

# **Low-Level Vision on Warp and the Apply Programming Model**

**Leonard G. C. Hamey, Jon A. Webb, and I-Chen Wu**

**CMU-RI-TR-87-17**

**The Robotics Institute  
Carnegie Mellon University  
Pittsburgh, Pennsylvania 15213**

**July 1987**

**Copyright © 1987 Carnegie Mellon University**

**This research was supported in part by the Defense Advanced Research Projects Agency (Dod), monitored by the Air Force Avionics Laboratory under Contract F33615-81-K-1539, and Naval Electronic Systems Command under Contract N00039-85-C-0134, in part by the U.S. Army Engineer Topographic Laboratories under Contract DACA76-85-C-0002, and in part by the Office of Naval Research under Contracts N00014-80-C-0236, NR048-659, and N00014-85-K-0152, NR SDRJ-007.**

---

**A version of this paper will appear in *Parallel Computation and Computers for Artificial Intelligence*, edited by Janusz Kowalik, Boston: Kluwer Academic Publishers, 1987.**

## Table of Contents

<b>1 Introduction</b>	<b>1</b>
<b>1.1 Warp Overview</b>	<b>2</b>
<b>2 Low-level vision on Warp</b>	<b>2</b>
<b>3 Introduction to Apply</b>	<b>4</b>
<b>3.1 The Apply Language</b>	<b>5</b>
<b>3.2 An Implementation of Sobel Edge Detection</b>	<b>7</b>
<b>3.3 Border Handling</b>	<b>8</b>
<b>3.4 Image Reduction and Magnification</b>	<b>8</b>
<b>3.5 Multi-function Apply Modules</b>	<b>9</b>
<b>4 Apply on Warp</b>	<b>12</b>
<b>5 Apply on Uni-processor Machines</b>	<b>12</b>
<b>6 Apply on the Hughes HBA</b>	<b>14</b>
<b>7 Apply on Other Machines</b>	<b>14</b>
<b>7.1 Apply on bit-serial processor arrays</b>	<b>14</b>
<b>7.2 Apply on distributed memory general purpose machines</b>	<b>15</b>
<b>8 Summary</b>	<b>15</b>
<b>9 Acknowledgments</b>	<b>16</b>
<b>10 References</b>	<b>16</b>

**List of Figures**

<b>Figure 1: Input Partitioning Method on Warp</b>	<b>3</b>
<b>Figure 2: Grammar of the Apply language</b>	<b>7</b>
<b>Figure 3: The Sobel Convolution Masks</b>	<b>7</b>
<b>Figure 4: An Apply Implementation of Thresholded Sobel Edge Detection</b>	<b>8</b>
<b>Figure 5: Image buffering for Apply</b>	<b>13</b>

## Abstract

In the course of implementing low-level (image to image) vision algorithms on Warp, we have understood the mapping of this class of algorithms well enough so that the programming of these algorithms is now a straightforward and stereotypical task. The partitioning method used is input partitioning, which provides an efficient, natural implementation of this class of algorithms. We have developed a special programming language called Apply, which reduces the problem of writing the algorithm for this class of programs to the task of writing the function to be applied to a window around a single pixel. Apply provides a method for programming Warp in these applications which is easy, consistent, and efficient. Apply is application specific, but machine independent—it is possible to implement versions of Apply which run efficiently on a wide variety of computers. We describe implementations of Apply on Warp, UNIX and the Hughes HBA, and sketch implementation on bit-serial processor arrays and distributed memory machines.

## 1 Introduction

In computer vision, the first, and often most time-consuming, step in image processing is *image to image* operations. In this step, an input image is mapped into an output image through some local operation that applies to a window around each pixel of the input image. Algorithms that fall into this class include: edge detection, smoothing, convolutions in general, contrast enhancement, color transformations, and thresholding. Collectively, we call these operations low-level vision. Low-level vision is often time consuming simply because images are quite large – a typical size is  $512 \times 512$  pixels, so the operation must be applied 262,144 times.

Fortunately, this step in image processing is easy to speed up, through the use of parallelism. The operation applied at every point in the image is often independent from point to point, and also does not vary much in execution time at different points in the image. This is because at this stage of image processing, nothing has been done to differentiate one area of the image from another, so that all areas are processed in the same way. Because of these two characteristics, many parallel computers achieve good efficiency in these algorithms, through the use of *input partitioning* [12].

We define a language, called *Apply*, which is designed for implementing these algorithms. *Apply* runs on the Warp machine, which has been developed for image and signal processing. We discuss Warp, and describe its use at this level of vision. The same *Apply* program can be compiled either to run on the Warp machine, or under UNIX, and it runs with good efficiency in both cases. Therefore, the programmer is not limited to developing his programs just on Warp, although they run much faster (typically 100 times faster) there; he can do development under the more generally available UNIX system.

We consider *Apply* and its implementation on Warp to be a significant development for image processing on parallel computers in general. The most critical problem in developing new parallel computer architectures is a lack of software which efficiently uses parallelism. While building powerful new computer architectures is becoming easier because of the availability of custom VLSI and powerful off-the-shelf components, programming these architectures is difficult.

Parallel architectures are difficult to program because it is not yet understood how to “cover” parallelism (hide it from the programmer) and get good performance. Therefore, the programmer either programs the computer in a specialized language which exploits features of the particular computer, and which can run on no other computer (except in simulation), or he uses a general purpose language, such as FORTRAN, which runs on many computers but which has additions that make it possible to program the computer efficiently. In either case, using these special features is necessary to get good performance from the computer. However, exploiting these features requires training, limits the programs to run on one or at most a limited class of computers, and limits the lifetime of a program, since eventually it must be modified to take advantage of new features provided in a new architecture. Therefore, the programmer faces a dilemma: he must either ignore (if possible) the special features of his computer, limiting performance, or he must reduce the understandability, generality, and lifetime of his program.

