Alpha and the Polyhedral Model

This is a section drawn from a larger paper, "Optimizing Memory Usage in the Polyhedral Model" by Quilleré and Rajopadhye. It is relatively self contained and presents the main ideas behind the ALPHA language and how one compiles it is (sequential or parallel) imerative code.

Categories and Subject Descriptors: []:

Additional Key Words and Phrases: Dataflow Analysis, Memory Management, Parallel Code Generation, Polyhedral Model

1. INTRODUCTION

The polyhedral model is a formalism for reasoning about massively parallel computations (potentially even infinite) with a compact (finite) representation. It was originally developed in the context of systolic array synthesis, and uses a generalization and extension of the systems of *Uniform Recurrence Equations* first proposed by Karp Miller and Winograd [1967]. This generalized formalism is called *Systems of Affine Recurrence Equations* (SAREs) over *polyhedral domains* (the *domain* of a recurrence equation is the set of index points where it is defined), and forms the foundation of the polyhedral model.

Feautrier paved the way for its use in parallelizing compilers by showing that by performing exact data flow analysis [Feautrier 1991] on a certain class of loops in an imperative program (called *static control loops*), one obtains an intermediate (dynamic single assignment) form which is isomorphic to an SARE over polyhedral domains. The PAF (*paralléliseur automatique de Fortran*) project underway in his group uses a generalization of systolic scheduling and allocation techniques for generating parallel code. A number of researchers propose using similar single assignement intermediate forms for parallelizing compilers. Independently of Feautrier, Maydan et al. [Maydan et al. 1993] propose an analogous array data flow analysis for the SUIF project. Knobe and Daly [Knobe and Dally 1995] introduce the notion of index subspaces, which could be viewed as a restricted class of domains. They give an algorithm to transform a sequential code into a single assignement form (this form is not necessarily sematically equivalent to the original program, it simply has the information about the subspaces, and can therefore be computed more efficiently at the price of reduced precision). The LOOPO project at the University of Passau [Griebl and Lengauer 1996] also explores parallelization in the polyhedral model and its extensions, as does the OPERA project [Mongenet 1994] at the University of Strasbourg.

In 1989 Mauras designed the ALPHA language based on the SARE formalism. ALPHA is a functional data parallel language, and has been used at IRISA for automatic derivation of systolic and regular VLSI arrays, and also for parallel code generation. MMALPHA is a prototype transformation system written in Mathematica for reading and manipulating ALPHA programs through correctness preserving transformations and (eventually) generating VHDL description of regular VLSI circuits or (sequential or parallel) code for programmable processors. We will use the expressions "SAREs over polyhedral domains" and "ALPHA programs" interchangeably.

2. THE ALPHA LANGUAGE

ALPHA is a strongly typed data-parallel functional language based on SAREs, originally designed in the context of systolic array synthesis [Le Verge et al. 1991]. ALPHA variables are type declared at the beginning of a program, and represent "polyhedral shaped" multidimensional arrays. The polyhedra specified in the declaration are called the *domains* of the variables. For example, a (strictly) lower triangular, real matrix is specified by the declaration:

"`A: {i,j| 0<j<i<=N} of real`"

To introduce the main features of ALPHA, consider solving, using forward substitution, a system of linear inequalities, $Ax = b$, where A is a lower triangular, $n \times n$ matrix with unit diagonal. A high-level, mathematical descrition of the program would be

$$\text{for } i = 1 \ldots n, \quad x_i = \begin{cases} \text{if } i = 1 & b_j \\ \text{if } i > 1 & b_i - \displaystyle\sum_{j=1}^{i-1} A_{i,j} x_j \end{cases} \tag{1}$$

The corresponding ALPHA program (Fig. 1) is identical, except for syntactic sugar. The first line names the system and declares that it has a positive integer parameter, `N`, which can take any integer value greater than 1. The next three lines are declarations of the input and output variables of the system (respectively, before and after the `returns` keyword). The domain of `A` is triangular, while `B` and `X` are one dimensional variables (vectors). A system may also have local variables (not present here), which are declared after the system header, using the keyword `var`. The body of the program is a number of equations delineated by the `let` and `tel` keywords. Here, we have a single equation corresponding to (1) above. The `case` construct has the usual meaning and allows us to define conditional expressions. The `restrict` construct has the syntax <domain> : <expr>, and denotes the expression, <expr> but restricted to the subset of index points in <domain>.

