NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

CMU-CS-86-137

MAPPING IMAGE PROCESSING OPERATIONS ONTO A LINEAR SYSTOLIC MACHINE

H. T. Kung and Jon A. Webb
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

15 March 1986

Abstract

A high-performance systolic machine, called Warp, is operational at Carnegie Mellon. The machine has a programmable systolic array of linearly connected cells, each capable of performing 10 million floating-point operations per second. Many image processing operations have been programmed on the machine. This programming experience has yielded new insights in the mapping of image processing operations onto a parallel computer. This paper identifies three major mapping methods that are particularly suited to a Warp-like parallel machine using a linear array of processing elements. These mapping methods correspond to partitioning of input dataset, partitioning of output dataset, and partitioning of computation along the time domain (pipelining). Parallel implementations of several important image processing operations are presented to illustrate the mapping methods. These operations include the fast Fourier transform (FFT), connected component labelling, Hough transform, image warping and relaxation.

The research was supported in part by Defense Advanced Research Projects Agency (DOD), monitored by the Air Force Avionics Laboratory under Contract F33615-84-K-1520, and Naval Electronic Systems Command under Contract N00039-85-C-0134, in part under ARPA Order number 5147, monitored by the US Army Engineer Topographic Laboratories under contract DACA76-85-C-0002, and in part by the Office of Naval Research under Contracts N00014-80-C-0236, NR 048-659, and N00014-85-K-0152, NR SDRJ-007.

1 Introduction

Warp is a programmable systolic array machine designed at Carnegie Mellon [1] and built together with its industrial partners—GE and Honeywell. The first large scale version of the machine with an array of 10 linearly connected cells is operational at Carnegie Mellon. Each cell in the array is capable of performing 10 million 32-bit floating-point operations per second (10 MFLOPS). The 10-cell array can achieve a performance of 50 to 100 MFLOPS for a large variety of signal and low-level vision operations.

Warp's first applications are low-level vision for robots, laboratory processing of images [4] and analysis of large images (10K×10K). It has therefore been important for us to understand how to map typical image processing operations onto the Warp array so that work can be partitioned evenly between all the cells.

We have identified three major mapping schemes. They include two methods of space division (that is, the assignment of part of the data to each cell): one is *input partitioning*, or the assignment of part of the input to each cell, and the other is *output partitioning*. The third method is *pipelining*: each cell does one stage of a multi-stage computation. All the mapping schemes that we have used on Warp for image processing operations are covered by these three mapping methods.

We begin with a brief review of the Warp machine and architectural features of its linear processor array that have allowed efficient mapping of many image processing operations onto the machine. Then we consider each mapping method, describe how it is done, and list its advantages and disadvantages and other characteristics. More precisely, in Section 3 we consider the input partitioning method, and present a new parallel implementation for the connected component labelling computation. Section 4 deals with the output partitioning method, illustrated by Hough transform and image warping. Section 5 discusses the pipelining method using the FFT and relaxation as examples.

2 The Warp Machine

In this section we first give an overview of the Warp system, and then describe the linear processor array used by the system.

2.1 System Overview

The Warp machine has three components—the Warp processor array, or simply Warp array, the interface unit, and the host, as depicted in Figure 1. We describe this machine only briefly here; more detail is available separately [1]. The Warp processor array performs the bulk of the computation—in this case, low-level vision routines. The interface unit handles the input/output between the array and the host. The host has two functions: carrying out high-level application routines and supplying data to the Warp processor array.

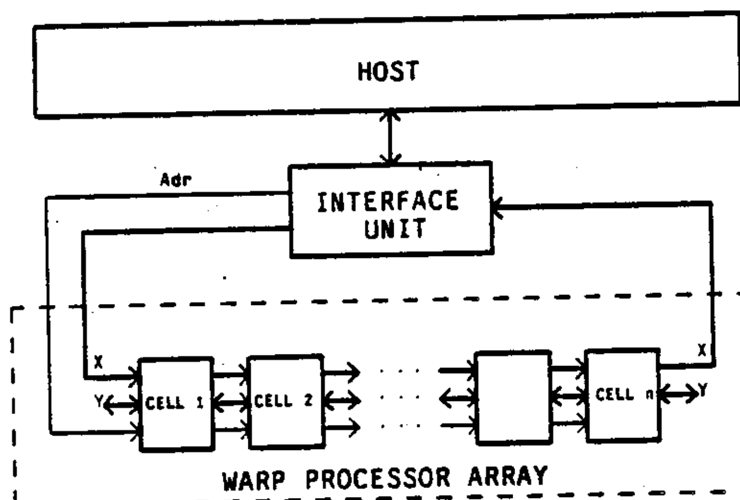

Figure 1. Warp machine overview.

The Warp processor array is a programmable, linear systolic array, in which all processing elements (Warp cells) are identical. Data flow through the array on two data paths (X and Y), while addresses and systolic control signals travel on the Adr path (as shown in the Figure 1). The data path of a Warp cell is depicted in Figure 2. Each cell contains two floating-point processors: one multiplier and one ALU [16]. These are highly pipelined; they each can deliver up to 5 MFLOPS. This performance translates to a peak processing rate of 10 MFLOPS per cell or 100 MFLOPS for a 10-cell processor array. To ensure that data can be supplied at the rate they are consumed, an operand register file is dedicated to each of the arithmetic units, and a crossbar is used to support high intra-cell bandwidth. Each input path has a queue to buffer input data. A 32K-word memory (expandable to 64K-word) is provided for resident and temporary data storage.

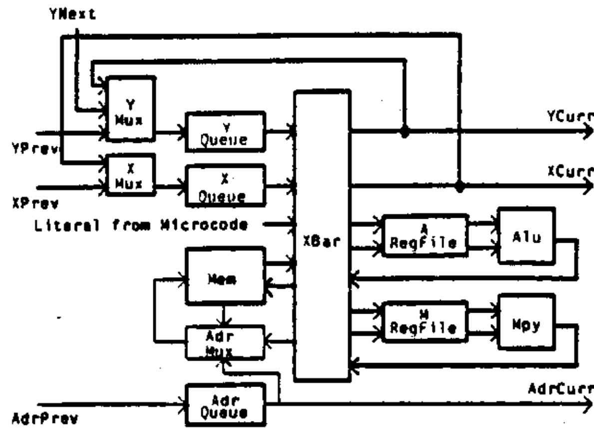

Figure 2. Warp cell data path.

