INSIGHT: A Dataflow Language for Programming Vision Algorithms

Linda G. Shapiro, Robert M. Haralick, Michael J. Goulish

Machine Vision International
Ann Arbor, Michigan 48104

Abstract

Machine vision systems used in industrial applications must execute their algorithms in real time to perform such tasks as inspecting a wire bond or guiding a robot to install a part on a car body moving along a conveyor. The real time speed is achieved by employing simple or very clever algorithms and by designing parallel architectures and parallel algorithms for some tasks. The majority of the work on parallel architectures has been limited to architectures that support image processing, but not mid- or high-level vision. In order for more complex vision algorithms to execute in real time, a more flexible architecture is needed.

Our conceptual approach to the problem is a reconfigurable systolic network. Each configuration of the network implements an algorithm or class of algorithms. A high-level language expresses the algorithms in a relational form that can be easily translated to the specification for a configuration. The language must be able to encode low-, mid-, and high-level vision algorithms and to efficiently handle not only pixel data, but also higher level structures. In this paper we describe a dataflow language called INSIGHT, which we have designed to meet these needs, and give several examples of parallel machine vision algorithms expressed in the language.

I. INTRODUCTION

The main difference between industrial machine vision and general computer vision research is that the industrial applications must run in real time. Real time execution is achieved either by employing very simple or very clever algorithms or by designing algorithms that can make use of parallel architectures. The parallel machines that have been designed for vision are mainly pipeline machines and cellular array machines [1,3]. The pipeline machines consist of a sequence of stages. The pixels of an image pass sequentially through each of the stages where they may be delayed or may become one of the operands of the operation performed by that stage. If there are n stages, then n operations at a time can be performed in parallel.

A cellular array machine consists of an interconnected array of processors. A subimage enters the array machine, one pixel per processor and neighborhood operations are performed on all the pixels in parallel. Again the amount of parallelism depends on the number of processors. Both pipeline and array machines have mainly been used for image processing, not for mid-level or high-level vision.

In order to execute more complex vision algorithms in real time, a more flexible parallel architecture is needed. The architecture must be able to handle more than 8 bit integers representing pixels. It should be able to handle a variety of datatypes representing a number of entities such as region labels and attributes, arc segment properties, and relational types. It should be able to execute efficiently not just from classes of processing operations, but a large variety of vision algorithms. It must be a multi-instruction, multi-datastream, reconfigurable architecture capable of operating systolically to maintain efficiency. Its reconfiguration capability must include connections of processor to processor or memory to processor.

Traditionally, useful algorithms that must run at a high speed have been translated to a hardware implementation designed to maximize parallelism and minimize execution time. The translation is done for a small specified set of algorithms by a skilled hardware engineer. Eventually a machine is produced that executes only this set of algorithms.

This approach is not feasible for building a vision system that must perform a variety of different algorithms and should be able to incorporate new algorithms as they are developed. Here, there must be an automatic translation of each algorithm from a high-level language into an efficient hardware configuration for a machine that reconfigures itself for each task it is commanded to perform. Such a flexible machine is called a reconfigurable systolic network (RSN), and the type of language that translates directly into a configuration of the network is called a dataflow language.

The language that is needed must allow the expression of vision algorithms in a form that can be easily translated to an architectural configuration for both pixel pushing and higher level vision data structure processing. Hardware programming languages and graph description languages are at too low a level. The interesting thing about the data flow in a systolic network is that a high level specification of the configuration of the network is a specification of the program the network is executing. This is different from Von Neumann architectures in which a specification of the architecture tells nothing about what program is executing. The low level specification of a network is a graph having labeled arcs and nodes and has nothing about it which is sequential or procedural. Likewise, a high level specification need not be sequential or procedural. A high level specification of a network is just a specification of the relations which hold in the network. So specification of the configuration of a systolic network amounts to specifying...
relations. Since the specification is the program which the network executes, the language used to program a systolic network is a language of relations. The language must be naturally non-procedural. From a high level perspective, the semantics of the language describe the essence of the architecture.

INSIGHT is a language in the LUCID family of data flow languages [2] that we have developed to meet the needs described above. In section II we describe the reconfigurable network. In section III we describe the INSIGHT language, and in section IV we give some preliminary examples of algorithms encoded in INSIGHT.

II. Reconfigurable Systolic Network: Important Concepts

The design of a reconfigurable systolic network architecture for computer vision is being done in the opposite order from the design of a traditional machine. Instead of designing the hardware first and then the languages and software, we are designing the language first. The exact design of the RSN will depend on simulation results obtained after encoding a representative set of algorithms. However, in order to specify the language, there must be some understanding of the concepts that will be embodied in the architecture.