The `reduce` construct corresponds to the summation in (1) and has three parts: an associative and commutative operator, a projection function and an expression. Here, the operator is `+`. The projection is `(i,j->i)`, and denotes (intuitively) the fact that within the body of the `reduce`, there are two indices, `i` and `j`, but only `i` is visible outside the scope of the `reduce`, i.e. the two dimensional expression of the body is projected by the mapping `(i,j->i)` to a one-dimensional one. The body of the `reduce` is the expression, `A * X.(i,j->j)`.

Here, `(i,j -> j)`, is a *dependency* and denotes the fact that to compute the body at index point, `[i,j]`, we need the value of `X` at index point `[j]` (the dependency on `A` is not explicitly written—it is the identity). Dependencies are an important aspect of ALPHA. They have the syntax, `(idx, idx, ...  -> i-expr, i-expr, ...)`, where each `idx` is an index name, and `i-expr` is an *affine* expression of the system parameters and the `idx`'s. ALPHA and MMALPHA use this syntax for specifying a multidimensional affine function in many different contexts (for example, the

```
system ForwardSubstitution : { N | N>1 }         -- comments are like this
               ( A : { i,j  | 0<j<i<=N } of real; -- a 2D input variable
                 B : { i    | 0<i<=N } of real)   -- a 1D input variable
       returns ( X : { i    | 0<i<=N } of real ); -- a 1D output variable
let
  X = case
        {| i=1} : B;
        {| i>1} : B - reduce(+, (i,j -> i), A * X.(i,j->j));
      esac;
tel;
```

Figure 1: ALPHA *program for the forward substitution algorithm*

projection function in a `reduce` expression). This syntax can be viewed as a special kind of lambda expression, restricted to affine mappings from $\mathcal{Z}^n$ to $\mathcal{Z}^m$. Such a function, f may be equivalently represented by a $m \times n$ matrix, A and an m-vector, a, i.e., $f(z) = Az + a$, and we will later use this form for analysis purposes.

2.1 Semantics of Alpha Expressions

An ALPHA expression is either (i) a varaible or constant, (ii) pointwise operators applied to one (or more, but typically two) expressions, (iii) the restriction of an expression to a domain, (iv) the casewise definition of multiple expressions (v) the composition of an expression with a dependency, or (vi) a reduction. We will now briefly and intuitively summarize the semantics of ALPHA expressions.

ALPHA is a data-parallel language and hence expressions denote collections of values. Indeed *all* expressions (not just variables) can be viewed as multi-dimensional arrays and denote a function from indices to values. ALPHA semantics [Mauras 1989] consist of two parts: a semantic function and a domain. The semantic function is defined using classic methods and is fairly obvious (the only subtle point being dependencies), but the domains are a unique aspect of ALPHA. *Every* (sub) expression in a program has a domain, and it can be determined from the domains of its subexpressions, as summarized below.

—*Identifiers and Constants:* An identifier simply denotes the variable it identifies, and is defined over its *declared* domain. In general this is a finite union of polyhedra. A constant expression denotes the constant itself and its domain is $\mathcal{Z}^0$, the zero-dimensional polyhedron.

—*Pointwise Operators:* The expression $E \oplus F$ denotes the pointwise application of the operator $\oplus$ to the corresponding elements of the expressions E and F and hence its domain is $\mathrm{Dom}(E) \cap \mathrm{Dom}(F)$, the intersection of the domains of its subexpressions. There may be unary or ternary (the *if-then-else* which is *strict*) pointwise operators, with similar semantics.

—*Restriction:* The expression $\mathcal{D} : E$ denotes the expression E, but restricted to the domain $\mathcal{D}$, and hence its domain is $\mathcal{D} \cap \mathrm{Dom}(E)$.