As address patterns are typically data-independent and common to all the cells, full address generation capability is factored out from the cell architecture and provided in the interface unit. Addresses are generated by the interface unit and propagated from cell to cell (together with the control signals). In addition to generating addresses, the interface unit passes data and results between the host and the Warp array, possibly performing some data conversion in the process.

The host is a general purpose computer (currently a Sun workstation, with added MC68020 cluster processors for I/O and control of the Warp array). It is responsible for executing high-level application routines as well as coordinating all the peripherals, which might include other devices such as the digitizer and graphics displays. The host has a large memory in which images are stored. These images are fed through the Warp array by the host, and results from the Warp array are stored back into memory by the host. This arrangement is flexible. It allows the host to do tasks not suited to the Warp array, including low-level tasks, such as initializing an array to zero, as well as tasks that involve a lot of decision making but little computation, such as processing a histogram (computed by the Warp array) to determine a threshold.

2.2 The Warp Processor Array

The Warp array is a linear array of identical cells. The linear configuration was chosen for several reasons. First, it is easy to implement. Second, it is easy to extend the number of cells in the array. Third, a linear array has modest I/O requirements since only the two end-cells communicate with the outside world.

The advantages of the linear interconnection are outweighed, however, if the constraints of interconnection render the machine too difficult to use for programmers. The concern is whether or not we can efficiently map applications on the linear array of cells. While many algorithms have been

designed specifically for linear arrays [6], computations designed for other interconnection topologies can often be efficiently simulated on the linear array mainly because the Warp cell, the building-block of the array, is itself a powerful engine. In particular, a single Warp cell can be programmed to perform the function of a column of cells, and therefore the linear array can, for example, implement a two-dimensional systolic array in an efficient way.

A feature that distinguishes the Warp cell from many other processors of similar computation power is its high I/O bandwidth—an important characteristic for systolic arrays. Each Warp cell can transfer up to 20 million words (80 Mbytes) to and from its neighboring cells per second. (In addition, 10 million 16-bit addresses can flow from one cell to the next cell every second.) We have been able to implement this high bandwidth communication link with only modest engineering efforts, because of the simplicity of the linear interconnection structure and clocked synchronous communication between cells. This high inter-cell communication bandwidth makes it possible to transfer large volumes of intermediate data between neighboring cells and thus supports fine grain problem decomposition.

For communicating with the outside world, the Warp array can sustain a 80 Mbytes/sec peak transfer rate. In the current setup, the interface unit can communicate with the Warp array at a rate of 40 Mbytes/sec. This assumes that the Warp array inputs and outputs a 32-bit word every (200 ns) instruction cycle. However the current host can only support up to 10 Mbytes/sec transfer rates. The smaller transfer rate supported by the host is not expected to affect the effective use of the Warp array for our applications for the following reasons: First, the Warp array typically does not perform an input and and output more than every two or more cycles. Second, for most signal and image processing applications, the host deals with 8-bit integers rather than 32-bit floating-point numbers, and therefore the I/O bandwidth for the host needs only be a quarter of that for the Warp array (assuming the image can be sent in raster-order). This implies that 10 Mbytes/sec transfer rate for the host is sufficient.

Each cell has a large local data memory that can hold 32K or 64K words; this feature is seldom found in special-purpose, systolic array designs, and is another reason why Warp is powerful and flexible. It can be shown that by increasing the data memory size, higher computation bandwidth can be sustained without imposing increased demand on the I/O bandwidth [7]. The large memory size, together with the high I/O bandwidth, makes Warp capable of performing *global* operations in which each output depends on any or a large portion of the input [8]. Examples of global operations are FFT, component labelling, Hough transform, image warping, and matrix computations such as matrix multiplication. Systolic arrays are known to be effective for local operations such as a 3×3

convolution [6]. The additional ability to perform global operations significantly broadens the applicability of the machine.

In summary, the simple linear processor array used in Warp is a powerful and flexible structure largely because the array is made of powerful, programmable processors with high I/O capabilities and large local memories.

Carnegie Mellon is continuing development of the Warp hardware with its industrial partners. A major effort of developing a VLSI Warp chip that can implement the entire Warp cell except its local memory has started. The integrated Warp system will achieve an order of magnitude reduction in cost, size, and power consumption, and a similar increase in computing power. It will consist of tens of cells with a total computing power of over a billion floating-point operations per second.

3 Input Partitioning

In this section, we describe the first mapping method—input partitioning. In input partitioning each cell receives part of the input data, and generates all of the output corresponding to its set of inputs. Each processor does the same operation on its portion of the input. The outputs are then combined. We will illustrate this mapping method by considering a rather elaborate example, that is, the labelling of connected components in an image.

3.1 Issues in the Input Partitioning Method

Input partitioning is natural in low-level image processing. Image operations are often local and regular, or produce data structures that are fairly easy to combine, as in histogram. Image sizes tend to be large (for example, 512×512) so that much parallelism is available even if the image is divided only along one dimension (for example, each cell gets one set of adjacent columns). In addition, this method lends itself naturally to raster-order processing; if the image is divided according to columns, the image can be sent in in raster order, while each cell accepts its portion of the inputs as they are pumped through the array.

The overhead in computation time for input partitioning comes from three sources: first, the cost of dividing the image into its parts; second, the cost of any additional bookkeeping on the Warp cell to allow outputs to be recombined later (see below)); and third, the cost of recombining outputs.

For many computations, on a linear systolic array, these costs are negligible. Consider convolution-like operations with a small kernel, for example. In these computations, each output depends only on a local, corresponding area of the input image. The computation is partitioned so that each cell takes a set of adjacent columns from the image. The cost of partitioning the data is simply the cost of sending a row through the array while each cell picks off its part of the data, which is negligible since systolic arrays have sufficient internal bandwidth to allow the communication of the data through all cells at little cost. There is no additional bookkeeping, and the cost of recombining outputs is simply the cost of concatenating the outputs from the different columns of the image, which again is negligible.

