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.

Object Recognition on a Systolic Array

Claire M. Bono and Jon A. Webb

CMU-RI-TR-87-21

Department of Computer Science
The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

September 1987

e 1987 Carnegie Mellon University

The research was supported in part by Defense Advanced Research Projects Agency (DOD), monitored by the Air Force Avionics Laboratory under Contract F33615-81-K-1539, and Naval Electronic Systems Command under Contract N00039-85-G-0134, in part by the U.S. Army Engineer Topographic Laboratories under Contract DACA76-85-C-QQQ2, and in part by the Office of Naval Research under Contracts MG0014-80-C-Q236, NR 048-659, and N00014-8S-K~0152, NR SDRJ-007. CMrc Bcmo was supported by an Office of Naval Research graduate fellowship.

Table of Contents

1. Introduction 1
2. An object recognition system 1
3. A VLSI Systolic Array Machine 3
3.1. Models of computation on iWarp 4
4. Intermediate-Level Vision on a Systolic Array 4
4.1. Techniques for performing joins and selections on a systolic array 7
4.1.1. Finding values in $A \times A$ 7
4.1.2. Finding values in $A \times B$ 8
4.1.3. Finding values in A 10
5. Mapping Koezuka and Kanade's Algorithm onto iWarp 10
6. Simulation Results 12
7. Conclusions 13
8. Acknowledgement 14

UNIVERSITY LIBRARIES
CARNEGIE-MELLON UNIVERSITY
PITTSBURGH, PENNSYLVANIA 15213

List of Figures

Figure 2-1: General flow-chart for Koezuka and Kanade's recognition system [8] 2
Figure 3-1: Some computation models on iWarp 5
Figure 3-2: The multi-function pipeline model as used on Koezuka and Kanade's algorithm 6

List of Tables

Table 2-1: Matching with and without filtering 2
Table 6-1: Operations, Memory, and I/O for one run of Koezuka and Kanade's Algorithm 13
Table 6-2: Recognition times for several images (in milliseconds) 13

Abstract

Computer vision systems for recognition include both the extraction of features and the matching of those features with a known model. Traditionally, the most time consuming step has been feature extraction, but new parallel architectures are removing the bottleneck at this level. Once features have been extracted from an image considerable geometric search is still necessary to form relationships between the extracted features and to match those features and feature aggregates with a model. One can take advantage of certain constraints about the appearance of an object, but with complex images or multiple models intensive processing is still required. We have developed some algorithms for doing these geometric search operations in parallel on iWarp, a long linear array of VLSI processing elements currently being designed by Carnegie Mellon and Intel Corporation. We have simulated a system which uses these algorithms to do an object recognition task (after low-level vision) almost completely on a 72 processor iWarp array. An analysis of this system indicates a speedup by a factor of roughly 100 to 250 over a sequential version running on a VAX 8650.

1. Introduction

A common paradigm for computer vision recognition systems, which was originally used by Roberts [10] is:

1. Extract features from an image.
2. Use partial feature matches to constrain possible model matches.
3. Match image features with model features, and report most likely matches.

Because of the size of images, traditionally most of the processing time is spent in the first step. However, because of the independence of the feature extraction process across the image, it is relatively easy to use fast parallel computers to speed up this step [7]. Once this is done, the second and third steps dominate the processing time. The characteristics of this part of the vision problem which make it slow are that it involves forming and testing *combinations* of image features as well as requiring geometric processing on these features. If we can use parallelism to speed up these steps, we can afford to do more of this type of processing, and build more robust vision systems, or we can develop vision systems that are as reliable as those we have now, but much faster.

We are studying the use of systolic processors at this level. We believe the pipelining effect of systolic arrays provides a powerful and natural programming model for these steps. In this paper we will focus on a particular computer vision algorithm, used for a bin-picking task in which a single object model is matched with an image to determine the orientation of an object in a bin of identical objects. This algorithm exhibits the characteristics of object recognition algorithms we discussed above. Our study of this task helps give insight into this level of vision in general. We mapped this algorithm onto a long systolic array, consisting of 72 iWarp cells. iWarp is a VLSI implementation of Warp, a systolic array developed at Carnegie Mellon and General Electric [1,2,3], where each cell of Warp is implemented by a single iWarp chip. Our simulation results give speedup estimates of a factor of roughly 100 to 250 over the VAX 8650 implementation of the algorithm.