—*Case:* The expressions case $E; F$ esac denotes an expression which has alternative definitions (of course, there may be more than two alternatives). Its domain is $\mathrm{Dom}(E) \cup \mathrm{Dom}(F)$, the (disjoint) union of the domains of its subexpressions.

—*Dependency:* As mentioned above, the dependency $z \to f(z)$ itself denotes the affine function, f. Now, $E.(z \to f(z)$ denotes an expression whose value at z is the value of E at $f(z)$. Hence its domain must be the set of points which are mapped by f to a point in the domain of E, i.e., the preimage of $\mathrm{Dom}(E)$ by f. The dependency can be viewed abstractly as a mechanism for *aligning* data-parallel oblects. For example consider the expression `A+B.(i,j->j,i)`. The standard view of this is an expression whose value at the point `[i,j]` is the sum of `A[i,j]` and `B[j,i]`. Alternatively we may also view the subexpression `B.(i,j->j,i)` as the transpose of `B` with is aligned with `A` by the `+` operator.

—*Reductions:* The expression $\mathrm{reduce}(\oplus, (z \rightarrow f(z)), E)$ denotes an expression whose domain is the *image* of the domain of E by f. Its value at any point, z, in this domain is obtained by taking all points in the domain of E which are mapped to z by f, and applying the associative and commutative operator $\oplus$ to the values of E at these points.

Due to the mathematical properties of polyhedra and affine functions [Mauras 1989], the domain of any ALPHA expression is a finite union of integral polyhedra, and is determined automatically by a library for manipulating polyhedra [Wilde 1993].

These closure properties have an important consequence. Every expression can be simplified into a unique *normal form*, called the *case-restriction-dependency* form. It consists of an (optional) outer case, each of whose branches is a (possibly restricted) simple expression. A simple expression consists of either variables or constants composed with a single dependency function (if this is the identity, it may be omited), or pointwise operators applied to such subexpressions. This simplification is obtained by a set of rewrite rules which combine dependencies together, eliminate empty domain expressions, etc.

Let `X = {i,j | 0<i,j<=N} : X.(p,q->q,p) + 0.(i,j->)` be an equation in an ALPHA program (observe that the domain of the constant `0` is $\mathcal{Z}^0$, whose preimage by the function $(i, j \rightarrow)$ is $\mathcal{Z}^2$ which is coherent with the rest of the expression). Since the right hand side of this equation is a normalized expression, we can always (i) rename the indices in the restrictions and in the dependencies to be identical (say, by by replacing `p` and `q` by `i` and `j`, respectively), and (ii) moving the index names in the dependencies (left of the "`->`") and in domains (left of the "`|`") to the left of the entire equation, yielding the following "sugarized" syntax, which is often more readable.

```
X[i,j] = {| 0<i,j<=N} : X[q,p] + 0[]
```

This is called the *array notation*, and indeed, all subsequent examples in this paper are normalized and presented in this notation.

2.2 Transformations in Alpha

We now introduce some of the ALPHA transformations used in the MMALPHA system for converting our program into a single assignment C program. The first one is used to **serialize** the reduction operation, and requires two parameters: the "direction" in which we accumulate the partial sums, and the name of the temporary variable. For our example, if we choose to accumulate in the *increasing* direction of `j` (specified as "`(i,j->i,j+1)`") and name the new variable, `f`, we obtain the ALPHA program shown in Fig. 2.

```
            ⋮
var
  f : {i,j | (2,j+1)<=i<=N; 0<=j} of real;
let
  f[i,j] = case
        {| j=0} : 0[];
        {| 1<=j} : f[i,j-1] + A * X[j];
           esac;
  X[i] = case
        {| 2<=i} : B[i] - f[i,i-1];
        {| i=1} : B[i];
         esac;
tel;
```

