DTIC FILE COPY

4

AD-A199 305

End of Year Report for Parallel Vision Algorithm Design and Implementation

January 15, 1987 - January 14, 1988

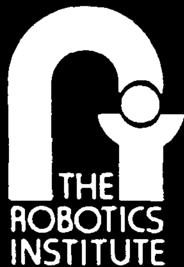
Takeo Kanade and Jon A Webb

11 August 1988
CMU-RI-TR-88-11

DTIC
ELECTE
SEP 1 4 1988
S D
H

Carnegie Mellon University

The Robotics Institute



Technical Report

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

(4)

End of Year Report for Parallel Vision Algorithm Design and Implementation

January 15, 1987 - January 14, 1988

Takeo Kanade and Jon A Webb
11 August 1988
CMU-RI-TR-88-11

DTIC
ELECTE
SEP 1 4 1988
S D
H

The research was supported by the Defense Advanced Research Projects Agency (DOD), monitored by the US Army Engineer Topographic Laboratories under Contract DACA76-85-C-0002.

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

88 9 12 099

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

A199 305

REPORT DOCUMENTATION PAGE

Field	Content
1a. REPORT SECURITY CLASSIFICATION	Unclassified
1b. RESTRICTIVE MARKINGS	
2a. SECURITY CLASSIFICATION AUTHORITY	
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
3. DISTRIBUTION / AVAILABILITY OF REPORT	Approved for public release; distribution unlimited
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	CMU-RI-TR-88-11
5. MONITORING ORGANIZATION REPORT NUMBER(S)	DACA76-86-C-0002
6a. NAME OF PERFORMING ORGANIZATION	The Robotics Institute Carnegie Mellon University
6b. OFFICE SYMBOL (If applicable)	
7a. NAME OF MONITORING ORGANIZATION	U.S. Army Engineer Topographic Laboratories
6c. ADDRESS (City, State, and ZIP Code)	Pittsburgh, PA 15213
7b. ADDRESS (City, State, and ZIP Code)	
8a. NAME OF FUNDING/SPONSORING ORGANIZATION	DARPA
8b. OFFICE SYMBOL (If applicable)	
9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
8c. ADDRESS (City, State, and ZIP Code)	
10. SOURCE OF FUNDING NUMBERS	PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.
11. TITLE (Include Security Classification)	End of Year Report for Parallel Vision Algorithm Design and Implementation
12. PERSONAL AUTHOR(S)	Takeo Kanade and Jon A. Webb
13a. TYPE OF REPORT	Technical
13b. TIME COVERED	FROM 1/15/87 TO 1/14/88
14. DATE OF REPORT (Year, Month, Day)	11 August 1988
15. PAGE COUNT	77
16. SUPPLEMENTARY NOTATION	

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)

parallel vision, Apply, language, WEB library, Warp

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Progress on the Parallel Vision project is reported. Three major accomplishments are noted: the development of the Apply language, the WEB library, and benchmarks of Warp for the DARPA image understanding architecture comparisons. The Apply language development includes a description of the language and its implementation on Warp, the Sun, and the Hughes HBA, together with benchmark comparisons of these very different architectures. The WEB library includes over 100 routines: included in this report are performance numbers of these routines on the CMU Warp machine. Finally, a detailed analysis of the Warp routines implemented for the DARPA Image Understanding benchmarks is given.

Field	Content
20. DISTRIBUTION / AVAILABILITY OF ABSTRACT	☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS
21. ABSTRACT SECURITY CLASSIFICATION	Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	
22b. TELEPHONE (Include Area Code)	
22c. OFFICE SYMBOL	

DD FORM 1473, 84 MAR

83 APR edition may be used until exhausted.
All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

Table of Contents

1. Introduction 2
  1.1 Overview 2
  1.2 Warp Vision Software 2
  1.3 The Apply Language and WEB 2
  1.4 Support for Other Programs 2
  1.5 Acknowledgments 3
2. An Architecture Independent Programming Language for Low-Level Vision 4
  2.1 Introduction 4
  2.2 Introduction to Apply 5
    2.2.1 The Apply Language 5
    2.2.2 An Implementation of Sobel Edge Detection 6
    2.2.3 Border Handling 7
    2.2.4 Image Reduction and Magnification 7
    2.2.5 Multi-function Apply Modules 8
      2.2.5.1 An Efficient Sobel Operator 9
      2.2.5.2 An Efficient Median Filter 9
  2.3 Apply on Warp and Warp-like Architectures 11
    2.3.1 Low-level vision on Warp 12
    2.3.2 Apply on FT Warp 13
    2.3.3 Apply on iWarp 14
  2.4 Apply on Uni-processor Machines 14
  2.5 Apply on the Hughes HBA 15
  2.6 Apply on Other Machines 16
    2.6.1 Apply on bit-serial processor arrays 16
    2.6.2 Apply on distributed memory general purpose machines 16
  2.7 Summary 17
  2.8 Grammar of the Apply Language 17
3. Architecture-Independent Image Processing: Performance of Apply on Diverse Architectures 20
  3.1 Introduction 20
  3.2 The WEB Library 20
  3.3 Apply Code Compared with Hand-written Code 21
    3.3.1 Apply code compared with SPIDER code 21
    3.3.2 Apply code compared with W2 code 23
  3.4 Comparison of Diverse Architectures 24
    3.4.1 Warp Compared with Sun 24
    3.4.2 Warp Compared with Hughes HBA 26
  3.5 Conclusions 27
4. The WEB Library 30
  4.1 Introduction 30
  4.2 Calling Programs in WEB 30
  4.3 Classification by Area 31
5. Performance of Warp on the DARPA Image Understanding Architecture Benchmarks 32
  5.1 Introduction 32
  5.2 Warp Status 33
  5.3 Vision Programming On Warp 34
    5.3.1 Input Partitioning 34
    5.3.2 Output Partitioning 34
    5.3.3 Pipelining 34
  5.4 Laplacian 34
  5.5 Zero Crossings Detection 36
  5.6 Border following 37
  5.7 Connected components labelling 38
    5.7.1 Sketch of the Algorithm 38

Accession For
NTIS GRA&I ☑
DTIC TAB ☐
Unannounced ☐
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or Special
A-1

5.7.1.1 Vocabulary and Notation 38
5.7.1.2 The Algorithm 38
5.7.2 Asymptotic Running Time 42
5.7.2.1 Parallel-Sequential-Systolic Algorithm 42
5.7.2.2 Parallel-Sequential-Parallel Algorithm 42
5.7.3 Implementation Details 42
5.7.3.1 Warp Architectural Constraints 42
5.7.3.2 Vax Implementation 44
5.7.3.3 Warp Implementations 44
5.8 Hough transform 45
5.9 Convex Hull. 47
5.10 Voronoi Diagram 47
5.11 Minimum spanning tree 48
5.12 Visibility 49
5.13 Graph Matching 49
5.14 Minimum-cost Path 50
5.15 Warp Benchmarks Summary 51
5.16 Evaluation of the Warp Architecture 51
5.16.1 Memory 51
5.16.2 Number of processing elements 52
5.16.3 External host 53
6. References 54
I. WEB Listing 57

List of Figures

Figure 2-1: The Sobel Convolution Masks. 6
Figure 2-2: An Apply Implementation of Thresholded Sobel Edge Detection. 7
Figure 2-3: A More Efficient Sobel Operator 9
Figure 2-4: Input Partitioning Method on Warp 12
Figure 2-5: Processing the first row by the cyclic-scroll buffering 15
Figure 2-6: Processing the second row by the cyclic-scroll buffering 15
Figure 3-1: Ratio of execution times of hand-generated SPIDER FORTRAN to Apply code. Vertical line indicates a ratio of one. 21
Figure 3-2: Scatter diagram of execution times of hand-generated SPIDER FORTRAN and Apply code. Diagonal line indicates equality. 22
Figure 3-3: Ratio of execution times of hand-generated W2 code to Apply code. Vertical line indicates a ratio of one. 24
Figure 3-4: Scatter diagram of execution times of hand-generated W2 code and Apply code. Diagonal line indicates equality. 25
Figure 3-5: Ratio of execution times of Sun Apply code to Warp Apply code. 26
Figure 3-6: Scatter diagram of execution times of Sun Apply code and Warp Apply code. 27
Figure 3-7: Ratio of execution times of Hughes HBA Apply code to Warp Apply code. Vertical line indicates a ratio of one. 28
Figure 3-8: Scatter diagram of execution times of Hughes HBA Apply code and Warp Apply code. Diagonal line indicates equality. 29
Figure 5-1: Folding columns 35
Figure 5-2: Using results from previous steps 35
Figure 5-3: Convolving and storing column sums 36
Figure 5-4: Adding appropriate column sums 36
Figure 5-5: Input 40
Figure 5-6: Labels after parallel phase 40
Figure 5-7: Labels after sequential phase 41

List of Tables

Table 5-1: Optimized Symmetric Convolution 36
Table 5-2: Final maps 41
Table 5-3: Label Computation 41
Table 5-4: Vax implementation timings 44
Table 5-5: Estimated WW Warp timings 45
Table 5-6: Estimated PC Warp timings 46
Table 5-7: Estimated *i*Warp timings 46
Table 5-8: Operation counts for Voronoi diagram 48
Table 5-9: Warp Benchmark Summary 51

Abstract

Progress on the Parallel Vision project is reported. Three major accomplishments are noted: the development of the Apply language, the WEB library, and benchmarks of Warp for the DARPA image understanding architecture comparisons. The Apply language development includes a description of the language and its implementation on Warp, the Sun, and the Hughes HBA, together with benchmark comparisons of these very different architectures. The WEB library includes over 100 routines; included in this report are performance numbers of these routines on the CMU Warp machine. Finally, a detailed analysis of the Warp routines implemented for the DARPA Image Understanding benchmarks is given.

1. Introduction

This report reviews progress at Carnegie Mellon from January 15, 1987 to January 14, 1988 on research supported by the Defense Advanced Research Project Agency (DOD), monitored by the US Army Engineer Topographic Laboratories under Contract DACA76-85-C-0002, titled "Research on Parallel Vision Algorithm Design and Implementation." The report consists of an introduction and four detailed reports on specific areas of research.

1.1 Overview

During this contract year our research has had three main themes:

- Support, development, and evaluation of Warp-related vision software.
- Development of the Apply programming language and WEB library of low- and mid-level vision algorithms.
- Support for the use of parallel vision software in related DARPA-sponsored programs.

Warp gives us a powerful, existing parallel computer on which to develop parallel vision software; with this basis, we are able to evaluate our work and see it applied to important problems in related programs that use Warp. But we have not limited ourselves strictly to Warp software development. The Apply programming language has proved to be a useful tool for parallel vision algorithm development on many parallel computers, especially since a substantial portion of the WEB library of low- and mid-level vision algorithms is implemented using it. These two efforts have led to significant application of our work in several DARPA-sponsored programs.

1.2 Warp Vision Software

We have implemented Warp software that allows the use of the Warp computer in the Carnegie Mellon vision environment, including remote access to the Warp computer from any Sun computer in the environment. This software has been used to develop parallel vision algorithms at Carnegie Mellon throughout the year.

Our implementation of Warp vision algorithms led to the evaluation of the Warp computer in the DARPA Image Understanding Architectures Benchmark Workshop. Several programs were implemented in order to compare Warp with other parallel vision architectures, including The Connection Machine and Butterfly. The results of this study are described in Section 5.

1.3 The Apply Language and WEB

In the summer of 1987, Apply was reimplemented to generate efficient code for Warp, the Sun/3, and FT Warp, a 2-dimensional Warp array. This reimplementation used a common front-end for all Apply programs, and different back-ends for the different target architectures and languages. Section 2 describes Apply and its implementations on Sun, Warp, and the Hughes HBA. It proved possible to directly compare the performance of Apply programs on Warp with Apply on the Sun and Apply in a previous implementation on the Hughes HBA. Results are reported in Section 3.

WEB was also reimplemented in the summer of 1987. This implementation used Apply for about 80% of the programs, and W2 code for the remainder, most of which are global image processing operations not suited for Apply. Section 4 describes WEB, and Appendix I lists the current status of each WEB routine. Comparison with last year's report shows enormous progress in making the routines implemented, validated, and made available.

1.4 Support for Other Programs

The Warp computer is used in several DARPA-sponsored programs: SC Vision, ALV, ADRIES, and SCORPIUS. In many of these programs, image processing and related functions are a primary concern. Work on parallel vision algorithms on the Warp machine at Carnegie Mellon has often been directly transferable to these other programs, often by using Apply and WEB.

For example, this software was used in the demonstration of the NAVLAB autonomous land vehicle of May 7, 1987 at the Warp/Butterfly User's Group, which demonstrated a 5-to-1 speedup over the previous NAVLAB demonstration of November 1986.

1.5 Acknowledgments

Several people contributed significantly to the parallel vision effort, and deserve special mention for their work:

- Leonard Hamey developed the original Apply concept and language, based on discussions with Steve Shafer. He also implemented the Generalized Image Library.
- I-Chen Wu implemented the current Apply compiler.
- Hudson Ribas wrote most of the current WEB library.
- Richard Wallace and Mike Howard implemented Apply on the Hughes Hierarchical Bus Architecture (HBA).
- Ravi Mosur implemented Warp Generalized Images and the Warp User Package.
- Francois Bitz installed and supported the Warp computer on the NAVLAB robot vehicle.

In addition, the parallel vision effort benefitted from its association with the Warp group at Carnegie Mellon and General Electric Corporation, and the Image Understanding Systems and Road Following groups at Carnegie Mellon.

2. An Architecture Independent Programming Language for Low-Level Vision

2.1 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 512×512 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* [24].

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 [23] such as connected components, Hough transform, FFT, and histogram. However, Apply is in daily use at Carnegie Mellon and elsewhere, and has been used to implement a significant library (100 programs) of algorithms covering most of low-level vision. A companion paper [33] describes this library and evaluates Apply's performance.

We begin by describing the Apply language, and its utility for programming low-level vision algorithms. Examples of Apply programs and Apply's syntax are presented. Finally, we discuss implementations of Apply on various architectures: Warp and Warp-like architectures, uni-processors, the Hughes HBA, bit-serial processor arrays, and distributed memory machines.

2.2 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 embedded. 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.

2.2.1 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.

Section 2.8 gives 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. 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.

The operators `^`, `|`, `&`, and `!` refer to the exclusive or, or, and, and not operations, respectively. Variable and function names are alpha-numeric strings of arbitrary length, commencing with an alphabetic character. The `INTEGER` and `REAL` pseudo-functions convert from real to integer, and from integer (or byte) to real types. Case is not significant, except in the preprocessing stage which is implemented by the *m4* macro processor [22].

`BYTE`, `INTEGER`, and `REAL` 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.

2.2.2 An Implementation of Sobel Edge Detection

As a si··ple 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 3×3 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 2-1.

```
|  1  2  1  |        |  1  0 -1  |
|  0  0  0  |        |  2  0 -2  |
| -1 -2 -1  |        |  1  0 -1  |
```

Horizontal *Vertical*

Figure 2-1: The Sobel Convolution Masks.

An Apply implementation of Sobel edge detection is shown in Figure 2-2. The lines have been numbered for the purposes of explanation, using the comment convention. Line numbers are not a part of the language.

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 3×3 window centered 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.

```
procedure sobel (inimg  : in array (-1..1, -1..1) of byte    -- 1
                          border 0,
                 thresh : const real,
                 mag    : out real)
is                                                            -- 2
    horiz, vert : integer;                                    -- 3
begin                                                         -- 4
    horiz := inimg(-1,-1) + 2 * inimg(-1,0) + inimg(-1,1) -  -- 5
             inimg(1,-1) - 2 * inimg(1,0) - inimg(1,1);
    vert := inimg(-1,-1) + 2 * inimg(0,-1) + inimg(1,-1) -   -- 6
            inimg(-1,1) - 2 * inimg(0,1) - inimg(1,1);
    mag := sqrt(horiz*horiz + vert*vert);                     -- 7
    if mag < thresh then                                      -- 8
        mag := 0.0;                                           -- 9
    end if;                                                   -- 10
end sobel;                                                    -- 11
```

Figure 2-2: An Apply Implementation of Thresholded Sobel Edge Detection.

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 2-1.

2.2.3 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 2-2 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.

2.2.4 Image Reduction and Magnification

Apply allows the programmer to process images of different sizes, for example to reduce a 512×512 image to a 256×256 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 4×4 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 + inimg(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 $(0,0),(0,s2),\ldots,(0,n\times s2),\ldots,(m\times s1,n\times s2)$. Thus, `SAMPLE (1,1)` is equivalent to omitting the `SAMPLE` option entirely.

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.

2.2.5 Multi-function Apply Modules

In many low-level image processing algorithms, results from an adjacent pixel are saved in order to be used to calculate the results at an adjacent pixel; this results in a more efficient algorithm. Because Apply programs do not share results from adjacent pixels (doing so would violate Apply's order-independence, which is what makes it easy to implement in parallel), Apply programmers cannot take advantage of this trick. However, many of these algorithms can be factored into multiple passes in a way that results in an efficient program without needing to introduce order dependence.

These multiple functions can be efficiently implemented in Apply. Where memory use is not a concern, the intermediate results can be saved, and used by the next Apply program. In cases where memory is limited, multiple Apply functions can be compiled together into a single pass.

2.2.5.1 An Efficient Sobel Operator

A simple example is the Scbel operator. In the program shown in Figure 2-2, at each pixel the row and column *sums must* be recalculated. But at every pixel, one of the row and column sums are shared with pixels two to the left, right, top, and bottom. This is inefficient.

Figure 2-3 shows the same Sobel operator implemented as multiple functions. In the ROWCOL procedure, the row and column sums are calculated–each is calculated only once per pixel. In the SOBEL procedure, the row and column differences are summed, and the result is computed just as before. This program does 6 fewer additions and 2 fewer multiplications than the program in Figure 2-2.

```
procedure rowcol (inimg  : in array (-1..1, -1..1) of byte
                           border 0,
                  rowsum : out integer,
                  colsum : out integer)
is
begin
    rowsum := inimg(0,-1) + 2 * inimg(0,0) + inimg(0,1);
    colsum := inimg(-1,0) + 2 * inimg(0,0) + inimg(1,0);
end rowcol;

procedure sobel (rowsum : in array(-1..1, 0..0) of integer,
                 colsum : in array(0..0, -1..1) of integer,
                 thresh : const real,
                 mag    : out real)
is
    horiz, vert : integer;
begin
    horiz := rowsum(-1,0) - rowsum(1,0);
    vert := colsum(0,-1) - colsum(0,1);
    mag := sqrt(horiz*horiz + vert*vert);
    if mag < thresh then
        mag := 0.0;
    end if;
end sobel;
```

Figure 2-3: A More Efficient Sobel Operator

2.2.5.2 An Efficient Median Filter

Another example is median filter. 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. Apply multiple functions lead to the following 3×3 median filter.

The algorithm works in two steps. The first step (MEDIAN1) 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 (MEDIAN2) sorts, based on the middle element in the column, the three elements produced by the first step. Note the use of the SAMPLE clause in this step to place MEDIAN2 at every third row produced by MEDIAN1–this causes MEDIAN2 to produce an image the same size as the input to MEDIAN1. MEDIAN2 produces the following relationships among the nine pixels at and surrounding a pixel:

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

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 3×3 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, 0..0) of byte,
                  si    : out array(-1..1, 0..0) of byte)
is
        byte a, b, c;
begin
     if image(-1,0) > image(0,0)
         then if image(0,0) > image(1,0)
              then si(1,0) := image(-1,0);
                   si(0,0) := image(0,0);
                   si(-1,0) := image(1,0); end if;
              else if image(-1,0) > image(1,0)
                   then si(1,0) := image(-1,0);
                        si(0,0) := image(1,0);
                        si(-1,0) := image(0,0);
                   else si(1,0) := image(1,0);
                        si(0,0) := image(-1,0);
                        si(-1,0) := image(0,0);
                   end if;
              end if;
         else if image(0,0) > image(1,0)
              then if image(-1,0) > image(1,0)
                   then si(1,0) := image(0,0);
                        si(0,0) := image(-1,0);
                        si(-1,0) := image(1,0);
                   else si(1,0) := image(0,0);
                        si(0,0) := image(1,0);
                        si(-1,0) := image(-1,0);
                   end if;
              else si(1,0) := image(1,0);
                   si(0,0) := image(0,0);
                   si(-1,0) := image(-1,0);
              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;
```

2.3 Apply on Warp and Warp-like Architectures

The Warp-like architectures have in common that they are systolic arrays, in which each processor is a powerful (10 MFLOPS or more) computer with high word-by-word I/O bandwidth with adjacent processors, arranged in a simple topology. Apply is implemented on these processors in similar ways, so we first describe the basic model of low-level image processing on Warp, and then sketch the implementations on FT Warp and *i* Warp.

We briefly describe each of the Warp-like architectures; a complete description of Warp is available elsewhere [3]. Warp is a short linear array, typically consisting of ten cells, each of which is a 10 MFLOPS computer. The array has high internal bandwidth, consistent with its use as a systolic processor. Each cell has a local program and data memory, and can be programmed in a Pascal-level language called W2, which supports communication between cells using asynchronous word-by-word `send` and `receive` statements. The systolic array is attached to an external host, which sends and receives data from the array from a separate memory. The external host in turn is attached to a Sun computer, which provides the user interface.

Fault-tolerant (FT) Warp is a two-dimensional array, typically a five-by-five array, being designed by Carnegie Mellon. Each cell is a Warp cell. Each row and column can be fed data independently, providing for a very high bandwidth. As the name suggests, this array has as a primary goal fault-tolerance, which is supported by a virtual channel mechanism mediated by a separate hardware component called a switch.

i Warp is an integrated version of Warp being designed by Carnegie Mellon and Intel. In *i* Warp each Warp cell is implemented by a single chip, plus memory chips. The baseline *i* Warp machine is a 72 cell linear array, although two-dimensional designs are also being considered. *i* Warp includes support for distant cells to communicate as if they were adjacent, while passing their data through intermediate cells.

2.3.1 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 2-4. This gives each cell a tall, narrow region to process; for 512×512 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.

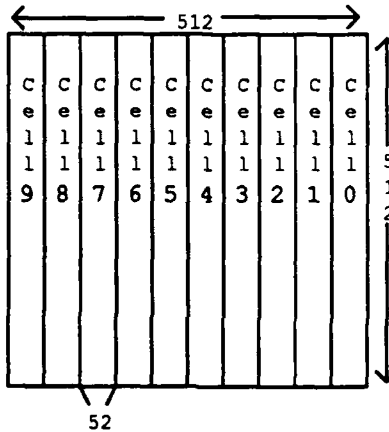


Figure 2-4: Input Partitioning Method on Warp

The image is divided in this way using a series of macros called GETROW, PUTROW, and COMPUTEROW. 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 2-4.

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 $1/4^{th}$ 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 [17]) 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 $1/10^{th}$ of a row. This was important in the prototype Warp machines, since they had only 4K words memory on each cell. On PC Warp, with 32K words of memory per cell, this approach makes it possible to implement very large window operations.
- 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!

2.3.2 Apply on FT Warp

The 2-dimensional FT Warp array can be viewed as several 1-dimensional arrays. An image is usually divided into several swaths (adjacent groups of rows) on FT Warp. The data of each swath are fed into the corresponding row of these 2-dimensional processors, as an image is fed into a 1-dimensional array. This results in each cell of FT Warp in seeing a rectangular portion of the image.

To make the bandwidth as high as possible and to use the COMPUTEROW model, we input the data along the horizontal path and output data along the vertical path.

The typical FT Warp array is a five-by-five array, as opposed to ten cells in Warp, and each cell is as powerful as

a Warp cell. FT Warp, however, has much higher bandwidth than Warp. Therefore, for complex image processing operations where I/O bandwidth is not a factor, we expect FT Warp Apply programs to be 2.5 times faster than Warp programs, and even faster in simple image processing operations where I/O bandwidth limits Warp performance.

2.3.3 Apply on iWarp

The *i* Warp implementation of Apply uses a virtual pathway mechanism to allow each cell to process only data intended for that cell. This eliminates much of the complication of Apply on Warp; there is no need for a cell to explicitly pass data on to other cells, instead it can simply direct the rest of the data to pass on to later cells without further intervention.

Our description of Apply on *i* Warp will be clear if we describe the action of GETROW and PUTROW on this machine. In GETROW, each cell accepts data intended for that cell, and then releases control of the data to be passed on to the next cell automatically, until the arrival of the start of the next row. After releasing control, it goes on to process the data it has just received. In the meantime, it is allowing data to pass by on the output channel until the end of the output row arrives. It then tacks on its computed output to the end of this output row, completing PUTROW.

We expect this method of implementing Apply to be at least as efficient as the COMPUTEROW model on Warp. Since the baseline *i* Warp machine has 72 cells, each of which is 1.6 to 2 times as powerful as a Warp cell, total performance should be from about 10 to 14 times greater than Warp. *i* Warp's I/O bandwidth is much higher than Warp's, so this performance should be achievable for all but the most simple image processing operations.

2.4 Apply on Uni-processor Machines

The same Apply compiler that generates Warp code also generates 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. The computation time of the Apply-generated code is usually faster than that of hand coded programs. This efficiency results from the expert knowledge which is built into the Apply implementation but which is too complex for the programmer to work with explicitly. In addition, Apply focuses the programmer's attention on the details of the computation, which often results in improved design of the basic computation.

The Apply implementation for uni-processor machines employs a technique, called *cyclic-scroll buffering* here, which efficiently uses small space and time to buffer the rows of the image. The technique allows the kernel to be shifted and scrolled over the buffer with low cost.

The cyclic-scroll buffering technique which we developed for Apply on uni-processor machines is described as follows. For an $N \times N$ input image which will be processed with an $M \times M$ kernel, a buffer with $(N+M-1) \times M + (N-1)$ elements is required.

Figure 2-5 and 2-6 display the column-major arrangement for processing a 3×3 kernel. The pointers represent successive positions in memory. In addition, we keep two base pointers for the buffer. One, called *row base*, points to the first pixel of the three rows of the image and the other, called *kernel base*, points to the first pixel of the kernel. C language subscripting can be used to directly access the elements of the kernel except that the indices of row and column must be exchanged because the rows of the images are stored in column-major order.

Initially, we put the first M rows of the image, including the border, into the buffer in *column-major order*. When the first kernel is processed, row base points to the first element of the buffer, and kernel base points to the center element of the window to be processed. After the first kernel has been processed, the kernel base is incremented by M to point to the first pixel of the next kernel. It is thus possible to shift the kernel across the entire buffer of data with a cost of only one addition.

When processing an entire row is completed, the first row in the buffer from the row base is discarded and the next row of the image is input into the discarded row with a column displacement of one (i.e. beginning at the

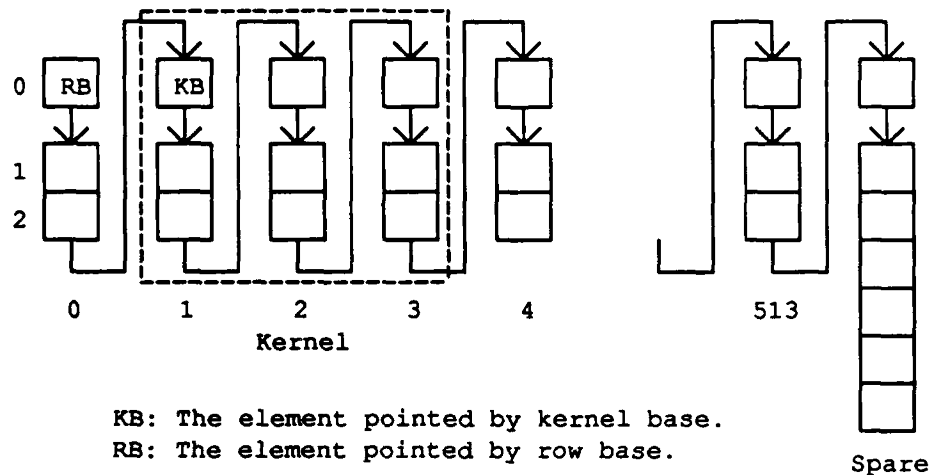


Figure 2-5: Processing the first row by the cyclic-scroll buffering

second element). Then the row base is incremented by one. The purpose of column displacement 1 is that the input row can be considered to be the Mth row of the buffer starting from the new row base. Effectively, the rolling is done at the same time. After the kernel base is reset to point to the center element of the new window, we can do another row operation in the same way as the first until all the rows are processed. Figure 2-5 and 2-6 show the processing of the first and second row.

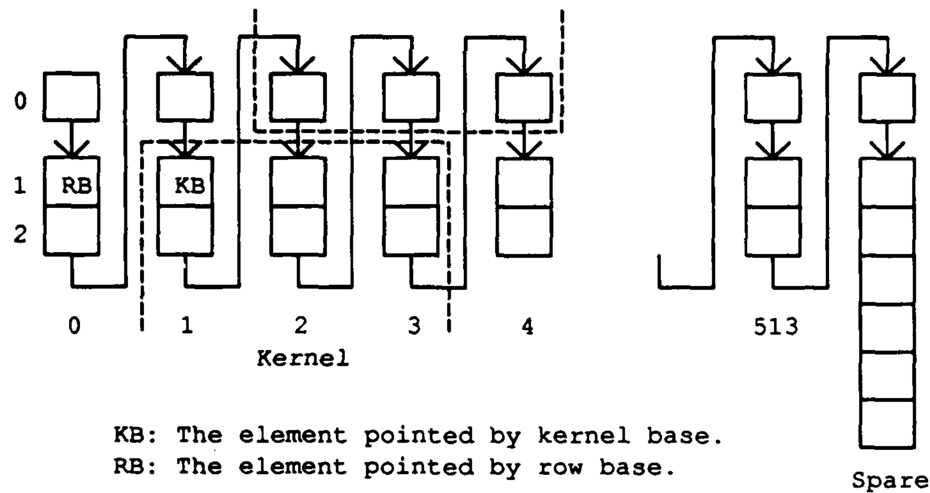


Figure 2-6: Processing the second row by the cyclic-scroll buffering

For each row operation, one more memory element is needed in the buffer. Therefore. the total number of the elements in the buffer is $M \times (N+M-1) + (N-1)$.

2.5 Apply on the Hughes HBA

Apply has been implemented on the Hughes HBA computer [32] 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 2.2.4.

2.6 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.

2.6.1 Apply on bit-serial processor arrays

Bit-serial processor arrays [6] 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* [16].

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 prestoring all window elements on each virtual processor.

2.6.2 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 [18] or the Cosmic Cube [29]. Other parallel architectures, such as the shared-memory BBN Butterfly [7, 25], 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.

2.7 Summary

The Apply language crystallizes our ideas on low-level vision programming on parallel machines. 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.

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 and elsewhere 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.

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. This language could then be mapped onto other computers, including both uni-processors and parallel computers.

One of the most exciting characteristics of Apply is that it is 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. Several implementation efforts are underway at other sites to map Apply onto other parallel machines than those described here.

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.

2.8 Grammar of the Apply Language

```
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 ]*

integer                  ::=    [sign] digit [ digit ]*
sign                     ::=    + | -
digit                    ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

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

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

range                    ::=    integer-expr .. integer-expr

elementary-type          ::=    sign object
                         |      object

sign                     ::=    SIGNED
                         |      UNSIGNED
                         |      Empty

object                   ::=    BYTE
                         |      INTEGER
                         |      REAL

statements               ::= [ statement ; ]*

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

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

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

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

expr                     ::=    expr + expr
                         |      expr - expr
                         |      expr * 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 integer-var IN range LOOP
                           statements
                       END LOOP

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

3. Architecture-Independent Image Processing: Performance of Apply on Diverse Architectures

3.1 Introduction

Low-level vision is an area of computer science that is ripe for the use of parallel computers. This class of operations is easily parallelizable. Indeed, many parallel computers are already being developed for use at this level of vision. These computers offer enormous speedup to the developer of computer vision algorithms, since these operations are so time-consuming, but software development is necessary before they can be used.

We have developed a language called Apply [14] which can generate efficient programs for a variety of parallel machines given a single source code. Apply therefore allows machine independent programming, for a limited, application-specific, set of algorithms.

Apply has been used to develop a library of vision programs called WEB, which includes routines for many low-level vision operations. Over 130 programs exist in WEB, 80% of which are written in Apply. The Apply routines include basic image operations, convolution, edge detection, smoothing, binary image processing, color conversion, pattern generation, and multi-level image processing. This library is therefore a machine-independent software base for low-level image processing.

Because of the machine independence of the Apply language, programs written in Apply can be ported from one machine to another simply by recompilation. Moreover, the Apply compiler and the WEB library allow the comparison of the performance of vision machines, since the same source code will be running on both machine, which is the strongest possible basis for comparison of two computers.

In this paper, we demonstrate this by studying the performance of Apply on three diverse architectures, by examining the execution times of programs from WEB. The architectures are the Carnegie Mellon Warp machine, a 100 MFLOPS systolic array machine [4]; a Sun workstation; and the Hughes Aircraft Corporation Hierarchical Bus Architecture (HBA) [32], a MIMD computer specifically designed for image processing applications. These architectures differ in the number of processors, in the processor topology, and in the underlying processor, but Apply generates efficient code for all of them. The implementation of Apply on each of them is described elsewhere [14].

We discuss the WEB library, which has been the basis of our performance experiments with Apply. Using WEB, we establish a baseline of Apply's performance by comparing Apply code with code generated by hand for some of the computers. Then we use execution time as a basis for evaluating the performance of Apply, and for studying the suitability of these machines as image processors.

3.2 The WEB Library

Apply has been used to implement a large portion of the WEB library of vision programs, which is a large library of vision programs implemented for use on the Carnegie Mellon Warp machine. The original purpose of the library was to facilitate vision programming on the Warp machine.

WEB currently consists of over 130 routines, 80% of which are written in Apply. The rest are written in W2, which is the standard Warp programming language. All of the local image-to-image vision routines in WEB are written in Apply; the W2 routines include non-local routines such as histogram, image warping, and connected components.

WEB is based on the SPIDER library of FORTRAN programs [30]. This is a subroutine library, developed in Japan, for image processing using FORTRAN. Routines from SPIDER will be compared here in performance with equivalent routines from WEB in order to measure Apply's performance as a code generator for Sun.

3.3 Apply Code Compared with Hand-written Code

Our primary purpose in this paper is to develop a comparison of different parallel processing machines for vision using Apply as a vehicle. In order to base this comparison on solid ground, we must first evaluate Apply's performance compared with hand-written code for the same machine. If Apply produces code that is comparable to hand-written code, then our comparison will be solidly based, since the code generated by Apply represents the peak performance of the machine. On the other hand, if Apply code is not as good, then the comparison will not be solidly based; it could be argued that the measured performance would not actually be seen, since the user would not use Apply.

3.3.1 Apply code compared with SPIDER code

We begin by comparing Apply performance on WEB routines with a set of routines of similar function from the SPIDER FORTRAN library. The SPIDER library is professionally written and distributed, and the code is of high quality; therefore, this comparison pits Apply's code against the code of expert programmers.

We are comparing the actual execution times (user time plus system time) of the FORTRAN programs, called as a subroutine from C, with execution times of C programs generated by Apply, called in the same way. The time is measured from the point at which the input images are ready (have been stored in the Sun's memory) to the point at which the output images are ready, in both cases. This time does not include the I/O time for the images from disk, or the code download time from disk into the Sun. All times are for 512×512 images.

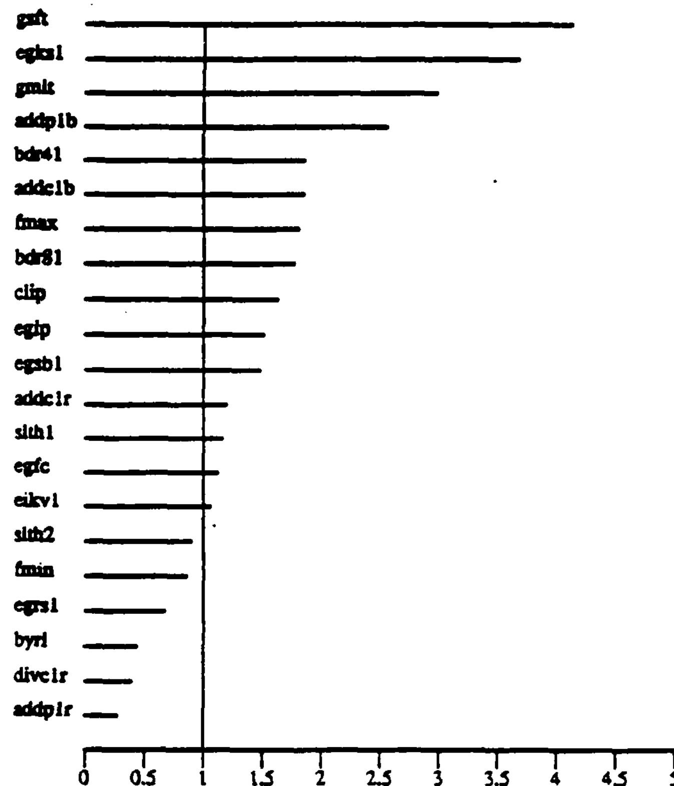


Figure 3-1: Ratio of execution times of hand-generated SPIDER FORTRAN to Apply code. Vertical line indicates a ratio of one.

Figure 3-1 gives the ratios of execution times for these programs, and Figure 3-2 shows the distribution of times for all programs. We can see from these figures the following phenomena:

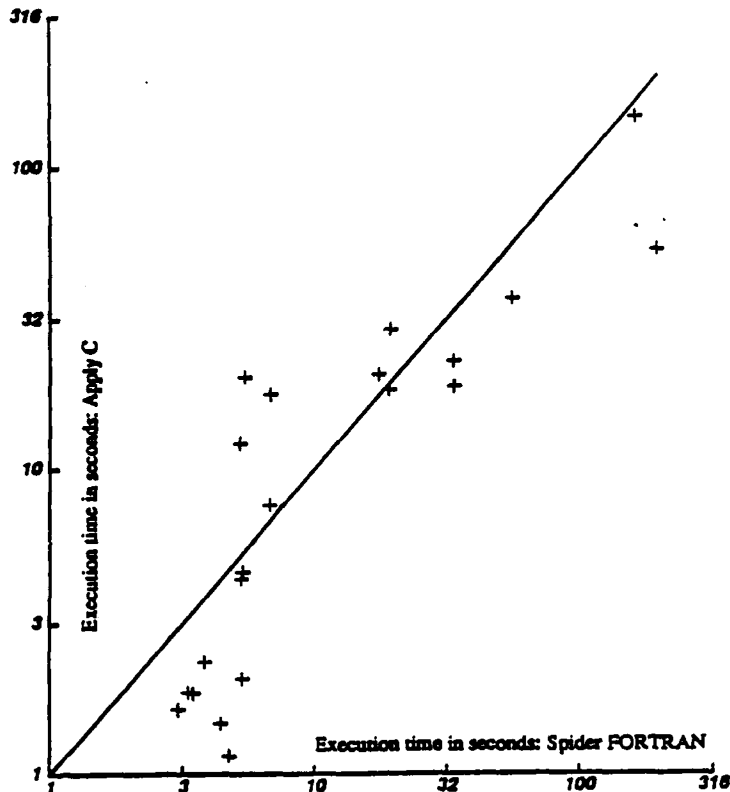


Figure 3-2: Scatter diagram of execution times of hand-generated SPIDER FORTRAN and Apply code.
Diagonal line indicates equality.

- The Apply programs are generally faster. There are four factors that can account for this: (1) Cyclic-scroll buffering; (2) The superiority of the Sun C compiler to the Sun FORTRAN compiler; (3) The FORTRAN code is written to be readable, at the expense of efficiency; the code generated by Apply need not satisfy such a constraint, since the Apply input code is quite readable. Apply can sacrifice legibility for speed.

- In some cases such as addp1r and divc1r the Apply code is slower. In these programs the algorithm is processing a single pixel from the input image to produce a single pixel in the output image. The cyclic-scroll buffering technique introduces a significant overhead in this case. (The same does not apply for addp1b and addc1b since here the FORTRAN code is processing integer images, while the Apply program is processing byte images. Thus, these programs are not strictly comparable).

- Apply has some limitations in its programming model that affect performance. In the FORTRAN subroutines, it is common to write several different ways of computing the output depending on switches. There is little overhead for this in FORTRAN since the code can be generated as follows:

```
        IF (SW.EQ.2) GOTO 100
        DO 10 I=RMIN, RMAX
        DO 10 J=CMIN, CMAX
         ..compute using method 1..
10      CONTINUE
        GOTO 200

100     DO 20 I=RMIN, RMAX
        DO 20 J=CMIN, CMAX
         ..compute using method 2..
20      CONTINUE
200     CONTINUE
```

That is, in FORTRAN the switch is tested only once, and different code is used in the different cases.

In Apply, the equivalent code would test the value of the switch once per pixel, since the Apply procedure is executed in its entirety for every pixel. This can limit performance in some cases, for example egrs1 and slth2.

In general, we see our intuitions about Apply performance compared with hand-written code to be correct. Apply can generate better code than hand-written, even on an easily programmed machine such as the Sun.

3.3.2 Apply code compared with W2 code

Next we compare performance on the Carnegie Mellon Warp machine. This machine is programmed by hand in W2, a Pascal-level language in which the user is explicitly aware of the different processors and the communication between them. (Send and receive statements are used to send words of data between cells).

While many programs have been and continue to be written for Warp in W2, the availability of Apply has *significantly eased the programmer's burden.* Apply *hides the explicit parallelism of cells, the number of cells in* use, and the communications between cells from the programmer. This has made it possible to develop WEB for Warp; without Apply, it is doubtful that such a library could have been built.

Figures 3-3 and 3-4 give the performance for hand-written W2 programs compared with WEB programs of equivalent function. The times are measured from the moment the input data is available for processing in the external host memory by the Warp array to the moment the output data is stored into the external host memory. All times are for 512×512 images. There are three main phenomena responsible for the wide distribution of execution time ratios:

- Some programs, such as egrs1, egpw3, and egpw4, are much slower in W2 than in Apply. This is because the W2 programmer made less optimizations to the code (such as unrolling innermost loops) than the Apply programmer. The Apply programs are smaller, and do not include statements for I/O, so that the programmer's effort is focused on making the heart of his code as efficient as possible.
- The Apply-generated programs consistently overlap I/O with computation on the cell, while the W2 programs do not. This is because, while it is possible to communicate between cells in the same Warp microinstruction where computation is done, doing so involves some careful placement of I/O statements which can be hard for the programmer.
- Some programs are slightly faster in W2 than in Apply. This is because of the limitation of the Apply programming model discussed earlier; the W2 programmer can initialize state based on the values of global variables, and avoid retesting switches, etc., during processing of the images, while the Apply programmer cannot do this.

Here we see two principal effects of the Apply language on Warp programming: (1) The Apply programs are simpler and easier to write, so the programmer makes them more efficient, and Apply in turn generates better code because it can deal with the machine complexity better; (2) The limitation of the Apply programming model for preprocessing data can lead to some loss of performance.

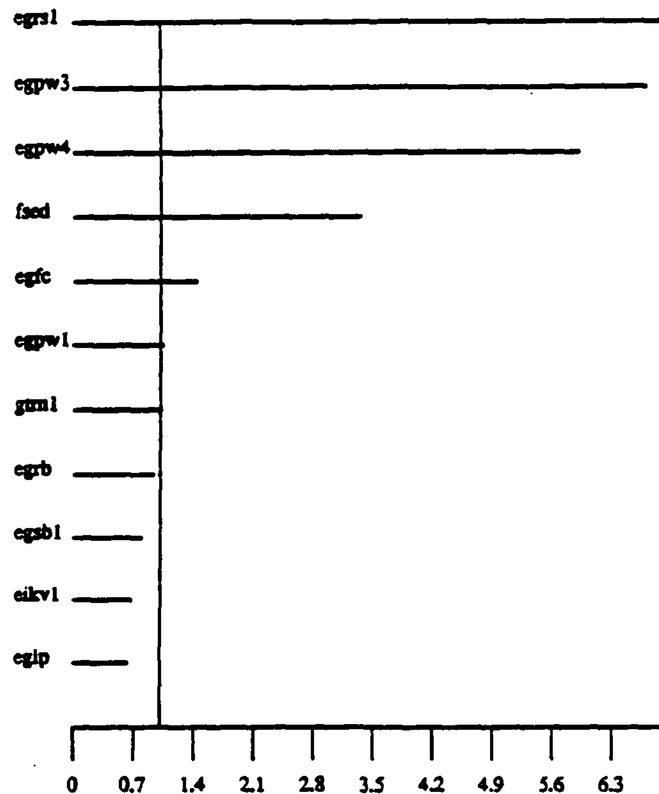


Figure 3-3: Ratio of execution times of hand-generated W2 code to Apply code. Vertical line indicates a ratio of one.

