ABSTRACT

Image Processing problems are often ideally suited to implementation on massively parallel machines. Providing a machine independent image processing language that can be readily targeted at massively parallel machines can be of great benefit in aiding researchers to use such machines. Such a language can free the user from having to learn the details of directly programming such a complex machine. We discuss the implementation and use of such a language, the AFATL (Air Force Armament Laboratory) image algebra, on a massively parallel machine. We introduce the problem of specifying image processing algorithms in a machine independent way, introduce the image algebra, provide an overview of how image algebra constructs are implemented in Connection Machine *lisp and provide examples of the use of image algebra for a variety of image processing operations. Finally, we discuss the generality, level of portability, and the efficiency of the existing implementation.

INTRODUCTION

Massively parallel computers offer great potential benefits to end users. Sometimes, however, users who could benefit directly from the use of such massively parallel machines are reluctant to use them. In some cases the decision not to use a massively parallel machine is driven by the high cost of such architectures, but in other cases the primary motivation for not using massively parallel machine is a lack of understanding of the computing paradigm on which such architectures are based or a reluctance to tackle the problem of adapting current programs and programming techniques to a radically different architecture. To confuse the issue even further, different massively parallel machines have different architectures, constraints, host systems, and so forth, making the porting of programs prepared for massively parallel machines difficult.

One approach to solving this problem is to develop a general purpose high-level language that expresses massively parallel computations in a machine independent way. This solution suffers from the problem of being architecture class specific. Programs in such a language will not in general express algorithms in a way that easily permits them to be ported to different classes of architectures such as sequential or pipeline machines.

One can overcome this problem of architecture class dependence in a specific application domain by developing a high level application specific language that is architecture class independent. In this paper we discuss implementation and use of such a language, the AFATL (Air Force Armament Laboratory) image algebra [1,2,3], on a massively parallel machine. The architecture independence of this algebra is a result of its formal mathematical structure and method of development. Implementations of the image algebra have been developed for three kinds of processor architectures: sequential, vector parallel, and massively parallel. We discuss a massively parallel implementation in this paper.

In the next section, we introduce the image algebra. We follow that with an overview of a Common Lisp implementation of the image algebra on the CM2 Connection Machine. We then provide examples of the use of this image algebra implementation. We summarize with a discussion of the generality, level of portability, efficiency, and extensibility of the existing implementation.

IMAGE ALGEBRA

The image algebra was designed to provide a mathematical system to support implementation, comparison, and analysis of image processing transformations. A number of architecture specific image processing notations have been developed in recent years [4], but programs developed in such languages are not easily ported to other architectures. Other image processing notations have been based on mathematical models of images [5], but most have not been demonstrated to be capable of describing certain classes of image processing transformations. The AFATL image algebra (henceforth referred to simply as image algebra) was developed with the intent of avoiding these pitfalls. The algebra was developed after carrying out a study of over 200 image processing algorithms. The operands and operations employed in these algorithms were analyzed to determine the fundamental components upon which they are built. The image algebra was then developed by extracting the relevant components discovered during that study.

The image algebra has been shown to be sufficient to express all image transformations over images with finitely many gray-values [6] as well as all image transformations described by finite program schemes [7]. Informally stated, these results mean that the image algebra can express any usable image transformation. Of greater interest, however, is the algebra's coherent set of operands and operations and the ease with which the algebra can be used to describe image
transformations. In the rest of this section we describe the image algebra briefly, concentrating on those components most relevant to our discussion of massively parallel implementation of the algebra.

An algebra consists of a set of operands together with a set of finitary operations on those operands. We briefly describe the operands and operations comprising the image algebra in the following paragraphs. In the limited space provided, we are unable to give a complete presentation of the image algebra. The interested reader should refer to [2] for a more complete discussion.

The operands of the image algebra are values and values sets, coordinates and coordinate sets, images, and generalized templates. Informally speaking, a value set can be any set that might be used for image pixel values such as the reals (R), integers (Z), binary representations of subsets of the integers, and so forth. The coordinate sets of the image algebra are subsets of R^n, that is, they are subsets of real n-space. In implementations of the image algebra, we restrict our attention to finite coordinate sets such as a rectangular subset of integral cartesian space Z^2. We use symbols X and Y to represent coordinate sets, with elements x and y respectively. An image is simply the graph of a function from a coordinate set into a value set. The set of all real valued images over coordinate set X is denoted RX.