The operation of the RSN involves the flow of sequences of values through a network of architectural primitives. More formally, a configuration consists of a set of processors \( P \) and a specification \( c \) of the interconnections between the processors. In this discussion a memory is considered a processor. Each processor \( P \in P \) is a pair \( P = (f_p, O_p) \) where \( f_p \) is a named set of input lines and \( O_p \) is a named set of output lines. Each connection \( c \in C \) is a quadruple \( c = (o_i, p_1, l, p_2) \) specifying that output line \( o \) of processor \( p_1 \) connects to input line \( l \) of processor \( p_2 \). Since different processors may take different amounts of time to process their inputs and produce their outputs, there must be some conventions that insure a processor will only execute when it has valid data. For this purpose, there is a state associated with each data line. A state is a pair of values \( s = (r, q) \) where \( r \) is an indication of readiness and \( q \) is an indication of acceptance. Legal values for \( r \) are:

1) preactive: the processor that produces the data on this line has not yet produced any values,

2) active and ready: valid data, ready to be used by the processors that require it,

3) active and not ready: the data on the line is an old value that was already consumed, but the new value is not yet ready for consumption,

4) postactive: the processor that produces the data on this line has terminated production; no new values will appear in this execution of the algorithm.

Legal values for \( q \) are consumed and un consumed.

A process can execute when all of its inputs are in the state (active and ready, un consumed) and all of its outputs from its previous execution have been consumed by every process to which they are inputs. The execution of the process takes some finite amount of time. When the execution is just starting, the output lines are

in the state (active and not ready, un consumed). During execution, each one eventually becomes (active and ready, un consumed). As soon as an output line reaches this state, it is available to the processes that wish to consume it. It reaches the state (active and ready, consumed) only when all of its potential consumers have consumed it.

If at least one input to a process is in state (preactive,*), then all of its outputs are in state (preactive,*), after its previous outputs have been consumed. If at least one input to a process is in state (postactive,*), then all of its outputs are in state (postactive,*) after its previous outputs have been consumed. In general, INSIGHT programmers do not have to worry about these states; the synchronization is taken care of by the hardware.

A sequence is an ordered stream of values that are either input to the RSN or are produced by one of the processors of the RSN. In either case, the values of the sequence are associated with a group of data lines. There are two kinds of conceptual orderings associated with the data lines. The first ordering is based on the counting of the real time clock. Each group of data lines necessarily has a value at each real time clock tick. The value of the data line group at the \( i \)th clock tick consists the \( i \)th element in the real time sequence associated with those lines. However, in the RSN, it is a processor that puts a value onto a group of data lines. As described above, a processor generates a new value only when each of its output values has been consumed by all of its consumers and when the inputs it requires are all active and ready. A clock tick occurring when all outputs have been consumed and all inputs are ready generates a permission tick. The second kind of ordering is based on the permission ticks. The value of the data line group at the \( i \)th permission tick is the \( i \)th element in the process time sequence associated with those lines.

When thinking logically about an algorithm in terms of sequences of values that the processors must generate, the INSIGHT programmer generally thinks about the process time sequence. Only when real time signals to or from an external device are input to or output from the RSN might the programmer wish to deal with real time sequences. From the point of view of the RSN hardware, however, it is the real time sequences that must be handled.

III. The INSIGHT Language

III.1 Programs, Activities, and Functions

INSIGHT programs specify relationships among sequences that will translate to relationships among architectural primitives of the RSN. A program is a sequence of configurations of the RSN designed to achieve some goal. The arguments of a program are supplied by and its results must be received by entities outside the RSN such as frame buffers, other external memories, and CRTs. In the simplest case, a program maps to a single configuration of the RSN that can produce the desired results. If the available hardware is not sufficient, then the program maps to several configurations called activities, each of which write temporary results to memories and which, when performed sequentially, produce the desired results. INSIGHT programs are modular, they may invoke INSIGHT functions to per-
form subtasks. An INSIGHT function translates to a subgraph of the configured hardware that is useful in one or more parts of the total algorithm. It is, however, closer to the usual concept of a macro than a function in a procedural language, since a new copy of the subgraph must be included wherever the results of the function are needed. This allows all such copies of the function to operate in parallel. The following simple example illustrates the program and the function in INSIGHT.

```plaintext
program add_and_mult(a[0:99], b[0:99], c[0:99],
                     d[0:99] : integer memory)
    result[0:99] : integer memory
where
    function add(x1, x2 : integer sequence) : integer sequence
where
    add = x1 + x2;
end;
declare sa, sb, sc, sd, t1, t2, index: integer sequence;
relations
    index = 0 fby index + 1 until index == 99
    sa = a[index];
    sb = b[index];
    sc = c[index];
    sd = d[index];
    t1 = add(sa, sb);
    t2 = add(sc, sd);
    result[index] = t1 * t2;
endwhere;
The inputs to the program are four separate integer memories named a, b, c, and d, which can be thought of as vector arrays, of 100 elements each. The output is a single integer memory called result. The purpose of the program is to produce the equivalent of the procedural code