3.4 Comparison of Diverse Architectures

It is very rare that widely different computer architectures are compared directly for performance; the best previous examples have been FORTRAN studies of supercomputer performance [20]. These have depended on the implementation of a large language designed for use on sequential computers, and so have been limited to those computers in which significant software development has occurred to bring up FORTRAN, and which are suitable for implementation of a sequential computer language.

The comparisons presented here differ from these because the Apply language is designed for use on parallel computers, so that a wider range of computers can be compared, and because Apply is application-specific, so that it is small and does not require an enormous effort to bring up on a new system. Thus, we are able to directly compare the Sun 3/75, the Carnegie Mellon Warp machine, and the Hughes HBA.

3.4.1 Warp Compared with Sun

Figures 3-5 and 3-6 give the performance of a large number of programs implemented both on the Sun 3/75 computer (16 MHz MC68020 with MC68881 coprocessor) and the Warp machine. This allows us to evaluate the Warp's performance for image processing compared with a high-performance workstation.

Warp execution time was measured from the point at which the arrays of input data were available in Warp's external host to the point at which the arrays of output data became available. This is consistent with the measurement method for the Sun 3/75. Code download time was not included. All times are for 512x512 images.

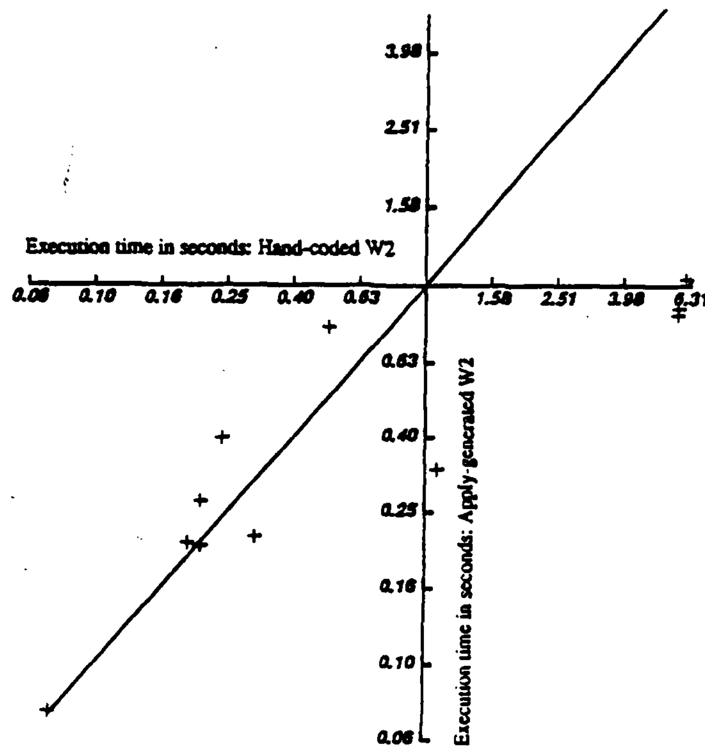


Figure 3-4: Scatter diagram of execution times of hand-generated W2 code and Apply code.
Diagonal line indicates equality.

We observe the following from these data:

- There are a few cases where Warp's performance far exceeds expectations: in the case of egfc and egks2, for example. Based on the comparative floating point rates, we would expect Warp's performance to be one to two hundred times that of the Sun, but here the execution times is 666 and 304 times less than the Sun. These large factors are due to the internal parallelism of the Warp cell; it consists of many independent units, which can be individually controlled with a wide horizontal microinstruction. In the best case, a Warp cell can do I/O with other cells, read and write memory, compute an integer ALU operation, and compute a floating point add and a floating point multiply, all in the same 200 nanosecond cycle. The success of this design (and of the compiler in packing instructions together) is shown in the ratios for egfc and egks2.

- In the majority of cases, the execution time ratio is tens of times the Sun 3/75. (The average ratio is 67, with a median of 40). This reflects the raw processing power of Warp combined with the effects of the applications mix (which includes a large amount of integer processing) and the efficiency of the Warp compiler.

- In some cases, the ratio is ten or less. In these cases, the Apply program cannot make use of Warp's highly pipelined floating point units, because of a large amount of conditional branching within the program, and also because the computation is mainly additions, so that the separate multiplier cannot be used. Here we are seeing the effects of using a highly pipelined machine to implement what is essentially a scalar operation. The multiple independent Warp cells can still be used effectively, since the computation is independent between each cell; but the pipelining of the computation within the cell is not successful.

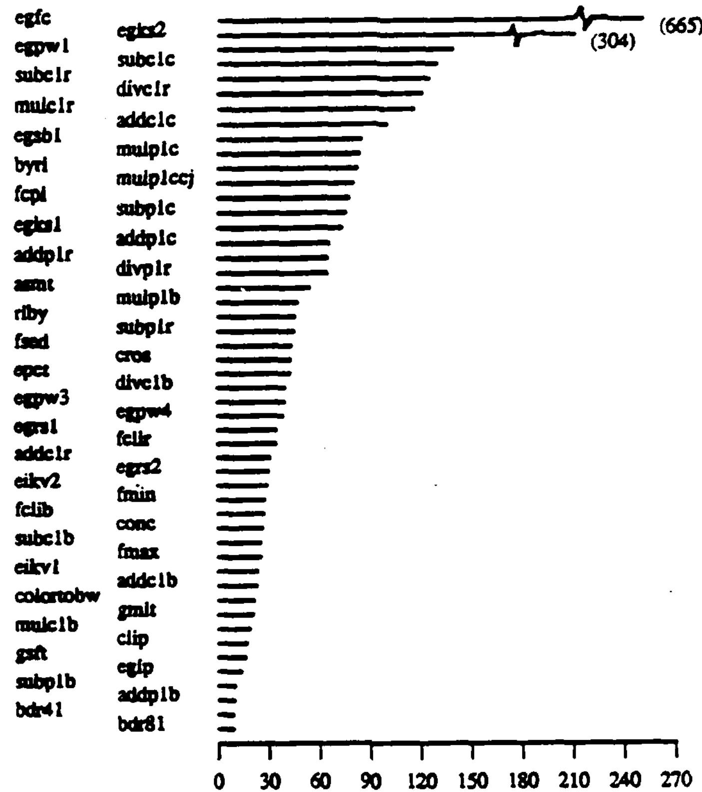


Figure 3-5: Ratio of execution times of Sun Apply code to Warp Apply code.

3.4.2 Warp Compared with Hughes HBA

The Hughes HBA and Warp satisfy very different applications requirements. One is a machine specifically designed for image processing, with a special video interface, and all high-speed I/O through a frame buffer; the other is a machine interfaced to a general-purpose external host, which is suited for scientific computing and signal processing as well as image processing. The high-speed floating point in Warp is largely a reflection of the desire to satisfy all of these applications areas.

The HBA times are measured from the time the image is available for processing in the frame buffer of the HBA to the time the output image is stored there. At the time of this study, the HBA processed 240×256 images; to be consistent with the Warp times, the HBA times have been multiplied by 4.27. The Warp times are for 512×512 images, measured as before.

We have done only preliminary work on the comparison between the Hughes HBA and Warp. Only a few programs have been tested, and most of those are integer applications, biasing the data against Warp, since it has higher-speed floating point than the HBA. Taking this into account, we can study the data shown in Figure 3-7 and 3-8:

- The Warp times are, on average, 3.2 times better than the HBA (the median is also 3.2). This reflects the greater total computational power of the Warp compared with the HBA, together with the application bias towards conditional, integer applications – the HBA can execute approximately 25 MFLOPS versus the Warp's 100 MFLOPS.
- In the fmin and fmax algorithms, which compute the minimum and maximum of a 3×3 window around each pixel, the HBA time is slightly better than the HBA time. This is because in such a highly conditional operation, the use of the long pipeline inside the Warp cell is a barrier to good performance.

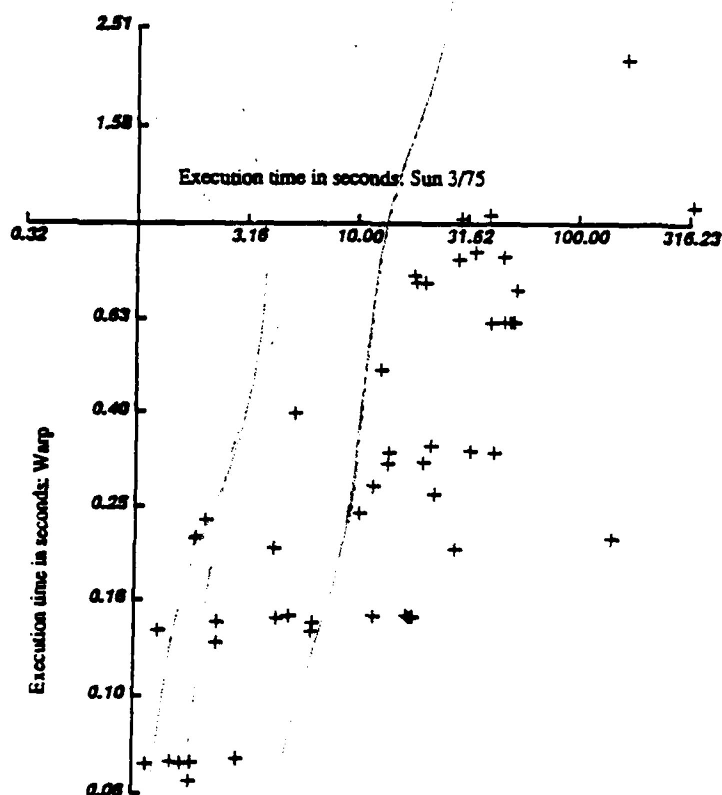


Figure 3-6: Scatter diagram of execution times of Sun Apply code and Warp Apply code.

Moreover, the total number of ALU operations in the HBA is greater, since there are 24 processors instead of 10, of comparable integer performance.

3.5 Conclusions

This is the first study which we are aware of in which highly diverse architectures have been compared using the same source code. We can make several conclusions based on this study:

- Apply and WEB are clearly good tools for comparing these architectures. Quite apart from the utility of having Apply and WEB available on an architecture, which is considerable, using the same source code eliminates many factors that significantly affect performance but which are irrelevant to performance analysis. Apply is easy to implement on a parallel processor, which makes it possible to evaluate the performance of a large number of parallel machines with little effort. We look forward to evaluating other parallel architectures in the future.

- The main performance limitation of the Apply programming model is the inability to manipulate constant-type parameters once per image rather than once per pixel. This deficiency will have to be corrected in future versions of Apply or its successors.

- Most of the performance limitations of one architecture over another arise from intra-processor characteristics, rather than inter-processor characteristics. This is because this level of vision is easy to parallelize, so that different processors need to communicate very little. Much more significant is the ability of the processor to successfully implement the wide range of operations that is required in low-level vision, including integer, floating point, and conditional operations.

- Parallel processors deliver performance increases even over high-performance workstations at this level of vision, and Apply makes them no harder to program. The performance ratios vary from ten- to hundred-fold increases. This is a significant, cost-effective, performance increase.

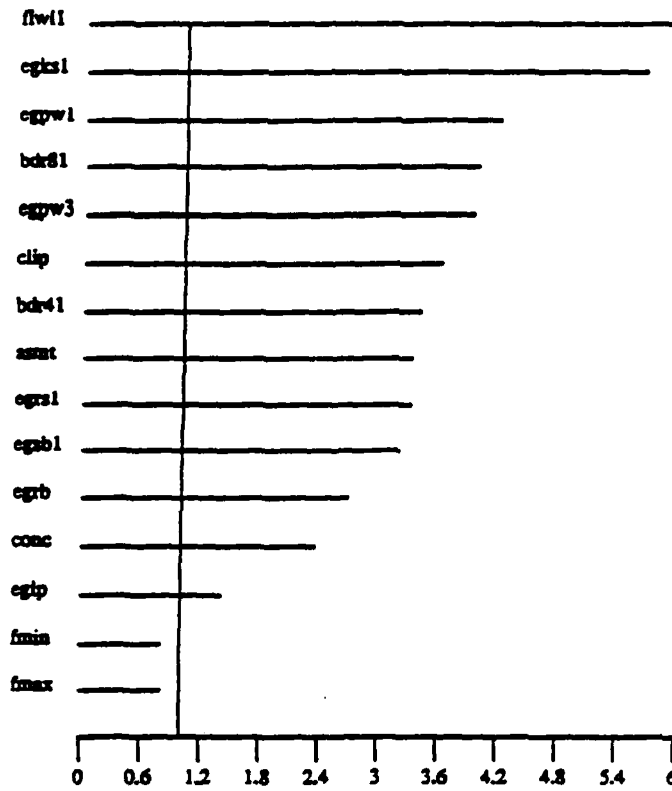


Figure 3-7: Ratio of execution times of Hughes HBA Apply code to Warp Apply code.
Vertical line indicates a ratio of one.

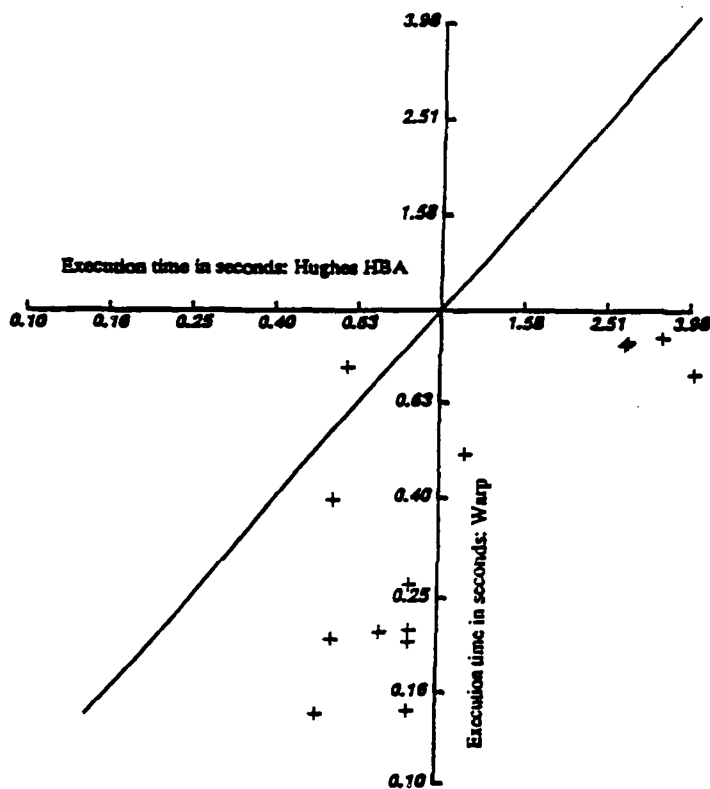


Figure 3-8: Scatter diagram of execution times of Hughes HBA Apply code and Warp Apply code.
Diagonal line indicates equality.

4. The WEB Library

4.1 Introduction

WEB is a basic library, based on the Spider library, for image processing on Warp. It consists currently of 134 programs, covering the following areas:

- Basic image operations: add, subtract, multiply, divide images by images and images by constants, assign zeroes, assign constant inside region.
- Conversions: byte to real, real to byte, polar to cartesian.
- Image grayvalue operations: clip, threshold, remap grayvalues, reduce graylevels.
- Image features: measure area of regions, center of gravity, circumscribing rectangle, histogram, moments, perimeter of regions.
- Edge detection: Roberts, Frei and Chen, Kirsch, Sobel, Laplacian, Prewitt, Robinson, Kasvand.
- Convolution: convolution with a given weight window and by a constant. Convolution and correlation using FFT.
- Smoothing: adaptive local smoothing, median filtering, local maximum and minimum, iterative enhancement, texture image processing.
- Orthogonal transformations: FFT, DCT.
- Warping: quadratic, affine.
- Pattern generation: checkerboard, stripe, bull's eye, diamond, grid.
- Multi-level image processing: generate pyramid, reduce by half, double.
- Binary image processing: detect borders, compute image of boundary points, connectivity, crossing, expand or contract, shrink components.
- Color conversion: color to black and white.

Approximately 80% of the routines are written in Apply, and the rest in W2. All of the Apply routines can be recompiled easily for W2 or C (Sun/Unix) code generation, and for different image sizes and number of cells. The W2 programs have been written to use macros, in a way that makes it possible to change image sizes and number of cells easily in most cases. As compiled, the WEB library does 512×512 image processing on a 10-cell Warp array.

4.2 Calling Programs in WEB

Any of the programs in WEB that are written in W2 or Apply can be called from C using `warp_call`. Parameters to `warp_call` are the file name of the program, and the data parameters to be passed in and out of Warp. The order of the parameters is given in Appendix I.

For any image parameter, a generalized image (type `IMAGE *`) can be passed to `warp_call`. The actual type and size of the generalized image can be of any type whatsoever. However, `warp_call` will process only a 512×512 region of the image, if is larger than that, and will pad the image with zeroes to produce a 512×512 image if it is smaller than that. Moreover, for reasons of efficiency, the user may want to manage the memory storage class and type of the image. For example, if a byte image is passed to a program that expects a real image, `warp_call` automatically converts it, using a C routine. The user may wish to use the WEB routine `byrl` instead. Also, all generalized images are converted to Warp generalized images before being passed to `warp_call`; this results in the image being copied to Warp's external host memory. For short programs, this can be inefficient, and the user may wish to create Warp images using `i_warp_image` or `i_warpcreat` instead.

`warp_call` expands environment variables in filenames, so that it is easy to write code that works no matter where the WEB library is stored. For example, the following code converts a byte image into a real image. The

byte image is the generalized image `in`, and the real image is the generalized image `out`:

```
warp_call("$WPEweb/byrl/byrl", GIMAGE, in, GIMAGE, out);
```

The following call applies two-dimensional convolution to an image. The input is in `in`, the output is in `out`, and the weight matrix is the matrix of floats `weights`:

```
warp_call("$WPEweb/flwl0/flwl0", GIMAGE, in, OTHER, weights,
                                 GIMAGE, out);
```

As documented in the man page, `warp_call` also takes parameters that are memory pointers or cluster memory descriptors. Any parameter that is an image can also be passed as an ordinary two-dimensional array of data using these methods. However, this array will not automatically be converted by `warp_call`, nor will the bounds be checked.

Programs in WEB that are written in W2 or Apply can also be executed in the Warpshell using `w2-execute`. Parameter types are defined as in Appendix I.

4.3 Classification by Area

The programs in the library are distributed among the areas mentioned in the Introduction section as follows:

- Basic Operations: *addclb, addclc, addclr, addcls, addplb, addplc, addplr, addpls, divclb, divclr, divcls, divplb, divplr, divpls, fclib, fclir, flog, fsed, mulclb, mulclc, mulclr, mulcls, mulplb, mulplc, mulplccj, mulplr, mulpls, rplalb, rplalr, rpla2, subclb, subclc, subclr, subcls, subplb, subplc, subplr, subpls, tferlb, tferlr.*
- Conversions: *byrl, fcpl, rlby.*
- Grayvalue Operations: *clip, gmlt, gsft, gtrnl, log, pted, rqnt, sclp, slthl, slth2, slth2m, slth3.*
- Image Features: *areal, cgrvl, cqltl, crcll, ersr3, histl, mmntl, mmnt4, prmtl, sizel.*
- Edge Detection: *egfc, egksl, egks2, eglp, egpr, egpwl, egpw2, egpw3, egpw4, egrb, egrsl, egrs2, egrs3, egsbl, egsb2, eikvl, eikv2.*
- Convolution/Correlation: *fcon, fcor, flwl0, flwl1, flwl2, xconv, yconv.*
- Smoothing: *asmt, fmax, fmin, itenl, iten2, medi, temx2, tepa, txav, txav2, txdfl, txdf2, txeg2.*
- Transforms: *dct, fft.*
- Warping: *afinl, afin2, afin3, nolnl, noln2, noln3.*
- Pattern Generation: *pgenl, pgen2, pgen3, pgen4, pgen5.*
- Multi-Level Image Processing: *expand, pyramid, reduce.*
- Binary Image Processing: *bdr41, bdr81, bflpl, conc, cros, epct, grassfire, srnkl, srnk2, srnk3.*
- Miscellaneous: *colortobw, mag, magdir, nonmax, sumrcb, sumrcr.*

5. Performance of Warp on the DARPA Image Understanding Architecture Benchmarks

5.1 Introduction

The DARPA Architecture Workshop Benchmark Study was conceived for these reasons:

- To arrive at an initial understanding of the general strengths and weaknesses for image understanding (IU) of the architectures represented.
- To project needs for future development of architectures to support IU.
- To promote communication and collaboration between various groups within the CS community which are expected to contribute to development of real-time IU systems.

The benchmarks chosen represented common image processing operations from low and middle level vision, but did not include high level image processing operations, such as recognition; these operations were felt to be too ill defined at present to properly evaluate machine architectures.

Warp was one of the participants in the study. This paper is a summary of our results, which reflect the performance on Warp on this level of vision, and can also serve as a guide for programming Warp in this area.