The advantages of input partitioning are its ease of programming and the good speedup it usually gives to an algorithm. Using a simplified cost model, we can calculate the speedup from input partitioning as follows. Suppose computation of the output for the complete input set on one cell takes time n. Then computation of the input on k cells takes time $n/k+kc$, where c is the time required to combine two cell's outputs, assuming the outputs are combined in sequence over the

cells. If we are allowed to optimize this by adjusting the number of cells the optimimum point has $k=\sqrt{n/c}$ Computation is wasted if the number of cells is greater than this, and as the number of cells approaches this point adding more cells becomes less cost-effective. For 512×512 image processing, we have $n=512\times512\times d$, where d is the cost of computing the operation on one input pixel. Assume that each cell takes one operation to read in the previous cell's outputs, and another operation to insert its outputs and send them to the next cell. Then $c=2$. Hence the optimal number of cells is $362\times\sqrt{d}$, much more than the number of Warp cells in the Warp machine, so that no time is wasted. For 8-bit histogram, $c=256$ (the cost adding two histograms), and $d=3$ (based on the current implementation of histogram on Warp) so the optimal number of cells is approximately 55, again much more than the number of Warp cells.

There are, however, two potential disadvantages to the input partitioning method. First, intermediate data structures may be duplicated at each cell, wasting memory. For example, consider labelled histogram: in this algorithm the input is a standard grayvalue image and a labelled image (produced by a connected components algorithm). The output is a set of histograms, with each histogram i counting the frequency of occurrence of pixel values in the standard image whose location has the label i in the labelled image. If each cell computes the histogram for a portion of the input image, then the same table of histograms spanning all the labels (which may be in the hundreds) must be maintained at every cell. Second, it is not always possible to find a simple method to recombine outputs. In fact, devising an efficient way for output recombination can be quite a challenge.

Consider for example one important algorithm in image processing—the extraction of region features from a labelled image. In this algorithm, significant regions of the image have been labelled, so that the input is an image with each region identified by having its pixels all be the same label, and the task is to reduce this labelled image to a set of symbolically described regions and their properties. Once this is done, the region property set is processed, for example, to find a large region of a particular color that may correspond to a road. Many different properties may be required to do this, and the particular property set depends on the application: properties may include area, center of gravity, moment of inertia, grayvalue mean and variance, boundary properties such as number of concavities, and internal properties such as number of holes. Some of these properties (such as area) are quite easy to compute separately and recombine later, while others (such as number of holes) may require some careful thought to figure out how they can be fit into the input partitioning method.

For example, how can a cell determine the number of holes lying on the boundary of the portion of

the image for which the cell is responsible, as shown in Figure 3? Luckily in this case, there is a relatively simple solution. Assume that we have also labelled the complement of the shaded region using a connected components algorithm as described below. Then a cell can determine whether a region on the boundary is hole or not by examining the label of the pixels of the region. This can be done using the following method: first, make sure that the outer border of the image is labelled consistently by creating a border around the image that is all zero. Then each cell can determine the "inside" relationships of all the labels in any column as follows: it scans the column from the border in, and the first time a new label is encountered, the region that label represents is inside the region represented by the label in the row just above. (This method can be rigorously proved using the Jordan Curve Theorem [2]).

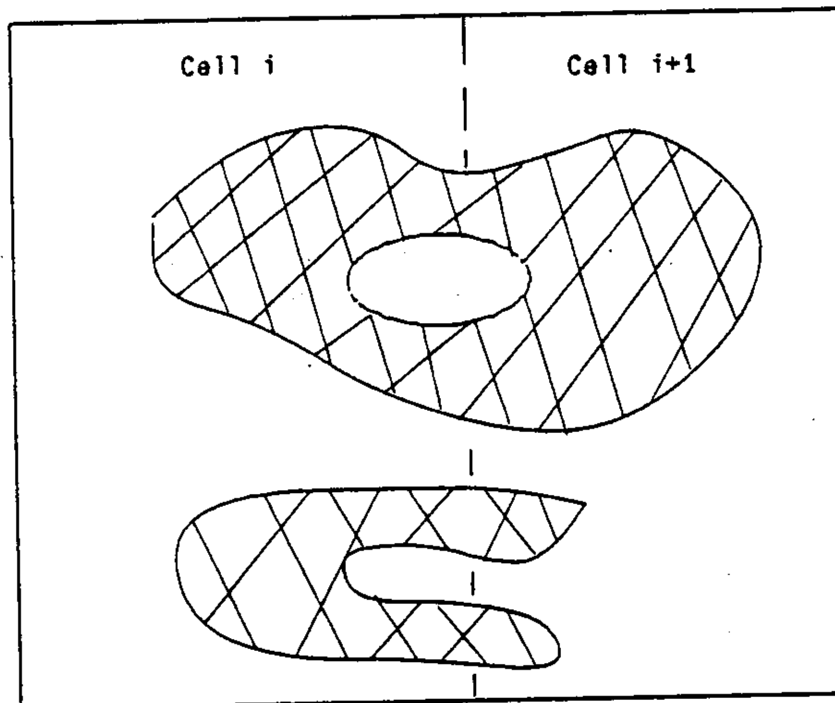

Figure 3. Determining the number of holes on the boundary of two sub-images.

However, combining outputs in an input partitioned method is not always this easy, as we shall see in the next section.

3.2 Component labelling

Assume that significant pixels in a given image are identified by some means. (For example, in a binary image, the significant pixels have the value 1.) All the significant pixels form a set of connected components; the component labelling problem is to produce a labelled image in which two pixels have the same label if, and only if, they belong to the same connected component. We first describe a

serial labelling algorithm suited to implementation on one Warp cell. We then show that a parallel labelling algorithm that can be efficiently implemented on an array of Warp cells, each essentially executing the serial labelling algorithm on a sub-image.