It is the thesis of *Apply* that *application dependence*, in particular *programming model dependence*, can be exploited to cover this parallelism while getting good performance from a parallel machine. Moreover, because of the application dependence of the language, it is possible to provide facilities that make it easier for the programmer to write his program, even as compared with a general-purpose language. *Apply* was originally developed as a tool for writing image processing programs on UNIX systems; it now runs on UNIX systems, Warp, and the Hughes HBA. Since we include a definition of *Apply* as it runs on Warp, and because most parallel computers support input partitioning, it should be possible to implement it on other supercomputers and parallel computers as well.

*Apply* also has implications for benchmarking of new image processing computers. Currently, it is hard to

compare these computers, because they all run different, incompatible languages and operating systems, so the same program cannot be tested on different computers. Once Apply is implemented on a computer, it is possible to fairly test its performance on an important class of image operations, namely low-level vision.

Apply is not a panacea for these problems; it is an application-specific, machine-independent, language. Since it is based on input partitioning, it cannot generate programs which use pipelining, and it cannot be used for global vision algorithms [11] such as connected components, Hough transform, FFT, and histogram.

We begin by reviewing the structure of the Warp machine, and then discuss our early work on low-level vision, where we developed the input partitioning method on Warp. Then we define and discuss Apply. Following this, we describe how Apply might be implemented on other computers.

### 1.1 Warp Overview

This is a brief overview of Warp; more detail is available elsewhere [2]. Warp has three components—the Warp processor array (*Warp array*), the interface unit (*IU*), and the *host*. The Warp array performs the computation-intensive routines, for example, low-level vision routines. The IU handles the input/output between the array and the host, and generates addresses and control signals for the Warp array. The host executes the parts of the application programs that are not mapped onto the Warp array and supplies the data to and receives the results from the array.

The Warp array is a linear array of ten cells, called Warp cells, which are identical and which include local data and microcode memory, input and output ports, and a 5 MFLOPS ALU and 5 MFLOPS multiplier, for a total of 10 MFLOPS per cell. The Warp array therefore has 100 MFLOPS peak power.

The Warp programming environment is based on Common Lisp. A compiler, debugger, and execution environment are included. The programming language, called W2, is approximately at the level of PASCAL. Data structures such as arrays and scalars are included. Control structures include IF, WHILE, and FOR. The compiler hides from the programmer all the parallelism in the Warp machine except for the parallel execution of the Warp cells themselves. Communication between cells is implemented using SEND and RECEIVE, which transfer words between adjacent cells using an asynchronous protocol. The debugger allows single stepping and source-level breakpoints, and allows the programmer to examine data structures within the Warp array. The execution environment manages the microcode and programs for the stand-alone processors, and aids the programmer in managing the memory of the external host.

Warp is integrated into the vision programming environment at Carnegie Mellon. Vision programming is based on the Generalized Image Library [6] which supports uniform access to images in files, frame buffers, memory, and printers. Presently, most vision programming is done in C/UNIX, using Suns and Vaxes; we are presently moving to a Sun/Warp/Common Lisp based environment.

## 2 Low-level vision on Warp

We map low-level vision algorithms onto Warp by the *input partitioning* method. On a Warp array of ten cells, the image is divided into ten regions, by column, as shown in Figure 1. This gives each cell a tall, narrow region to process; for  $512 \times 512$  image processing, the region size is 52 columns by 512 rows. To use technical terms from weaving, the Warp cells are the “warp” of the processing; the “weft” is the rows of the image as it passes through the Warp array.

The image is divided in this way using a series of macros called GETROW, PUTROW, and COMPUTEROW.

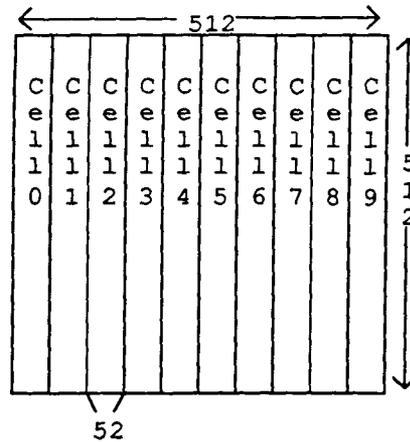


Figure 1: Input Partitioning Method on Warp

GETROW generates code that takes a row of an image from the external host, and distributes one-tenth of it to each of ten cells. The programmer includes a GETROW macro at the point in his program where he wants to obtain a row of the image; after the execution of the macro, a buffer in the internal cell memory has the data from the image row.

The GETROW macro works as follows. The external host sends in the image rows as a packed array of bytes—for a 512-byte wide image, this array consists of 128 32-bit words. These words are unpacked and converted to floating point numbers in the interface unit. The 512 32-bit floating point numbers resulting from this operation are fed in sequence to the first cell of the Warp array. This cell takes one-tenth of the numbers, removing them from the stream, and passes through the rest to the next cell. The first cell then adds a number of zeroes to replace the data it has removed, so that the number of data received and sent are equal.

This process is repeated in each cell. In this way, each cell obtains one-tenth of the data from a row of the image. As the program is executed, and the process is repeated for all rows of the image, each cell sees an adjacent set of columns of the image, as shown in Figure 1.