```plaintext
for i = 0 to 99
    result[i] = (a[i] + b[i]) * (c[i] + d[i])
using an architectural configuration that performs the two adds in parallel. Add, which was written as a function just to illustrate what functions look like, takes in two sequences z1 and z2 of integers and outputs a single sequence of integers. Because add has only one output sequence, we don't have to give its output a separate name; the variable name add refers to the output sequence. The program add_and_mult generates a sequence called index of the integer values from 0 to 99 using the fby operation, which will be defined in section III.4. It uses this sequence of index values to address all five memories in parallel. The values that are indexed in memories a, b, c, and d become elements of the sequences sa, sb, sc, and sd. Pairwise elements of sa and sb are added together to produce temporary result sequence t1. In parallel, pairwise elements of sc and sd are added to produce temporary result sequence t2. Pairwise elements of sequences t1 and t2 are multiplied together to form an unnamed sequence which is routed back into the output memory, result. Figure 1 illustrates the architectural configuration generated for program add_and_mult. The program could be expressed more concisely as:

```plaintext
program add_and_mult(a[0:99], b[0:99], c[0:99],
                     d[0:99] : integer memory)
    result[0:99] : integer memory
where
    declare index : integer sequence;
relations
    index = 0 fby index + 1 until index == 99;
    result[index] = (a[index]+b[index])*(c[index]+d[index]);
endwhere;
```

![Figure 1 illustrates the configuration that would be produced for program add_and_mult.](image)

377
III.2 Memories and Data Structures

A memory is a random access storage device that is external to, but accessible by the RSN. The purpose of a memory is to store results produced by one INSIGHT program or activity and used by another. Memories are assumed to have only one port, so that only one element of a single memory can be accessed at a time.

Memories can be used to store more complex data structures than arrays. In particular, linked lists can easily be implemented by using two parallel memories, one for the head of each cell and one for the tail. A list stored in this fashion is accessed through the generation of a sequence of memory addresses; the first is the address of the first cell in the list, and the nth comes from the tail field of the (n-1)st. Corresponding to the sequence of memory addresses is a sequence of data values that constitute the list. Thus, not only can a stream of pixels flow through the RSN, but also streams of list elements can flow through the network. We expect to provide primitives for list handling, so that mid and high-level vision will be as natural in INSIGHT as image processing.

III.3 Sequence Arrays

It will often be the case that the programmer wants to configure part of the RSN as a pipeline of several stages. We provide the notion of sequence arrays as a convenience tool, so that the programmer will not have to repeat any relations that are the same for each stage. A sequence array is an array of sequences declared as a translator time entity that expresses a configuration of several sequences that have some relation to each other. For example, the following INSIGHT code generates the configuration shown in Figure 2. The output of each stage is its input plus one. Instead of having to repeat this relationship three times for three stages, the repetition is accomplished by the foreach construct.

```
declare
  stage: translate integer;
output[0:3]: integer seqarray;
relations
  output[0] = 0;
foreach stage = 1 to 3
  output[stage] = output[stage - 1] + 1;
endfor;
```

In this example, the translator integer stage is used in the translation time foreach loop that defines the elements of the sequence array. The sequence array itself is merely a shorthand notation for defining four related sequences. In this case, the relationship is that the first of the four sequences is the input and the remaining three are outputs of three stages in a pipeline configuration. Sequence arrays can also be used to express sequences whose relationship is parallel.

Figure 2 illustrates an example of a pipeline configuration expressed in INSIGHT with the sequence array notation.

III.4 Sequence Operators

INSIGHT provides a variety of operations for manipulating sequences. The if-then-else expression allows a sequence to be generated, each of whose elements is chosen from one of two input sequences, depending on the value of a boolean expression sequence. The conditional expression allows a sequence to be generated, each of whose elements is chosen from one of many input sequences. This construct is similar to the cond function in LISP. All the standard numeric operators and functions such as +, -, *, /, **, abs, cos, sin, log, etc., are provided, as are all the standard logical operators and the comparative character operators. The interesting operators in INSIGHT are the sequence operators, a set of special sequence generation and manipulation operators. Some of these come from LUCID [2], and others were invented to express constructions that can occur in a configuration of the RSN.