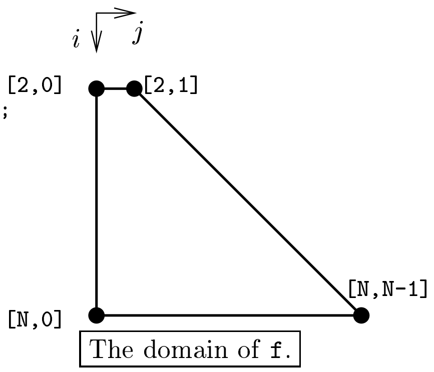

Figure 2: *The forward substitution program after serialization*

Observe that the domain of `f` is `{i,j| 0<=j<i; 2<=i<=N}` (a "nearly triangular" trapezium defined by the points `[2,0]` `[2,1]` `[N,N-1]` and `[N,0]`). Also note that in the new equation for `X`, the reduction is replaced by the expression `f[i,i-1]`, and the new equation for `f` consists of an initialization at the boundary, and an accumulation using the *body of the original reduction*. All this is automatically determined by MMAlpha using a library for manipulating polyhedra [Wilde 1993].

Perhaps the most important transformation in the Alpha system is the **change of basis** (COB). The intuition behind it is as follows. Since an Alpha variable can be viewed as a multidimensional array defined over a polyhedral domain, we should be able to "move" (or otherwise change the "shape" of) its domain and construct an equivalent program. In MMAlpha, a COB can be applied to any local variable, say `X`, using any affine function, $\mathcal{T}$ that admits a *left inverse* $\mathcal{T}^{-l}$ for all points in the domain of `X`. The following changes are made to the program, yielding an equivalent program

—Replace D_{X}, the domain of `X` by $\mathcal{T}(D_{\mathrm{X}})$, its image by $\mathcal{T}$.

—If E is the expression on the right hand side of the equation defining `X`, replace it by $E.\mathcal{T}^{-1}$.

—Replace all occurrences of `X` on the right hand side of *any* equation by $\mathtt{X}.\mathcal{T}$.

Many program transformations such as alignment, scheduling, processor allocation, etc., can be viewed as special cases of COB.

2.3 Compiling Alpha to Imperative Code

An Alpha program can be evaluated using the simple interpretive strategy outlined in Sect. 1, but it suffers from the standard disadvantages of demand driven evaluation, namely the overhead of context switching. Quinton et al. [1995] showed how the overhead of suspensions can be avoided by

—statically determining a schedule for each variable using well known methods (the schedule may be one-dimensional as in classic systolic synthesis [Rajopadhye and Fujimoto 1990; Quinton and Van Dongen 1989; Mauras et al. 1990] or multidimensional as proposed by Feautrier [1992]); let us assume that the schedule is

```
            ⋮
x: {t,j| t=2j; 1<=j<=N} of real;
f: {t,j| (2j+1,j+2)<=t<=j+N; 0<=j; 5<=N} of real;
let
  f[t,j] =
      case
        {| j=0} : 0[];
        {| 1<=j} : f[t-1,j-1] + A[t-j,j] * x[2j,j];
      esac;
  x[t,j] =
      case
        {| j+2<=t} : B[t-j] - f[2t-2j-1,t-j-1];
        {| t=j+1} : B[t-j] - 0[];
      esac;
tel;
```

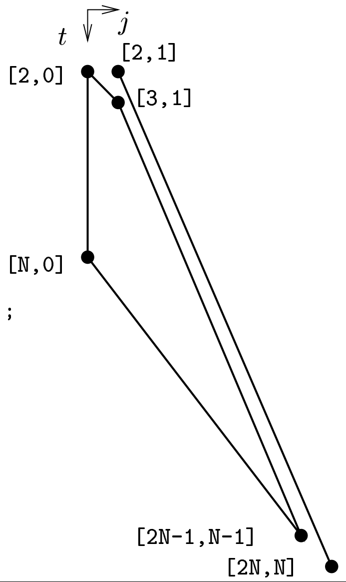

Figure 3: *The scheduled and transformed program and new domains of* f *and* x

k-dimensional.

—rewriting the program, by means of an appropriate COB, such that the first k indices of the domains of *all* variables are the k time dimensions; and

—visiting the domains of all the variables (i.e., a union of polyhedra, each of whose first k indices are now common and represent time) in lexicographic order of these k indices (and other indices, if any, in any order), and evaluating the body of the recurrence *without* suspensions (the arguments are guaranteed to have been already evaluated).

We now illustrate these steps for the forward substitution example above.