We have omitted certain details of GETROW—for example, usually the image row size is not an exact multiple of ten. In this case, the GETROW macro pads the row equally on both sides by having the interface unit generate an appropriate number of zeroes on either side of the image row. Also, usually the area of the image each cell must see to generate its outputs overlaps with the next cell's area. In this case, the cell copies some of the data it receives to the next cell. All this code is automatically generated by GETROW.

PUTROW, the corresponding macro for output, takes a buffer of one-tenth of the row length from each cell and combines them by concatenation. The output row starts as a buffer of 512 zeroes generated by the interface unit. The first cell discards the first one-tenth of these and adds its own data to the end. The second cell does the same, adding its data after the first. When the buffer leaves the last cell, all the zeroes have been discarded and the first cell's data has reached the beginning of the buffer. The interface unit then converts the floating point numbers in the buffer to zeroes and outputs it to the external host, which receives an array of 512 bytes packed into 128 32-bit words. As with GETROW, PUTROW handles image buffers that are not multiples of ten, this time by discarding data on both sides of the buffer before the buffer is sent to the interface unit by the last cell.

During GETROW, no computation is performed; the same applies to PUTROW. Warp's horizontal microword, however, allows input, computation, and output at the same time. COMPUTEROW implements this. Ignoring the complications mentioned above, COMPUTEROW consists of three loops. In the first loop, the data for the cell is read into a memory buffer from the previous cell, as in GETROW, and at the same time the first one-tenth of the output

buffer is discarded, as in `PUTROW`. In the second loop, nine-tenths of the input row is passed through to the next cell, as in `GETROW`; at the same time, nine-tenths of the output buffer is passed through, as in `PUTROW`. This loop is unwound by `COMPUTEROW` so that for every 9 inputs and outputs passed through, one output of this cell is computed. In the third loop, the outputs computed in the second loop are passed on to the next cell, as in `PUTROW`.

There are several advantages to this approach to input partitioning:

- Work on the external host is kept to a minimum. In the Warp machine, the external host tends to be a bottleneck in many algorithms; in the prototype machines, the external host's actual data rate to the array is only about  $1/4^{\text{th}}$  of the maximum rate the Warp machine can handle, even if the interface unit unpacks data as it arrives. Using this input partitioning model, the external host need not unpack and repack bytes, which it would have to if the data was requested in another order. On the production Warp machine, the same concern applies; these machines have DMA, which also requires a regular addressing pattern.
- Each cell sees a connected set of columns of the image, which are one-tenth of the total columns in a row. Processing adjacent columns is an advantage since many vision algorithms (e.g., median filter [8]) can use the result from a previous set of columns to speed up the computation at the next set of columns to the right.
- Memory requirements at a cell are minimized, since each cell must store only  $1/10^{\text{th}}$  of a row. This is important in the prototype Warp machines, since they have only 4K words memory on each cell.
- The image is processed in raster order, which has for a long time been a popular order for accessing data in an image. This means that many efficient algorithms, which have been developed for raster-order image processing, can be used.
- An unexpected side effect of this programming model was that it made it easier to debug the hardware in the Warp machine. If some portion of a Warp cell is not working, but the communication and microsequencing portions are, then the output from a given cell will be wrong, but it will keep its proper position in the image. This means that the error will be extremely evident—typically a black stripe is generated in the corresponding position in the image. It is quite easy to infer from such an image which cell is broken!

### 3 Introduction to Apply

The Apply programming model is a special-purpose programming approach which simplifies the programming task by making explicit the parallelism of low-level vision algorithms. We have developed a special-purpose programming language called the Apply language which embodies this parallel programming approach. When using the Apply language, the programmer writes a procedure which defines the operation to be applied at a particular pixel location. The procedure conforms to the following programming model:

- It accepts a window or a pixel from each input image.
- It performs arbitrary computation, usually without side-effects.
- It returns a pixel value for each output image.

The Apply compiler converts the simple procedure into an implementation which can be run efficiently on Warp, or on a uni-processor machine in C under UNIX.

The idea of the Apply programming model grew out of a desire for efficiency combined with ease of programming for a useful class of low-level vision operations. In our environment, image data is usually stored in disk files and accessed through a library interface. This introduces considerable overhead in accessing individual pixels so algorithms are often written to process an entire row at a time. While buffering rows improves the speed of algorithms, it also increases their complexity. A C language subroutine implementation of Apply was developed as a way to hide the complexities of data buffering from the programmer while still providing the efficiency benefits. In fact, the buffering methods which we developed were more efficient than those which would otherwise

be used, with the result that apply implementations of algorithms were faster than previous implementations.

After implementing Apply, the following additional advantages became evident.

- The Apply programming model concentrates programming effort on the actual computation to be performed instead of the looping in which it is imbedded. This encourages programmers to use more efficient implementations of their algorithms. For example, a Sobel program gained a factor of four in speed when it was reimplemented with Apply. This speedup primarily resulted from explicitly coding the convolutions. The resulting code is more comprehensible than the earlier implementation.
- Apply programs are easier to write, easier to debug, more comprehensible and more likely to work correctly the first time. A major benefit of Apply is that it greatly reduces programming time and effort for a very useful class of vision algorithms. The resulting programs are also faster than the programmer would probably otherwise achieve.

### 3.1 The Apply Language

The Apply language is designed for programming image to image computations where the pixels of the output images can be computed from corresponding rectangular windows of the input images. The essential feature of the language is that each operation is written as a procedure for a single pixel position. The Apply compiler generates a program which executes the procedure over an entire image. No ordering constraints are provided for in the language, allowing the compiler complete freedom in dividing the computation among processors.