3.2.1 Serial labelling Algorithm

The serial algorithm presented here appears to be more efficient than many previously known methods in the crucial step of sorting components that are connected into equivalence classes [11, 14]. Our algorithm is similar to a new algorithm by Jack Schwartz, et al. [13] Their algorithm has the potential that it can be efficiently implemented in a special-purpose hardware with a small memory. Our algorithm, on the other hand, readily leads to a parallel version as we shall see in the next subsection.

Our algorithm works in two passes. The first pass takes the image in raster-scan order, and produces a partially labelled image and a label mapping table (LMT) for each row. The partially labelled image has the property that the labels for each row are correct (in the sense that two pixels have the same label if and only if they belong to the same connected component) if only that row and all rows above are taken into account. The second pass takes the partially labelled image and LMT's in reverse order (that is, reverse raster-scan order) and produces a final labelled image. The LMT translates labels on one row into the labels on the previous row that correspond to the same connected component.

In the first pass, we scan each row twice, in two phases. The first phase scans the row left to right and produces a partially labelled row, a (tentative) LMT, and a re-labelling table (RLT). The RLT is used to unify labels in a row that belong to the same connected component; RLT(a) = RLT(b) if a and b belong to the same connected component based on information from the current and previous rows. The second phase scans the row right to left and applies the RLT to produce the final labellings and LMT for that row.

For each row the first phase works as follows: scan the row from left to right, and for each significant pixel, label it using the following strategy:

> Let P be the pixel we are considering, A the left neighbor of P on the row above, and B the left neighbor of P on the same row, as shown in Figure 4. Let $l(X)$ denote the label of pixel X, and let LMT(l) denote the value of the LMT table for label l. A pixel P gets a label $l(P)$ according to the following procedure:
>
> 1. If LMT($l(A)$) exists, then $l(P)$ = LMT($l(A)$).

2. If method 1 is not possible, and B is a significant pixel, $l(P) = l(B)$.

3. If methods 1 and 2 are not possible, $l(P)$ gets a new label.

A	C	C
B	P	

Figure 4. Pixel labels.

Suppose that $l(C)$ is the label of a neighboring pixel on the row above (either above or immediately to the right of P). Then LMT($l(C)$) = $l(P)$ should be added to the LMT if this entry did not exist in the table. If the entry already existed in the table with LMT($l(C)$) = d and $l(P)$ not equal to d, then the relabelling table (RLT) for the current row should be updated to include the entry RLT($l(P)$) = d and $l(P)$ should be set to d.

Similarly, suppose that $l(P)$ is the new label of a pixel, and $l(B)$ is the label of the left neighbor. Then RLT($l(B)$) = $l(P)$ should be added to the RLT if this entry did not exist in the table.

In the second phase, we rescan the row right to left and produce the final labellings and LMT for the first pass. This phase works as follows:

Let P be the pixel we are considering, and $l(P)$ its label from the first phase. We will replace $l(P)$ with a label from the RLT. If RLT($l(P)$) has no value, no replacement is done. Otherwise, we consider RLT(RLT($l(P)$)). If this has no value, we replace $l(P)$ with RLT($l(P)$). If RLT(RLT($l(P)$)) has a value, we replace $l(P)$ with it and, at the same time, set RLT($l(P)$) = RLT(RLT($l(P)$)) in the RLT table.

After completing the rescan of the row, we use the RLT table to update the LMT table obtained from the first phase. That is, if $l(P)$ is a value in the LMT table and RLT($l(P)$) has a value, then we replace $l(P)$ with RLT($l(P)$) in the LMT table.

The following property holds during the first pass processing:

P1. When finishing the second phase for a row, any two pixels on the row that can be connected by a path lying totally on or above the row must have the same label.

We prove property *P1* by induction on rows, starting from the top row.

- Property *P1* clearly holds for the first row.

- Suppose that property *P1* holds for the kth row. We want to prove it for the $(k+1)^{st}$ row. Consider any two pixels *x* and *y* on $(k+1)^{st}$ row that can be connected by a path (without loops) lying totally on or above the $(k+1)^{st}$ row, as illustrated by Figure 5. Assume that the entire path does not lie on the $(k+1)^{st}$ row, otherwise *P1* clearly holds.

  Imagine that the portion of the path that lie on the $(k+1)^{st}$ row is removed. Then the remaining path is broken into segments which lie entirely above the $(k+1)^{st}$ row. The end points of each segment are on the kth row, and by the induction hypothesis, they have the same label.

  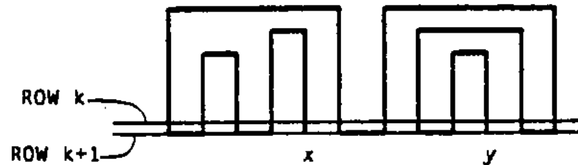

  Figure 5. A path connecting two pixels *x* and *y* on the $(k+1)^{st}$ row.

  Because the original path has no loops, the segments are all disjoint. Therefore, they cannot cross each other, but can be nested as shown in Figure 5.

  Consider those segments, called outer segments, which are not nested inside any other segments. The outer segments must all be connected by line segments on the $(k+1)^{st}$ row, which are part of the original path. It is easy to check that the first pass algorithm will label all the pixels on these line segments with the same label. The operation of the first pass algorithm for these labellings is totally independent from other possible segments which are nested inside the outer segments.

  Now consider any arbitrary outer segment, together with those segments, called the inner segments, that are *immediately* nested inside the outer segment. These segments must all be connected by line segments on the $(k+1)^{st}$ row, which are part of the original path. The following fact holds after first phase of the first pass. All pixels on the line segments that are connected to the left end point of the outer segment will receive the same label, say, *a*; and pixels on the line segments that are connected to the right end point of the outer segment will receive label *a* or another label, say, *b*, with RLT(*b*) = *a* in the RLT table. After the second phase, all the pixels on the line segments receive the same label, namely, *a*. Again, these labelling operations are totally independent from whatever segments that may be nested inside the inner segments.

  To complete the proof, we continue applying the above arguments to cover all the segments of the original path.