2. An object recognition system

Although it is not the primary focus of this paper, we describe in some detail the sequential object recognition system we studied. Tetsuo Koezuka and Takeo Kanade developed this system [8] while Koezuka was visiting Carnegie Mellon in 1986. The task is to find the most likely match (viewing position) of a single known polyhedral object model in an image which may contain several instances of the object Koezuka and Kanade's system uses a pre-compilation technique [6] that greatly reduces the amount of work required when a potential match is compared between the object and the model. A CAD model of the object is constructed, and 2D shape data for many different views (distributed regularly over a sphere) are generated. In each view, the visible line segments are stored, and are used in the matching step. (There is no attempt to choose just those views that show different collections of image lines.) The system was tested using an L-shaped block model and real images. In this system relationships between lines, such as angles and distances, are pre-compiled and are used to narrow down the search of the possible viewing directions. By drastically reducing the total number of matching trials, this technique gives good performance: *in* the VAX 8650 implementation, a complete match of an image took only 5 seconds. Figure 2-1 shows the model tested, and an example image and corresponding output from the system.

like many systems of this sort, the matching between the image and the model is done on the basis of a feature that has already been extracted from the image. Koezuka and Kanade used Hoes because they assumed their objects were industrial parts which include a great'many flat surfaces and straight edges. Additional assumptions made in this system are that the distance between the camera and the object is known (presumably because the object is lying on a known surface) and the camera is distant enough from the object that scaled orthographic projection can be assumed.

In Koezuka and Kanade's system impossible matches of lines in the image with lines in the modal are discarded