Each procedure has a parameter list containing parameters of any of the following types: *in*, *out* or *constant*. Input parameters are either scalar variables or two-dimensional arrays. A scalar input variable represents the pixel value of an input image at the current processing co-ordinates. A two-dimensional array input variable represents a window of an input image. Element (0,0) of the array corresponds to the current processing co-ordinates.

Output parameters are scalar variables. Each output variable represents the pixel value of an output image. The final value of an output variable is stored in the output image at the current processing co-ordinates.

Constant parameters may be scalars, vectors or two-dimensional arrays. They represent precomputed constants which are made available for use by the procedure. For example, a convolution program would use a constant array for the convolution mask.

The reserved variables ROW and COL are defined to contain the image co-ordinates of the current processing location. This is useful for algorithms which are dependent in a limited way on the image co-ordinates.

Figure 2 is a grammar of the Apply language. The syntax of Apply is based on Ada [1]; we chose this syntax because it is familiar and adequate, and because we do not wish to create yet another new language syntax, nor do we consider language syntax to be an interesting research issue. However, as should be clear, the application dependence of Apply means that it is not an Ada subset, nor is it intended to evolve into such a subset.

Apply does not allow assignment of fixed expressions to floating variables or floating expressions to fixed variables. Expressions mixing fixed and floating values are also disallowed. A fixed expression may be explicitly converted to float by means of the pseudo-function FLOAT and a floating expression can be converted to fixed by using the pseudo-function INTEGER.

```

procedure ::= PROCEDURE function-name ( function-args )
              IS
                variable-declarations
              BEGIN
                statements
              END function-name;

function-args ::= function-argument [ , function-argument ]*

function-argument ::= var-list : IN type
                       [ BORDER const-expr ]
                       [ SAMPLE ( integer-list ) ]
                       | var-list : OUT type
                       | var-list : CONST type

var-list ::= variable [ , variable ]*

integer-list ::= integer [ , integer ]*

variable-declarations ::= [ var-list : type ; ]*

type ::= ARRAY ( range [ , range ]+ ) OF elementary-type
        | elementary-type

range ::= int-expr .. int-expr

elementary-type ::= sign object
                   | object

sign ::= SIGNED
          | UNSIGNED
          | Empty

object ::= BYTE
          | INTEGER
          | FLOAT

statements ::= statement [ ; statement ]*

statement ::= assignment-stmt
              | if-stmt
              | for-stmt
              | while-stmt

assignment-stmt ::= scalar-var := expr

scalar-var ::= variable
              | variable ( subscript-list )

subscript-list ::= int-expr [ , int-expr ]*

expr ::= expr + expr
         | expr - expr
         | expr * expr
         | expr / expr
         | ( expr )
         | pseudo-function ( expr )
         | variable ( subscript-list )

```

```

if-stmt      ::=  IF bool-expr THEN
                  statements
                  END IF
              |  IF bool-expr THEN
                  statements
                  ELSE
                  statements
                  END IF

bool-expr    ::=  bool-expr AND bool-expr
              |  bool-expr OR bool-expr
              |  NOT bool-expr
              |  ( bool-expr )
              |  expr < expr
              |  expr <= expr
              |  expr = expr
              |  expr >= expr
              |  expr > expr
              |  expr /= expr

for-stmt     ::=  FOR int-var IN range LOOP
                  statements
                  END LOOP

while-stmt   ::=  WHILE bool-expr LOOP
                  statements
                  END LOOP

```

Figure 2: Grammar of the Apply language

Variable names are alpha-numeric strings of arbitrary length, commencing with an alphabetic character. Case is not significant, except in the preprocessing stage which is implemented by the *m4* macro processor [10].

BYTE, INTEGER, and FLOAT refer to (at least) 8-bit integers, 16-bit integers, and 32-bit floating point numbers. BYTE values are converted implicitly to INTEGER within computations. The actual size of the type may be larger, at the discretion of the implementor.

### 3.2 An Implementation of Sobel Edge Detection

As a simple example of the use of Apply, let us consider the implementation of Sobel edge detection. Sobel edge detection is performed by convolving the input image with two  $3 \times 3$  masks. The horizontal mask measures the gradient of horizontal edges, and the vertical mask measures the gradient of vertical edges. Diagonal edges produce some response from each mask, allowing the edge orientation and strength to be measured for all edges. Both masks are shown in Figure 3.

	1	2	1			1	0	-1	
	0	0	0			2	0	-2	
	-1	-2	-1			1	0	-1	

*Horizontal*

*Vertical*

Figure 3: The Sobel Convolution Masks.

An Apply implementation of Sobel edge detection is shown in Figure 4. The lines have been numbered for the purposes of explanation, using the comment convention. Line numbers are not a part of the language.

```

procedure sobel (inimg : in array (-1..1, -1..1) of byte  -- 1
                 border 0,
                 thresh : const float,
                 mag : out float)
is
  horiz, vert : integer;  -- 2
begin
  horiz := inimg(-1,-1) + 2 * inimg(-1,0) + inimg(-1,1) -  -- 3
           inimg(1,-1) - 2 * inimg(1,0) - inimg(1,1);
  vert := inimg(-1,-1) + 2 * inimg(0,-1) + inimg(1,-1) -  -- 4
           inimg(-1,1) - 2 * inimg(0,1) - inimg(1,1);
  mag := sqrt (FLOAT(horiz)*FLOAT(horiz)  -- 5
              + FLOAT(vert)*FLOAT(vert));
  if mag < thresh then  -- 6
    mag := 0.0;  -- 7
  end if;  -- 8
end sobel;  -- 9

```

Figure 4: An Apply Implementation of Thresholded Sobel Edge Detection

Line 1 defines the input, output and constant parameters to the function. The input parameter `inimg` is a window of the input image. The constant parameter `thresh` is a threshold. Edges which are weaker than this threshold are suppressed in the output magnitude image, `mag`. Line 3 defines `horiz` and `vert` which are internal variables used to hold the results of the horizontal and vertical Sobel edge operator.