Scheduling. Alpha is based systems of affine recurrence equations (SAREs), and the past decade has seen considerable research on the SARE *scheduling* problem [Rajopadhye and Fujimoto 1990; Feautrier 1992]. It may formulated as follows. For each variable X in a SARE, determine a function $t_{\mathrm{X}}(z)$ which gives a k-dimensional vector representing the time instant at which $\mathrm{X}(z)$ is computed. The time function must respect causality: whenever $\mathrm{X}(v)$ depends on $\mathrm{Y}(v')$, then $t_{\mathrm{X}}(v) \prec t_{\mathrm{Y}}(v')$. In general, the form of the schedule is a *multidimensional affine function.* The schedule provides the first k rows of the COB transformation, and allows one to then interpret the k first dimensions of a variable domain as the time instant of its computation.

For our example, it turns out that $t_{\mathtt{f}}(i,j) = i + j$ and $t_{\mathtt{x}}(i) = 2i$ are valid (one dimensional) affine schedules, and thus a parallel program would be able to compute the forward substitution in $2n$ steps. However, in order to illustrate the transformations needed, let us choose to derive sequential code. Observe that we will now have a doubly nested (sequential) loop, i.e., the variable `f` will have a two-dimensional schedule, say, $t_{\mathtt{f}}(i,j) = \begin{pmatrix} i+j \\ j \end{pmatrix}$. In order to generate code, we must therefore "embed" the one-dimensional variable `x` into this two-dimensional (temporal) index-space. This can be done by choosing a schedule $t_{\mathtt{x}}(i) = \begin{pmatrix} 2i \\ i \end{pmatrix}$. In other words, we apply a COB, namely `(i,j -> i+j,j)` and `(i -> 2i,i)` to `f` and `x` respectively. This yields the program shown in Fig. 3 (the first index has now been renamed `t`). "Embedding" `x` into the two dimensional index space is a transformation called *alignment* [Li and Chen 1991], and our COB implemented a specific schedule and alignment simultaneously.

The same approach can be used to derive parallel code. For example, if we retain the schedule, $t_{\mathtt{f}}(i,j) = i+j, t_{\mathtt{x}}(i) = 2i$ and use the functions $p_{\mathtt{x}}(i,j) = p_{\mathtt{f}}(i,j) = j$ as *allocation functions* (i.e., `Y[i,j]` and `x[j]` are both computed on processor `j`), we will end up with exactly the *same* COB, and the same transformed program except that the j index is a parallel loop.

Code generation. This step has been detailed in [Wilde 1995; Quinton et al. 1995], and we give a very brief intuitive summary. First of all, we generate declarations of multidimensional arrays to store the variables of the program. For this we allocate memory corresponding to the bounding box of the domains of the variables. This is deduced directly from the ALPHA variable declarations.

Recall that after the COBs, we have a program where, for all variables, some indices (in a specific order) denote time, and the others (if any) are guaranteed to carry no dependencies. The program is now rewritten in a slightly different form (called **r-normal form**) which has no `case`'s, but variables may be defined by multiple equations, possibly qualified by polyhedral domains. Next, the time indices of all equations are separated, and the equations are merged into groups (qualified by mutually disjoint domains). Although this may begin to resemble sequential code, it is still functional, and the order of the equations has no significance. The next step, called **code ordering** "sorts" the equations so that whenever an equation group textually precedes another, it is also temporally before the other (see [Quinton et al. 1995] for details). At this point, the "`=`" may be safely interpreted as an assignment rather than an equation. Fig. 4 shows this stage (called **imperative normal form**) for our example. C code may now be generated directly from it by a special pretty printer.

REFERENCES

FEAUTRIER, P. 1991. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming 20,* 1 (Feb.), 23–53 (or 23–52??).

FEAUTRIER, P. 1992. Some efficient solutions to the affine scheduling problem, part I, one dimensional time, and part II, multidimensional time. *International Journal of Parallel Programming 21,* 5-6 (Oct.), 313–348, 389–420.