The precise definition of the image processing operations as given to the participants was as follows:

1. **Laplacian.** (Edge detection is done by this and the following two tasks. For edge detection, the input is a 8-bit digital image of size 512×512 pixels.) Convolve the image with an 11×11 sampled "Laplacian" operator [15]. (Results within 5 pixels of the image border can be ignored.)
2. **Zero-crossings Detection.** Detect **zero-crossings** of the output of the operation, i.e. pixels at which the output is positive but which have neighbors where the output is negative.
3. **Border Following.** Such pixels lie on the borders of regions where the Laplacian is positive. Output sequences of the coordinates of these pixels that lie along the borders. (On **border following** see [27, Section 11.2.2].)
4. **Connected component labeling.** Here the input is a 1-bit digital image of size 512×512 pixels. The output is a 512×512 array of nonnegative integers in which
   a. pixels that were 0's in the input image have value 0.
   b. pixels that were 1's in the input image have positive values; two such pixels have the same value if and only if they belong to the same connected component of 1's in the input image. (On connected component labeling see [27, Section 11.3.1].)
5. **Hough transform.** The input is a 1-bit digital image of size 512×512. Assume that the origin (0,0) image is at the lower left-hand corner of the image, with the x-axis along the bottom row. The output is a 180×512 array of nonnegative integers constructed as follows: For each pixel (x, y) having value 1 in the input image, and each i, $0 < i < 180$, add 1 to the output image in position (i, j), where j is the perpendicular distance (rounded to the nearest integer) from $(0, 0)$ to the line through (x, y) making angle i-degrees with the x-axis (measured counterclockwise). (This output is a type of Hough transform; if the input image has many collinear 1's, they will give rise to a high-valued peak in the output image. On Hough transforms see [27, Section 10.3.3].)
6. **Convex Hull.** (For this and the following two geometrical constructions tasks the input is a set S of 1000 real coordinate pairs, defining a set of 1000 points in the plane, selected at random, with each coordinate in the range $[0, 1000]$. Several outputs are required as follows.) An ordered list of the pairs that lie on the boundary of the convex hull of S, in sequence around the boundary. (On convex hulls see [26, Chapters 3-4].)
7. **Voronoi Diagram.** The Voronoi diagram of S, defined by the set of coordinates of its vertices, the set of pairs of vertices that are joined by edges, and the set of rays emanating from vertices and not terminating at another vertex. (On Voronoi diagrams see [26, Section 5.5].)
8. **Minimal Spanning Tree.** The minimal spanning tree of S, defined by the set of pairs of points of S

that are joined by edges of the tree. (On minimal spanning trees see [26, Section 6.1].)

9. **Visibility.** The input is a set of 1000 triples of triples of real coordinates $((r,s,t),(u,v,w),(x,y,x))$, defining 1000 opaque triangles in three-dimensional space, selected at random with each coordinate in the range [0,1000]. The output is a list of vertices of the triangles that are visible from (0,0,0).

10. **Graph matching.** The input is a graph G having 100 vertices, each joined by an edge to 10 other vertices selected at random, and another graph H having 30 vertices, each joined by an edge to 3 other vertices selected at random. The output is a list of the occurrences of (an isomorphic image of) H as a subgraph of G. As a variation on this task, suppose the vertices (and edges) of G and H have real-valued labels in some bounded range; then the output is that occurrence (if any) of H as a subgraph of G for which the sum of the absolute differences between corresponding pairs of labels is a minimum.

11. **Minimum-cost path.** The input is a graph G having 1000 vertices, each joined by an edge to 100 other vertices selected at random, and where each edge has a nonnegative real-valued weight in some bounded range. Given two vertices P, Q of G, the problem is to find a path from P to Q along which the sum of the weights is minimum.

In what follows, we first describe the current Warp status, and then describe our work on each of the algorithms. We do not review the Warp architecture or programming environment here, since complete reviews are available elsewhere [2, 3, 5, 8, 9].

5.2 Warp Status

At the time this comparison was done, there are three operating Warp machines at Carnegie Mellon. Two of them were prototypes. One was built by General Electric Radar Systems Department (Syracuse) and the other by Honeywell Marine Systems Department (Seattle). Both consist of a linear array of ten cells, each giving 10 MFLOPS, for a total of 100 MFLOPS, and operate in an identical software environment. These machines are referred to as WW Warp, since they are of wirewrap construction. The machines are fed data by MC68020 processors, called the "external host," and the whole system is controlled from a Sun 3/160.

The third machine was a production machine, one of several being constructed by General Electric Corporation. (Currently, all the Warp computers in existence are of this type; there are two of The production machines are built from printed-circuit boards, and are called PC Warp. The baseline power of these machines is also 100 MFLOPS, although they can easily be expanded to 160 MFLOPS by simply adding more cells. (The array can be expanded still further, but this requires a special repeater board and a second rack). The PC Warp is changed in several ways from the WW Warp: cell data and program memories are larger, there is on-cell address generation, and there is a large register overflow file to provide a second memory for scalars. Some of these improvements imply an increased speed on some of the benchmarks, as will be noted. For example, because of on-cell address generation, the cells is able to tolerate an arbitrary skew in computation, which makes it possible to overlap input, computation, and output in many algorithms. Also, improved processor boards in the external host allow improved I/O rates between Warp and the host through DMA, removing the host I/O bottleneck in many cases. Finally, since each cell has more local control, it is possible to make Warp computation more data dependent, by allowing data-dependent I/O between cells, as well as heterogeneous computation (different programs on different cells).

Carnegie Mellon and Intel Corporation are developing the "integrated" version of Warp, called *i* Warp. In this machine, each cell of Warp will be implemented on a single chip. The clock rate will be increased so that each chip will support at least 16 MFLOPS computation, as opposed to 10 MFLOPS in WW and PC Warp. In the baseline machine the cells will be organized into a linear array of 72 cells, giving a total computation of 1.152 GFLOPS. In the following analysis, it has been assumed that each *i* Warp cell can do everything a PC Warp cell can, with an increase of 1.6 in speed (this is a design goal). When I/O bottlenecks have led to a maximum performance time on a benchmark, this has been noted.

All the benchmarks listed below as being implemented on Warp are written in W2, the Warp programming language. W2 is a procedural language, on about the same level as C or Pascal. Arrays and scalars are supported, as are `for` loops, and `if` statements. The programmers are aware that they are programming a parallel machine, since

each program is duplicated to all cells and then executed locally (with local sequencing) on each cell.

5.3 Vision Programming On Warp

We have studied vision programming at various levels on Warp for some time now, and developed and documented several different models [12, 24]. In this section we briefly review the various models of Warp programming, for reference in later sections.

All the programs in this paper use the cells in a *homogeneous* programming model: that is, all cells execute the same program, although the program counters on the different cells can differ, and each has its own local data memory. This is a restriction imposed by the hardware of WW Warp. Programs on PC Warp need not follow this restriction.

5.3.1 Input Partitioning

In this model, which is used for local operations like convolutions, the image is divided into a number of portions by column, and each of the ten cells takes one-tenth of the image. Thus, in 512×512 image processing cell 0 takes columns 0-51 of the image, cell 1 takes columns 52-103, and so on (a border is added to the image to take care of images whose width is not a multiple of ten). The image is divided in this way because it makes it possible to process a row of the image at a time, and because the host need only send the image in raster-order, which is important because the host tends to be a bottleneck in many algorithms.

5.3.2 Output Partitioning

This model is used for algorithms in which the operation to be performed is global, so that any output can depend on any input, but can still be computed independently. In this model, each cell sees the complete input image, and processes it to produce part of the output. Generally, the output data set produced by a cell is stored in the cell's local memory until the complete input image is processed. Hough transform is implemented in this way.

5.3.3 Pipelining

In this model, which is the classic type of "systolic" algorithm, the algorithm is divided into regular steps, and each cell performs one step. This method can be used when the algorithm is regular. (Because the cell code must be homogeneous, this method is of less use on the wire-wrap Warp machine than it usually is in systolic machines). When this method can be used, it is generally more efficient in terms of input and output overlap with computation and local memory use than either of the two models above.

5.4 Laplacian

Laplacian. Convolve the image with an 11×11 sampled "Laplacian" operator [15]. (Results within 5 pixels of the image border can be ignored.)

The Laplacian given [15] is symmetric, but not separable. (Separable filters can be computed more efficiently, in general, than non-separable filters). In this section we describe a series of optimizations we applied to the Laplacian filter in the Warp implementation, which led to an efficient implementation. These optimizations can be applied to any symmetric filter, and will lead to efficient implementations on many different computer architectures.

Since most filters use masks with an odd number of rows and columns, the rest of this discussion will deal with this case. Let the size of the mask be represented by $N = 2M + 1$.

In order to see where the optimizations come from, we first notice that an unoptimized $N \times N$ convolution takes N^2 multiplies and $N^2 - 1$ additions per pixel. A separable convolution of the same size would take only $2N$ multiplications and $2(N-1)$ additions.

One way to compute the Laplacian is to compute it as a series of column convolutions. Each column takes N

multiplications and $N-1$ additions, and then $N-1$ additions are required to add all of the partial sums. The total number of multiplications is $N \times N = N^2$, and the number of additions is $N \times (N-1) + (N-1) = N^2 - 1$.

Due to symmetry, we can add the pairs of corresponding pixels within a column before multiplying them by the weights, as shown in Figure 5-1. Each of the N columns contains M pixels that can be added in this way, and one pixel in the middle which is not part of a pair. We call this column of $M+1$ pixels a "folded" column. After the multiplication, the pixels in each folded column must be added, and then all the columns must be added as before. This saves multiplications, but not additions: the number of multiplications is $N(M+1) = (N^2+N)/2$, while the additions sum to $N \times M + N \times M + (N-1) = N^2 - 1$.

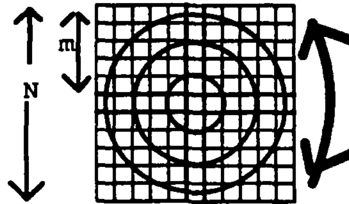


Figure 5-1: Folding columns

Now note that calculations for a given pixel can share partial results with neighboring calculations in the same row. As we shift the convolution window from the left to the right one step, we can retain all but one of the folded columns from the previous convolution, and sum just one new folded column, as shown in Figure 5-2. The rest of the algorithm is unchanged. Multiplications are unaffected, but additions are reduced almost by half, to $M + N \times M + (N-1) = (N^2 + 3N - 3)/2$.

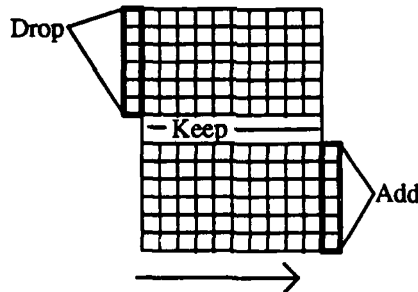


Figure 5-2: Using results from previous steps

Finally, we notice that the column convolutions are not done with N unique column weights, but rather with $M+1$ unique weights. As we shift the window to the right, we can compute and store the convolution of the new column with all $M+1$ column weights, as shown in Figure 5-3. Then, as we shift the window up to N pixels to the right, we will only have to add the appropriate convolved column sums, as shown in Figure 5-4. Thus again, nearly half of the multiplications and additions can be saved. Thus for each pixel, only $M+1$ partial weighted column sums need be generated, and then $N-1$ additions are required to add the proper partial sums together. The number of multiplications is then $(M+1) \times (M+1) = ((N+1)/2)^2$, while the additions come to $(M+1) \times M + M + (N-1) = M^2 + 4 \times M$

Table 5-1 summarizes our result by comparing the number of multiplications and additions by our method with N^2, the number required for an unoptimized kernel, and $2N$, the number for a separable kernel:

An algorithm based on the above model was implemented using input partitioning on the WW Warp, and gave a

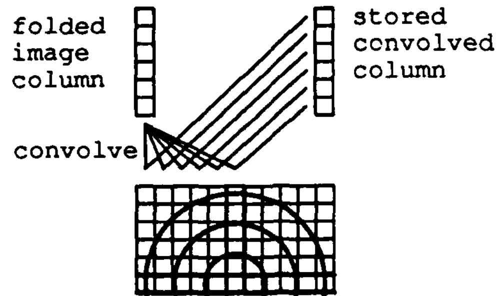


Figure 5-3: Convolving and storing column sums

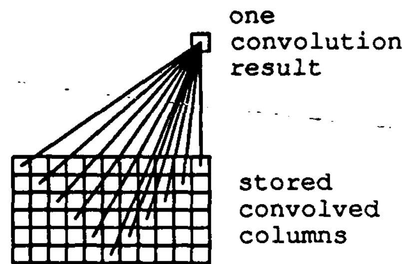


Figure 5-4: Adding appropriate column sums

Mask Size	Multiplications	Additions	N^2	$2N$
3×3	4	5	9	6
5×5	9	12	25	10
7×7	16	21	49	14
9×9	25	32	81	18
11×11	36	45	121	22
15×15	64	77	225	30
25×25	144	192	625	50

Table 5-1: Optimized Symmetric Convolution

runtime of 432 milliseconds. The same algorithm was compiled for the PC Warp, and gave a runtime of 350 milliseconds. The change was due to overlap of I/O with computation in PC Warp, which is not possible for this algorithm on the WW Warp. On *i* Warp, assuming a straightforward speedup arising from a 72-cell array with a 16 MHz clock, the time will be 30 milliseconds.

5.5 Zero Crossings Detection

Zero-crossings Detection. Detect zero-crossings of the output of the operation, i.e. pixels at which the output is positive but which have neighbors where the output is negative.

Zero crossing was implemented using the input partitioning model. A three by three window was taken around each pixel. If any elements of the window were negative, but the central pixel was positive, a zero crossing was declared and a "1" was output, otherwise "0" was output. This computation was performed by transforming each 9-element window into a 9-bit integer, with which a table lookup was performed. Input and output were represented as 8-bit pixels. Execution time on the WW Warp was 172 milliseconds; on the PC Warp the time will be

approximately 92 milliseconds, due to overlap of I/O with computation. On *i*Warp, the time will be limited by I/O bandwidth to the array to at least 7.8 milliseconds.

In many cases, it is desirable to perform the Laplacian and zero crossing computations in sequence, without saving the results of the Laplacian. In this case, on *i*Warp, the computation can be done more quickly than by performing each individually. We estimate that such a computation will take 31 milliseconds, fast enough for video rate image processing.

5.6 Border following

Border Following. Output sequences of the coordinates of pixels that lie on the borders of regions where the Laplacian is positive. (On **border following** see [27, Section 11.2.2].)

The algorithm is mapped in two steps. First each Warp cell performs the border following technique on part of the image. Then, these partial results are then combined within the array to produce the complete border trace for the image. The full algorithm is:

- Each cell sends its bottom row to its successor.
- Starting with the bottom row, on the left, each cell inspects the pixels on this row. If the pixel is turned on, the cell begins to trace this connected component. As it traces the component, it builds a list of of its pixels to the next cell. As it visits pixels, it turns them off, so they will not be visited on scans of higher rows.
- Either this component extends to the cell's top row, or it does not. If not, then the list of pixels eventually terminates within the cell's strip; the cell queues the whole list of pixels for output to the next cell, marking the component as complete. But if the component extends to the top row, it may join with a component of the preceding cell. The cell checks its copy of the previous cell's last row to see if this is a possibility. If not, again the list may be passed to the next cell. But if it is, the cell stacks the list it has built so far, and begins processing another component, bottom to top.

This completes the parallel phase of the computation. Each cell now has two lists of borders: those ready for output, and those that must be merged with borders in preceding cells. The cells now run the following merge phase.

- Each cell tries to do two things: (1) empty its ready-for-output queue, and (2) move all the components on its stack to this queue. Operation (1) happens asynchronously, depending upon the next cell's input queue. Operation (2) is performed as follows.
- Eventually, the preceding cell will emit a list of the component touching the stacked component. When this happens, the component may be unstacked, the stacked pixels attached to the proper end of the list received from the previous component (note that this may involve attaching lists to both ends), and pass the now completed list at least, complete in its path through the given cell and its predecessor to the ready-for-output queue.

This algorithm must terminate, since the first cell never has any stacked components. Hence it will eventually flush all the components on its output queue to the second cell, giving the second cell all the information it needs to move all its stacked components to its output queue. By iterating this argument, it follows that each cell must eventually clear its stack and then its output queue.

Finally, we must provide a time estimate for this algorithm. The first step is essentially a connected components computation. This will take no longer than the parallel step of a UNION-FIND based connected components program below. For PC Warp, with 10 cells, this is 73 milliseconds; for *i*Warp, with 72 cells, this is 6.3 milliseconds. These estimates were obtained by dividing the uniprocessor time of the Hughes HBA [32] implementation of a pure UNION-FIND algorithm by the number of cells, and again by suitable numbers to correct for processor speed.

The second step is a serial merge (in the worst case). We estimate this step will take about 1.02 second for PC Warp, and 690 milliseconds for *i*Warp. These estimates are based on our experience with similar merge steps for

the connected components algorithm, and the i/o bandwidth of each machine. Hence our estimates are:

PC Warp: 1.1 seconds
i Warp: 690 milliseconds

5.7 Connected components labelling

Connected component labeling. Here the input is a 1-bit digital image of size 512×512 pixels. The output is a 512×512 array of nonnegative integers in which

1. pixels that were 0's in the input image have value 0.
2. pixels that were 1's in the input image have positive values; two such pixels have the same value if and only if they belong to the same connected component of 1's in the input image. (On connected component labeling see [27, Section 11.3.1].)

In this section we present our parallel-sequential-systolic algorithm for this computation, our timings of a C simulation of the algorithm, and our estimates of its execution time on Warp, PC Warp, and *i* Warp.

Section 5.7.1 gives the algorithm. Section 5.7.2 presents the asymptotic running time of the parallel–sequential–systolic algorithm. We also show how to modify this work to get a parallel–sequential–parallel algorithm, and give its running time. Section 5.7.3 discusses the implementation, covering both our existing C simulation and our planned Warp implementations; here we give the actual execution time of the simulations and the estimated execution times for the Warp implementation, and discuss the constraints imposed by the Warp architecture.

5.7.1 Sketch of the Algorithm

5.7.1.1 Vocabulary and Notation

The input to the algorithm is a $N \times N$ array (512×512 in this case) of binary pixels. A 1-valued pixel is called *significant*; all others are *insignificant*. We label the rows and columns consecutively from 0 to N-1, starting in the upper-left-hand corner. The *4-neighbors* of a pixel are the pixels that lie immediately above, below, left and right of it; its *8-neighbors* are the eight pixels that surround it. Two significant pixels x and y lie in the same *connected 4-component (connected 8-component)* of the image iff there is a sequence of significant pixels $p_0, \ldots, p_n$ with $p_0=x$, $p_n=y$, and p_{i-1} a 4-neighbor (8-neighbor) of p_i for each $i=1, \ldots, N$. The algorithm we present here computes connected 4-components. It is straightforward to modify it to compute connected 8-components; the timing estimates we present later are for the connected 8-component version.

Our algorithm executes on a linear systolic array of K processing cells, numbered consecutively from 0 to $K-1$. Each cell processes a set of adjacent rows of the image, called a *slice*. We assume that K divides N, and that the slices are of uniform size N/K rows. The 0th cell processes the first N/K rows of the image, called slice 0, and so on. When data flows from cell i to cell $i+1$, we will say it crosses the $i, i+1$ *boundary*, or simply, an *inter-cell boundary*. A cell's *label space* is the set of all labels that it may assign to any pixel; cell i's label space is denoted L_i. We choose suitable bounds on the label spaces so that they are guaranteed disjoint.

5.7.1.2 The Algorithm

The algorithm proceeds in three phases: parallel, sequential, and systolic.

In the parallel phase, each cell computes labels for its slice of the image.

In the sequential phase, computation proceeds serially over each $i-1, i$ boundary, for $i=1, \ldots, K$. The ith stage of this computation effectively passes information about the connectivity of slices 0 through $i-1$ to slice i. The actual computation consists of scanning the $i-1, i$ boundary to construct two maps, which record connectivity information, then applying the second of these maps along the bottom row of slice i to propagate this information downward. Note that after this phase finishes, lower-numbered slices still lack information about higher-numbered

slices. We perform this computation in K serial steps because of the limited interconnection topology of Warp.

In the systolic phase, the labels are pumped out of the cell array. As each label crosses into or out of a cell, the cell applies the maps generated in the sequential phase. Since the labels assigned to slice i must pass through cells i+1, . . . , K, this permits higher-numbered cells to modify the labels assigned by lower-numbered cells, completing the computation. Each phase of the algorithm is explained in greater detail below.

Parallel Phase. In this phase, each cell computes preliminary labels for its slice of the image. These labels are drawn from the cell's label space, which are guaranteed not to be used by any other cell. We use a modification of the Schwartz-Sharir-Siegel algorithm [28], which runs in linear time in the size of the slice.

Sequential Phase. In this phase, processing proceeds sequentially in $K-1$ stages over each of the $K-1$ inter-cell boundaries. The function of stage i, when we compute along the $i-1,i$ boundary, is to pass information about the connectivity of slices 0 to $i-1$, inclusive, to cell i. This information is recorded in the two maps that are built for each boundary. Cell i builds the maps for the $i-1,i$ boundary. We call the first map σ_i; it is used by cell i to relabel pixels when they enter the cell. We call the second map ϕ_i; it is used by cell i to relabel the pixels when they leave the cell.

The maps have intuitive meanings, as follows. Each ϕ_i tells how to relabel the pixels of slice i to make them consistent with the connections in the i–1 preceding slices. Specifically, suppose x and y are two significant pixels of slice i such that there is a path from x to y that passes through slices 0 to i, but no path that lies entirely within slice i. Then after the parallel phase, x and y will bear distinct labels. However, ϕ_i is constructed such that $\phi_i(x) = \phi_i(y)$ iff there is a path from x to y that lies wholly within slices 0 through i. Thus ϕ_i encodes the influence of slices 0 through i–1 on slice i.

Similarly, σ_i contains information about connectivity across the $i-1,i$ boundary. Let w and v be significant pixels on the bottom row of slice i–1, and let x and y be significant pixels on the top row of slice i, such that w and x are adjacent, and v and y are adjacent. Suppose that x and y are connected by a path that lies wholly within slices 0 through i, but that w and v are not connected by any path that lies wholly within slices 0 through i–1. Then after the parallel phase, w and v will bear distinct labels. However, σ_i is constructed such that $\phi_i(x)=\phi_i(\sigma_i(w))=\phi_i(\sigma_i(v))=\phi_i(y)$. Thus σ_i encodes the influence of slice i on slices 0 through i–1.