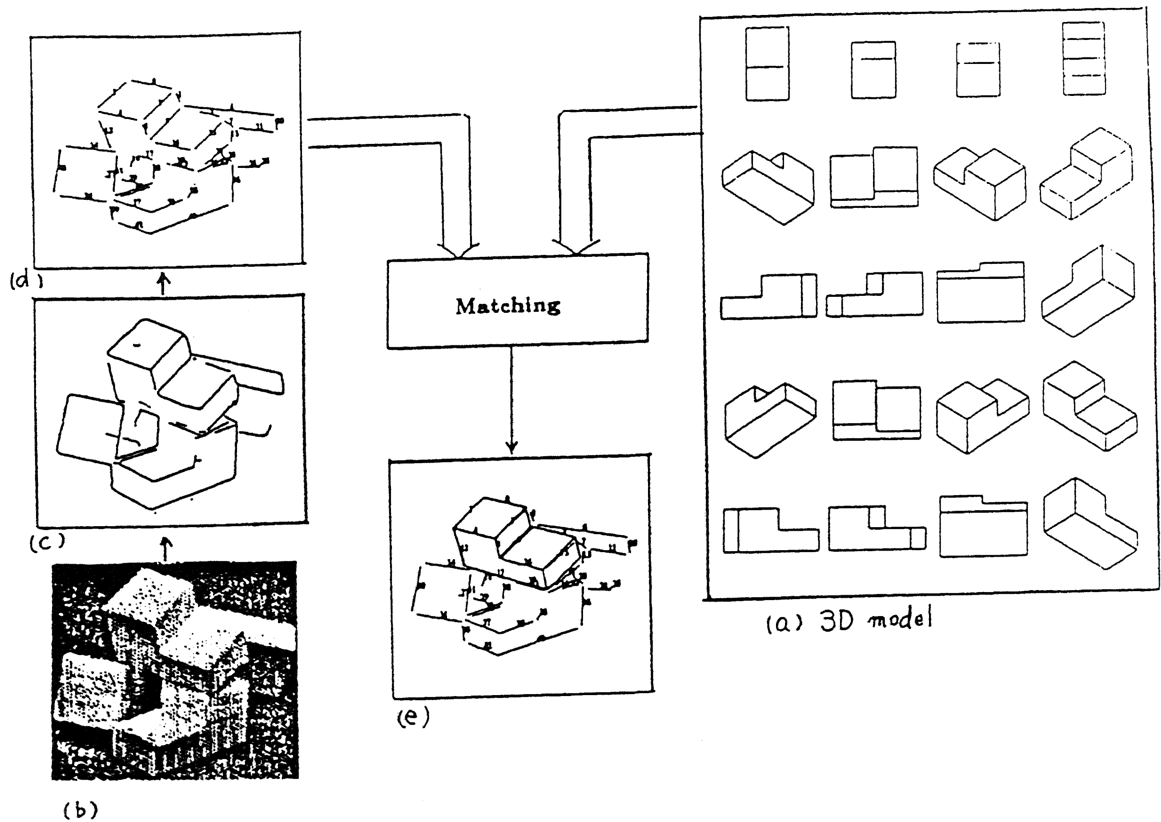

Figure 2-1: General flow-chart for Koezuka and Kanade's recognition system [8]

as soon as possible. This is a common technique, and it is important to be able to take advantage of it in a parallel implementation. Table 2-1 shows results for a number of images. It gives the major result of Koezuka and Kanade's work, which is the enormous reduction in the number of matches with filtering; the ratio is from 10^8 to 10^{11}. No conceivable parallel machine could make up for this large a factor reduction in the number of matches; therefore, it is very important that we structure the parallel implementation so as to take advantage of this reduction. We cannot simply propose to build a machine which does the recognition by considering every possible match in parallel [5].

Table 2-1: Matching with and without filtering

Number of Image Lines	Potential Matches	Actual Matches	Ratio
42	1.1×10^{14}	30047	3.8×10^{9}
37	5.7×10^{12}	7759	7.3×10^{8}
41	6.5×10^{13}	16256	4.0×10^{9}
37	5.7×10^{12}	15791	3.6×10^{8}
45	5.5×10^{14}	25676	2.1×10^{10}
48	2.3×10^{15}	5835	4.0×10^{11}
43	2.0×10^{14}	1336	1.5×10^{11}

Matching proceeds in several steps:

1. **Parallel lines.** Parallel lines are found in the image.
2. **Distance-length condition.** Parallel image lines are matched with the model. Under scaled orthographic projection, given the distance between the object and the camera, it is known that the image lines must appear to be no farther apart than the matching lines. Similarly, the length of the image lines must be less than or equal to the matching model lines. These constraints are used to filter the match.
3. **Orientation condition.** Two pairs of parallel image lines are matched with two pairs of parallel model lines. Constraints on the angles between the pairs are used to filter the match.
4. **Verify.** Given the match from parallel pairs match, we compute the possible viewing directions of the model. If there are no possible viewing directions, the match is rejected.
5. **Matching.** All the line segments in the pre-compiled 2D shape data from the determined viewing direction are matched with corresponding line segments in the image. The 2D shape with the largest matching degree is selected as the result.

We have shown the flow of data via these steps because we will be elaborating on them later in the paper. In the actual Koezuka and Kanade implementation, which was optimized for speed, the different steps are merged and structured as follows:

```
for each image line A
    for each image line B
        if A and B are parallel
            for each model line pair (C,D) which
               can appear to be parallel in the scene
               if (A,B) and (C,D) can match
                  ...
```

This is a natural programming style in a language such as C. In our parallel implementation the series of nested **for** and **if** statements are replaced by a series of stages, most of which generate combinations of data objects and filter them. These different steps can be done in parallel by mapping them onto different groups of processors. As background for understanding the mapping, in the next section we describe the hardware we are using.