For the second pass, we scan the image in reverse order, from bottom to top, and use the LMT tables produced by the first pass to propagate labels upwards. We use two working tables (WT and WTold), which are assumed to be empty initially. The algorithm works by scanning each row from right to left as follows:

Let *P* be the pixel we are considering and *l*(*P*) its label from the first pass. If WT(*l*(*P*)) has a

value, label P with that value. If not, check to see if WTold(LMT($l(P)$)) has a value, say a. If so, assign P the label a, and insert WT($l(P)$)$=a$ in the WT. If WTold(LMT($l(P)$)) has no value, assign P the next available label number, say, b, and insert WT($l(P)$)$=b$ in the WT.

At the end of the scanning of each row, we assign WTold the current value of WT, and set WT to be empty.

The following property holds during the second pass processing:

P2. When finishing the scan of a row, any two pixels on or below the row that can be connected by a path must have the same label.

We prove property *P2* by induction on rows, starting from the bottom row.

- Property *P2* clearly holds for the bottom row.
- Suppose that *P2* holds for the $(k+1)^{st}$ row. We want to prove that it holds for the kth row. Consider any two pixels x and y on or below the kth row that can be connected by a path. We can assume at least one of x and y is on the kth row.
    - Without loss of generality, assume that y is on the kth row and x is not. We trace the path from x to y. Consider the point v where the path is on the $(k+1)^{st}$ row for the last time. Let w be the next point of the path. Then w must be on the kth row. By the induction hypothesis, x and v have the same label. By the algorithm of the second pass, w has the same label as v. The proof is complete if y is w. If y is not w, then y and w are connected by a segment which lies totally on or above the $(k+1)^{st}$ row. By the property *P1* of the first pass, y and w have the same label at the end of the first pass, and therefore also at the end of the second pass.
    - Assume that both x and y are on the kth row. We trace the path from x to y. Consider points u and v where the path is on the $(k+1)^{st}$ row for the first and last time, respectively. Then by an argument similar to the above one, we can show that u and v, x and u, and v and y have the same label.

Implementation of this algorithm on a single Warp cell is straightforward. The most complex data structures are one-dimensional arrays whose size is one scan line, which easily fits a Warp cell's memory, and which can be addressed using Warp's addressing facilities. The algorithm must be modified, however, to run on multiple cells. This modification is explained in the next section.

3.2.2 Parallel labelling Implementation

Assume that there are k cells in the Warp array. We split the image into k equal size regions horizontally, and assign regions to cells in order from top to bottom.

- Step *A*: All the cells label their own regions simultaneously, using the serial labelling algorithm presented above.

- Step B: Cells 2 to k perform the following operations in the sequential order:

  - Step B_2: Cell 2 performs the following operations on the first and last rows of region 2.

    1. Input from cell 1 the last row of region 1 and its labels.

    2. Re-label the first row of region 2 (assuming that above it is the last row of region 1 input from cell 1 in the previous step), remember the re-labelling, and save the resulting LMT table as LMT_2. (This labelling is done by the same scheme as the one applied to a typical row in the first pass of the serial labelling algorithm.)

    3. Apply the same re-labelling (remembered from the previous step) to the last row of region 2.

  - Step B_3: Cell 3 performs operations similar to those of step B_2 on the first and last rows of region 3, and saves the resulting LMT table as LMT_3.

  - ...

  - Step B_k: Cell k performs operations similar to those of step B_2 on the first and last rows of region k, and saves the resulting LMT table as LMT_k.

- Step C: Cells k to 1 perform the following operations in the sequential order:

  - Step C_k: Cell k performs the following operations:

    1. Initialize the working table WT to be empty.

    2. For each pixel P on the last and first rows of region k, if $WT(l(P))$ has a value, assign it to be the label of P. If not, assign P the next available label number, say, b, and insert $WT(l(P))=b$ in the WT.

    3. Save WT as WT_k.

    4. Form a new working table WTnew by inserting to it $WTnew(a)=WT(LMT_k(a))$ for each label a assigned to the last row of region $k-1$. (Note that cell k input all the labels assigned to the last row of region $k-1$ in the first step of step B_k.)

  - Step C_{k-1}: Cell $k-1$ sets WT to be the WTnew input from cell k, performs operations similar to those in the second step of step C_k on the last and first row of region $k-1$, save WT as WT_{k-1}, and forms WTnew for cell $k-2$.

  - ...

  - Step C_1: Cell 1 sets WT to be the WTnew input from cell 2, performs operations similar to those in the second step of step C_k on the last row of region 1, and save WT as WT_1

- Step D: All the cells re-label their own regions simultaneously, with different labels used for each region. More precisely, for each $i=1,\ldots,k$, cell i re-labels rows in region i in order from bottom to top. Consider the operations of cell i. For the bottom row, the cell assigns Wold to the value of WT_i from step C, initializes WT to be empty, and re-labels the row in the following manner: for each pixel P on the row, if $WT(I(P))$ has a value, assign it to be the label of P. If not, check to see if $WTold(I(P))$ has a value, say a. If so, assign P the label a, and insert $WT(I(P))=a$ in the WT. If $WTold(I(P))$ has no value, assign P the next available label number, say, b, and insert $WT(I(P))=b$ in the WT. At the end of the scanning of the row, cell i assigns WTold to the current value of WT, and sets WT to be empty. Cell i then performs similar operations on the row above.

We see that steps B and C are essentially the first and second passes, respectively, of the serial algorithm applied only to those rows which are on the boundaries of adjacent regions. Since after step A pixels belonging to the same connected component in each region have the same label, the labels produced by steps B and C are the same as those produced by the first and second passes, respectively, of the serial algorithm applied to the *whole* image. Therefore after step C, the bottom row of each region will have the final labels, and as a result all other rows of each region will receive the final labels during step D.

Note that sub-steps in steps B and C must be carried out sequentially, that is, by only one cell at a time. However, the time taken by these steps is not significant with respect to the total time of the parallel algorithm, when the number of rows n in the given image is much larger than the number of cells k in the Warp machine. More precisely, suppose that the serial algorithm takes time T_1. Then the time for the parallel algorithm is roughly:

$$T_k=(T_1/k)+(T_1k/n),$$

where the first term is the time for steps A and D, and the second term is that for steps B and D. Therefore the speedup ratio is

$$\frac{T_1}{T_k}=k\frac{n}{n+k^2}.$$

For $n=512$ and $k=10$ (as in the case of the current Warp machine), the speedup ratio is more than 8.