These maps are constructed by the following procedure. We use some special notation. Let $f: M \rightarrow N$; then f is a subset of $M \times N$. We write $f+\langle m,n \rangle$ for the function obtained by deleting the pair $\langle m, f(m) \rangle$ from f and adding the pair $\langle m,n \rangle$ to the resulting set. For the purposes of the UNION-FIND portion of the algorithm, we assume that each $l \in L_i$ lies in a singleton set $\{l\}$ that bears the name l. We also assume that each map is initialized to the identity map.

```
for i = 1 to K do begin
        get B, the bottom row of slice i-1
        get T, the top row of slice i
        for col = 0 to N-1 do
                if B[col] and T[col] are significant then
                        Call Update(B[col], T[col])
        for col = 0 to N-1 do
                if T[col] is significant then begin
                        φi = φi + <T[col], FIND(T[col])>
                end
        if i ≠ K then apply φi to the bottom boundary of slice i
end

procedure Update(PrevCell, CurrCell)
begin
        if σi(PrevCell) = PrevCell then σi = σi + <PrevCell, CurrCell>
        else UNION(CurrCell, σi(PrevCell))
end
```

Note that each of σ_i, ϕ_i may be computed locally by cell i, requiring only the bottom row of slice $i-1$. This is not done in practice, because we want to use path-compression for the UNION-FIND computations, and the cells cannot implement this algorithm efficiently. Because the UNION-FIND operations are performed on the data structures that embody the ϕ_i, we will refer to these operations, when we are accounting for the algorithm's running time, as ϕ lookups and additions, or simply ϕ updates.

The correctness proofs for these algorithms are tedious and are omitted here (a correctness proof for a similar algorithm can be found in Kung and Webb [23]). It remains to show how these maps are used to compute the connected components of the entire image. This is done in the next section.

Systolic Phase. In this phase, the pixel labels are pumped out of the cells. Each significant pixel receives its final label through the following systolic labelling procedure. First, as a label enters cell i, crossing the $i-1, i$ boundary, it is passed through the map σ_i. Note that labels belonging to slice i do not cross this boundary, so are not mapped this way within cell i. Second, as a label leaves cell i, crossing the $i, i+1$ boundary, it is passed through the map ϕ_i. This happens whether the label was received from cell $i-1$, or originated within cell i.

It is not difficult to give an inductive proof that this procedure correctly labels the connected components of the image. However, we believe it is more illuminating to work through an example.

Figure 5-5 depicts the binary input to the algorithm. Here N=9, K=3. Significant pixels are marked "X." Rows and columns are numbered consecutively from 0, starting in the upper-left-hand corner; we give the coordinates of a pixel as (row, column). Figure 5-6 shows the labels for the significant pixels after completion of the parallel phase.

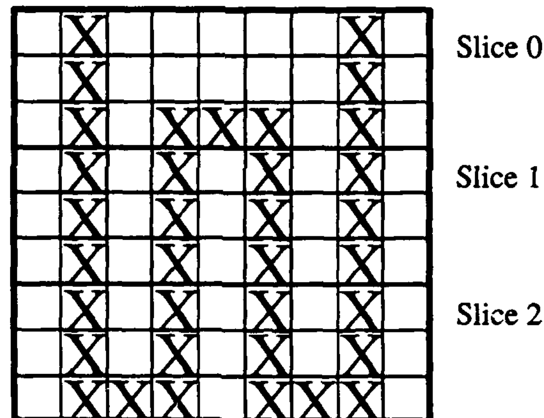


Figure 5-5: Input

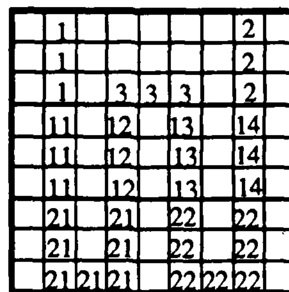


Figure 5-6: Labels after parallel phase

Now we work through the computation of ϕ_1 and σ_1. To begin, each map is initialized to the identity map on its domain. When we reach column 1, we note that pixels (2,1) and (3,1) are adjacent and significant. This prompts a call to Update, where we note that σ_1 fixes 1. Hence we add the pair <1,11> to σ_1. Likewise, we add <3,12> to σ_1. But on the call to Update for the pair of labels <3,13>, we note that σ_1 is not the identity on 3. Instead we take the UNION of {13} and {12}; creating the set {12,13} that bears the label 12. At column 7, we add <2,14> to σ_1. Finally, we perform FINDs on the labels that appear on the top row of slice 1 to determine ϕ_1 (it is the identity except for the ordered pair <13, 12>), and apply this map to the bottom row of slice 1. The resulting labels appear in Figure 5-7.

<table>
<tr><td></td><td>1</td><td></td><td></td><td></td><td></td><td></td><td>2</td><td></td></tr>
<tr><td></td><td>1</td><td></td><td></td><td></td><td></td><td></td><td>2</td><td></td></tr>
<tr><td></td><td>1</td><td></td><td>3</td><td>3</td><td>3</td><td></td><td>2</td><td></td></tr>
<tr><td></td><td>11</td><td></td><td>12</td><td></td><td>13</td><td></td><td>14</td><td></td></tr>
<tr><td></td><td>11</td><td></td><td>12</td><td></td><td>13</td><td></td><td>14</td><td></td></tr>
<tr><td></td><td>11</td><td></td><td>12</td><td></td><td>12</td><td></td><td>14</td><td></td></tr>
<tr><td></td><td>21</td><td></td><td>21</td><td></td><td>22</td><td></td><td>22</td><td></td></tr>
<tr><td></td><td>21</td><td></td><td>21</td><td></td><td>22</td><td></td><td>22</td><td></td></tr>
<tr><td></td><td>21</td><td>21</td><td>21</td><td></td><td>22</td><td>22</td><td>22</td><td></td></tr>
</table>

Figure 5-7: Labels after sequential phase

We perform a similar computation for the 2,3 boundary. The final maps appear in Table 5-2. (We do not display the ordered pairs associated with the identity portion of each map.)

ϕ_1	σ_1	ϕ_2	σ_2
<13, 12>	<1, 11> <3, 12> <2, 14>	<22, 21>	<11, 21> <12, 21> <14, 22>

Table 5-2: Final maps

Now we show how the systolic computation works. Consider pixel (2,4), with label 3. As it leaves cell 0, it passes through ϕ_0, which is the identity map. When it enters cell 1, it passes through σ_1, where it is relabeled 12; when it leaves cell 1, it passes through ϕ_1, which fixes 12. When it enters cell 2, it passes through σ_2, where it is relabeled 21; when it leaves cell 2, it passes through ϕ_2, which fixes 21. Hence the final pixel label is 21. Table 5-3 summarizes this computation on each of the labels in the example; note that each significant pixel has the same final label.

Labels				Action
1	2	3		enter slice 1, map through σ_1 to
11	14	12	13	leave slice 1, map through ϕ_1 to
11	14	12	12	enter slice 2, map through σ_2 to
21	22	21	21	leave slice 2, map through ϕ_2 to
21	21	21	21	final values

Table 5-3: Label Computation

5.7.2 Asymptotic Running Time

This discussion is divided into two parts. In the first, we give the running time of the parallel–sequential–systolic algorithm. In the second, we show how to transform this approach into a parallel–sequential–parallel algorithm, and give its running time.

5.7.2.1 Parallel-Sequential-Systolic Algorithm

The asymptotic running time of this algorithm is $O(N^2/K + KNG(N) + N^2) = O(KNG(N) + N^2)$, where G is the inverse Ackermann's function. The terms of this expression represent the running times of the parallel phase, the sequential phase, and the systolic phase respectively.

These expressions are obtained as follows. The parallel phase estimate is immediate, since it runs in linear time in the slice size, which is N^2/K.

For the sequential phase, observe that we must perform $K-1$ computations along boundaries. As we process a boundary, we will perform no more than $N/2$ additions to a σ-map, and no more than $N/2-1$ updates to a ϕ-map. Now no ϕ_i or σ_i will map more than $N/2$ elements of their respective domains away from themselves. If the σ_i maps are maintained as linear arrays, each σ_i operation takes constant time. If the ϕ_i maps are maintained as linear arrays with path compression, a sequence of N lookups and additions takes $O(NG(N))$ time. Hence the total time for this phase is $O(KNG(N))$.

For the systolic phase, observe that the last cell must perform a constant-time lookup on each pixel in the image, and all other cells may perform their lookups in parallel. Hence the time for this phase is $O(N^2)$.

Thus a pure systolic implementation has no asymptotic advantage over a linear-time uniprocessor algorithm. This statement is deceptive. Ultimately, any machine must run in $O(N^2)$ time on this problem, since it takes that much time to pump the data in and out. The advantage of the systolic algorithm is that it performs a useful constant-time computation step during the output, reducing the time spent in the sequential phase.

5.7.2.2 Parallel-Sequential-Parallel Algorithm

Here we introduce a straightforward modification of the algorithm to obtain an improved asymptotic time bound. It is based on the following simple observation: the systolic phase consists of computing, for each pixel p of slice i, the value of the composed function

$$\lambda_i = (\phi_K \circ \sigma_K \circ \cdots \circ \phi_{i+1} \circ \sigma_{i+1} \circ \phi_i)$$

Hence it suffices to compute each λ_i for $i = K, K-1, \ldots, 0$. But since

$$\lambda_{i-1} = \lambda_i \circ (\sigma_i \circ \phi_{i-1})$$

it is straightforward to compute the λ_i sequentially in $O(KN)$ time. (In fact, this can be done in parallel in $O(N\log(K))$ time.) Once this is done, cell i can obtain the *final* labels for slice i by applying λ_i to each significant pixel. Since this step can be performed in parallel among the cells, the running time of this modified algorithm is $O(N^2/K + KNG(N) + N^2/K) = O(N^2/K + KNG(N))$.

5.7.3 Implementation Details

In this section we discuss two implementations of this algorithm: a C-language Vax implementation and a Warp implementation. We begin with a sketch of the architectural constraints imposed by the various Warp machines; these constraints motivate some of the design decisions of both the Vax and Warp implementations. Then we discuss the Vax implementation, which was undertaken to learn about the algorithm in a familiar environment. We close with a treatment of the Warp implementation on each of Warp, PC Warp, and *i* Warp.

5.7.3.1 Warp Architectural Constraints

Two key factors determine most of the design decisions in a Warp implementation of this algorithm: the Warp cell's synchronization requirements and memory size. Let us consider these in turn.

Synchronization. The WW Warp requires compile-time synchronization. Thus an `if-then-else` statement will always take the time required by the slower of the two alternatives, and any loop must run for a fixed (maximal) number of iterations.

As a consequence, the WW Warp runs poorly on algorithms that exhibit good behavior only in the off-line sense. To see this, recall that techniques with good off-line performance – notably path-compression – derive their advantage by performing a few of the operations in a sequence slowly, so that the remaining operations in the sequence will be fast. But Warp forces each step of a procedure to take the time of the slowest possible alternative. Hence an implementation of an off-line algorithm will behave as if the most expensive operation were performed at each step of the sequence. Thus the algorithm with the best on-line behavior is always preferred.

This means that we must either abandon the path-compression approach to the sequential stage, or perform the sequential portion of the algorithm on a computation engine that does not have these constraints. Since there is no inherent advantage to performing the sequential phase on the cells (for there is no parallelism to exploit), and since the cost of shipping the necessary data to a suitable processor is low, we choose to do this phase on one of the Warp's MC68020-based cluster processors.

For similar reasons, we cannot improve the execution time by using sophisticated data structures to implement the σ-maps in the systolic phase. Unfortunately, this problem cannot be avoided. The best we can do here is use a data structure with good constant-time performance. This is discussed more fully below.

Memory Constraints. Our formulae for the asymptotic running times of these algorithms are based on the assumption of unit-access time to the data structures that hold the σ and ϕ maps. If memory is not a consideration, this speed can be attained by representing each map by a large array. The PC Warp and *i* Warp machines have enough cell memory to represent the maps this way. The speed estimates below for these machines are based on this assumption.

The WW Warp cell does not have enough memory to do this. Instead we must use an approach that gives good update and access times, with only moderate memory requirements. This is easy to do for the ϕ-maps. If each cell begins the assignment of the initial labels on the top boundary of its slice, the labels of this row will be drawn from the first $N/2$ elements of the slice's label space. Now note that though each ϕ_i is defined on the set $L_0 \cup \cdots \cup L_i$, which contains $(i+1)\lceil N/2 \rceil \lceil N/2K \rceil$ elements, ϕ_i will fix all of $L_0 \cup \cdots \cup L_{i-1}$, and also all of L_i, except possibly those elements of L_i that appear on the top row of slice i. Thus we can maintain ϕ_i as an array of size $N/2$, indexed by offset from the first element of L_i. To compute $\phi_i(r)$, we need only check to see if r lies in the range of interest, then find its offset and look up the value. This approach uses a small amount of memory, with only minor sacrifice of speed. Also, it is efficient for both the sequential phase of the computation, when the algorithm builds each ϕ_i using path-compression techniques, and the systolic phase, when the only operations are look-ups.

The situation is not as nice when we consider the σ maps. It is true that no σ_i will map more than $N/2$ elements away from themselves. This is because only labels that appear on the bottom row of slice $i-1$ may be moved by σ_i. However, these labels are no longer guaranteed to be drawn from some small subset of L_{i-1}. For instance, it is easy to construct an example so that labels drawn from both the first $N/2$ elements and the last $N/2$ elements of L_{i-1} will appear on the bottom row of slice i.

One solution is to maintain each σ_i as an array of ordered pairs, sorted by the first element of each pair. This permits lookup in worst-case $\log(N)$ time, and is well-suited to the systolic stage of the algorithm. In fact, it is the approach we use there. However, it does not permit fast addition to the map, and we must do both lookup and addition operations in the sequential stage. For this reason, we use the self-adjusting binary tree data structure to implement the σ_i in this stage. This data structure exhibits only good off-line performance. However, we use it only during the sequential phase of the calculation, when we build the map. The efficiency and simplicity of this data structure is another reason for doing the sequential calculation elsewhere than the Warp cells.

5.7.3.2 Vax Implementation

We have implemented the algorithm in C on a Vax 780, simulating the operation of a 10-cell Warp array. Each phase of the algorithm is implemented as one or more procedures, parameterized by cell number. A cell's local memory is represented by several large arrays; systolic communication is simulated by explicit data movement in and out of these arrays.

Note that the value of the σ maps themselves are never needed directly. We are interested only in the ϕ maps, and in the composition maps $\phi \circ \sigma$. For this reason we compute these compositions explicitly ahead of time. This way, each label that traverses a cell is mapped only once, through $\phi \circ \sigma$, rather than through σ and ϕ successively.

The simulation program processes a typical 512×512 image in about 4 1/2 minutes of CPU time. Fortunately, most of this represents the simulation of inter-cell communication.

To learn how the program was spending its time, we used the Unix *prof* [21] performance-monitoring program. The results are summarized in Table 5-4. The total is less than 4 1/2 minutes because the time for simulating communication is not included.

Phase	Time (seconds)	Time (seconds)
Parallel Phase	33	33
Sequential Phase		
Boundary Scan	.16	
ϕ Update	.10	
σ Update	.53	
$\phi \circ \sigma$ Computation	.06	
Total	.85	.85
Systolic Phase		
ϕ Lookups	3.8	
$\phi \circ \sigma$ Lookups	25	
Total	28	28
Total		62

Table 5-4: Vax implementation timings

5.7.3.3 Warp Implementations

We have not yet completed a Warp implementation. In this section we discuss the partial implementation for the WW Warp architecture, and give execution time estimates for the planned PC Warp and *i* Warp implementations. All our estimates are for the parallel–sequential–parallel version of the algorithm, computing connected 8-components.

WW Warp Our implementation for the WW Warp divides the computational burden between the linear systolic array and the cluster processors. The initial and final labellings are done by the systolic array; the sequential step is done by the cluster processors. This permits us to use algorithms with fast amortized time in the sequential step.

After the initial labelling, we would like to retain the initial results in cell memory, transmitting only each cell's boundary rows to the external host for generating the necessary maps. Unfortunately, the WW Warp cell memory is not large enough to hold a labelled slice, and barely large enough to hold the intermediate result required by the initial marking algorithm. This forces us to send the entire contents of each cell's slice to the external host as the labels are generated, then pump these slices back through the array for the final labelling.

We have written, but not yet debugged, all the code for the cell array. We have accurate estimates of the running time of this code, provided by the compiler. We have also estimated the running time of the sequential phase. We

derived this estimate from the sequential phase running time of the C implementation, allowing for a slight speed-up of the cluster processors over the Vax, and also for the extra work (computing the λ_i) done in this phase by the parallel–sequential–parallel version of the algorithm. The resulting estimate appears in Table 5-5.

Phase	Time (milliseconds)	Time (milliseconds)
Initial Parallel Phase		
Pump in Image	50	
Initial Labelling	2400	
Total	2450	2400
Sequential Phase		
Boundary Scan	150	
ϕ Update	90	
σ Update	490	
λ Computation	110	
Total	840	840
Final Parallel Phase		
λ Lookups	2200	
Pump Out Labels	50	
Total	2290	2300
Total		5600

Table 5-5: Estimated WW Warp timings

PC Warp and *i* Warp Architectures. In this section we derive estimated execution times for these architectures. There are three key differences between the design of these cells and those of the WW Warp. The first is that each cell has enough memory to maintain a full slice of labels. This means that we do not need to pump the intermediate labels to an external memory. The second is that the cells are not bound by the synchronization constraints of the WW machine. This means that the sequential phase computation can be performed on the cell array. This saves time because we no longer have to do i/o to the cluster processors for this phase, and because the cells run 2.8 times faster than the cluster processors. The third is that each of these machines is more powerful than the WW Warp. Both the PC Warp and the *i* Warp can do arithmetic directly on integers; this speeds up any integer arithmetic computation by a factor of 3. Furthermore, the *i*Warp cells run 1.6 times faster than the Warp and PC Warp cells.

The only other salient difference between PC Warp and *i* Warp, for our purposes, is that the *i* Warp contains 72 cells. Thus we can potentially attain more parallelism on *i* Warp. However, because the time taken in the merge phase varies linearly with the number of cells, while the time taken in each parallel phase varies inversely with this number, it is not necessarily best to use the greatest possible number of processors. If the execution time of the algorithm as a function of the number of cells is $T(K)=A/K+BK$, then the best time will be obtained with $K=\sqrt{A/B}$. In the case of *i* Warp, we have $A\approx 4.994, B\approx .00812$, so the best K is 25. The estimate below for *i* Warp execution time was made using this value.

The resulting estimates appear in Tables 5-6 and 5-7.

5.8 Hough transform

Hough transform. The input is a 1-bit digital image of size 512×512. Assume that the origin (0,0) image is at the lower left-hand corner of the image, with the x-axis along the bottom row. The output is a 180×512 array of nonnegative integers constructed as follows: For each pixel (x,y) having value 1 in the input image, and each i, $0<i<180$, add 1 to the output image in position (i,j), where j is the perpendicular distance (rounded to the nearest integer) from (0,0) to the line through (x,y) making angle i-degrees with the x-axis (measured counterclockwise). (This output is a type of Hough transform; if the input image has many collinear 1's, they will give rise to a high-valued peak in the output image. On Hough transforms see [27, Section 10.3.3].)

The Hough transform algorithm has been previously described [23]. Briefly, each of the ten cells gets one-tenth

Phase	Time (milliseconds)	Time (milliseconds)
Initial Parallel Phase		
Pump In Image	53	
Initial Labelling	710	
Total	760	760
Sequential Phase		
Boundary Scan	53	
ϕ Update	33	
σ Update	3	
λ Computation	41	
Total	130	130
Final Parallel Phase		
λ Lookups	36	
Pump Out Labels	53	
Total	89	89
Total		980

Table 5-6: Estimated PC Warp timings

Phase	Time (milliseconds)	Time (milliseconds)
Initial Parallel Phase		
Pump In Image	33	
Initial Labelling	191	
Total	224	224
Sequential Phase		
Boundary Scan	83	
ϕ Update	4.7	
σ Update	52	
λ Computation	63	
Total	200	200
Final Parallel Phase		
λ Lookups	9.0	
Pump Out Labels	33	
Total	42	42
Total		470

Table 5-7: Estimated *i*Warp timings

of the Hough array, partitioned by angle. The input image flows through the Warp array, and each cell increments its portion of the Hough array for all image pixels which are "1". Once the image has been processed, the Hough array is concatenated and output to Warp's external host.

For the particular parameters of this benchmark, which uses an array of 180×512 data, this requires each cell store 18×512=9 K words of data. This will not fit on the WW machine, which has a memory of 4K words/cell. But on PC Warp, each cell will have a memory of 32K words, so that the Hough array fits easily. On *i*.Warp 60 cells are used (60 being the largest number less than 72 which evenly divides 180), so that each cell needs to store only 3×512=1536 bytes of data.

In order to derive estimates, we implemented a Hough transform program (with a smaller number of angles than in the benchmark) and ran it on the WW machine. The algorithm does not change for more angles, so the estimates given by this method are accurate for the PC Warp with the benchmark parameters.

By derivation from this program, the time per pixel with value "1" is 13 microseconds. Assuming 10% of the image is one, on PC Warp the benchmark will execute in 340 milliseconds. On *i* Warp, the estimated execution time is 60 milliseconds. These times scale linearly with the number of "1"'s in the image.

5.9 Convex Hull.

Convex Hull. The input is a set S of 1000 real coordinate pairs, defining a set of 1000 points in the plane, selected at random, with each coordinate in the range [0,1000]. The output is an ordered list of the pairs that lie on the boundarv *of the convex hull of S, in sequence around the boundary. (On convex hulls see [26, Chapters 3-4].)*

R. A. Jarvis's [19] algorithm was used. This algorithm works as follows:

- Sort the points according to (x,y)-coordinate. The first point is a convex hull point. Call it A_0.
- Let i=0. Repeat the following until $A_{i+1}=A_0$:
  - For each point B in the set, do the following:
    - Calculate the angle from the vector $A_i - A_{i-1}$ to the vector $B - A_i$. (If i=0 we take the second vector to be (−1,0)).
    - The point with smallest angle is a convex hull point. Call it A_{i+1}.

This algorithm obviously has time complexity O(KN), where K is the number of convex hull points, and N is the number of points in the set. The time consuming step in the algorithm is the scan through the set of points to find the next convex hull point.