3. A VLSI Systolic Array Machine

The iWarp machine is a reimplementation of the Warp machine in VLSI. The machine is being designed by Carnegie Mellon and Intel Corporation. It differs in several ways from the Warp machine:

- More total computing power: a 72 cell iWarp machine (the baseline machine) will have 1.1 to 1.4 GFLOPS power, versus 100 MFLOPS for Warp.
- More general computational model: the floating point units on the cells have no pipeline delays. There is also a general purpose ALU for address computation. These two changes will make it possible to write complicated algorithms that use complex data structures.
- Larger array: the iWarp baseline machine has 72 cells, versus 10 for Warp. This implies the need for software tools to aid the programmer in distributing his program.
- More memory: each cell has 64K words of memory in the baseline machine, four times the amount in Warp. This enables us to store larger images than in the past on one cell for applications where the whole image is needed locally.
- Flexible I/O model: there is hardware support for communication in parallel with computation. While the baseline iWarp machine is still a linear array, this hardware support allows distant cells to communicate without the intervention of intermediate cell's computational units.

The size, power, and cost of the iWarp machine are expected to be within a factor of two of the Warp machine.

The most significant change in programming model between Warp and iWarp is the flexible I/O model. In this model, different cells communicate via "messages" which pass through intermediate cell's communication units, but are not seen by the computation units. Once a message is created, it defines a point-to-point communication pathway through which "blocks" of data may be sent. These blocks may be read and processed a word at a time, or they may be stored into memory by a background process for later processing.

An extension to this model allows multiple cells to communicate using the same message. A cell may open a connection in a message in which it is not a termination point, but rather an intermediate stop. Once this is done, a cell can read blocks of data from the message, write blocks of data, and redirect blocks on to other cells sharing the same message. This extension is important because it reduces the number of point-to-point communications which are necessary; by time-multiplexing a message, the hardware cost of support for messages can be reduced.

The general communication between arbitrary cells is important to effectively use the large iWarp array, and will be used extensively in our mapping of the object recognition system onto the array. Now that we have described communication mechanisms at the hardware level, we will explain communications usage at the software level.

3.1. Models of computation on iWarp

There are several models of computation in use on iWarp, which we discuss in this paper. These models are useful guidelines for a programmer using the machine, as well as for building programming tools which incorporate the models automatically. Some of these models have been defined previously [9]. We will be using all but the first model in our mapping of Koezuka and Kanade's algorithm. Refer to Figures 3-1 and 3-2.

1. In *pipelined* computation, the classic systolic array model, the algorithm is broken down into nearly identical stages, each cell doing one stage of the computation. Each cell communicates with its immediate neighbor, passing intermediate results along the way, with final results emerging from the rightmost cell. This model is used for regular computations such as convolution, fast Fourier transform, and dynamic programming.
2. In *output partitioning* each cell computes partial results on all the input data, but in this case the partial results are independent, i.e. each cell computes part of the set of outputs, but needs to see all of the inputs. This model is used for global operations in which each output can depend on any input, such as image warping and Hough transform.
3. In *input partitioning* the input is divided up among the cells; input sent to a particular cell is processed only at that cell. This model is used for non-regular local operations, such as edge detection and smoothing, and has been implemented as the Apply programming model [7]. It has been used extensively on Warp for image processing.
4. *Addressed input* computation is a special form of input partitioning where the cell to which a data item is sent is determined at run time. One can think of the destination of the datum as its address. This model is supported only on iWarp, not Warp, and is used to partition a problem among cells without the overhead of having the receiving cells do the partitioning themselves.
5. In *multi-function pipelined* computation there are two or more blocks of cells computing with one of the models above, and the output of one block is used as the input to the next. This model has been used in Warp, for example, to compute the histogram of an image in one stage, and apply a translation table in a later stage. This model will be more important in iWarp, where the long linear array forces the use of multiple functions to utilize the array efficiently.

4. Intermediate-Level Vision on a Systolic Array

Intermediate-level vision differs in important ways from low-level vision:

- All operations are global; any input can affect any output. In contrast, in low-level image processing, such as edge detection, the result of an edge detection at a pixel depends only on a corresponding window around the input.