4 Output Partitioning

In this section we consider image processing operations where the value of each output depends on the entire input dataset, or a subset that cannot be determined *a priori*. The input partitioning method described in the preceding section does not work well here, because any subset of the input dataset alone will not have all the information needed for generating any output. For these operations, it is often natural to use the second mapping method—output partitioning. In output partitioning the output dataset is partitioned among the cells. Each cell sees the entire input dataset and generates part of the output set. Outputs from different cells are concatenated as they are fed out of the array. We will use the Hough transform and image warping to illustrate the output partitioning method.

4.1 Issues in the Output Partitioning Method

An advantage of this method is that it does not require the duplication of intermediate data structures necessary in input partitioning, such as the LMT in component labelling. It is less wasteful of memory. Each cell must maintain only the data structures it needs to compute its outputs, which it would have to anyway (there is no cost to this unless output data structures overlap).

In this method, for maximum efficiency on a synchronous machine like Warp, each cell must be able to make use of each data item. The reason for this is that as the data are sent in, each datum must be either used or rejected; but because the decision to accept or reject a datum must be made at run-time, enough time must be left to process the datum even if it is rejected to maintain synchrony. As a result, no time is saved by rejecting data. Consider the labelled histogram computation. As we observed in the input partitioning section, it may not be possible to partition this algorithm by input because duplication of data structures may lead to too large memory requirements (since hundreds of histograms must be stored at each cell). If output partitioning is used in this algorithm, and the histograms are split across cells (by, for example, storing a range of the histogram elements at each cell) then each datum is used by only one cell (the cell that holds the histogram elements for that data range) and so there is no speedup from parallelism. There appears to be no good method of implementing this algorithm on a synchronous distributed memory machine like Warp if the local memory of a cell is not large enough for the intermediate data structures needed for input partitioning. In this case either the algorithm must be output partitioned, without an increase in speed over one cell, or the algorithm must be implemented on Warp's cluster processors.

In output partitioning, the only overhead is duplication of effort when intermediate results for one output could contribute to the computation of another output, but these intermediate results are on

different cells and so must be recomputed. For example, in FFT the computation of any single output requires nearly the same effort as to calculate all outputs. There is no overhead associated with broadcasting the input data or combining the output data: on Warp, data can be broadcast at little cost. Since each cell has all the information it needs to compute each output, all that has to be done is to concatenate outputs, which can also be implemented at negligible cost on Warp.

In input partitioning there was an advantage if each cell took a set of adjacent columns of the input data; for local neighborhood operations, it is necessary to have a set of adjacent columns that form the neighborhood. No such input or output advantage exists for output partitioning; since each cell sees the entire dataset, it can produce any set of outputs that is necessary.

As examples of output partitioned algorithms, we present the Hough transform and image warping.

4.2 The Hough Transform

The Hough transform is a template matching algorithm originally invented to find lines in cloud chamber photographs [5] and later generalized to find arbitrary parameterized curves [3]. The algorithm works by mapping each significant pixel into a set of locations in a table representing different locations in the parameter space. The mapping takes each pixel in the image into all possible combinations of parameters generating curves that pass through the image pixel.

For example, in line-finding, lines are parameterized by two values, θ and ρ. The line described by a particular pair of values of these parameters is

$$x \cos \theta + y \sin \theta = \rho.$$

Thus, for line finding, the Hough transform takes the (x,y) location of each significant pixel and, over a range of θ values, calculates the ρ value for this (x,y) using the formula above. It then increments a table at location (θ, ρ). Once the entire data set has been processed, the table is scanned and peaks are found. These peaks are the most likely lines in the image.

This algorithm can be used to find any other curve by changing the mapping between the image space and the parameter space. For example, to find circles of a certain size one maps each significant pixel into the (x, y) locations of centers of circles that pass through this pixel (which is itself a circle in the parameter space).

The time-consuming step in this algorithm is the mapping between the image and the parameter space. This can involve floating-point computation and must be done once for each significant pixel in the image, which can be a good portion of the image. Also, the parameter space searched can be quite large, depending on its dimensionality and the granularity of the parameter search.

The Warp implementation of the Hough transform works by dividing the parameter space into different segments to a cell's memory, then allocating each segment to one Warp cell. The host preprocesses the image by selecting significant pixels and sending their locations to Warp (alternatively, these pixel locations can be generated in a pass of the image through Warp). The location of each significant pixel is sent to every cell systolically. At each cell, the segment of the Hough space that belongs to the cell is indexed by some set of parameters $p_1, p_2, \ldots, p_n$. The pixel location is fed into a formula with some particular value of the first $n-1$ parameters and the nth parameter is generated. Table lookup can be used to simplify computation of the nth parameter. The table element at this location is then incremented. This process is repeated until the computation for the entire segment belonging to the cell is completed, and then the pixel location is sent on to the next cell. Thus in the steady state, all the Warp cells carry out computations simultaneously for different segments.

After all the data has been sent through the Warp cell, each cell selects its significant peaks and sends them to the host where the maxima of all the peaks can be found.

4.3 Image Warping

In some applications, it is desirable to remap or "warp" an image according to some transformation. This may be done, for example, if the lens distortions introduced by a particular calibrated lens are to be undone. A more interesting application of this technique [15] is in a robot vehicle which is following a road; the camera is mounted on the vehicle, and of course sees an image distorted by perspective projection. If we know the position of the ground plane relative to the camera, we can warp the image so that it appears as it would from an aerial view. Once this is done, it is relatively easy to find the road edges. For example, after this transformation road edges appear as parallel lines.

In a typical application of this technique a "fish-eye" lens is used (because this lens gives a very wide angle view of the road, making it easier to keep the road in view when going around turns). The point $(-1,i,j)$ on the ground plane is first projected onto the unit sphere centered at the origin, then perpendicularly to the image plane which is tangent to the sphere at (0,0,1). The overall transformation is $M(i,j)=(-1,i)/\sqrt{1+i^2+j^2}$. Figure 6 shows the center line and edges of a road being first projected onto the surface of a unit sphere, and then onto the image plane.

