of processors represents the maximum space size of the ALPHA equations, at the end of the design, before the VHDL translation. However, the actual VHDL design is not organized as a collection of identical processing units, as each equation produces an independent `generate` statement. As a consequence, the number of DSPs of the design depends on resource usage optimization done by the FPGA tools.

[2]The VHDL code of the optimized version was "patched" to implement the direct access to the columns of the A^{-1} matrix. In ALPHA, addressing an element of a variable through an index–say, writing `X[A]` where `A` is the result of an equation–is not direct, whereas the translation of such a statement in VHDL is obvious. The modification of the final code by hand concerned a few declarations and instructions. Obtaining the same code automatically is not, in theory, a difficulty, but it requires the MMALPHA tool to be extended and checked.

The full Sherman-Morrison algorithm[3] produces a space-linear design, with linear latency. The complexity is dominated by the matrix-vector product, which requires N processing units.

The optimized algorithm produces a design with N^2 processing units required to update the A^{-1} matrix, which is kept in registers. The latency is constant, independent on the size of the design.

As expected, the synthesis time–i.e., the time needed to produce the VHDL code–is independent on the size parameter N.

The VHDL code, generated directly from the final ALPHA equations, is very readable, as each statement is a direct translation of an ALPHA equation, as shown by the example of Appendix B.

9 Other Works

Implementations of the Sherman-Morrison formula to accelerate Electrical simulation on FPGA has been considered by several authors (see [9] for example). The systematic exploration and optimization of FPGA architectures to implement this method is more recent, and either is based on handwritten VHDL code as in [9], or on `C` code annotated for the use of high-level-synthesis tools, as in [16].

The generation of architectures from Polyhedral models has been less addressed than the generation of parallel programs. Most research on the polyhedral model start from affine loops in imperative programs, and not on equations. Most often, the target is the production of parallel programs, but the success of high-level synthesis tools have led some projects to target the generation of `C` code adapted to FPGA compilation [6, 29–31].

Recent research on parallel accelerators for machine learning consider *systolic arrays* as a target [26]), and make use of polyhedral optimization to obtain efficient designs.

Research on the polyhedral equational approach to target VHDL has been pursued in [4, 19], and the research presented in this paper results of a large extension of the MMALPHA software done to implement such methods.

10 Discussion

Several aspects of this research deserve comments.

- The choice of implementing the computation of the σ coefficient using a look-up table is motivated by the difficulty of computing the division on a FPGA, and the observation that the number of values taken by this coefficient is very limited (a few hundreds of values, depending on the size of the A matrix). However, this depends largely on the FPGA platform that is used for the implementation, and such a choice might be reconsidered if needed.

[3]Actually, without the calculation of the σ term, as it is supposed to be pre-calculated.

Program	Size	#DSP	#LUT	#FF	Latency (cycles)	# Processors	#ALPHA Lines	#VHDL lines	Synthesis time (seconds)
Full SM	N=3	15	750	452	$8 + 2N = 14$	$N + 1 = 4$	125	920	18.32
Full SM	N=7	35	1615	804	$8 + 2N = 22$	$N + 1 = 8$	125	920	18.64
Full SM	N=13	65	3115	1488	$8 + 2N = 34$	$N + 1 = 14$	125	920	18.23
Opt. SM	N=10	120	2888	2248	4	$N^2 = 100$	53	566	9.13
Opt. SM	N=16	288	6732	5130	4	$N^2 = 256$	53	566	9.64

Table 1. Results for the synthesis of several versions of the Sherman-Morrison formula, showing the size of the A matrix, the FPGA resources used by the design (number of DSPs, of Look-up-tables, and of flip-flops), the latency of the design in number of cycles, the number of *processors* of the design, the number of lines of ALPHA code and the number of VHDL lines of the code generated.