We implemented the above algorithm on the WW Warp, using C code to program the cluster processors and W2 to program the Warp array. In our implementation, the Warp array performs the inner loop in the algorithm, which finds a new convex hull point by calculation of the angle with all points. This is done in parallel on all cells, by partitioning the set of data points across the array and finding the best point in each cell's dataset individually, then finding the best point of the cell's points. The cluster processors repeatedly accept the new point from the Warp array and pass in this new convex hull point for the next step of the computation.

To test this algorithm, we generated a 1000 node random graph, which had 13 hull points. The measured time on the WW machine was 6.76 milliseconds, with the same execution time on PC Warp. The time for this algorithm scales linearly with the number of hull points.

Assuming a 16 MHz clock time and 72 cells in *i* Warp, each point location will take 26 microseconds, based on an operation count from the Warp implementation. Loading the initial array to the cells will take 250 microseconds, for a total time of 590 microseconds for our sample problem.

5.10 Voronoi Diagram

Geometrical constructions. The input is a set S of 1000 real coordinate pairs, defining a set of 1000 points in the plane, selected at random, with each coordinate in the range [0,1000]. The output is the Voronoi diagram of S, defined by the set of coordinates of its vertices, the set of pairs of vertices that are joined by edges, and the set of rays emanating from vertices and not terminating at another vertex. (On Voronoi diagrams see [26, Section 5.5].)

We consider the computation of the Voronoi diagram of a set of 1000 real points [13]. The algorithm is:

1. The coordinates of the points are sorted divided equally among the cells so that each cell has 100 points. The sorting is done systolically on the Warp array, using a heapsort algorithm in which each cell builds a heap of 100 points as the data values stream in, passing the rest of the data on to the next cell.
2. Each cell computes the Delauney triangulation of 100 points using a standard sequential algorithm.
3. Cells 1, 3, 7, and 9 receive the Delauney triangulation of their left neighbors. The two Delauney triangulations are then merged to form a single Delauney triangulation in these receiving cells. At the end of this stage we have four Delauney triangulations of 200 points each and two Delauney triangulations of 100 points each in cells 4 and 5. Six cells will be idle during this step.

4. The 200 point triangulations are merged to form 400 point triangulations. At the end of this step we have two triangulations of 400 points each and two triangulations of 100 points each. Eight cells are idle during this step. The mergings are carried out in the in the third and eighth cells.

5. The 400 point and 100 point triangulations are merged to form 500 point triangulations in cells 4 and 6. At the end of this step there are two triangulations of 500 points each. Eight cells are idle during this step.

6. The two 500 point triangulations are merged to give the Delauney triangulation of 1000 points. This operation is carried out in the fifth cell. Nine cells are idle.

7. The dual of the Delauney triangulation thus obtained will give the Voronoi diagram.

Table 5-8 gives operation counts for each of the steps in the Voronoi diagram algorithm above. These counts were obtained through a C program which computed the Voronoi diagram.

Step	Assignments	Array References	Comparisons	Arithmetic operations	Logical Operations
2	86897	192695	60149	71572	36290
3	89529	198309	60209	74221	36343
4	91754	202898	60264	76401	36388
5	94504	208420	60326	79030	36441
6	97313	214221	60394	81733	36502

Table 5-8: Operation counts for Voronoi diagram

*i*Warp will have 72 cells instead of 10. Since the time for intermediate data transfers is small we ignore any changes in that and assume linear speedup in the Delauney triangulation computation.

Since the computation of addresses for the array references appears to be the critical path we considered this as the bottleneck in the computation. (PC Warp and *i*Warp will have parallel address computation engines in each cell). Each array reference takes 300ns on PC Warp (100ns for the address computation and 200ns for the memory access) and 100ns on the baseline *i*Warp. The total computation time therefore comes to 64 milliseconds on PC Warp and 8.9 milliseconds on *i*Warp. The initial sort step requires 24 milliseconds on PC Warp and 10 milliseconds on *i*Warp. The number of floating-point data transfers internal to the computation is 3600 (400 in step 3, 800 in step 4, 400 in step 5, and 2000 in step 6). This will take 800 microseconds on PC Warp and 63 microseconds on *i*Warp.

Since the Voronoi diagram computation is taking the dual of the Delauney triangulation, this can be done in parallel. This can be done in pipelined mode (concurrent I/O and computation in a cell) so that the total time of computation will be around the total time for I/O which is around 200 milliseconds on PC Warp, and 120 milliseconds on *i*Warp. The conversion to Voronoi diagram will be part of a pipeline at the end of which Voronoi diagram edges will be transmitted to the host. Hence time for transmission to the host will be included in this.

The total times for the computation are, on PC Warp, 64 milliseconds + 24 milliseconds + 800 microseconds + 200 milliseconds = 290 milliseconds, while on *i*Warp the time is 8.9 milliseconds + 10 milliseconds + 63 microseconds + 120 milliseconds = 140 milliseconds.

5.11 Minimum spanning tree

Geometrical constructions. The input is a set S of 1000 real coordinate pairs, defining a set of 1000 points in the plane, selected at random, with each coordinate in the range [0,1000]. The output is the minimal spanning tree of S, defined by the set of pairs of points of S that are joined by edges of the tree. (On minimal spanning trees see [26, Section 6.1].)

We use Shamos's algorithm [26], in which we have only to examine edges in the Delauney triangulation to find an incremental edge in the minimum spanning tree. In the worst case 1000 vertices can correspond to 3000 edges,

implying an average of 3 edges per vertex. This means that we have to make a maximum of 2 comparisons to find the edge of minimum length out of a vertex. Since there are 1000 vertices we have to make only 2000 comparisons per stage of the algorithm and, since there are log (N) stages, we have to make 20000 comparisons in all. Also, as part of the initialization step we have to compute the lengths of all the 3000 edges, which will involve 6000 floating-point multiplications and 3000 floating-point additions. We also have to prepare a data structure which will give the out-degree of a particular vertex. This will involve 2 comparisons per edge, for a maximum of 6000 comparisons in all. We assume that the minimum spanning tree shall be computed. We also assume that a floating-point multiplication takes 5 microseconds and a floating-point addition takes 2.5 microseconds, and each comparison takes 1 microsecond. Adding up the respective times the total comes to about 65 milliseconds. This time is the worst case since the Delauney triangulation of 1000 points will typically contain much less than 3000 edges.

5.12 Visibility

Visibility. The input is a set of 1000 triples of triples of real coordinates $((r,s,t),(u,v,w),(x,y,x))$, defining 1000 opaque triangles in three-dimensional space, selected at random with each coordinate in the range [0,1000]. The output is a list of vertices of the triangles that are visible from (0,0,0).

An input partitioning method is used. Each vertex is simply tested to see if it is obscured by any of the triangles. This is done by taking the four planes defined by the triangle vertices and the origin and any two of them, and testing to see if the vertex point lies in the interior of the region defined by the three planes including the origin, but on the far side of the triangle. The mapping onto Warp is to broadcast the set of triangle points to all cells, and then to send to each of the ten cells one-tenth of the vertex set, with each cell testing its portion to see if it is visible. The execution time on the WW Warp is 825 milliseconds (however, the WW Warp machine cannot hold the entire dataset due to memory limitations – this time is a compiler estimated execution time). Some improvement (probably a factor of two to three) is expected on PC Warp, since the algorithm will be able to stop testing a vertex when it is found that a vertex is definitely not obscured by a particular triangle. On *i* Warp, we estimate a speedup of about 10, giving an execution time of 40 milliseconds.

5.13 Graph Matching

Graph matching. The input is a graph G having 100 vertices, each joined by an edge to 10 other vertices selected at random, and another graph H having 30 vertices, each joined by an edge to 3 other vertices selected at random. The output is a list of the occurrences of (an isomorphic image of) H as a subgraph of G. As a variation on this task, suppose the vertices (and edges) of G and H have real-valued labels in some bounded range; then the output is that occurrence (if any) of H as a subgraph of G for which the sum of the absolute differences between corresponding pairs of labels is a minimum.

This problem includes two subproblems. The first is to find isomorphic embeddings of one the smaller graph in the larger one. Finding one such embedding (or determining the existence of one) is known to be NP-complete [11]. Finding all isomorphisms actually grows exponentially. For example, in one set of randomly generated data, we found about 10^{16} solutions. Because there are too many solutions, no presently existing machine can produce all the solutions in one year.

The second problem is to find the one isomorphism to the graph with the least differences between the corresponding edge and vertex costs. The complexity of the second problem is obviously between finding one and finding all. This problem has not been completed because there were too many solutions to the first problem.

Our parallel algorithm is based on Ullmann's refinement procedure [31] which can prune the search tree by eliminating mappings that are infeasible because of connectivity requirements. The method eliminates mappings as early as possible.

In addition, we developed a more powerful method to cut the search tree as early as possible. The new method uses graph analysis and makes use of some special features of the graph.

We implemented the problem on the Warp host, which is a Sun workstation. Running on a set of randomly generated data for over one hour, we obtained 1188174 solutions, giving 267 solutions/second or about 3.75

milliseconds/solution. At this point, by counting the branching factors of the tree above the portion we had processed, we estimated we had found only about $1.2\times10^{-9}\%$ of the solutions, leading to our estimate of 10^{16} solutions for this example.

In the Warp implementation, we parallelize the exploration of the search tree. This is easy to do because the search tree is so large that we can easily assign each subtree to a processor. By straightforward extrapolation of cycle time, we estimate the solution rate in PC Warp to be 2700 solutions/second. Similarly, we estimate the solution rate in *i* Warp to be 19000 solutions/second.

5.14 Minimum-cost Path

Minimum-cost path. The input is a graph G having 1000 vertices, each joined by an edge to 100 other vertices selected at random, and where each edge has a nonnegative realvalued weight in some bounded range. Given two vertices P, Q of G, the problem is to find a path from P to Q along which the sum of the weights is minimum. (Dynamic programming may be used, if desired.)

The algorithm used here is the best known sequential algorithm, Dijkstra's Single Source Single Destination [10] (SSSD). The algorithm works by repeatedly "expanding" nodes (adding all their neighbors to a list) then finding the next node to expand by choosing the closest unexpanded node to the destination.

The lack of a `while` loop on the WW Warp results in a significant loss of performance, compared to PC Warp and *i* Warp. PC Warp and *i* Warp have very similar mappings:

- **WW Warp.** The WW Warp cannot execute a loop a data dependent number of times, so that the outer loop of SSSD must be mapped into the cluster processors. In this case, the Warp array is used for expanding nodes, and for calculating which node should be expanded next. Node expansion is done by feeding from the cluster processor the descendants of the node to be expanded, and by calculating the distance to the goal of each of these nodes. The computation is extremely simple, and I/O bound on the Warp array. Each node expansion involves the transfer of 200 words of data, which takes 200×1.2 microseconds $=240$ microseconds, since the transfer of a single word takes 1.2 microseconds.

  To find the next node to be expanded, the entire set of nodes must be scanned, and the node nearest the goal is selected. On the WW Warp this means 1000 nodes must be scanned. Again, the computation is I/O bound, so that the execution time is 1000×1.2 microseconds $=1.2$ milliseconds. In the worst case, 1000 nodes must be expanded, for a total time of $1000\times(1.2$ milliseconds $+240$ microseconds $=1.44$ s). This number scales linearly with the number of nodes that must be expanded to find the goal.

- **PC Warp.** In PC Warp it is possible to map the outer loop of SSSD into the Warp array, giving a much better time.

  Node expansion is done by prestoring at each cell the costs, giving each cell 100 data. Node expanding is done in parallel in all cells. In the worst case, the slowest cell will have to expand 100 nodes, so that the time for one node expansion is 100×0.25 microseconds $=25$ microseconds.

  The global minimum is calculated in parallel in all cells, and then the minimum among cells is found in one pass through the array. Finding the minimum on each cell takes 0.4 microseconds $\times100=40$ microseconds. Finding the minimum among cells takes 0.4 microseconds $\times10=4$ microseconds.

  The total time for one node expansion is therefore 69 microseconds. In the worst case, when 1000 nodes are expanded, the time is 69 milliseconds. This time scales linearly with the number of nodes that must be expanded to find the goal.

- ***i* Warp.** Following the same algorithm partitioning method as for PC Warp, we use 72 cells instead of 10. Now each cell need store only 14 data. The faster cycle time of *i* Warp gives a 10 microsecond time for one node expansion, 7 microseconds to find the global minimum in each cell, and 8 microseconds to find the global minimum across cells. (The minimum across cells is done sequentially from cell to cell, so it takes longer on longer arrays). The total time for one node expansion on *i* Warp is 25 microseconds. In the worst case, the total time for the solution will be 25 milliseconds. This number scales linearly with the number of nodes that must be expanded to find the goal.

5.15 Warp Benchmarks Summary

In Table 5-9 we summarize Warp's performance on the IU Architecture benchmarks. With each time, we give its source – from an actual run of WW Warp, from compiled code, or by an estimate (all *i* Warp times are estimated). The times from an actual run are, of course, the most reliable – they are observed times, from an actual run on our WW Warp at Carnegie Mellon, and include I/O. Times marked "compiled code" are just as reliable; the W2 compiler for Warp produces a time estimate, which gives the actual execution time for the algorithm on Warp (we have modified these times as appropriate when the Warp array is not the bottleneck in the execution time of the algorithm). Finally, "estimate" indicates a time which is not based on compiled code, but on some other method, which may not be as reliable. The source of the time is given in the relevant section. We have tried to be as accurate as possible in these estimates, and have tried to err on the side of caution.

Algorithm	WW Warp	PC Warp	*i* Warp
Laplacian	430 ms actual run	350 ms compiled code	7.8 ms
Zero crossing	170 ms actual run	50 ms estimate	7.8 ms
Border following	N/A	1.1 s estimate	690 ms
Connected Components	5.6 s compiled code	980 ms estimate	470 ms
Hough transform	N/A	340 ms compiled code	60 ms
Convex Hull	9 ms actual run	9 ms compiled code	3.2 ms
Voronoi diagram	N/A	290 ms estimate	140 ms
Minimal spanning tree	N/A	160 ms estimate	43 ms
Visibility	830 ms compiled code	400 ms estimate	40 ms
Graph matching	N/A	1800 soln./s estimate	19,000 soln./s
Minimum-cost path	1.4 s estimate	69 ms estimate	25 ms

Table 5-9: Warp Benchmark Summary

5.16 Evaluation of the Warp Architecture

In this section we will use the data generated by these benchmarks to evaluate the Warp architecture, by considering the effect of various reasonable design changes. The intent is to explore the design space around the PC Warp. We will consider all of the benchmark algorithms except for minimal spanning tree, which is not performed on the Warp array.

5.16.1 Memory

In PC Warp, each cell has 32K words of memory, for a total memory in the Warp array of 320K. What is the effect on performance of decreasing the memory size?

Laplacian and zero crossing are input partitioned algorithms. This implies that each cell needs only enough

memory to compute the result for the area of the image assigned to that cell – in this case, approximately $11\times52+5\times52=832$ words for the Laplacian, and $3\times52+512=668$ data for zero crossing. If we decrease the memory per cell below this point, the computation can still be done, but only by processing a strip of the image at a time. For example, the Laplacian could process two 512×256 images and need only $11\times26+5\times26=416$ words of memory. (The computation would actually process a slightly wider image, because of the need for overlap at the interior edge. This makes it less efficient.)

Border following and connected components both must store the entire image (distributed through the array) at once to do their processing. This means the total array storage must be at least 256K, plus whatever is needed to store their local tables. If less memory is available than this, the computation becomes exceedingly complex – either the image must be compressed for storage, or several passes must be performed, with a new merge step. This sort of complexity is frustrating for a programmer to deal with.

Hough transform and visibility display the standard behavior of output partitioned algorithms; as memory is reduced, the computation grows proportionately less efficient. For example, for Hough transform the current benchmark requires $180\times512=90K$ words of memory in the array. If only, say, 45K words of memory are available, the computation can be done in two passes, each building half the Hough array; but each pass takes as long as the whole thing on a machine with sufficient memory. Similarly, visibility needs 27K; if less is available than this, multiple passes must be made, each pass deleting some of the points from the visibility set.

The other algorithms (Voronoi diagram, minimal spanning tree, graph matching, and minimum-cost path) all share the characteristics that they require the entire dataset to be stored in the array at once, their computation is fairly complex, and they have small datasets. In a well-designed machine, memory is unlikely to be a problem; but if it is too small to store the complete dataset, programming any of these problems will become very difficult.

5.16.2 Number of processing elements

PC Warp has ten cells in its array, a fairly small number as parallel machines go. What happens if we increase this number?

The effect on Laplacian, zero crossing, Hough transform, convex hull, and visibility is straightforward; their speed changes approximately linearly, increasing or decreasing as the number of cells is increased or reduced, as long as I/O is not a bottleneck. This bottleneck occurs when the data transfer rate between the external host and the Warp array reaches 12 MB/second, which occurs when the number of Warp cells is 168 for Laplacian, 24 for zero crossing, 180 for Hough transform (since the partitioning is by angle, this is the bottleneck), 130 for convex hull, and 530 for visibility. (Actually, due to the effects of rounding, some of these numbers do not actually represent peaks in performance. For example, we will not observe any change in performance between 128 cells, or four pixels per cell, and 171 cells, or three pixels per cell.) By this point, effects we ignored in our initial time estimate, such as the cost of overlapping data with an adjacent cell, or the buffer sizes in the interface unit, probably dominate. Except possibly for zero crossing, these limits on the number of cells exceed the practical limits of building and maintaining such a PC Warp array.

Graph matching is similarly partitioned, and it should display the same sort of behavior as the above algorithms. We have not done enough analysis to determine the optimal number of cells.

The case of connected components is quite different. This algorithm consists of two parts, one of which is partitioned like the algorithms above, and the other of which is a merge step. The total time for both steps is $O(A/N+BN)$, where A and B are constants depending on the algorithm for the partitioned and merge steps, respectively, and N is the number of cells. This formula has a minimum when $N=\sqrt{A/B}$. For connected components, this occurs when $N=25$, as shown in Section 5.7.3.3.

Similarly remarks apply to Voronoi diagram and border following. We do not have accurate enough estimates to give a definite maximal number of cells in these cases.

5.16.3 External host

The external host is based on standard MC68020 processors and the VME bus. This is convenient for programming, but may be undesirable for performance. What is the effect of making the external host more powerful?

Naturally, as the external host grows more and more powerful, more and more of the computation can be mapped onto it – in the most extreme case, it can perform the entire computation. We will restrict ourselves to considering the qualitative effects of making the external host more powerful, but still less powerful than the Warp array.

There is no benchmark in which the external host actually creates an I/O bottleneck. However, there are many ways in which a more powerful external host would significantly affect the program mapping. This is most evident in Section 5.14. Here, on the WW machine, the external host is used to control the outer loop of the program, while on PC Warp and *i* Warp, the Warp array itself controls this outer loop. In many ways, it is convenient to use the external host for this computation; there is no reason not to split the computation in this way, and it is in some ways easier to program. However, the poor computational abilities of the external host make it advantageous to map as much computation onto the Warp array as possible, even when it is somewhat inconvenient.

Similar remarks apply to border following, connected components, convex hull, and Voronoi diagram. All of these algorithms could use a more powerful external host in the merge phase of their computation.

However, it is interesting to consider alternatives to a more complex external host. It is unlikely that the ratio of power between the external host and the Warp array will shift towards the external host in future versions of Warp or similar systems. Rather, as our ability to build larger Warp arrays grows, it will likely shift in the other direction. We must try to find alternatives to mapping important parts of the computation onto a sequential processor if we are to see further speedup in these algorithms. It seems that a much better alternative to making the external host more powerful is to make the Warp array more flexible, for example by making the communications between the cells more powerful (allowing higher dimensional arrays or logically connecting distant cells).

6. References

[1] *Reference Manual for the Ada Programming Language*
MIL-STD 1815 edition, United States Department of Defense, AdaTEC, SIGPLAN Technical Committe on Ada, New York, N.Y. AdaTEC, 1982.
Draft revised MIL-STD 1815. Draft proposed ANSI Standard document.

[2] Annaratone, M., Arnould, E., Cohn, R., Gross, T., Kung, H.T., Lam, M., Menzilcioglu, O., Sarocky, K., Senko, J., and Webb, J.
Warp Architecture: From Prototype to Production.
In *Proceedings of the 1987 National Computer Conference.* AFIPS, 1987.

[3] Annaratone, M., Arnould, E., Gross, T., Kung, H.T., Lam, M., Menzilcioglu, O., Sarocky, K. and Webb, J.A.
Warp Architecture and Implementation.
In *Conference Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 346-356. June, 1986.

[4] Annaratone, M., Arnould, E., Gross, T., Kung, H. T., Lam, M., Menzilcioglu, O. and Webb, J. A.
The Warp Computer: Architecture, Implementation and Performance.
IEEE Transactions on Computers C-36(12), December, 1987.

[5] Annaratone, M, Arnould, E., Cohn, R., Gross, T., Kung, H.T., Lam, M., Menzilcioglu, O., Sarocky, K., Senko, J., and Webb, J.
Architecture of Warp.
In *COMPCON Spring '87*, pages 264-267. IEEE Computer Society, 1987.

[6] Batcher, K. E.
Bit-serial parallel processing systems.
IEEE Trans. Computer C-31(5):377-384, May, 1982.

[7] BBN Laboratories.
The Uniform System Approach to Programming the Butterfly Parallel Processor
1 edition, Cambridge, MA, 1985.

[8] Bruegge, B., Chang, C., Cohn, R., Gross, T., Lam, M., Lieu, P., Noaman, A. and Yam, D.
The Warp Programming Environment.
In *Proceedings of the 1987 National Computer Conference.* AFIPS, 1987.

[9] Bruegge, B., Chang, C., Cohn, R., Gross, T., Lam, M., Lieu, P., Noaman, A. and Yam, D.
Programming Warp.
In *COMPCON Spring '87*, pages 268-271. IEEE Computer Society, 1987.

[10] Dijkstra, E.
A note on two problems in connexion with graphs.
Numerische Mathematik 1:269-271, 1959.

[11] Garey, M. R., and Johnson, D. S.
Computers and Intractibility: A guide to the theory of NP-completeness.
W. H. Freeman, 1979.