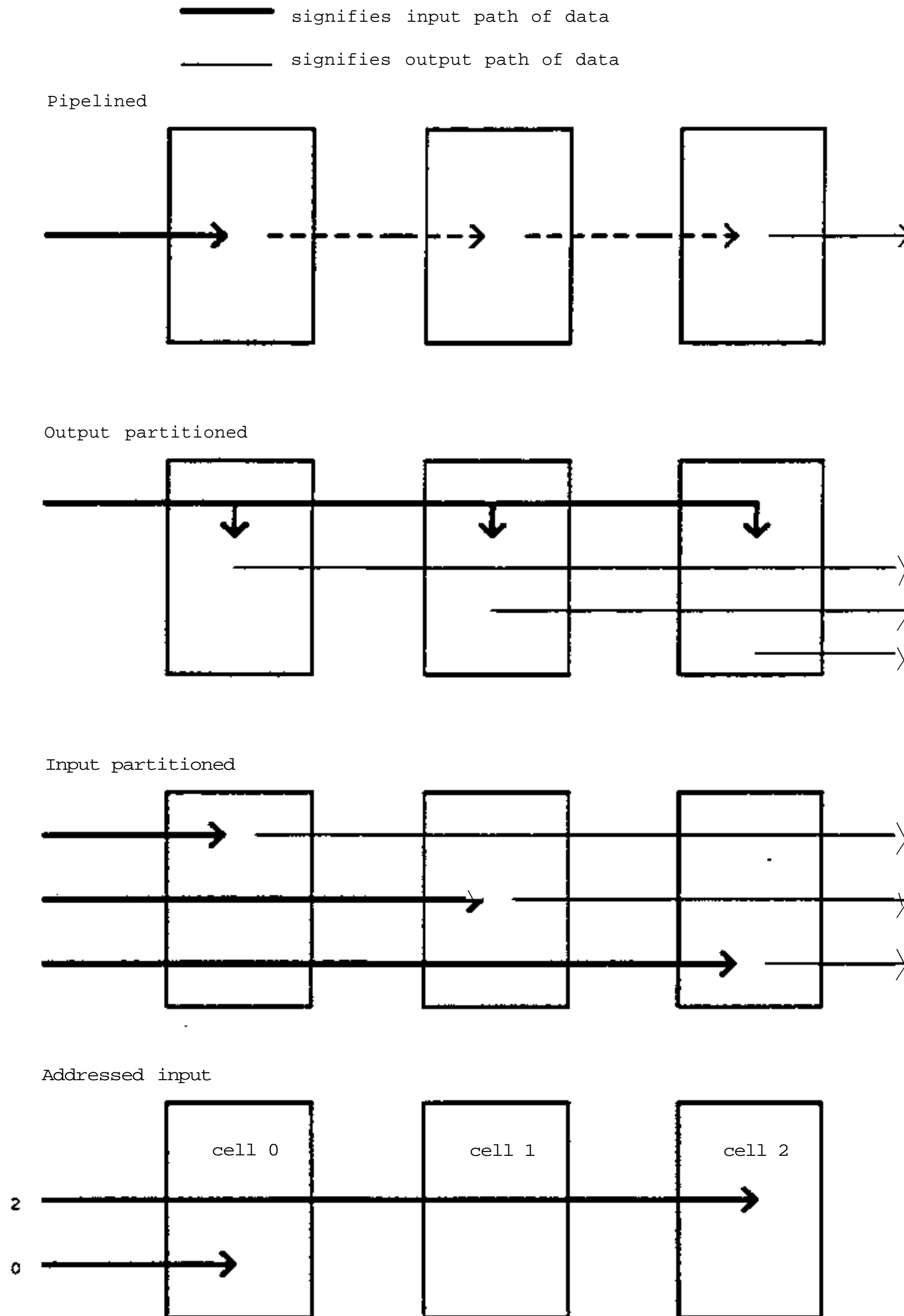

Figure 3-1: Some computation models on iWaip

- The data size grows and shrinks during the processing, as combinations are generated and discarded. Moreover, this growing and shrinking is data dependent, and may be impossible to predict in advance.