Line 1 also defines the input image window. It is a  $3 \times 3$  window centred about the current pixel processing position, which is filled with the value 0 when the window lies outside the image. This same line declares the constant and output parameters to be floating-point scalar variables.

The computation of the Sobel convolutions is implemented by the straight-forward expressions on lines 5 through 7. These expressions are readily seen to be a direct implementation of the convolutions in Figure 3.

### 3.3 Border Handling

Border handling is always a difficult and messy process in programming kernel operations such as Sobel edge detection. In practice, this is usually left up to the programmer, with varying results—sometimes borders are handled in one way, sometimes another. Apply provides a uniform way of resolving the difficulty. It supports border handling by extending the input images with a constant value. The constant value is specified as an assignment. Line 1 of Figure 4 indicates that the input image `inimg` is to be extended by filling with the constant value 0.

If the programmer does not specify how an input variable is to be extended as the window crosses the edge of the input image, Apply handles this case by not calculating the corresponding output pixel.

We plan to extend the Apply language with two other methods of border handling: extending the input image by replicating border pixels, and allowing the programmer to write a special-purpose routine for handling border pixels.

### 3.4 Image Reduction and Magnification

Apply allows the programmer to process images of different sizes, for example to reduce a  $512 \times 512$  image to a  $256 \times 256$  image, or to magnify images. This is implemented via the `SAMPLE` parameter, which can be applied to input images, and by using output image variables which are arrays instead of scalars. The `SAMPLE` parameter specifies that the apply operation is to be applied not at every pixel, but regularly across the image, skipping pixels

as specified in the integer list after `SAMPLE`. The window around each pixel still refers to the underlying input image. For example, the following program performs image reduction, using overlapping  $4 \times 4$  windows, to reduce a  $n \times n$  image to an  $n/2 \times n/2$  image:

```

procedure reduce(inimg : in array (0..3, 0..3) of byte sample (2, 2),
                 outimg : out byte)
is
  sum : integer;
  i, j : integer;
begin
  sum := 0;
  for i in 0..3 loop
    for j in 0..3 loop
      sum := sum + in(i, j);
    end loop;
  end loop;
  outimg := sum / 16;
end reduce;

```

Magnification can be done by using an output image variable which is an array. The result is that, instead of a single pixel being output for each input pixel, several pixels are output, making the output image larger than the input. The following program uses this to perform a simple image magnification, using linear interpolation:

```

procedure magnify(inimg : in array(-1..1, -1..1) of byte border 0,
                 outimg: out array(0..1, 0..1) of byte)
is
begin
  outimage(0,0) := (inimg(-1,-1) + inimg(-1,0)
                   + inimg(0,-1) + inimg(0,0)) / 4;
  outimage(0,1) := (inimg(-1,0) + inimg(-1,1)
                   + inimg(0,0) + inimg(0,1)) / 4;
  outimage(1,0) := (inimg(0,-1) + inimg(0,0)
                   + inimg(1,-1) + inimg(1,0)) / 4;
  outimage(1,1) := (inimg(0,0) + inimg(0,1)
                   + inimg(1,0) + inimg(1,1)) / 4;
end magnify;

```

The semantics of `SAMPLE (s1, s2)` are as follows: the input window is placed so that pixel  $(0,0)$  falls on image pixel  $(0,0), (0,s2), \dots, (0,n \times s2), \dots, (m \times s1, n \times s2)$ . Thus, `SAMPLE (1,1)` is equivalent to omitting the `SAMPLE` option entirely. If only one `SAMPLE` parameter exists, it applies to the last image dimension.

Output image arrays work by expanding the output image in either the horizontal or vertical direction, or both, and placing the resulting output windows so that they tile the output image without overlapping. If only one dimension is specified, it applies to the last image dimension, as with `SAMPLE`.

### 3.5 Multi-function Apply Modules

It is a topic of current research to allow `Apply` to efficiently implement multiple functions. The current version of `Apply` requires a separate pass, producing intermediate output images, for each `Apply` function. If multiple `Apply` functions can be compiled together in a single pass, it will be possible to perform some operations much more efficiently. For example, many median filter algorithms use results from an adjacent calculation of the median filter to compute a new median filter, when processing the image in raster order. This cannot be done with a single `Apply` function, since it requires the algorithms to make no restrictions on the order pixels are processed. However, we can define an efficient median filter using multiple `Apply` functions, and allow the compiler to figure out how to

efficiently execute this program on a particular machine, by taking advantage of adjacent results [5]. The following  $3 \times 3$  median filter has been carefully optimized for speed.