- One may worry about the numerical stability of the Sherman-Morrison method used here. This issue, shared by all matrix inversion methods, can be solved by several approaches. In the context of an FPGA implementation, one can try to find out, experimentally, the best number representation, either as fixed-point or floating-point. Another flexibility is given by the possibility to choose the initial A^{-1} matrix among a set of carefully selected ones, in such a way that the error of the inversion is minimized. All these aspects are currently under investigation.
- The data types of the ALPHA variables can be chosen according to the needs of the algorithm being implemented: integers, fixed point or floating point numbers of various size. In fact, the whole synthesis process is largely independent on the choice of these datatypes, which matters only when generating the VHDL code.
- The synthesis method that is described here has been applied successfully to many other algorithms: matrix operations, string matching, signal processing algorithms for example. It has currently been validated by a functional simulation of the VHDL code.
- Due to lack of space, we do not detail here, the time needed by each step of the synthesis. The scheduling step, which is supposed to be the critical step of the synthesis, takes only a small fraction of the total synthesis time, on the order of 10%, using the linear solver of the MATHEMATICA software. Most of the time is spent in the transformation of the time-space mapped ALPHA code into a multi-dimensional RTL description, which is essentially due to the use of the interpreted Wolfram language.
- In its current status, the MMALPHA software does not provide a facility to generate input/outputs and memorization of data. In the context of the present application, this is not a limitation, since the generated code is meant to be included in a schematic description used by the System Generator tool. The solution to overcome this limitation would be to implement transformations allowing I/O and memorization to be performed. In general, obtaining efficient implementations is a difficult problem, as it depends heavily on the hardware platform that is available.

11 Conclusion and Future Work

We have shown how the Sherman-Morrison formula for recomputing the inverse of a square matrix subject to a one-order perturbation can be expressed using a Polyhedral equational description, and then, transformed into a VHDL hardware description suitable for FPGA implementation.

We have presented results for the full Sherman-Morrison algorithm, and for a version optimized for the simulation of electrical circuits.

Our results show that the synthesis time is independent of the size of the problem, which is a main advantage of the polyhedral model, where calculations, be they expressed as loops or as multi-dimensional equations, are represented by operations on collections of data.

We have shown that a representation by polyhedral equations allows efficient parallel designs to be obtained automatically.

Future directions of research are to enlarge the set of available transformations, in order to target various architecture organizations, on FPGA on or other types of parallel architectures. To this end, an interconnection between MMALPHA and ALPHAZ would represent an interesting experimentation platform.

Another direction would be to take advantage of the functional nature of the ALPHA languages to explore algebraic transformations–for example, linear algebra properties–since the rewriting of equations certainly offers large opportunities of code simplifications.

12 Acknowledgments

This work was funded by the Natural Sciences and Engineering Research Council of Canada grants, Prompt, Opal-RT, Hydro-Québec, CMC Microsystems, and the Research Chair in Signals and Intelligence of High-Performance Systems.

The authors would like to answer the referees for their insightful comments and suggestions to improve the clarity of this paper.

References

[1] Corinne Ancourt and François Irigoin. 1991. Scanning Polyhedra with DO Loops. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Williamsburg, Virginia, USA) *(PPOPP '91)*. Association for Computing Machinery, New York, NY, USA, 39–50. https://doi.org/10.1145/109625.109631

[2] Cédric Bastoul. 2004. Code Generation in the Polyhedral Model Is Easier Than You Think. In *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*. Juan-les-Pins, France, 7–16.

[3] Belanger, J. and Venne, P. and Paquin, J.N. 2010. The what, where and when of real-time simulations. *Planet RT* (2010), 37––49.

[4] F. Charot, M. Nyamsi, P. Quinton, and C. Wagner. 2004. Modeling and Scheduling Parallel Data Flow Systems using Structured Systems of Recurrence Equations. In *Proceedings of ASAP 2004* (Galveston, Texas, USA). IEEE, 6–16.

[5] Florent de Dinechin, Patrice Quinton, and Tanguy Risset. 1995. Structuration of the ALPHA language. *Programming Models for Massively Parallel Computers* (1995), 18–24. https://api.semanticscholar.org/CorpusID:5828501

[6] Harald Devos, Kristof Beyls, Mark Christiaens, Jan M. Van Campenhout, Erik H. D'Hollander, and Dirk Stroobandt. 2007. Finding and Applying Loop Transformations for Generating Optimized FPGA Implementations. *Trans. High Perform. Embed. Archit. Compil.* 1 (2007), 159–178. https://api.semanticscholar.org/CorpusID:23749500

[7] Paul Feautrier. 1988. Parametric integer programming. *RAIRO-Operations Research* 22, 3 (1988), 243–268.