We therefore had to consider very different mappings for intermediate-level vision from low-level vision, and had to make use of iWarp's flexibility to efficiently implement this level of vision. We considered several mappings of this problem onto the iWaip array. Let us examine each of these in mm:

1. Dividing computation among cells by dividing the input data, and nmnkig the complete algorithm on

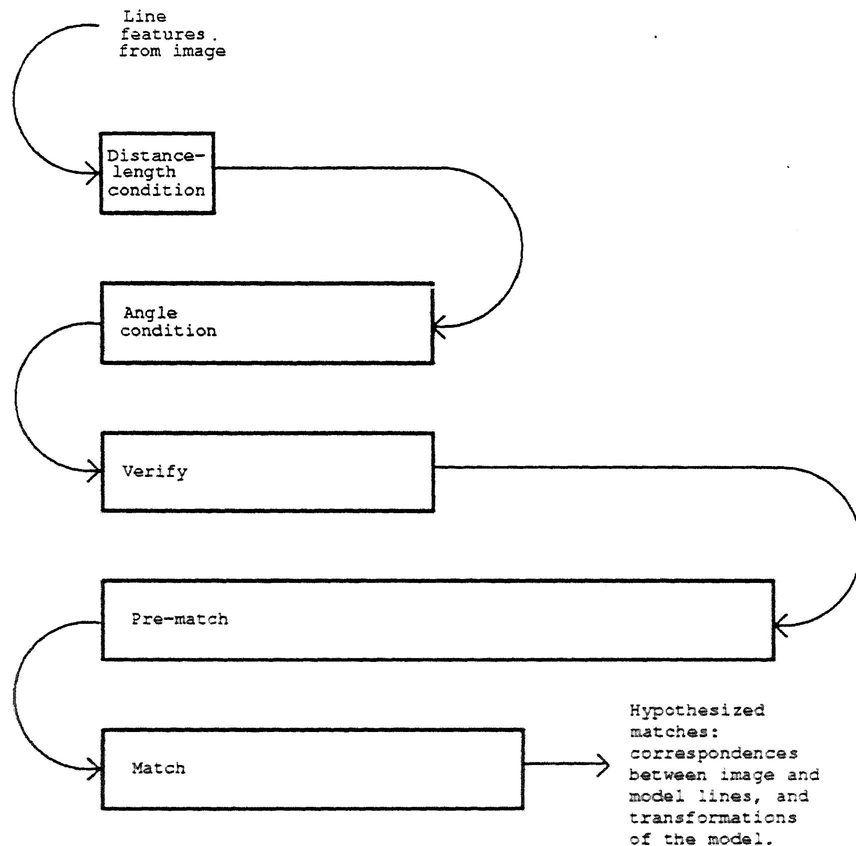

Figure 3-2: The multi-function pipeline model as used on Koezuka and Kanade's algorithm

each portion of the data on a single cell.

2. Multiple passes through the array, by spreading each step in the algorithm across the complete array.
3. A multi-function pipeline, using either input or output partitioning as appropriate for each step in the algorithm.

In the first scheme, all of the decisions about some data are made local to a cell. However, in Koezuka and Kanade's algorithm many operations are made that combine different data items – to create pairs of items, pairs of pairs, etc. – so that, either cells must duplicate work of other cells, or a complex communications pattern must be used to communicate results between cells. Implementing such a communications pattern on a linear array like iWarp is extremely difficult, and there are even possibilities of deadlock. This mapping does not take advantage of locality within the array; distant cells would have to communicate as often as adjacent cells. Also, this method requires the duplication of large data structures, which could not fit in the 64K word memory of an iWarp cell.