The fundamental operations on images in RX are, for the most part, pixelwise induced operations on the reals. That is, functions f:R→R and g:R×R→R induce similar functions RX→RX and RX×RX→RX, also denoted by f and g, and defined by

\[ f(a) = \{(x,c(x)): c(x) = f(a(x))\} \]
\[ g(a,b) = \{(x,c(x)): c(x) = g(a(x),b(x))\} \]

For example, \( \sin(a) = \{(x,\sin(a(x))): x \in X \} \), and \( a*b = \{(x,a(x)*b(x)): x \in X\} \).

In addition to these pixel-wise operations on images, several unary operations on images and image dot product serve to map images into real values. The image sum operation, \( \Sigma \), is defined on image \( a \in RX \) as

\[ \Sigma a = \Sigma \sigma X a(x) \]

Besides these unary and binary operations on images, the Image Algebra supports what are called generalized template operations. Each of these operations takes an image and a generalized template. The generalized templates formalize and extend the concept of mask or template entities used in neighborhood image processing algorithms.

Roughly speaking, if one wants to compute a generalized template operation on a real-valued image over coordinate set \( X \), giving as its result a real-valued image over coordinate set \( Y \), one uses a real-value template in the set \((RX)^Y\) (the set of functions from coordinate locations in \( Y \) into real-valued images in \( X \). That is, if template \( t \) is an element of \((RX)^Y\), then if \( y \in Y \), \( t(y) \in RX \), that is, \( t(y) \) is an image in \( X \). For convenience sake, we write \( t_y \) to mean \( t(y) \). The image assigned to a result location is used to weight each of the values in the source image and then gather those values together to yield the result image value at that location. The weighting and gathering of values is specified by an operation taking an image and template as its operands and yielding an image result.

Although there are three fundamental image template operations in the image algebra, we consider only one of them, generalized convolution, in this paper. The other image-template operations provided by the image algebra are multiplicative maximum and additive maximum. Additive maximum generalizes the gray-scale morphology to non-rigid structuring elements [8], and multiplicative maximum provides a pseudo-linear operation with characteristics that are currently under exploration.

Given an image \( a \) on coordinate set \( X \), and a template \( t \in (RX)^Y \), we define the generalized convolution of \( a \) with \( t \), written \( a \circ t \), as follows:

\[ a \circ t = \sigma \text{ where } c \in \text{R}^Y \text{ and for all } y \in Y, c(y) = \Sigma (a*t_y) \]

In Figure 1, we show an image \( a \), that will be convolved with template \( t \) to yield a result image \( e \), that is, \( e = a \circ t \).

Figure 2 shows how the generalized convolution operation is applied to determine the value of \( e \) at the pixel location \( y_1 \). First one obtains the image \( t_{y_1} \), then one multiplies this image pointwise with the source image \( a \). The sum of the pixel values in this image are added together, the result providing the value of \( e_{y_1} \).

![Figure 1](image1.png)

![Figure 2](image2.png)
mapping each result image location into its source support. In this sense, then template operations of the image algebra are backward driven, since the result pixel location is first mapped to a support, then an operation involving the support and the image operand is performed to yield the result pixel value.

Many different formulations of neighborhood operations have been developed and implemented in computer systems. Such formulations may provide an operation similar to the image algebra image-template operation but restricted in some sense. One type of restriction is so prevalent we have given it a special name. A template \( t \) is said to be translation invariant if given \( x, y, \) and \( z \) satisfying \( x + z \in X \) and \( y, y + z \in Y \) we have \( t_y(x) = t_{y+z}(x+z) \), i.e. when translating the result location argument of the template, the template image yielded is identical (up to the same translation). Translation invariance is not required of image algebra templates. As shown in Figure 3, the example template \( t \) of Figures 1 and 2 may assign to point \( y_2 \) an image completely different from that assigned to point \( y_1 \).

One important generalization of templates is to provide for the definition of families of templates where each particular element can be distinguished by parameter values. If \( P \) is a set of parameter values, a real-valued parameterized template \( t \) with parameters in \( P \) is a function of the form

\[
t: P \rightarrow \mathbb{R}^S
\]

The parameterized template maps its parameter into a normal template. The ability to define parameterized templates is of great benefit in describing complex image processing and vision algorithms. We discuss parameterized template definitions further in the next section.