[8] Gautam and S. Rajopadhye. 2006. Simplifying Reductions. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '06)*. 30–41. https://doi.org/10.1145/1111037.1111041

[9] A. Hadizadeh, M. Hashemi, M. Labbaf, and M. Parniani. 2019. A Matrix-Inversion Technique for FPGA-based Real-Time EMT Simulation of Power Converters. *IEEE Transactions on Industrial Electronics* 2 (Feb. 2019), 1224–1234.

[10] H. Le Verge, C. Mauras, and P. Quinton. 1991. The ALPHA Language and its Use for the Design of Systolic Arrays. *The Journal of VLSI Signal Processing* 3, 3 (1991), 173–182.

[11] Vincent Lefebvre and Paul Feautrier. 1998. Automatic storage management for parallel programs. *Parallel computing* 24, 3-4 (1998), 649–671.

[12] C. Mauras. 1989. Alpha : un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones. Thèse de l'Université de Rennes 1, IFSIC.

[13] C. Mauras, P. Quinton, S. Rajopadhye, and Y. Saouter. 1990. Scheduling Affine Parameterized Recurrences by Means of Variable Dependent Timing Functions. In *Application Specific Array Processors*, S.Y. Kung, E.E. Schwartzlander, J.A.B. Fortes, and K.W. Przytula (Eds.). IEEE Computer Society Press, Princeton University, 100–110.

[14] Farid N. Najm. 2010. *Circuit Simulation*. John Wiley & Sons, Ltd., Hoboken, New Jersey.

[15] POLYLIB 2023. The Polylib Library. https://icps.u-strasbg.fr/polylib/ (online)

[16] J. Poupart, M. Lemaire, and D. Massicotte. 2023. FPGA Implementation of Sherman-Morrison Formula Using High-Level Synthesis and Graphical Blocks Programming . In *Proceeding of DCIS 2023* (Malaga, Spain).

[17] Fabien Quilleré and Sanjay Rajopadhye. 2000. Optimizing Memory Usage in the Polyhedral Model. *ACM Trans. Program. Lang. Syst.* 22, 5 (sep 2000), 773–815. https://doi.org/10.1145/365151.365152

[18] Patrice Quinton. 1984. Automatic Synthesis of Systolic Arrays from Uniform Recurrent Equations. In *Proceedings of the 11th Annual International Symposium on Computer Architecture (ISCA '84)*. 208–214. https://doi.org/10.1145/800015.808184

[19] P. Quinton, A. Chana, and S. Derrien. 2012. Efficient hardware implementation of data-flow parallel embedded systems. In *2012 International Conference on Embedded Computer Systems (SAMOS)*. 364–371. https://doi.org/10.1109/SAMOS.2012.6404202

[20] Patrice Quinton and Tanguy Risset. 2001. Structured Scheduling of Recurrence Equations: Theory and Practice. In *Proceedings of the Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation (SAMOS '01)*. 112–134. https://doi.org/10.1007/3-540-45874-3_7

[21] P. Quinton and V. Van Dongen. 1989. The Mapping of Linear Recurrence Equations on Regular Arrays. *The Journal of VLSI Signal Processing* 1 (1989), 95–113.

[22] S. Rajopadhye, G. Gupta, and DG. Kim. 2011. Alphabets: An Extended Polyhedral Equational Language. In *Proceedings of the 13th Workshop on Advances in Parallel and Distributed Computational Models*, Fujiwara Nakano, Bordim (Ed.).

[23] Sanjay V Rajopadhye and Richard M Fujimoto. 1990. Synthesizing systolic arrays from recurrence equations. *Parallel Comput.* 14, 2 (1990), 163–189. https://doi.org/10.1016/0167-8191(90)90105-I

[24] S. V. Rajopadhye, S. Purushothaman, and R. M. Fujimoto. 1986. On Synthesizing Systolic Arrays from Recurrence Equations with Linear Dependencies. In *Proceedings, Sixth Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer Verlag, LNCS 241, New Delhi, India, 488–503.