The algorithm works in two steps. The first step produces, for each pixel, a sort of the pixel and the pixels above and below that pixel. The result from this step is an image three times higher than the original, with the same width. The second step sorts, based on the middle element in the column, the three elements produced by the first step, producing the following relationships among the nine pixels at and surrounding a pixel:

$$\begin{array}{ccc}
 a & d & g \\
 \vee & \vee & \vee \\
 b & < e & < h \\
 \vee & \vee & \vee \\
 c & f & i
 \end{array}$$

From this diagram, it is easy to see that none of pixels  $g$ ,  $h$ ,  $b$ , or  $c$  can be the median, because they are all greater or less than at least five other pixels in the neighborhood. The only candidates for median are  $a$ ,  $d$ ,  $e$ ,  $f$ , and  $i$ . Now we observe that  $f < \{e, h, d, g\}$ , so that if  $f < a$ ,  $f$  cannot be the median since it will be less than five pixels in the neighborhood. Similarly, if  $a < f$ ,  $a$  cannot be the median. We therefore compare  $a$  and  $f$ , and keep the larger. By a similar argument, we compare  $i$  and  $d$  and keep the smaller. This leaves three pixels:  $e$ , and the two pixels we chose from  $\{a, f\}$ , and  $\{d, i\}$ . All of these are median candidates. We therefore sort them and choose the middle element; this is the median.

This algorithm computes a  $3 \times 3$  median filter with only eleven comparisons, comparable to many techniques for optimizing median filter in raster-order processing algorithms.

```

-- Sort the three elements at, above, and below each pixel
procedure median1(image in array(-1..1) of byte,
                  si out array(-1..1, 1) of byte)
is
    byte a, b, c;
begin
    if image(-1) > image(0)
    then if image(0) > image(1)
        then si(1,0) := image(-1);
            si(0,0) := image(0);
            si(-1,0) := image(1); end if;
        else if image(-1) > image(1)
            then si(1,0) := image(-1);
                si(0,0) := image(1);
                si(-1,0) := image(0);
            else si(1,0) := image(1);
                si(0,0) := image(-1);
                si(-1,0) := image(0);
            end if;
        end if;
    else if image(0) > image(1)
        then if image(-1) > image(1)
            then si(1,0) := image(0);
                si(0,0) := image(-1);
                si(-1,0) := image(1);
            else si(1,0) := image(0);
                si(0,0) := image(1);
                si(-1,0) := image(-1);
            end if;
        else si(1,0) := image(1);
            si(0,0) := image(0);
            si(-1,0) := image(-1);
        end if;
    end if;
end median1;

```

```

procedure median2(si in array(-1..1, -1..1) of byte sample (3, 1),
                  median out byte)
-- Combine the sorted columns from the first step to give the median.

is
  int l, m, h;
  byte A, B;
begin
  if si(-1, 0) > si(0, 0)
    then if si(0, 0) > si(1, 0)
      then h := -1; m := 0; l := 1; end if;
      else if si(-1,0) > si(1,0)
        then h := -1; m := 1; l := 0;
         else h := 1; m := -1; l := 0; end if; end if;
    else if si(0, 0) > si(1, 0)
      then if si(-1,0) > si(1,0)
        then h := 0; m := -1; l := 1;
         else h := 0; m := 1; l := -1; end if;
        else h := 1; m := 0; l := -1; end if; end if;

  if si(l, -1) > si(m, 1)
    then A := si(l, -1);
     else A := si(m, 1); end if;
  if si(m, -1) < si(h, 1)
    then B := si(m, -1);
     else B := si(h, 1); end if;

  if A > si(m, 0)
    then if si(m, 0) > B
      then median := si(m, 0); end if;
      else if A > B
        then median := B;
         else median := A; end if; end if;
    else if si(m, 0) > B
      then if A > B
        then median := A;
         else median := B; end if;
        else median := si(m, 0); end if; end if;

end median2;

```

#### 4 Apply on Warp

The implementation of Apply on Warp employs straight-forward raster processing of the images, with the processing divided among the cells as described in Section 2. The Sobel implementation in Figure 4 processes a 512×512 image on a 10 cell Warp in 330 ms, including the I/O time for the Warp machine.

#### 5 Apply on Uni-processor Machines

The same Apply compiler that generates Warp code also can generate C code to be run under UNIX. We have found that an Apply implementation is usually at least as efficient as any alternative implementation on the same machine. This efficiency results from the expert knowledge which is built into the Apply implementation but which is too verbose for the programmer to work with explicitly. In addition, Apply focuses the programmer's attention on the details of his computation, which often results in improved design of the basic computation.

The Apply implementation for uni-processor machines relies upon a subroutine library which was previously developed for this purpose. The routines are designed to efficiently pass a processing kernel over an image. They employ data buffering which allows the kernel to be shifted and scrolled over the buffer with a low constant cost, independent of the size of the kernel. The Sobel implementation in Figure 4 processes a  $512 \times 512$  image on a Vax 11/785 in 30 seconds.

The buffering technique which we developed for Apply on uni-processor machines operates as follows. Initially, a buffer is allocated and indexed by an Illiffe vector of pointers as shown in figure 5. For an  $N \times N$  input image which will be processed with an  $M \times M$  kernel,  $N \times M + (N+M-1)/M - 1$  pointers are required. The cost of computing these pointers is negligible compared to the  $N^2$  cost of the actual computation being performed at all pixel locations.