[12] Gross, T., Kung, H.T., Lam, M. and Webb, J.
Warp as a Machine for Low-level Vision.
In *Proceedings of 1985 IEEE International Conference on Robotics and Automation*, pages 790-800. March, 1985.

[13] Guibas, L. J., and Stolfi, J.
Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams.
ACM Transactions on Graphics 4, 1985.

[14] Hamey, L. G. H., Webb, J. A., and Wu, I-C.
An Architecture Independent Programming Language for Low-Level Vision.
Submitted to Computer Graphics and Image Processing.

[15] Haralick, R. M.
Digital Step Edges from Zero Crossings of Second Directional Derivatives.
IEEE Transactions on Pattern Analysis and Machine Intelligence 6:58-68, 1984.

[16] Hillis, W. D.
The Connection Machine.
The MIT Press, Cambridge, Massachusetts, 1985.

[17] Huang, T. S., Yang, G. J., and Tang, G. Y.
A fast two-dimensional median filtering algorithm.
In *International Conference on Pattern Recognition and Image Processing*, pages 128-130. IEEE, 1978.

[18] *iPSC System Overview*
Intel Corporation, 1985.

[19] Jarvis, R. A.
On the identification of the convex hull of a finite set of points in the plane.
Information Processing Letters 2:18-21, 1973.

[20] Jordan, K. E.
Performance comparison of large-scale scientific computers: Scalar mainframes, mainframes with integrated vector facilities, and supercomputers.
IEEE Computer 20(3):10-23, March, 1987.

[21] Joy, W. N., Babaoglu, O., Fabry, R. S., Sklower, K.
UNIX Programmer's Manual
4th Berkeley Distribution edition, University of California at Berkeley, 1980.

[22] Kernighan, B. W. and Ritchie, D. M.
The M4 Macro Processor.
In *Unix Programmer's Manual*. Bell Laboratories, Murray Hill, NJ 07974, 1979.

[23] Kung, H.T. and Webb, J.A.
Global Operations on the CMU Warp Machine.
In *Proceedings of 1985 AIAA Computers in Aerospace V Conference*, pages 209-218. American Institute of Aeronautics and Astronautics, October, 1985.

[24] Kung, H. T. and Webb, J. A.
Mapping Image Processing Operations onto a Linear Systolic Machine.
Distributed Computing 1(4):246-257, 1986.

[25] 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.

[26] Preparata, F. P. and Shamos, M. I.
Computational Geometry - An Introduction.
Springer, New York, 1985.

[27] Rosenfeld, A. and Kak, A. C.
Digital Picture Processing.
Academic Press, New York, 1982.

[28] Schwartz, J., Sharir, M., and Siegel, A.
An efficient algorithm for finding connected components in a binary image.
Technical Report 154, New York University Department of Computer Science, February, 1985.

[29] Seitz, C.
The Cosmic Cube.
Communications of the ACM 28(1):22-33, January, 1985.

[30] Electrotechnical Laboratory.
SPIDER (Subroutine Package for Image Data Enhancement and Recognition).
Joint System Development Corp., Tokyo, Japan, 1983.

[31] Ullman, J. R.
An algorithm for subgraph isomorphism.
Journal of the ACM 23(1):31-42, January, 1976.

[32] Wallace, R. S. and Howard, M. D.
HBA Vision Architecture: Built and Benchmarked.
In *Computer Architecures for Pattern Analysis and Machine Intelligence.* IEEE Computer Society, Seattle, Washington, December, 1987.

[33] Wallace, R. S., Webb, J. A. and Wu, I-C.
Architecture Independent Image Processing: Performance of Apply on Diverse Architectures.
Submitted to Computer Graphics and Image Processing.

I. WEB Listing

In the following, *image* is a 512×512 array of unsigned char and *realimage* is a 512×512 array of float. (These are the sizes compiled in the programs; to change these sizes, the programs have to be re-compiled.)

The "Status" given below is either coded, compiled, tested, or validated. "Coded" indicates the program is written (the source code is available in the directory) but not yet compiled. "Compiled" indicates the program has been written and successfully compiled by W2 and Apply (if necessary) but not necessarily tested. "Tested" indicates that the program has been written, compiled, and tested. "Validated" indicates that the program has been written, compiled, tested, and passed validation in this release. Over one-half of the library can now be validated.

For programs that have been compiled or tested but not validated, the execution time given is the time estimated for execution by the W2 compiler. *Actual run times are given for validated programs.* The run time of a Warp program is defined as the time from the start of the execution of Cluster1 (used for input) to the end of the execution of Cluster2 (used for output). This time includes all I/O from the external host to Warp. To distinguish these times from the estimated times, they are printed in **boldface**. In general, the actual run time for a program may differ from the compiler estimated execution time, for two reasons:

1. The compiler does not take I/O between cells or with the host in its estimate. Since I/O is almost completely overlapped with execution, this usually gives a very slight underestimate (about 2%) because of the skew between cells. However, for programs that process real images, and perform a very simple operation on them, I/O with the host may be a bottleneck. (Currently, using DMA in compiler-generated code, the host provides about 7 MB/S each of input and output to the Warp array. In the best case the array can process 20 MB/S each input and output. I/O is not a bottleneck for byte images because the interface unit unpacks bytes to floats, giving a factor of four increase in data).
2. In computing the estimated times, the compiler makes assumptions about branching in conditionals that are pessimistic.

Thus, the compiler will tend to *overestimate* the execution times of programs with greatly unbalanced conditionals, slightly *underestimate* the execution time of most other programs, and significantly *underestimate* the execution time of programs that perform very simple operations (e.g., add a constant) on real images.

"Size" is size in W1 instructions of the compiled code. The Warp machine has space for 7936 W1 instructions (8192 instructions are in the memory, and 256 are used for system purposes).

Program	Status / Description	Time	Size	Language	Access
Parameters					
addc1b	Validated	**109.4 ms**	94	Apply	
	Add a constant to a byte image.				
Parameters:	1st param: *image*				input
	2nd param: int constant				input
	3rd param: *image*				output
addc1c	Validated	**328.08 ms**	124	Apply	
	Add a complex constant to a complex image.				
Parameters:	1st param: *realimage* (real part)				input
	2nd param: *realimage* (imaginary part)				input
	3rd param: float constant (real part)				input
	4th param: float constant (imaginary part)				input
	5th param: *realimage* (real part)				output
	6th param: *realimage* (imaginary part)				output

addc1r Validated **146.0 ms** 78 Apply
Add a constant to a *realimage*.
Parameters: 1st param: *realimage* input
2nd param: float constant input
3rd param: *realimage* output

addc1s Validated **108.4 ms** 94 Apply
Add a constant to an signed byte image.
Parameters: 1st param: *image* input
2nd param: int constant input
3rd param: *image* output

addp1b Validated **161.5 ms** 99 Apply
Add two byte images.
Parameters: 1st param: *image* input
2nd param: *image* input
3rd param: *image* output

addp1c Validated **622.10 ms** 165 Apply
Add two complex images.
Parameters: 1st param: *realimage* (real part) input
2nd param: *realimage* (imaginary part) input
3rd param: *realimage* (real part) input
4th param: *realimage* (imaginary part) input
5th param: *realimage* (real part) output
6th param: *realimage* (imaginary part) output

addp1r Validated **309.8 ms** 99 Apply
Add two *realimage*'s.
Parameters: 1st param: *realimage* input
2nd param: *realimage* input
3rd param: *realimage* output

addp1s Validated **161.8 ms** 99 Apply
Add two signed byte images.
Parameters: 1st param: *image* input
2nd param: *image* input
3rd param: *image* output

afin1 Validated **4396.7 ms** 410 W2
Affine image warping using linear interpolation.
Parameters: 1st param: *image* input
2nd param: float array[2][3] input
Homogeneous transformation matrix.
3rd param: 1 (inverse) or 0 (direct transform) input
4th param: *image* output

afin2 Validated **4530.3 ms** 436 W2
Affine image warping using quadratic interpolation.
Parameters: 1st param: *image* input
2nd param: float array[2][3] input
Homogeneous transformation matrix.
3rd param: 1 (inverse) or 0 (direct transform) input
4th param: *image* output

afin3	Validated	**4413.1 ms**	827	W2	
	Affine image warping using max, min, or nearest neighbor interpolation.				
Parameters:	1st param: *image*				input
	2nd param: float array[2][3]				input
	Homogeneous transformation matrix.				
	3rd param: 1 (inverse) or 0 (direct transform)				input
	4th param: select type of interpolation				input
	5th param: *image*				output
andc1b	Validated	**109.6 ms**	99	Apply	
	Logically and an image with a constant.				
Parameters:	1st param: *image*				input
	2nd param: int constant				input
	3rd param: *image*				output
andp1b	Validated	**160.7 ms**	118	Apply	
	Logically and two images.				
Parameters:	1st param: *image*				input
	2nd param: *image*				input
	3rd param: *image*				output
area1	Validated	**224.7 ms**	317	W2	
	Measure area of regions in a labeled image.				
Parameters:	1st param: *image*				input
	2nd param: int value				input
	Label of region to be processed; if 0, all regions are processed.				
	3rd param: int array[256]				output
	array[i] is the area of region labeled i.				
asmt	Validated	**1201.9 ms**	824	Apply	
	Local selective averaging.				
Parameters:	1st param: *image*				input
	2nd param: *image*				output
bdr41	Validated	**124.7 ms**	552	Apply	
	Detect borders in binary picture (4-connectedness).				
Parameters:	1st param: *image*				input
	2nd param: 1 (inner) or 0 (outer borders)				input
	3rd param: *image*				output
bdr81	Validated	**126.5 ms**	677	Apply	
	Detect borders in binary picture (8-connectedness).				
Parameters:	1st param: *image*				input
	2nd param: 1 (inner) or 0 (outer borders)				input
	3rd param: *image*				output
bflp1	Validated	**329.0 ms**	728	Apply	
	Detect borders of regions in a labeled image.				
Parameters:	1st param: *image*				input
	2nd param: int value				input
	Label of region to be processed; if 0, all regions are processed.				
	3rd param: int value				input
	Label assigned to borders; if 0, keep label of the region.				
	4th param: 4 or 8 (connectedness)				input
	5th param: *image*				output

byrl	Validated	**146.9 ms**	75	Apply	
	Byte to real conversion.				
Parameters:	1st param: *image*				input
	2nd param: *realimage*				output

canny	Compiled	W2	
	Canny operator.		

Link with ${WPEweb}/libWEB.a.
This subroutine can handle any size input image, by processing 512x512 regions. Uses mag, magdir, nonmax.
C interface: canny(inimg, outimg,size, bounds, xgrad, ygrad, rgrad, dir, verbose, err)

Parameters:	inimg: input image	input
	outimg: nonmaxima suppressed edges	output
	size: int size of Canny (<=20)	input
	bounds: SUBIMAGE bounds to compute Canny	input
	xgrad: output X gradient	output
	ygrad: output Y gradient	output
	rgrad: gradient maxima	output
	dir: gradient direction	output
	verbose: int 0 (quiet) or 1 (verbose)	input
	err: Generalized image library error parameter	input

cgrv1	Validated	**221.9 ms**	388	W2	
	Measure coordinates of center of gravity of regions in a labeled image.				
Parameters:	1st param: *image*				input
	2nd param: int value				input
	Label of region to be processed; if 0, all regions are processed.				
	3rd param: int array[256]				input
	Array with areas of regions (output of AREA1).				
	4th param: float array[256]				output
	array[i] is the row coordinate of region labeled i.				
	5th param: float array[256]				output
	array[i] is the column coordinate of region labeled i.				

clip	Validated	**149.0 ms**	198	Apply	
	Set gray values in a range to zero.				
Parameters:	1st param: *image*				input
	2nd param: int value (lower bound)				input
	3rd param: int value (upper bound)				input
	4th param: *image*				output

colortobw	Validated	**215.9 ms**	185	Apply	
	Convert (r,g,b) image to black and white by averaging.				
Parameters:	1st param: *image* (red)				input
	2nd param: *image* (green)				input
	3rd param: *image* (blue)				input
	4th param: *image* (gray)				output

conc	Validated	**458.0 ms**	573	Apply	
	Compute connectivity number.				
Parameters:	1st param: *image*				input
	2nd param: 4 or 8 (connectedness)				input
	3rd param: *image*				output

connect	Validated	**2545.4 ms**	773	W2	
	Eight-connected components analysis. In the input image, pixels with different grayvalues are not considered to be connected. Grayvalue 0 is "background" and is not labelled. Output is a *realimage* where each grayvalue represents a different connected region.				
Parameters:	1st param: *image*				input
	2nd param: *realimage*				output
cqlt1	Validated	**943.1 ms**	461	W2	
	Measure coordinates of circumscribing rectangle of regions in a labeled image.				
Parameters:	1st param: *image*				input
	2nd param: int value Label of region to be processed; if 0, all regions are processed.				input
	3rd param: int value When 2nd param = 0, greatest label in the input labeled image.				input
	4th param: int array[256][4] array[i][4] contains the coordinates of the circumscribing rectangle of region labeled i.				output
crcl1	Validated	**2.7 ms**	186	W2	
	Measure compactness of regions.				
Parameters:	1st param: int array[256] Array with areas of regions (output of AREA1).				input
	2nd param: int array[256] Array with perimeters of regions (output of PRMT1).				input
	3rd param: float array[256] array[*i*] *is the compactness of region labeled i.*				output
cros	Validated	**257.1 ms**	427	Apply	
	Compute crossing number.				
Parameters:	1st param: *image*				input
	2nd param: 4 or 8 (connectedness)				input
	3rd param: *image*				output
dct	Validated	**232.4 ms**	583	W2	
	Two dimensional direct discrete cosine transform. Takes an input image and performs 8 x 8 discrete cosine transforms to produce the output image. Useful for image compression.				
Parameters:	1st param: *image*				input
	2nd param: *realimage*				output
display	Validated	**514.7 ms**	499	W2	
	Histogram-equalize and halftone image. Used by WPE for image display under X.				
Parameters:	1st param: *image*				input
	2nd param: *image*				output
divc1b	Validated	**107.5 ms**	136	Apply	
	Divide an image by a constant.				
Parameters:	1st param: *image*				input
	2nd param: int constant				input
	3rd param: *image*				output

divc1r	Validated	**145.7 ms**	87	Apply	
	Divide a *realimage* by a constant.				
Parameters:	1st param: *realimage*				input
	2nd param: float constant				input
	3rd param: *realimage*				output
divc1s	Validated	**107.2 ms**	136	Apply	
	Divide a signed byte image by a constant.				
Parameters:	1st param: *image*				input
	2nd param: int constant				input
	3rd param: *image*				output
divp1b	Validated	**161.8 ms**	144	Apply	
	Divide 1st input image by 2nd input image.				
Parameters:	1st param: 1st *image*				input
	2nd param: 2nd *image*				input
	3rd param: *image*				output
divp1r	Validated	**310.2 ms**	100	Apply	
	Divide 1st input *realimage* by 2nd input *realimage*.				
Parameters:	1st param: 1st *realimage*				input
	2nd param: 2nd *realimage*				input
	3rd param: *realimage*				output
divp1s	Validated	**160.8 ms**	144	Apply	
	Divide 1st signed byte image by 2nd signed byte image.				
Parameters:	1st param: 1st *image*				input
	2nd param: 2nd *image*				input
	3rd param: *image*				output
egfc	Validated	**849.7 ms**	593	Apply	
	Edge detection using orthogonal templates by Frei and Chen.				
Parameters:	1st param: *image*				input
	2nd param: *image*				output
egks1	Validated	**730.3 ms**	487	Apply	
	Edge detection using Kirsch operator (outputs magnitude only).				
Parameters:	1st param: *image*				input
	2nd param: *image*				output
egks2	Validated	**906.47 ms**	537	Apply	
	Edge detection using Kirsch operator (outputs magnitude and direction of gradient).				
Parameters:	1st param: *image*				input
	2nd param: *image* (magnitude)				output
	3rd param: *image* (direction)				output
eglp	Validated	**311.1 ms**	608	Apply	
	Edge detection using Laplacian.				
Parameters:	1st param: *image*				input
	2nd param: 1, 2, or 3 Select Laplacian operator.				input
	3rd param: *image*				output
egpr	Validated	**2080.5 ms**	873	Apply	
	Edge preserving smoothing.				
Parameters:	1st param: *image*				input
	2nd param: *image*				output

egpw1	Validated	**233.8 ms**	868	Apply	
	Edge detection using Prewitt operator (differential type) (outputs magnitude only).				
Parameters:	1st param: *image*				input
	2nd param: 0 or 1				input
	Select computation equation for magnitude.				
	3rd param: *image*				output
egpw2	Validated	**2031.3 ms**	1327	Apply	
	Edge detection using Prewitt operator (differential type) (outputs magnitude and direction of gradient).				
Parameters:	1st param: *image*				input
	2nd param: 0 or 1				input
	Select computation equation for magnitude.				
	3rd param: *image* (magnitude)				output
	4th param: *image* (direction)				output
egpw3	Validated	**703.5 ms**	478	Apply	
	Edge detection using Prewitt operator (template type). (outputs magnitude only).				
Parameters:	1st param: *image*				input
	2nd param: *image*				output
egpw4	Validated	**874.80 ms**	528	Apply	
	Edge detection using Prewitt operator (template type). (outputs magnitude and direction of gradient).				
Parameters:	1st param: *image*				input
	2nd param: *image* (magnitude)				output
	3rd param: *image* (direction)				output
egrb	Validated	**145.2 ms**	378	Apply	
	Roberts operator.				
Parameters:	1st param: *image*				input
	2nd param: 0 or 1				input
	Select computation equation for magnitude.				
	3rd param: *image*				output
egrs1	Validated	**673.3 ms**	465	Apply	
	Robinson operator (outputs magnitude only).				
Parameters:	1st param: *image*				input
	2nd param: *image*				output
egrs2	Validated	**840.67 ms**	515	Apply	
	Robinson operator (outputs magnitude and direction of gradient).				
Parameters:	1st param: *image*				input
	2nd param: *image* (magnitude)				output
	3rd param: *image* (direction)				output
egrs3	Compiled	0.880992 s	856	Apply	
	Robinson operator (checks local connectivity of edges and deletes those that do not meet the conditions).				
Parameters:	1st param: *image*				input
	Magnitude (output of EGRS2).				
	2nd param: *image*				input
	Direction (output of EGRS2).				
	3rd param: *image*				output

egsb1	Validated	**329.3 ms**	961	Apply	
	Sobel operator (outputs magnitude only).				
Parameters:	1st param: *image*				input
	2nd param: 0 or 1				input
	Select computation equation for magnitude.				
	3rd param: *image*				output
egsb2	Coded			Apply	
	Sobel operator (outputs magnitude and direction of gradient).				
Parameters:	1st param: *image*				input
	2nd param: 0 or 1				input
	Select computation equation for magnitude.				
	3rd param: *image* (magnitude)				output
	4th param: *image* (direction)				output
eikv1	Validated	**827.0 ms**	705	Apply	
	Iterative edge detection using Kasvand's method.				
Parameters:	1st param: *image*				input
	2nd param: *image*				output
eikv2	Validated	**903.08 ms**	678	Apply	
	Iterative line sharpening using Kasvand's method.				
Parameters:	1st param: *image*				input
	2nd param: *image*				output
epct	Validated	**456.1 ms**	619	Apply	
	Expand or contract binary pattern.				
Parameters:	1st param: *image*				input
	2nd param: 4 or 8 (connectedness)				input
	3rd param: 0 (contract) or 1 (expand)				input
	4th param: *image*				output
ersr3	Validated	**149.8 ms**	122	Apply	
	Erase small regions in a labeled image. Small regions are those whose area is less than a given threshold.				
Parameters:	1st param: *image*				input
	2nd param: int array[256]				input
	Array with areas of regions (output of AREA1).				
	3rd param: int value (threshold)				input
	4th param: *image*				output
expand	Validated	**114.7 ms**	675	Apply	
	Image doubling using linear interpolation.				
Parameters:	1st param: char array[256][256]				input
	2nd param: *image*				output
fclib	Validated	**66.70 ms**	71	Apply	
	Assign zero to an image.				
Parameters:	1st param: *image*				output
fclir	Validated	**145.3 ms**	71	Apply	
	Assign zero to a *realimage*.				
Parameters:	1st param: *realimage*				output

fcon	Validated	**400 ms**		W2

Two-dimensional convolution using FFT.
Image2 is replaced by the convolution of image1 with image2.
Image1 will be destroyed if it is a Warp real image.
The execution time reported is the Unix user time for a complete call.
C Calling sequence: fcon(image1, image2).
Link with ${WPEweb}/libWEB.a, the libraries needed for warp_call(3), and -lm.

Parameters: *realimage* — input
realimage — inout

fcor	Validated	**483 ms**		W2

Two-dimensional correlation using FFT.
Image2 is replaced by the correlation of image1 with image2.
Image1 will be destroyed if it is a Warp real image.
The execution time reported is the Unix user time for a complete call.
Link with ${WPEweb}/libWEB.a, the libraries needed for warp_call(3), and -lm.
C calling sequence: fcor(image1, image2)

Parameters: 1st *realimage* — input
2nd *realimage* — inout

fcpl	Compiled	1.7909992 s	558	Apply

Convert (real, imaginary) representation to (magnitude, phase) representation for complex images.

Parameters:
1st param: *realimage* (real part) — input
2nd param: *realimage* (imaginary part) — input
3rd param: *realimage* (amplitude part) — output
4th param: *realimage* (phase part) — output

fft	Validated	**3485.1 ms**	193	W2

In-place two-dimensional fast Fourier transform.
The image is replaced by its Fourier transform, defined so that the inverse of the forward transform gives the original image.
C calling procedure: fft(real-part, imaginary-part, flag)
Link with $WPEweb/libWEB.a and the libraries needed by warp_call(3).

Parameters:
1st param: *realimage* (real part) — inout
2nd param: *realimage* (imaginary part) — inout
3rd param: float 1.0 (direct) or -1.0 (inverse) — input

flog	Compiled	0.1864334 s	232	Apply

Compute logarithms of a *realimage*.
C Calling sequence: flog(imgin, type, imgout).
Link with $WPEweb/libWEB.a.