In the second scheme all cells would be working on the same step of Koezuka and Kanade's algorithm at the same time, but on different sets of data. The algorithm would proceed as multiple passes through the array, one pass per step. This method has the advantage of simplicity: each cell is running the same program, and the communication could proceed in a forward direction between the cells. In addition, this system is modular: it would be easy to put together arbitrary modules of this type to do more complex processing or if the task requirements changed. However, this scheme is not feasible for three reasons. First, the steps are determined based on their

feasibility for doing uniform operations, but the computation requirements of each step varies widely. This means that for some quick steps most of the array is wasted (i.e., we could do it just as quickly on one cell as 72). Second, there is a bottleneck between the host and the iWarp cell which slows down the whole computation: all the output for one step must be sent to the host before the next computation may begin. Third, the programs are duplicated across all cells, so that the complete algorithm may not fit in a cell's program memory. Therefore, it may be the case that we would have to reload the program memory during the computation, which could significantly degrade performance.

The mapping we finally chose is the third, where the same stages mentioned in the second mapping are used together on different sections of the array at the same time. A diagram illustrating this approach is shown in Figure 3-2. Each step can statically be given the number of processors necessary based on the amount of computation to be done. The steps are all chained together using the multi-function pipeline abstraction mentioned in the last section. In contrast to the second scheme, all the steps are pipelined, so they can execute concurrently. This mapping will be described in more detail in Section 5.

Given the above justification for mapping feature search operations as a series of steps in a heterogeneous array, we see that there are several different sorts of search operations which must be mapped. In this system much of the processing consists of forming pairs of image features, of forming pairs consisting of model features and image features, and of filtering out objects that do not satisfy some criteria. In the next section we discuss ways to perform these types of operations efficiently on a systolic array.

4.1. Techniques for performing joins and selections on a systolic array

We are interested in general techniques for pairwise operations, which may not be as efficient as the best-known specialized techniques for the same operations. For example, we find the set of parallel lines by an $O(n^2)$ operation in which each line is compared with each line, as opposed to a technique based on sorting by angle, which would take only $O(n \log(n))$ operations. The reason for this is that developing such specialized code for each set operation is difficult and would make the goal of an general system for such operations very hard to realize. For example, if at some point we wanted to change the operation from "parallel" to "nearly parallel and nearly same length" finding an $O(n \log(n))$ algorithm is difficult, but replacing the comparison operation in an $O(n^2)$ algorithm is trivial.

4.1.1. Finding values in $A \times A$

Given objects from domain A, we consider how to find all objects in $A \times A$ which satisfy some condition, possibly computing some value associated with the relation. (This is a join operation on A and A [4].)

For instance, the domain could be line segments, and the condition could be that the pairs satisfy the parallelism relationship, i.e., that they have the same slope. An additional value that might be computed is the distance between the parallel lines. The input to the array is a set of line segments. Each line segment structure input will include a line segment identification number which uniquely identifies it on the host processor. The output will be a set of pairs of line segment id's. These id's eliminate the need for the array to guarantee any ordering of the output.

The input can be presented in any order. Each new object is compared with all the objects that preceded it. This requires $m \times (m-1)/2$ comparisons for m objects. To do this on n cells, each cell eventually stores $1/n$ of the objects. Each object visits all the cells and is compared to all the objects previously stored at each cell. Each cell stores only every n^{th} object it receives (starting with the ith one for cell i) but compares every object it receives with all the ones stored there so far.

This technique is an example of the output partitioning model, where each cell generates up to $1/n$ of the output.

Here is the algorithm for ceil /:

```
(LIST is the list of objects to compare with other objects.)
ignore first i-1 inputs
input the I^th object
stone the I^th object in LIST.

repeat until end-of-data
  for/= 1 to / do
    input an object
    compare this input with all otheis in LIST and output
      a message for each pair that satisfies the condition,
  end for
  store the last object processed in LIST,
end repeat.
```

Number of operations required:

k = number of operations per comparison
m = number of input objects
i = how many words per input object
n = number of cells involved in queiy

Time in operations from first input from host, to last output to host:

$bnn/(2n\}+3km/2+im/n$ operations

4.1.2. Finding values in $A\,xB$

Given two sets of objects A and B, we find all objects in AxB which satisfy some condition. (This is a join -operation on A aod £.) For instance, A could be Hue segments extracted from an image, and B could be line segments in a two-dimensional model of an object, and the condition could be that *the* image line "matches" the model line.