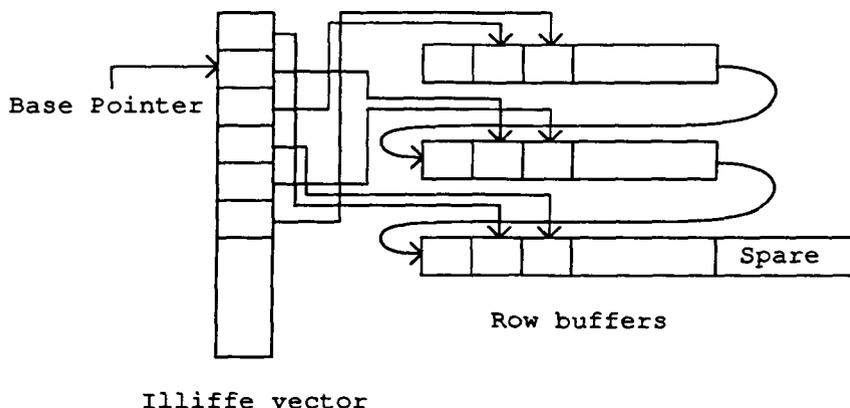


Figure 5: Image buffering for Apply

After establishing the pointers, which remain unchanged during the remainder of the algorithm, the first  $M$  rows of the image are copied into the buffer in preparation for processing.

Figure 5 displays the specific pointer arrangement for processing a  $3 \times 3$  kernel. When the pointer into the Illiffe vector is as shown in the figure, C language subscripting can be used to directly access the elements of the kernel surrounding the first pixel location. Two pointer dereference operations, possibly with small offsets, are needed for each access.

After the first pixel location has been processed, the base pointer is incremented by  $M$ . The  $3 \times 3$  kernel surrounding the second pixel location can then be directly accessed as before. It is thus possible to shift the kernel across the entire buffer of data with a cost of only one addition per pixel. The cost of relocating the kernel is independent of the size of the kernel, so large kernels can be processed very efficiently by Apply.

When processing of an entire row is completed, the base pointer is set back to its original position and then incremented by one. This has the effect of rolling the individual rows of the buffer upwards. The row which was previously in the centre of the  $3 \times 3$  kernel is now at the top and the row which was previously at the bottom is now in the centre. The row which was previously at the top is now the bottom row but, because it is being indexed by a new pointer, its origin has been shifted right one word. This shifting is not a problem because it affects the row into which new data must be read. The only constraints imposed are that there must be additional buffer space available and that the rows must be organized in memory so that there are no overlaps when the buffer is rolled in this manner. Figure 5 shows an arrangement which satisfies these constraints:  $(N+M-1)/M$  additional words of buffer space are provided, and the initial top row, which is shifted first, is placed last in memory.

Notice that once again, the cost of relocating the kernel is a single addition and does not depend on the size of the kernel being processed.

## 6 Apply on the Hughes HBA

Apply has been implemented on the Hughes HBA computer [15] by Richard Wallace of Carnegie Mellon and Hughes. In this computer, several MC68000 processors are connected on a high-speed video bus, with an interface between each processor and the bus that allows it to select a subwindow of the image to be stored into its memory. The input image is sent over the bus and windows are stored in each processor automatically using DMA. A similar interface exists for outputting the image from each processor. This allows flexible real-time image processing.

The Hughes HBA Apply implementation is straightforward and similar to the Warp implementation. The image is divided in “swaths”, which are adjacent sets of rows, and each processor takes one swath. (In the Warp implementation, the swaths are adjacent sets of columns, instead of rows). Swaths overlap to allow each processor to compute on a window around each pixel. The processors independently compute the result for each swath, which is fed back onto the video bus for display.

The HBA implementation of Apply includes a facility for image reduction, which was not included in earlier versions of Apply. The HBA implementation subsamples the input images, so that the input image window refers to the subsampled image, not the original image as in our definition. We prefer the approach here because it has more general semantics. For example, using image reduction as we have defined it, it is possible to define image reduction using overlapping windows as in Section 3.4.

## 7 Apply on Other Machines

Here we briefly outline how Apply could be implemented on other parallel machine types, specifically bit-serial processor arrays, and distributed memory general purpose processor machines. These two types of parallel machines are very common; many parallel architectures include them as a subset, or can simulate them efficiently.

### 7.1 Apply on bit-serial processor arrays

Bit-serial processor arrays [3] include a great many parallel machines. They are arrays of large numbers of very simple processors which are able to perform a single bit operation in every machine cycle. We assume only that it is possible to load images into the array such that each processor can be assigned to a single pixel of the input image, and that different processors can exchange information locally, that is, processors for adjacent pixels can exchange information efficiently. Specific machines may also have other features that may make Apply more efficient than the implementation outlined here.

In this implementation of Apply, each processor computes the result of one pixel window. Because there may be more pixels than processors, we allow a single processor to implement the action of several different processors over a period of time, that is, we adopt the Connection Machine’s idea of *virtual processors* [7].

The Apply program works as follows:

- Initialize: For  $n \times n$  image processing, use a virtual processor network of  $n \times n$  virtual processors.
- Input: For each variable of type IN, send a pixel to the corresponding virtual processor.
- Constant: *Broadcast* all variables of type CONST to all virtual processors.
- Window: For each IN variable, with a window size of  $m \times m$ , shift it in a spiral, first one step to the right, then one step up, then two steps two the left, then two steps down, and so on, storing the pixel

value in each virtual processor the pixel encounters, until a  $m \times m$  square around each virtual processor is filled. This will take  $m^2$  steps.

- **Compute:** Each virtual processor now has all the inputs it needs to calculate the output pixels. Perform this computation in parallel on all processors.

Because memory on these machines is often limited, it may be best to combine the “window” and “compute” steps above, to avoid the memory cost of pre storing all window elements on each virtual processor.

## 7.2 Apply on distributed memory general purpose machines

Machines in this class consist of a moderate number of general purpose processors, each with its own memory. Many general-purpose parallel architectures implement this model, such as the Intel iPSC [9] or the Cosmic Cube [14]. Other parallel architectures, such as the shared-memory BBN Butterfly [4; 13], can efficiently implement Apply in this way; treating them as distributed memory machines avoids problems with contention for memory.