The subalgebra of the image algebra providing only the operations of generalized convolution, image multiplication, and image sum is isomorphic to linear algebra \([8]\). Removal of the requirement for translation invariance provides the capability to express non-linear transformations such as warpings with image-template operations. The conceptual power of such template operations comes from the distinction drawn between the template operation, e.g. generalized convolution, and the assignment of images to result locations by the template.

**Figure 3.**

One important generalization of templates is to provide for the definition of families of templates where each particular element can be distinguished by parameter values. If \( P \) is a set of parameter values, a real-valued parameterized template \( t \) with parameters in \( P \) is a function of the form

\[
t: P \rightarrow \mathbb{R}^S
\]

The parameterized template maps its parameter into a normal template. The ability to define parameterized templates is of great benefit in describing complex image processing and vision algorithms. We discuss parameterized template definitions further in the next section.

**MASSIVELY PARALLEL IMPLEMENTATION**

The image algebra described briefly in the preceding section, while not in any way limited to implementation on massively parallel processors, can easily exploit the power of such computer architectures. The obvious paradigm for parallel implementation of the image algebra is to equate coordinates in coordinate sets with processors in the machine architecture. Unary and binary image operations can then be implemented in a SIMD fashion. The implementation of image-template operations, on the other hand, is somewhat more complicated.

Let us consider a model for implementing the expression \( a \circ t \). Our paradigm above indicates that each coordinate, whether in the source or result coordinate set, will be associated with some processor. As noted in the previous section, the template operations are backward driven, so we will consider the set of result processors as controlling the template operation. Consider the processor associated with result location \( y \). This processor will need to compute the support of \( t \) at \( y \), \( S(t_y) \), then for each location \( x \) in \( S(t_y) \) it must compute \( a(x) \circ t_y(x) \) and sum these values to determine the result value at \( y \). If template \( t \) is translation invariant then such an operation is clearly SIMD. If \( t \) is more complex then SIMD implementation becomes more difficult.

We now discuss a language interface for providing the image algebra to users. Let us first note that the image algebra is not a programming language. In comparison to scalar arithmetic, one can consider the image algebra to be the arithmetic of images. We have implemented a subset of the image algebra as an extension to Common Lisp \([10]\). Our massively parallel implementation of the image algebra has been conducted on a CM2 Connection Machine using *lisp \([11]\). This provides us with several key benefits: we have the benefit of working in the rich development environment provided by Common Lisp; and all image algebra constructs described in this paper have been able to undergo parallel development in a uniprocessor implementation carried out in Kyoto Common Lisp on various Sun workstations. This exemplifies the architecture independence of the image algebra. Aside from execution speed, there is no way of distinguishing what underlying architecture the image algebra implementor is using.

Our implementation has undergone several stages and is continuing to evolve. In the current implementation a seamless extension of Common Lisp is provided by execution of an image algebra read function. The user interface provided by this function can distinguish between forms that employ image algebra constructs and execute appropriate code to implement those image algebra operations. The new data type image is provided by this interface. While images are represented in the CM2 implementation as *lisp pairs, the user of the image algebra need not be concerned with this detail. Common Lisp arithmetic forms are extended to have appropriate meanings when operating on images. Image algebra operations with no analogues in Common Lisp are provided via defined functions. It is interesting to note that many of these functions are already directly available in *lisp, indicative of the close relationship between the Connection Machines model of computation and the image algebra. Parameterized templates are defined with a form analogous to defun and are used in image-template operations. As implementation of unary and binary image operations is relatively straightforward, we will concentrate in the rest of this section on definition and use of templates.
As noted in the previous section, a parameterized template is a function mapping a parameter value into a template, which is a function from coordinates into images. One can, however, more compactly specify a template by considering only the support (non-zero portion) of the image it yields. This view leads to a significantly more efficient implementation and permits more compact specification of templates. Parameterized templates are declared much like Common Lisp functions with the special form deftemplate. The deftemplate form specifies the template name, its parameter names, the result location coordinates, and a template body which constructs the support of the template when given bindings for parameter values and result coordinate location. The special form weight is evaluated to yield a single pixel in the support. The forms that may appear in a template body under our current image algebra implementation are limited. Scalar arithmetic functions may be used, as well as predicate functions, do, if, let, let*, when, and weight. The syntax for template definitions is given below:

```lisp
deftemplate template ((var)*) 
  [ (coordinates (var)* ) ]
body
```

The parameter names for the template are contained in the list following the template name, this is followed by an optional coordinate specification giving a sequence of names corresponding to the result location's coordinates, and the template body containing forms to assign weights to the support follows.