Parameters:
1st param: *realimage* — input
2nd param: 1 (natural) or 2 (base-10 logarithm) — input
3rd param: *realimage* — output

flwl0	Validated	**154.0 ms**	532	Apply

Execute linear filtering operation.

Parameters:
1st param: *image* — input
2nd param: float array[n][n] (weights) — input
n is compiled in the program. This one is 3.
3rd param: *image* — output

flwl1	Validated	**143.0 ms**	523	Apply

Execute linear filtering operation.

Parameters:	1st param: *image*	input
	2nd param: float array[n][n] (weights) n is compiled in the program. This one is 3.	input
	3rd param: float value (normalization coefficient)	input
	4th param: *image*	output

flwl2	Validated	**280.6 ms**	453	W2

Execute linear filtering operation using uniform weighting function.

Parameters:	1st param: *image*	input
	2nd param: float value (normalization coefficient)	input
	3rd param: *image*	output

fmax	Coded			Apply

Perform local max filtering.

Parameters:	1st param: *image*	input
	2nd param: *image*	output

fmin	Validated	**1038.8 ms**	685	Apply

Perform local min filtering.

Parameters:	1st param: *image*	input
	2nd param: *image*	output

fsed	Validated	**333.0 ms**	96	Apply

Convert an image to complex image (assigning zero to imaginary part).

Parameters:	1st param: *image*	input
	2nd param: *realimage* (real part)	output
	3rd param: *realimage* (imaginary part)	output

gmlt	Validated	**107.5 ms**	94	Apply

Multiply gray values (same as MULC1B).

Parameters:	1st param: *image*	input
	2nd param: int constant	input
	3rd param: *image*	output

grassfire	Coded			W2

Grassfire transform.
Input is binary image, output is distance from a 0.

Parameters:	1st param: *image*	input
	2nd param: *image*	output

gsft	Validated	**109.3 ms**	94	Apply

Shift gray values (same as ADDC1B).

Parameters:	1st param: *image*	input
	2nd param: int constant	input
	3rd param: *image*	output

gtrn1	Validated	**129.3 ms**	147	Apply

Apply grayvalue translation table.

Parameters:	1st param: *image*	input
	2nd param: int array[256] (table)	input
	3rd param: *image*	output

halftone	Compiled	0.4345898 s	218	W2

Image halftoning using Heckbert's algorithm

Parameters:	1st param: *image*	input
	2nd param: *image*	output

hist1 Validated **80.9 ms** 159 W2
Obtain histogram.
Parameters: 1st param: *image* input
2nd param: float array[256] (histogram) output

hyth Compiled W2
Hysteresis thresholding.
Link with ${WPEweb}/libWEB.a.
Uses byrl, connect.
C interface: canny(in, out, bounds, pbounds, low, high,
percent, verbose, err)
Parameters: in: input image input
out: thresholded output image output
bounds: SUBIMAGE bounds to perform thresholding input
bounds: SUBIMAGE bounds to compute percentages input
low: lower threshold input
high: higher threshold output
percent: int 0 (thresholds are absolute) input
or 1 (thresholds are percentages of range of values of image)
verbose: int 0 (quiet) or 1 (verbose) input
err: Generalized image library error parameter input

idct Validated **206.2 ms** 437 W2
Two dimensional inverse discrete cosine transform.
Takes an input image and performs 8 x 8 inverse discrete
cosine transforms to produce the output image.
Useful for image compression. (Inverse of dct).
Parameters: 1st param: *realimage* input
2nd param: *image* output

iten1 Compiled 4.598952 s 1348 Apply
Iterative enhancement of noisy image (method 1).
Parameters: 1st param: *image* input
2nd param: *image* output

iten2 Compiled 0.8456302 s 731 Apply
Iterative enhancement of noisy image (method 2).
C Calling sequence: iten2(imagein, imageout)
Link with $WPEweb/libWEB.a.
Parameters: 1st param: *image* input
2nd param: *image* output

mag Compiled 0.2611432 s 434 Apply
Gradient magnitude computation.
Parameters: 1st param: x gradient *realimage* input
2nd param: y gradient *realimage* input
3rd param: gradient magnitude *realimage* output

magdir Compiled 1.9610856 s 682 Apply
Gradient magnitude and direction computation.
Parameters: 1st param: x gradient *realimage* input
2nd param: y gradient *realimage* input
3rd param: gradient magnitude *realimage* output
4th param: gradient direction *realimage* output

medi Compiled 0.7777158 s 428 W2
Median filter.
Parameters: 1st param: *image* input
2nd param: *image* output

mmnt1 Compiled s 383 W2
Measure moments M_{pq} around center of gravity of regions in a labeled image. (p and q are the order of x and y components of the moment and are compiled in the program. This one is M_{11}.)
Parameters: 1st param: *image* input
2nd param: int array[256] input
Array with row coordinates of centers of gravity (output of CGRV1).
3rd param: int array[256] input
Array with column coordinates of centers of gravity (output of CGRV1).
4th param: int value input
Label of region to be processed; if 0, all regions are processed.
5th param: int array[256] output
Array[i] is the moment of region labeled i.

mmnt4 Validated **135.7 ms** 186 W2
Obtain 0th to 2nd moments of an image.
Parameters: 1st param: *image* input
2nd param: float array[6] output
Array with moments in the following order: 00, 10, 01, 20, 02, and 11.

mulc1b Validated **107.6 ms** 94 Apply
Multiply an image by a constant.
Parameters: 1st param: *image* input
2nd param: int constant input
3rd param: *image* output

mulc1c Validated **326.67 ms** 137 Apply
Multiply a complex image by a complex constant.
Parameters: 1st param: *realimage* (real part) input
2nd param: *realimage* (imaginary part) input
3rd param: float constant (real part) input
4th param: float constant (imaginary part) input
5th param: *realimage* (real part) output
6th param: *realimage* (imaginary part) output

mulc1r Validated **145.6 ms** 78 Apply
Multiply a real image by a real constant.
Parameters: 1st param: *realimage* input
2nd param: float constant input
3rd param: *realimage* output

mulc1s Validated **108.5 ms** 94 Apply
Multiply a signed byte image by a constant.
Parameters: 1st param: *image* input
2nd param: int constant input
3rd param: *image* output

mulp1b	Validated	**161.7 ms**	99	Apply	
	Multiply two images.				
Parameters:	1st param: *image*				input
	2nd param: *image*				input
	3rd param: *image*				output
mulp1c	Compiled	0.3194312 s	175	Apply	
	Multiply two complex images.				
Parameters:	1st param: *realimage* (real part)				input
	2nd param: *realimage* (imaginary part)				input
	3rd param: *realimage* (real part)				input
	4th param: *realimage* (imaginary part)				input
	5th param: *realimage* (real part)				output
	6th param: *realimage* (imaginary part)				output
mulp1ccj	Compiled	0.3194312 s	175	Apply	
	Multiply the 1st complex image by the complex conjugate of the 2nd complex image.				
Parameters:	1st param: 1st *realimage* (real part)				input
	2nd param: 1st *realimage* (imaginary part)				input
	3rd param: 2nd *realimage* (real part)				input
	4th param: 2nd *realimage* (imaginary part)				input
	5th param: *realimage* (real part)				output
	6th param: *realimage* (imaginary part)				output
mulp1r	Validated	**308.3 ms**	99	Apply	
	Multiply two *realimage*'s.				
Parameters:	1st param: *realimage*				input
	2nd param: *realimage*				input
	3rd param: *realimage*				output
mulp1s	Validated	**161.0 ms**	99	Apply	
	Multiply two signed byte images.				
Parameters:	1st param: *image*				input
	2nd param: *image*				input
	3rd param: *image*				output
noln1	Validated	**6449.1 ms**	319	W2	
	Nonlinear (quadratic) image warping using linear interpolation.				
Parameters:	1st param: *image*				input
	2nd param: float array[2][6]				input
	Homogeneous transformation matrix.				
	3rd param: *image*				output
noln2	Validated	**6928.4 ms**	345	W2	
	Nonlinear (quadratic) image warping using quadratic interpolation.				
Parameters:	1st param: *image*				input
	2nd param: float array[2][6]				input
	Homogeneous transformation matrix.				
	3rd param: *image*				output

noln3 Compiled 13.381153 s 727 W2
Nonlinear (quadratic) image warping using max, min, or nearest neighbor interpolation.
Parameters: 1st param: *image* input
2nd param: float array[2][6] input
Homogeneous transformation matrix.
3rd param: int value input
Select type of interpolation.
4th param: *image* output

nonmax Compiled 0.7827166 s 935 Apply
3x3 Canny-style non-maxima suppression.
Parameters: 1st param: x gradient *realimage* input
y gradient *realimage* input
gradient magnitude *realimage* input
non-maxima suppressed *realimage* output

not1 Validated **109.3 ms** 100 Apply
Logical negation of an image.
Parameters: 1st param: *image* input
2nd param: *image* output

orc1b Validated **107.8 ms** 99 Apply
Logically or an image with a constant.
Parameters: 1st param: *image* input
2nd param: int constant input
3rd param: *image* output

orp1b Validated **160.1 ms** 118 Apply
Logically or two images.
Parameters: 1st param: *image* input
2nd param: *image* input
3rd param: *image* output

pgen1 Validated **80.38 ms** 185 W2
Generate binary checkerboard pattern.
Parameters: 1st param: int value input
Width of checkerboard part.
2nd param: int value input
Height of checkerboard part.
3rd param: *image* output

pgen2 Tested 0.1542298 s 602 W2
Generate binary stripe pattern.
Parameters: 1st param: int value input
2nd param: int value input
Slope of stripes given by 1st-param/2nd-param.
3rd param: int value input
Width of stripes.
4th param: *image* output

pgen3	Tested	0.1395828 s	356	W2	
	Generate binary "bull's-eye" pattern.				
Parameters:	1st param: int value				input
	Center row-coordinate of concentric circles.				
	2nd param: int value				input
	Center column-coordinate of concentric circles.				
	3rd param: int value				input
	Interval between adjacent concentric circles.				
	4th param: 0 or 1				input
	Value in the innermost circle.				
	5th param: *image*				output
pgen4	Tested	0.9567508 s	454	W2	
	Generate binary diamond pattern.				
Parameters:	1st param: int value				input
	2nd param: int value				input
	3rd param: int value				input
	4th param: int value				input
	(1st-param/2nd-param and 3rd-param/4th-param give diamond edge slopes.)				
	5th param: int value				input
	6th param: int value				input
	(5th-param and 6th-param give diamond widths.)				
	7th param: *image*				output
pgen5	Tested	0.1302654 s	389	W2	
	Generate binary grid pattern.				
Parameters:	1st param: int value				input
	2nd param: int value				input
	(1st-param and 2nd-param specify size of rectangles.)				
	3rd param: *image*				output
prmt1	Tested	0.7775946 s	643	W2	
	Measure perimeter of regions in a labeled image.				
Parameters:	1st param: *image*				input
	2nd param: int value				input
	Label of region to be processed; if 0, all regions are processed.				
	3rd param: 4 or 8 (connectedness)				input
	4th param: int array[256]				output
	array[i] is the perimeter of region labeled i.				
pted	Tested	0.3178648 s	286	Apply	
	Extract or delete points in an image.				
Parameters:	1st param: *image*				input
	2nd param: 0 (deletion) or 1 (extraction)				input
	3rd param: int value (low threshold)				input
	4th param: int value (high threshold)				input
	5th param: 0 (within) or 1 (outside range)				input
	6th param: int value				input
	Label assigned to extracted points; if 0, keep input value.				
	7th param: *image*				output

pyramid	Compiled	0.1137702 s	1099	Apply	
	Pyramid reduction.				
Parameters:	1st param: *image*				input
	2nd param: char array[256][256]				output
	3rd param: char array[128][128]				output
	4th param: char array[64][64]				output
	5th param: char array[32][32]				output
	6th param: char array[16][16]				output
	7th param: char array[8][8]				output
reduce	Compiled	0.0907948 s	559	Apply	
	Image halving using linear interpolation.				
Parameters:	1st param: *image*				input
	2nd param: char array[256][256]				output
rlby	Validated	**131.9 ms**	75	Apply	
	Real to byte conversion.				
Parameters:	1st param: *realimage*				input
	2nd param: *image*				output
rlby1	Compiled	0.0709754 s	234	Apply	
	Real to byte conversion with wraparound at 256.				
Parameters:	1st param: *realimage*				input
	2nd param: *image*				output
rpla1b	Tested	0.0658724 s	59	Apply	
	Assign a constant to an image.				
Parameters:	1st param: int constant				input
	2nd param: *image*				output
rpla1r	Tested	0.0658702 s	48	Apply	
	Assign a real constant to a *realimage*.				
Parameters:	1st param: float constant				input
	2nd param: *realimage*				output
rpla2	Tested	0.248991 s	437	W2	
	Assign a constant to inside of an irregularly-shaped region in an image.				
Parameters:	1st param: *image*				input
	2nd param: *image*				output
	3rd param: int array[512] (top border)				input
	4th param: int array[512] (bottom border)				input
	5th param: int array[512] (left border)				input
	6th param: int array[512] (right border)				input
	7th param: int constant				input
	8th param: int value				output
	Number of points in the region.				
rqnt	Tested	0.0857254 s	160	Apply	
	Requantize image by reducing graylevels.				
Parameters:	1st param: *image*				input
	2nd param: int value				input
	Degree of reduction in gray levels.				
	3rd param: *image*				output
sclp	Tested	0.0669818 s	87	Apply	
	Binarize image by setting nonzero grayvalues to 1.				
Parameters:	1st param: *image*				input
	2nd param: *image*				output

size1	Tested	8.256E-4 s	125	W2	
	Measure size of regions.				
Parameters:	1st param: int array[256]				input
	Array with areas of regions (output of AREA1).				
	2nd param: int array[256]				input
	Array with perimeters of regions (output of PRMT1).				
	3rd param: float array[256]				output
	array[i] is the size of region labeled i.				
slth0	Compiled	0.0679034 s	106	Apply	
	Binarize gray-scale image using single threshold.				
	Output is 1 if image is greater than the threshold, 0 if less.				
Parameters:	1st param: *image*				input
	2nd param: int value (threshold)				input
	3rd param: *image*				output
slth1	Validated	**180.2 ms**	321	Apply	
	Binarize gray-scale image using single threshold.				
	Output depends on mode and threshold in the following manner:				
	Mode = 1: Output = 1 iff input > threshold				
	Mode = 2: Output = 1 iff input >= threshold				
	Mode = 3: Output = 1 iff input < threshold				
	Mode = 4: Output = 1 iff input <= threshold				
Parameters:	1st param: *image*				input
	2nd param: int value (threshold)				input
	3rd param: int value (binarization mode)				input
	4th param: *image*				output
slth2	Validated	**121.7 ms**	221	Apply	
	Binarize gray scale image using two thresholds.				
	mode = 0 : imageout = 1 iff thd1 >= imagein >= thd2				
	mode = 1 : imageout = 1 iff thd1 <= imagein or imagein <= thd2				
Parameters:	1st param: *image*				input
	2nd param: (thd1) int value (threshold)				input
	3rd param: (thd2) int value (threshold)				input
	4th param: (mode) int value (binarization mode)				input
	5th param: *image*				output
slth2m	Tested	0.1818864 s	215	Apply	
	Binarize gray scale image using two thresholds and a mask plane.				
Parameters:	1st param: *image*				input
	2nd param: *image* (mask plane)				input
	3rd param: int value (threshold)				input
	4th param: int value (threshold)				input
	5th param: int value (binarization mode)				input
	6th param: *image*				output
slth3	Tested	0.1892586 s	255	Apply	
	Binarize gray scale image using reference plane.				
Parameters:	1st param: *image*				input
	2nd param: *image* (reference plane)				input
	3rd param: int value (binarization mode)				input
	4th param: *image*				output
snns	Compiled		592	Apply	
	15 x 15 symmetric nearest-neighbor smoothing.				
Parameters:	1st param: *image*				input
	2nd param: *image*				output

srnk1	Tested	0.277216 s	519	Apply	
	Shrink using Levialdi's parallel algorithm.				
Parameters:	1st param: *image*				input
	2nd param: int connectedness (4 or 8)				input
	3rd param: *image*				output
srnk2	Compiled	3.549949 s	1526	W2	
	Shrink binary pattern using Rao's algorithm.				
Parameters:	1st param: *image*				input
	2nd param: *image*				output
srnk3	Compiled	0.2098556 s	596	Apply	
	Shrink binary pattern, separating touching blobs.				
Parameters:	1st param: *image*				input
	2nd param: *image*				output
subc1b	Validated	**109.1 ms**	94	Apply	
	Subtract a constant from an image.				
Parameters:	1st param: *image*				input
	2nd param: int constant				input
	3rd param: *image*				output
subc1c	Validated	**328.04 ms**	124	Apply	
	Subtract a complex constant from a complex image.				
Parameters:	1st param: *realimage* (real part)				input
	2nd param: *realimage* (imaginary part)				input
	3rd param: float constant (real part)				input
	4th param: float constant (imaginary part)				input
	5th param: *realimage* (real part)				output
	6th param: *realimage* (imaginary part)				output
subc1r	Validated	**146.4 ms**	78	Apply	
	Subtract a real constant from a real image.				
Parameters:	1st param: *realimage*				input
	2nd param: float constant				input
	3rd param: *realimage*				output
subc1s	Validated	**109.6 ms**	94	Apply	
	Subtract a constant from a signed byte image.				
Parameters:	1st param: *image*				input
	2nd param: int constant				input
	3rd param: *image*				output
subp1b	Validated	**161.7 ms**	99	Apply	
	Subtract 2nd input image from 1st input image.				
Parameters:	1st param: 1st *image*				input
	2nd param: 2nd *image*				input
	3rd param: *image*				output
subp1c	Validated	**623.19 ms**	165	Apply	
	Subtract 2nd input complex image from 1st input complex image.				
Parameters:	1st param: 1st *realimage* (real part)				input
	2nd param: 1st *realimage* (imaginary part)				input
	3rd param: 2nd *realimage* (real part)				input
	4th param: 2nd *realimage* (imaginary part)				input
	5th param: *realimage* (real part)				output
	6th param: *realimage* (imaginary part)				output

subp1r	Validated	**310.0 ms**	99	Apply	
	Subtract 2nd input *realimage* from 1st input *realimage*.				
Parameters:	1st param: 1st *realimage*				input
	2nd param: 2nd *realimage*				input
	3rd param: *realimage*				output
subp1s	Validated	**160.8 ms**	99	Apply	
	Subtract 2nd signed byte image from 1st signed byte image.				
Parameters:	1st param: 1st *image*				input
	2nd param: 2nd *image*				input
	3rd param: *image*				output
sumrcb	Validated	**80.83 ms**	124	W2	
	Sum the rows and columns of an image.				
Parameters:	1st param: *image*				input
	2nd param: float array[512] (row sums)				output
	3rd param: float array[512] (column sums)				output
sumrcr	Validated	**136.6 ms**	124	W2	
	Sum the rows and columns of an *realimage*.				
Parameters:	1st param: *realimage*				input
	2nd param: float array[512] (row sums)				output
	3rd param: float array[512] (column sums)				output
temx2	Compiled	*5.8877583 s*	749	Apply	
	Thin one-directional texture edges.				
Parameters:	1st param: *image*				input
	2nd param: float value Cosine of angle.				input
	3rh param: float value Sine of angle.				input
	4th param: *image*				output
tepa	Coded			Apply	
	Smooth image preserving texture edges.				
Parameters:	1st param: *image*				input
	2nd param: int value (threshold)				input
	3rd param: *image*				output
tfer1b	Tested	0.0661626 s	75	Apply	
	Transfer (copy) an image to another.				
Parameters:	1st param: *image*				input
	2nd param: *image*				output
tfer1r	Tested	0.0661626 s	75	Apply	
	Transfer (copy) a *realimage* to another.				
Parameters:	1st param: *realimage*				input
	2nd param: *realimage*				output
txav	Compiled	0.1398848 s	616	Apply	
	Average grayvalues in square neighborhood.				
Parameters:	1st param: *image*				input
	2nd param: *image*				output

txav2	Compiled	3.9827297 s	582	W2	
	Average grayvalues in square neighborhood with a certain angle.				
Parameters:	1st param: *image*				input
	2nd param: *image*				output
	3rd param: float value Cosine of angle.				input
	4th param: float value Sine of angle.				input
txdf1	Compiled	0.117301 s	430	Apply	
	Compute edge value of texture edge horizontally or vertically.				
Parameters:	1st param: *image*				input
	2nd param: 0 (horizontal) or 1 (vertical)				input
	3rd param: *image*				output
txdf2	Compiled	0.2342448 s	779	Apply	
	Compute edge value of texture edge of a specified size and direction.				
Parameters:	1st param: *image*				input
	2nd param: int value (size)				input
	3rd param: float value Cosine of angle.				input
	4th param: float value Sine of angle.				input
	5th param: *image*				output
txeg2	Compiled	0.795186 s	343	Apply	
	Compute best-edge size, direction, and value using results of TXDF1 or TXDF2.				
Parameters:	1st param: *image* Edge value.				input
	2nd param: *image* Old best edge value.				input
	3rd param: *image* Old best edge size.				input
	4th param: *image* Old best edge direction.				input
	5th param: int value (edge size)				input
	6th param: float value (edge direction)				input
	7th param: float value (lambda)				input
	8th param: *image* New best edge value.				output
	9th param: *image* New best edge size.				output
	10th param: *image* New best edge direction.				output
xconv	Validated	**1251.2 ms**	221	Apply	
	Convolution in the X (row) direction. 41-point convolution.				
Parameters:	1st param: *realimage*				input
	2nd param: array [41] of float				input
	3rd param: *realimage*				output
xorc1b	Validated	**109.1 ms**	99	Apply	
	Exclusive or an image with a constant.				
Parameters:	1st param: *image*				input
	2nd param: int constant				input
	3rd param: *image*				output

xorp1b	Validated	**161.1 ms**	118	Apply
	Exclusive or two images.			
Parameters:	1st param: *image*			input
	2nd param: *image*			input
	3rd param: *image*			output

yconv	Validated	**3086.7 ms**	649	Apply
	Convolution in the Y (column) direction. 41-point convolution.			
Parameters:	1st param: *realimage*			input
	2nd param: array [41] of float			input
	3rd param: *realimage*			output