Using the inverse of this transformation, each pixel in the output image is mapped into some pixel in the input image, which may not be at an integral location. Suppose M^{-1} maps each pixel location (i,j) of the output image to a location $(x+\Delta_x, y+\Delta_y)$ in the input image, where x, y are integers and

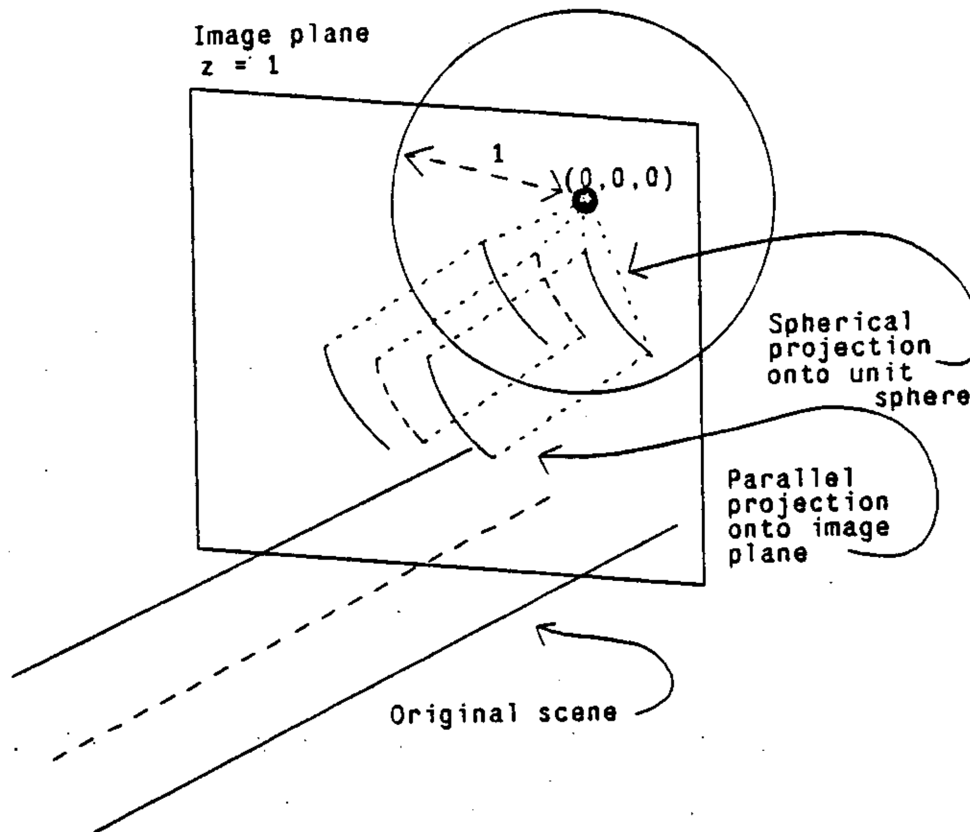

Figure 6. Fish-eye lens projection.

$0 \le \Delta_x, \Delta_x < 1$. The pixel value o at location (i,j) for the output image is the interpolation of the values of the four pixels—$i_{nw}, i_{ne}, i_{sw}, i_{se}$—surrounding $(x+\Delta_x, y+\Delta_y)$ in the input image. More precisely,

$$o = [i_{nw}\Delta_x + i_{ne}(1-\Delta_x)](1-\Delta_y) + [i_{sw}\Delta_x + i_{se}(1-\Delta_x)]\Delta_y.$$

To compute the output image, we must, for each location (i,j), first evaluate the function M to compute $(x+\Delta_x, y+\Delta_y)$ and then compute o.

In the Warp implementation, each cell first receives and stores the entire input image. Then the cells generate the output image in parallel. Each of the n cells takes every n^{th} output element, computes its value from the input image, and sends its value out of the array.

If a cell's memory is not large enough to hold a complete image, the algorithm can be done in several passes, with each cell taking the same portion of the input and generating what outputs it can from the given input portion. The partial output images can then be combined to create the complete output image. Note that this algorithm is memory limited; doubling the memory available to a cell leads to a doubling of the speed of the algorithm, up to the point that a cell can hold a complete image.

5 Pipelining

In pipelining, an algorithm is broken down into steps, and each cell does one step of the computation. Inputs flow into the first cell, are subjected to a series of operations, and eventually. become outputs that flow out of the last cell. Intermediate results flow between cells. This mode of computation is what is usually thought of as "systolic processing."

This method requires that the algorithm be regular, so that each cell can do nearly an identical operation. This is not often the case in image processing, so that this method has not been as widely used as the space partitioning methods mentioned before.

This method has the advantage that there is no duplication of data structures between cells; each cell maintains only the data structures necessary for its stage of the computation. Also, the input and output sets are not divided, so that there is no extra cost associated with splitting them up or recombining them.

We illustrate this method with the FFT. Other image processing tasks suited to this mode of computation are one-dimensional convolution [6] and relaxation [12], which is discussed in the next section.

5.1 Fast Fourier Transform

The problem of computing an n-point discrete Fourier transform (DFT) is as follows:

Given $x_0, x_1, \ldots, x_{n-1}$,
compute $y_0, y_1, \ldots, y_{n-1}$ defined by
$y_i = x_0\omega^{i(n-1)} + x_1\omega^{i(n-2)} + \cdots + x_{n-1}$
where ω is a primitive n-th root of unity.

Assume that n is a power of two. The well known fast Fourier transform (FFT) method solves an n-point DFT problem in $O(n\log n)$ operations, while the straightforward method requires $O(n^2)$ operations. The FFT involves $\log_2 n$ stages of $n/2$ butterfly operations, and data shufflings between any two consecutive stages. The so-called constant geometry version of the FFT algorithm allows the same pattern of data shuffling to be used for all stages [9], as depicted in Figure 7 with $n=16$. In the figure the butterfly operations are represented by circles, and number h by an edge indicates that the result associated with the edge must be multiplied by ω^h.