Templates are used in the context of image-template operations. The generalized convolution operation, gcon, has the following form:

```lisp
(gcon image (template (parameter)*))
```

Note that the template argument is evaluated with its parameters, indicating that the specific template associated with the parameter values has been computed. The weight special form has the following form:

```lisp
(weight expression (expression)* )
```

and assigns the value of first expression to the source image support location addressed by the coordinates given in the list of expressions.

Perhaps a simple example will help illustrate the use of deftemplate. Consider a translation invariant template with no parameters whose support contains the result pixel location, its immediate right neighbor, and its immediate left neighbor, each with weight 1/3. Such a template can be defined as follows:

```lisp
(let* ((w 1/3))
  (deftemplate smooth ()
    (coordinates (y0 y1))
    (weight 1/3 (y0 + y1))
    (weight 1/3 ((1+ y0) y1))
    (weight 1/3 ((1- y0) y1)))
)
```

Note that this template has an empty parameter list. Hence there is only one template in the family described by this template definition. Note also the coordinate specification naming the horizontal and vertical coordinates in the result image coordinate set. These pseudo-variables, y0 and y1, will take on values corresponding to result image coordinates when the template is evaluated. In this simple example, the first weight form assigns weight 1/3 to the source coordinate point in the same location as the result coordinate. The second and third weight forms assign 1/3 to the source coordinate points corresponding to the nearest right and left neighbor locations to the first support location. This template might be used in a generalized convolution operation to horizontally smooth an image as follows:

```lisp
(gcon a (smooth))
```

The user interface presents image-template operations and templates in a fashion that is completely analogous to their image algebra definitions. The implementation of image-template operations on the CM2 is not, however, directly analogous to this interface. Let us look at how the generalized convolution of an image and template is implemented. Consider the evaluation of:

```lisp
(gcon image (template arg))
```

In our CM2 implementation, template will have been declared by evaluating a deftemplate form. Deftemplate is a macro yielding a defun form as its result. This function takes the following parameters:

1. any parameters declared in the deftemplate specification,
2. parameters specifying with image-template operation is to be performed, and
3. an image.

Evaluation of the gcon form will dispatch the template function with the appropriate parameter values. In this case, the dispatched function will return a single pixel in nature. Consider the fact that do forms with initial and exit specifiers depending on the result location may be specified. This means that the number of elements in a template's support may vary from result location to result location. This has required the implementation of a general parallel do loop structure. The final example in the section following uses precisely such a template.

**EXAMPLES OF USE OF IMAGE ALGEBRA ON THE CM**

In this section we provide a few simple examples of the definition of templates in the Common Lisp image algebra notation and the use of these templates with the generalized convolution operation. It must be noted that the image algebra consists of much more than just a few convolution operations and
complete algorithms for such applications as tracking and identification of objects can be implemented using the image algebra.

The first example we present is the smoothing template of the last section. The source template is as follows:

```lisp
deftemplate smooth ()
  (coordinates (yo yi))
  (weight 1/3 (yo yi))
  (weight 1/3 ((1+ yo) yi))
  (weight 1/3 ((1-yo) yi))
```

The actual *lisp code generated for this template is listed here.

```lisp
(*DEFUN SMOOTH (#:SMOOTH-SOURCE-IMAGE-13
 #:SMOOTH-SOURCE-WEIGHTING-OP-135
 #:SMOOTH-ACCUMULATION-OP-136
 #:SMOOTH-OFF-SOURCE-RETURN-VALUE-134)
 (*ALL
  (*LET
   ((#:SMOOTH-TARGET-IMAGE-132
     #:SMOOTH-OFF-SOURCE-RETURN-VALUE-134))
    (X (SELF-ADDRESS-GRID!! (!! 0)))
    (Y (SELF-ADDRESS-GRID!! (!! 1))))
  (*SET #:SMOOTH-TARGET-IMAGE-132
    (#:SMOOTH-SOURCE-IMAGE-133
     #:SMOOTH-SOURCE-WEIGHTING-OP-135
     #:SMOOTH-OFF-SOURCE-RETURN-VALUE-134)))))
```

The second example demonstrates a translation variant template in which the support may be empty for some target pixel locations. The template we present can be used to cause all pixels within a given rectangular region in an image to take on the value 0. The rectangular region to be set to zero is specified by template parameters.