Two examples of INSIGHT sequence operators are defined below. The first operator, fby, comes from LUCID, while the second is LUCID-like, but new to INSIGHT. Traces given to show how the operators work are in process time. INSIGHT operators may be combined into multi-operation expressions.

1) followed by
R = X followed by Y;
R = X fby Y;

explanation:
R's first element will equal the first element of X. The next element of R will be the first element of Y, and so on: Ri = Yi - 1.

trace:
X = <X0 X1 X2 X3 ...>
Y = <Y0 Y1 Y2 Y3 ...>
R = <X0 Y0 Y1 Y2 Y3 ...>

2) as long as
R = X as long as C;
R = X ala C;

explanation:
C is a binary/boolean sequence. Ri = Xi as long as C == 1. When C becomes 0, R becomes eod. If the first value of C is 0, then R0 = eod.
trace:

\[
\begin{align*}
C &= 1 1 1 1 1 1 1 0 1 1 \\
X &= X_0 X_1 X_2 X_3 X_4 X_5 X_6 X_7 X_8 X_9 \\
R &= R_0 R_1 R_2 R_3 R_4 R_5 \text{ end ...}
\end{align*}
\]

IV. Examples

We are in the process of encoding vision algorithms in INSIGHT, to be sure the language is sufficient for our needs. In this section we present two of the simpler vision algorithms as examples.

Convolution

Convolve is an image processing function that illustrates one implementation of a convolution operation. The strategy is to use one stage per pixel in the neighborhood of the convolution. Each stage has an associated delay value, stored in memory delays and an associated coefficient, stored in memory coefficients. The cumulative delays of the input image have the effect of selecting a neighborhood of pixels. The coefficients form the kernel of the convolution.

```c
function convolve (image.in : integer sequence ;
                   delays : integer memory ;
                   coefficients : integer memory)
                 : integer sequence ;

where

declare
  size, stages: translator integer ;
  result[0:size] image[0:size]: integer seqarray ;
relations
  result[0] = coefficient[0] * image.in ;
  image[0] = image.in
foreach stage = 1 to size
  image[stage] = image[stage - 1] dby delays[stage];
  result[stage] = result[stage - 1] +
                  coefficients[stage] * image[stage];
endfor ;
convolve = result [size] ;
endwhere
```

Mapper

The second example is a very simplified form of an order dependent structural shape matching algorithm \([4]\). Two shapes are to be compared and an error of the match computed. Each shape is represented by a list of its parts and their attributes. The parts of each shape are numbered from 1 to max_parts. Function mapper inputs two integer sequences c.first and u.first. Each pair of corresponding elements \((c.first, u.first)\) represent the hypothesis that part number c.first of the first shape corresponds to part number u.first of the second shape. Thus the inputs can be thought of conceptually as a sequence of hypotheses. The three outputs are best.error, whose last value will be the error of the best match, and \((best.c.first, best.u.first)\) whose last values indicate the best hypothesis. Mapper uses a utility function compute.error which is assumed to compute the error between a pair of parts, based on their attributes which are stored in the local memory of each stage, so that parallel access is possible.

```c
function mapper (c.first, u.first : integer sequence)
                     besterr : real sequence; best.c.first,
                     best.u.first : integer sequence

where

declare
  c.part[1:max.parts], u.part[1:max.parts] :
                   integer seqarray;
  error[1:max.parts]: real seqarray;
  best.error: real sequence;
  best.c.first, best.u.first: integer sequence;
  max.parts, stage : translator integer;
relations
  c.part[1] = c.first ;
  u.part[1] = u.first ;
  error[1] = compute.error (c.first, u.first) ;
foreach stage = 2 to max.parts
  c.part[stage] = (c.part[stage-1] 
                     mod max.parts)+1 ;
  u.part[stage] = (u.part[stage-1] 
                     mod max.parts)+1 ;
  error[stage] = error[stage-1] +
                 compute.error(c.part[stage],
                               u.part[stage]));
endfor ;
best.error = MAXINT fby
  if error[max.parts] < best.error
    then error[max.parts]
    else best.error ;
best.c.first = 0 fby
  if error[max.parts] < best.error
    then c.first
    else best.c.first ;
best.u.first = 0 fby
  if error[max.parts] < best.error
    then u.first
    else best.u.first ;
endwhere
```

V. Conclusions

The INSIGHT language allows the expression of low, mid, and high-level vision algorithms in a relational form that can be easily translated to a configuration of a reconfigurable systolic network. The language is still under development and is being tested for utility by encoding a set of representative vision algorithms and simulating their execution. Using the language is, in
fact, teaching us how to write parallel algorithms for machine vision. We expect this research to lead to a fast, general purpose, machine vision system.

References