[25] Jack Sherman and Winifred J. Morrison. 1950. Adjustment of an Inverse Matrix Corresponding to a Change in One Element of a Given Matrix. *The Annals of Mathematical Statistics* 21, 1 (1950), 124 – 127. https://doi.org/10.1214/aoms/1177729893

[26] Jie Wang and Jason Cong. 2021. Search for Optimal Systolic Arrays: A Comprehensive Automated Exploration Framework and Lessons Learned. *CoRR* abs/2111.14252 (2021). arXiv:2111.14252 https://arxiv.org/abs/2111.14252

[27] Xilinx 2022. System Generator for DSP Overview (Online). https://docs.xilinx.com/r/en-US/ug948-vivado-sysgen-tutorial/System-Generator-for-DSP-Overview

[28] Tomofumi Yuki, Gautam Gupta, DaeGon Kim, Tanveer Pathan, and Sanjay Rajopadhye. 2013. AlphaZ: A System for Design Space Exploration in the Polyhedral Model. In *Languages and Compilers for Parallel Computing*, Hironori Kasahara and Keiji Kimura (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 17–31.

[29] Ruizhe Zhao, Jianyi Cheng, Wayne Luk, and George A. Constantinides. 2022. POLSCA: Polyhedral High-Level Synthesis with Compiler Transformations. In *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*. 235–242. https://doi.org/10.1109/FPL57034.2022.00044

[30] Wei Zuo, Peng Li, Deming Chen, Louis-Noël Pouchet, Shunan Zhong, and Jason Cong. 2013. Improving polyhedral code generation for high-level synthesis. In *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. 1–10. https://doi.org/10.1109/CODES-ISSS.2013.6659002

[31] Wei Zuo, Yun Liang, Peng Li, Kyle Rupnow, Deming Chen, and Jason Cong. 2013. Improving High Level Synthesis Optimization Opportunity through Polyhedral Transformations. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '13)*. Association for Computing Machinery, New York, NY, USA, 9–18. https://doi.org/10.1145/2435264.2435271

A Implementation of MMALPHA

MMALPHA is implemented in the Wolfram language, under the MATHEMATICA software. The reason of this choice is the existence of a long legacy software, since a first, very sketchy, version was designed around 1995.

MMALPHA makes use of the polyhedral library POLYLIB [15] and uses a parser and a pretty-printer written in C. The PIP software is also used.

The interest of using MATHEMATICA lies in some of the functions available in this software, for example, the linear programming solver which is directly available.

B VHDL Translation of one Register Equation

The following VHDL code is the translation of a *register equation* of the form $Y[t,p] = \text{CopyBMirrMovedIn}[t-p,p]$ that appears in the generation of the full Sherman-Morrison version, for $N = 13$. It contains a `Generate` statement for the p index, and another one for the i index needed for an intermediate register named `YReg`. Inside the `Generate` statements, there is a sequential process needed to produce registers. The $Y[t,p]$ value is obtained through an assignment which is controlled by the value of the time counter.

```
-- Translation of the definition of Y
-- Equation (register):
-- Y[t,p] = CopyBMirrMovedIn[t-p,p]
-- Domain: {t,p | p+1<=t<=p+13; 1<=p<=13}
G81: FOR p IN 1 TO 13 GENERATE
  G79: FOR i IN 1 TO 13 GENERATE
    PROCESS(clk) BEGIN
      IF (clk = '1' AND clk'EVENT) THEN
        IF CE = '1' THEN
          YReg(p)(i) <= YReg(p)(i-1);
          YReg(p)(0) <= CopyBMirrMovedIn(p);
        END IF;
      END IF;
    END PROCESS;
  END GENERATE;
  Y(p) <= YReg(p)(-1 + p)
  WHEN
     (counter - counterDelay >= p+1)
  AND
     (counter - counterDelay <= p+13);
END GENERATE;
```