GRIEBL, M. AND LENGAUER, C. 1996. The loop parallelizer LooPo. In *Proc. Sixth Workshop on Compilers for Parallel Computers,* M. Gerndt, Ed. Konferenzen des Forschungszentrums Jülich, vol. 21. Forschungszentrum Jülich, 311–320.

KARP, R., MILLER, R., AND WINOGRAD, S. 1967. The organization of computations for uniform recurrence equations. *Journal of the Association for Computing Machinery 14,* 3 (July), 563–590.

KNOBE, K. AND DALLY, W. J. 1995. The subspace model: A theory of shapes for parallel systems. In *5th Workshop on Compilers for Parallel Computers.* Malaga, Spain.

```
            ⋮
var
  x : {t,j | t=2j; 1<=j<=N} of real;
  f : {t,j | (2j+1,j+2)<=t<=j+N; 0<=j; 5<=N} of real;
let {t | t=2} ::
          let {t,j | j=0} : f[t,j] = 0[];
          {t,j | j=1} : x[t,j] = B[t-j] - 0[];
      tel;
    {t | t=3} ::
      let {t,j | j=0} : f[t,j] = 0[];
          {t,j | 2j+1<=t<=j+N; 1<=j} : f[t,j] = f[t-1,j-1] + A[t-j,j] * x[2j,j];
      tel;
    {t | 4<=t<=N} ::
      let {t,j | j=0} : f[t,j] = 0[];
          {t,j | 2j+1<=t<=j+N; 1<=j} : f[t,j] = f[t-1,j-1] + A[t-j,j] * x[2j,j];
          {t,j | t=2j} : x[t,j] = B[t-j] - f[2t-2j-1,t-j-1];
      tel;
    {t | N+1<=t<=2N-1} ::
      let {t,j | 2j+1<=t<=j+N; 1<=j} : f[t,j] = f[t-1,j-1] + A[t-j,j] * x[2j,j];
          {t,j | t=2j} : x[t,j] = B[t-j] - f[2t-2j-1,t-j-1];
      tel;
    {t | t=2N} :: {t,j | t=2j} : x[t,j] = B[t-j] - f[2t-2j-1,t-j-1];
tel;
```

Figure 4: *The program in imperative normal form*

LE VERGE, H., MAURAS, C., AND QUINTON, P. 1991. The Alpha language and its use for the design of systolic arrays. *Journal of VLSI Signal Processing 3,* 3 (Sept.), 173–182.

LI, J. AND CHEN, M. 1991. The data alignment phase in compiling programs for distributed-memorymachines. *Journal of Parallel and Distributed Computing 13,* 2 (Oct.), 213–221.

MAURAS, C. 1989. Thèse de l'Université de Rennes 1. Ph.D. thesis.

MAURAS, C., QUINTON, P., RAJOPADHYE, S., AND SAOUTER, Y. 1990. Scheduling affine parameterized recurrences by means of variable dependent timing functions. In *International Conference on Application Specific Array Processing*, S. Y. Kung and E. Swartzlander, Eds. IEEE Computer Society, Princeton, New Jersey, 100–110.

MAYDAN, D., AMARSINGHE, S. P., AND LAM, M. 1993. Array data flow analysis and its use in array privatization. In *Principles of Programming Languages*. ACM, 2–15.

MONGENET, C. 1994. Data compiling for systems of uniform recurrence equations. *Parallel Processing Letters 4,* 3, 245–257.

QUINTON, P., RAJOPADHYE, S., AND WILDE, D. 1995. Deriving imperative code from functional programs. In *7th Conference on Functional Programming Languages and Computer Architecture*. ACM, La Jolla, CA, 36–44.

QUINTON, P. AND VAN DONGEN, V. 1989. The mapping of linear recurrence equations on regular arrays. *Journal of VLSI Signal Processing 1,* 2, 95–113.

RAJOPADHYE, S. AND FUJIMOTO, R. 1990. Synthesizing systolic arrays from recurrence equations. *Parallel computing 14*, 163–189.

WILDE, D. 1993. A library for doing polyhedral operations. PI 785, Irisa. dec.

WILDE, D. 1995. From alpha to imperative code : A transformational compiler. Ph.D. thesis, Oregon State University.