This implementation of Apply works as follows:

- **Input:** If there are  $n$  processors in use, divide the image into  $n$  regions, and store one region in each of the  $n$  processors' memories. The actual shape of the regions can vary with the particular machine in use. Note that compact regions have smaller borders than long, thin regions, so that the next step will be more efficient if the regions are compact.
- **Window:** For each IN variable, processors exchange rows and columns of their image with processors holding an adjacent region from the image so that each processor has enough of the image to compute the corresponding output region.
- **Compute:** Each processor now has enough data to compute the output region. It does so, iterating over all pixels in its output region.

## 8 Summary

We have described our programming techniques for low-level vision on Warp. These techniques began with simple row-by-row image processing macros, which are still in use for certain kinds of algorithms, and led to the development of Apply, which is a specialized programming language for low-level vision on Warp.

We have defined the Apply language as it is currently implemented, and described its use in low-level vision programming. Apply is in daily use at Carnegie Mellon for Warp and vision programming in general; it has proved to be a useful tool for programming under UNIX, as well as an introductory tool for Warp programming.

The Apply language crystallizes our ideas on low-level vision programming on Warp. It allows the programmer to treat certain messy conditions, such as border conditions, uniformly. It also allows the programmer to get consistently good efficiency in low-level vision programming, by incorporating expert knowledge about how to implement such operators.

One of the most exciting characteristics of Apply is that it may be possible to implement it on diverse parallel machines. We have outlined such implementations on bit-serial processor arrays and distributed memory machines. Implementation of Apply on other machines will make porting of low-level vision programs easier, should extend the lifetime of programs for such supercomputers, and will make benchmarking easier.

We have shown that the Apply programming model provides a powerful simplified programming method which is applicable to a variety of parallel machines. Whereas programming such machines directly is often difficult, the Apply language provides a level of abstraction in which programs are easier to write, more comprehensible and

more likely to work correctly the first time. Algorithm debugging is supported by a version of the Apply compiler which generates C code for uni-processor machines.

## 9 Acknowledgments

We would like to acknowledge the contributions made by Steve Shafer who helped develop the Apply programming model. The Warp project is a large, and growing, project at Carnegie Mellon University and General Electric Corporation. The authors are greatly indebted to this group, which has designed, built, and maintained the Warp machine, as well as implemented the W2 programming language, which is the basis for the Warp implementation of Apply. Apply itself grew out of work in the standard vision programming environment at Carnegie Mellon, which is based on C/UNIX. Apply benefitted from the use and criticism of members of the Image Understanding Systems and Autonomous Land Vehicles group at Carnegie Mellon.

## 10 References

1. *Reference Manual for the Ada Programming Language*. MIL-STD 1815 edition, United States Department of Defense, AdaTEC, SIGPLAN Technical Committee on Ada, New York, N.Y. AdaTEC, 1982. Draft revised MIL-STD 1815. Draft proposed ANSI Standard document.
2. Annaratone, M., Arnould, E., Gross, T., Kung, H.T., Lam, M., Menzilcioglu, O., Sarocky, K. and Webb, J.A. Warp Architecture and Implementation. Conference Proceedings of the 13th Annual International Symposium on Computer Architecture, June, 1986, pp. 346-356.
3. Batcher, K. E. "Bit-serial parallel processing systems". *IEEE Trans. Computer C-31*, 5 (May 1982), 377-384.
4. BBN Laboratories. *The Uniform System Approach to Programming the Butterfly Parallel Processor*. 1 edition, Cambridge, MA, 1985.
5. Fisher, A. J. and P. T. Highnam. Communications, scheduling, and optimization in SIMD image processing. Computer Architectures for Pattern Analysis and Machine Intelligence, IEEE, 1987. (Submitted).
6. Hamey, L., H. Printz, D. Reece, and S. Shafer. A Programmer's Guide to the Generalized Image Library.
7. Hillis, W. D.. *The Connection Machine*. The MIT Press, Cambridge, Massachusetts, 1985.
8. T. S. Huang, G. J. Yang, and G. Y. Tang. A fast two-dimensional median filtering algorithm. International Conference on Pattern Recognition and Image Processing, IEEE, 1978, pp. 128-130.
9. *iPSC System Overview*. Intel Corporation, 1985.
10. Kernighan, B. W. and D. M. Ritchie. The M4 Macro Processor. In *Unix Programmer's Manual*, Bell Laboratories, Murray Hill, NJ 07974, 1979.
11. Kung, H.T. and Webb, J.A. Global Operations on the CMU Warp Machine. Proceedings of 1985 AIAA Computers in Aerospace V Conference, American Institute of Aeronautics and Astronautics, October, 1985, pp. 209-218.
12. Kung, H. T. and Webb, J. A. "Mapping Image Processing Operations onto a Linear Systolic Machine". *Distributed Computing* 1, 4 (1986), 246-257.
13. Olson, T. J. An Image Processing Package for the BBN Butterfly Parallel Processor. Butterfly Project Report 9, University of Rochester, Department of Computer Science, August, 1985.
14. Seitz, C. "The Cosmic Cube". *Communications of the ACM* 28, 1 (January 1985), 22-33.
15. Wallace, R. S. and M. D. Howard. HBA Vision Architecture: Built and Benchmarked. Computer Architectures for Pattern Analysis and Machine Intelligence, IEEE, 1987. (Submitted).