Figure 4 shows an image of an sr71 airplane, sr71. Figure 5 shows the result of evaluating the expression

```
gcon sr71 (smooth)
```
Note that when evaluating the result of a generalized convolution operation if the template support is empty at some location \( y \), then the result value at \( y \) will be 0. Any processor not executing a \texttt{weight} operation in evaluating a template will have empty support and hence be assigned this value. The matte template yields an empty support at each location inside the specified rectangle. Figure 6 shows the result of evaluating the expression:

\[
\text{(gcon sr71 (matte 32 64 64 96))}
\]

Figure 6.

The third example shows the definition of a template that can be used to achieve rotation of a source image. We demonstrate this technique with a simple nearest-neighbor rotation in which the nearest point to the ideal rotation source is chosen and placed in the target image. A more appropriate and physically accurate interpolation of the source gray-value could be described by suitable modification of the template definition. The template definition is as follows:

\[
\text{(deftemplate rotate (i j theta))}
\]

\[
\text{(coordinates (yo y1))}
\]

\[
\text{(weight 1 (+ (* (- yo i) (cos theta))
(- y1 j) (sin theta)))}
\]

The definition of the rotate template assigns to each result location a single support pixel which is the nearest neighbor to the inverse of the rotation specified. This, once again, brings up the backward-driven nature of the template definition. The template tells where to find the source values associated with a result location. Other warpings, such as projective transformations can be specified with template operations in a similar manner.

Figure 7 shows the result of executing the following expression:

\[
\text{(gcon sr71 (rotate 54 64 0.436))}
\]

Figure 7.

Our final example demonstrates the versatility of templates by providing a translation variant shrinking template definition. The support at different points in this template have different positions, weights, and numbers of elements, as determined from the template arguments. Each result pixel \( y \) is assigned a support containing the location \( y^2 \), where \( p \) is specified as a parameter to the template. The template definition is as follows:

\[
\text{(deftemplate power-shrink-y (power))}
\]

\[
\text{(coordinates (x y))}
\]

\[
\text{(if (\text{= y 0})

\text{(do ((i (expt (- y 0.5) power)
(1+ i)))

\text{((> i (expt (+ y 0.5) power)))

\text{(weight 1 (x (expt y power))))))))}
\]

The *lisp code generated for this template is shown here.

592
Note that this code has implemented the MIMD do of the template by iterating enough times to cover the range \( (\exp (-y 0.5) \text{power}) \) to \( (\exp (+y 0.5) \text{power}) \) for any processor with address \( y \). Figure 8 shows a simple line drawing of a skull, and Figure 9 contains the result of application of the following expression involving that image:

\[
gcon \text{skull} (\text{power-shrink-y 1.1})
\]

**SUMMARY**

As noted in the introduction, one can make massively parallel computers accessible to researchers in a specific application area by providing a domain specific user interface that will execute efficiently on such machines without requiring knowledge of the underlying architecture. In the case of the image algebra, this has been achieved by carefully choosing a set of operands and operations that, while capable of expressing all image processing transformations, is not dependent on the particular attributes of any special computer architecture or class of architectures. At the same time, the image algebra is clearly well suited to implementation on massively parallel machines such as the CM2.

The particular choice of Common Lisp for this implementation has provided great portability for user programs. The image algebra has been as easy to implement on uniprocessors as on massively parallel machines. The only differences discernible to the user are the dramatic difference in speed of execution of the developed algorithms and the different behaviors of systems when error conditions arise.
The current Connection Machine implementation, while relatively efficient, can be improved in several ways. Many of the constants appearing in template bodies could be assigned to pvars using *let. Templates could be implemented as functions that generate optimized versions of themselves when dispatched from particular image-template operations. A variety of data flow analysis techniques and peephole optimizations could be used to improve the code generated for templates. One example of where data flow analysis might pay off is in the substitution of front end control flow for distributed control and termination on execution of do's with constant bounds. Another is in combining the weighting and combining operations into single *lisp operations, rather than sequences of operations.

Despite the rudimentary nature of the current implementation, the results we have seen are quite promising. Interpreted execution speed of image algebra code compares favorably with hand coded versions of similar algorithms and the implementation can be easily modified to add further functionality. Work continues on expanding the subset of the image algebra currently supported.

REFERENCES