We show that the constant geometry version of the FFT can be implemented efficiently with Warp [6]. In the Warp array, all the butterfly operations in the i-th stage are carried out by cell i, and

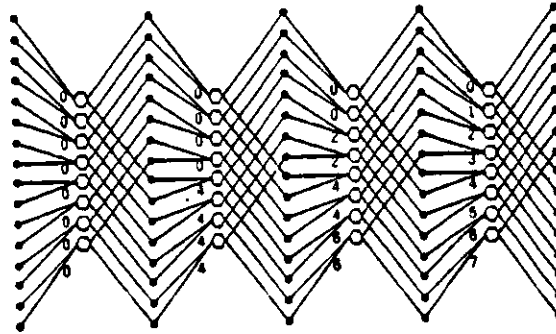

Figure 7. Constant geometry version of FFT.

results are stored to the data memory of cell $i+1$. While the data memory of cell $i+1$ is being filled by the outputs of cell i, cell $i+1$ can work on the butterfly operations in the $(i+1)^{st}$ stage of another FFT problem. Note that every cell accesses its memory in a shuffled order. As the same shuffling is performed for all the stages, the interface unit can send the same address stream to all the cells. In practical applications, there are often a large number of FFTs to be processed, or there are FFT problems being continuously generated. Thus it is possible that a new FFT problem can enter the first cell, as soon as the cell becomes free. In this way all the cells of the systolic array can be kept busy all the time.

5.2 Relaxation

In some image processing algorithms, the input image is subject to multiple passes of the same operation [10, 12]. This process is called *relaxation*, in which pass $i+1$ uses the results of pass i. A natural way to implement relaxation on Warp is to have cell i perform pass i and send results to cell $i+1$.

Relaxation could also be implemented on Warp using the input partitioning method. But this requires more involved communication and control between cells. For example, as each cell completes a pass of the relaxation method over its portion of the image, it must communicate the results of its computation to its neighbors—both its predecessor, and its successor. This requires bidirectional communication on the Warp array, which is physically possible, but difficult to schedule.

Therefore, it is better to implement relaxation by pipelining, with each cell performing one stage of the relaxation. In image processing, it is common for relaxation algorithms to require tens of passes over the image to stabilize (since images are large, it takes many passes for constraints to relax across the entire image). Thus, the ten-cell size of the Warp array is suited to image processing relaxation methods.

6 Conclusions

It is somewhat surprising and convenient that only three algorithm partitioning methods seem to work for all image processing algorithms that can be implemented on Warp. In this section we analyze the reasons for this result.

First, we must note that image processing has several characteristics that help make these partitioning techniques, or variants of them, work on any parallel computer. Images are large, which is a ready source of parallelism, and at the low level results from one part of the image do not influence what computation is done in another part of the image. This makes it easy to distribute computation among cells by splitting up the input or output images, since there is little communication among distant parts of the image.

Second, the short, linear structure of Warp and its high internal bandwidth make it easy to divide algorithms up among cells. The algorithm must be divided only along one dimension, and only among ten cells, rather than two-dimensionally among hundreds or thousands of cells. This means there is a negligible cost associated with broadcasting, splitting, or combining the data among all cells.

Third, the host can prepare inputs for Warp, as in the Hough transform, where it filters out non-significant pixels so only significant pixels will be processed by the Warp array. This makes load balancing in the array easier.

Fourth, each cell has a fairly large local memory. This memory can be used to store intermediate results for output after the entire data stream is processed. This is used in the FFT, component labelling, image warping and the Hough transform. The relatively large local memory of each cell (and fast inter-cell communication) are necessary for many operations as shown in a recent analysis [7].

Fifth, the cells have programmable local control. Different cells can do different things as long as the timing is the same. Therefore different global operations, or different computation stages needed for one global operation, can be processed at different cells in one pass. This alleviates the host I/O bottleneck problem.

References

[1] 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*. June, 1986.

[2] Apostol, T.
Mathematical Analysis.
Addison-Wesley, Reading, Massachusetts, 1974.

[3] Ballard, D. H. and Brown, D.M.
Computer Vision.
Prentice-Hall, 1982.
pp. 123-31.

[4] 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.

[5] Hough, P. V. C.
Method and Means for Recognizing Complex Patterns.
United States Patent Number 3,069,654, December, 1962.

[6] Kung, H.T.
Systolic Algorithms for the CMU Warp Processor.
In *Proceedings of the Seventh International Conference on Pattern Recognition*, pages 570-577. International Association for Pattern Recognition, 1984.

[7] Kung, H.T.
Memory Requirements for Balanced Computer Architectures.
Journal of Complexity 1(1):147-157, 1985.
(A revised version also appears in Conference Proceedings of the 13th Annual International Symposium on Computer Architecture, June 1985).

[8] 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.

[9] Rabiner, L.R. and Gold, B.
Theory and Application of Digital Signal Processing.
Prentice-Hall, Englewood Cliffs, New Jersey, 1975.

[10] Rosenfeld, A.
Iterative methods in image analysis.
In *Proceedings of the IEEE Computer Society Conference on Pattern Recognition and Image Processing*, pages 14-18. International Association for Pattern Recognition, 1977.

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

[12] Rosenfeld, A., Hummel, R. A., and Zucker, S. W.
Scene labelling by relaxation operations.
IEEE Trans. on Systems, Man, and Cybernetics SMC-6:420-433, June, 1976.

[13] Schwartz, J., Sharir M. and Siegel, A.
An Efficient Algorithm for Finding Connected Components in a Binary Image.
Robotics Research Technical Report 38, New York University, Courant Institiitue of Mathematical Sciences, February, 1985.

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

[15] Wallace, R., Matsuzaki, K., Goto, Y., Crisman, J., Webb, J. and Kanade, T.
Progress in Robot Road-Following.
In *Proceedings of 1986 IEEE International Conference on Robotics and Automation.* April, 1986.

[16] Woo, B., Lin, L. and Ware, F.
A High-Speed 32 Bit IEEE Floating-Point Chip Set for Digital Signal Processing.
In *Proceedings of 1984 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 16.6.1-16.6.4. 1984.