The input will be the sets of objects of type A and B. The output will be a set of pairs of objects, ooe from A and the other from B.

We consider three methods. The first is general, and worics for any comparison operation. The second is used in those cases where a structuring relationship is known on the values to be compared (for example, for finding line matching, we only need to compare with lines of the roughly the same angle). In the third method, it is assumed that such an ordering relationship exists, but that the index structure used may be too big to fit in a cell, and we can afford the time to pre-stoie the first set into the cells. This last method is best used when the A dataset is relatively static

Method I. Broadcast input; $A^f\$$ are resident on cells

First, send the objects in A to the cells, so thai there are 1 At of them oa each c®E, Second, broadcast each B object to all the participating cells. At each cell* a B object will get compared to all the A objects stored at that cell. An output message wil be pnxiaced for every pair from AxB which passes the test TMs is another example of the output partitioning model

Number of ©persons required:

k,n,i same as before
l = number of A objects
m = number of B objects

Time in operations from first host input, to last host output:

$il+kml/n+ikl(1-1/n)$ operations

The first term is for loading the A objects, the second for sending the B objects, and the third term is the time for the last object to be processed from start to finish (i.e., clearing the pipeline).

Method 2. Broadcast input; A's are in buckets on cells

In certain situations we can avoid some comparisons by coarsely sorting the objects into some data structure with a fixed number of buckets. Each B object needs to be compared only with the A objects in the appropriate bucket(s). This data structure is indexed by the value to be compared, the orientation and distance from origin of the line in the case of matching lines.

We can build this structure on the iWarp array by allocating a part of each bucket onto each cell. In the first step of the algorithm the objects from domain A are stored on the cells into the bucket structure. We can build the data structure on the cells as the objects arrive by sending $1/n$ of the data to each cell, and storing them in the appropriate bucket as they arrive (assuming a random input order, the objects for each bucket are evenly distributed among the cells). Each cell contains the whole index for the bucket structure, but only $1/n$ of the objects from each bucket.

The set of B objects now get broadcast to the cells, at each cell a B object is compared with all the A objects which are relevant using the data structure (e.g., all the objects in the same orientation bucket). The satisfying pairs are output as individual messages. In this scheme the load is balanced by giving every B object a little work to do at each cell.

Number of operations required:
k,n,i,l,m same as before
b = number of buckets

Time in operations from first host input, to last host output:

$il+kml/(bn)+kl(1-1/n)/b$ operations

Method 3. Addressed input

Another way to use an indexed data structure is to store only one part of the index structure, and all the associated buckets full of objects, on each cell. Each of the new objects that comes in is only processed at one cell.

We can construct the index structure on the host, and then send the appropriate parts of the index to the correct cells.

The processor (it can be the host or another cell) which sends the a B object will address the input message to the appropriate cell. On iWarp this is done using addressed blocks in a multi-drop channel.

Addressed input works best if we know that the incoming objects will be going to a different cell each time. If the input comes from one processor, then this is easier to arrange; but otherwise it is difficult to predict the frequency of

outputs destined for a particular cell.

The operation count for this method can be estimated by the one given for method 2.

4.1.3. Finding values in A

Given a single set of features, we may want to report those that satisfy some particular condition, such as satisfying a length or angle constraint. (This is a selection operation on A.) In this case, we use a straightforward data partitioning method; the dataset is distributed across the cells, and each cell calculates the condition on its portion of the dataset. This method is better than a sequential implementation only if the computation time is larger than the time to send the datum to the cell.

Number of operations required:
k,n,i,l,m same as before

Time in operations from first host input, to last host output:

$kl/n+in$ operations

Now we will describe how these methods were used in the multi-function pipeline mapping of Koezuka and Kanade's algorithm.

5. Mapping Koezuka and Kanade's Algorithm onto iWarp

The 72 processor array is split up into a multi-function pipeline of sections of consecutive cells which work together on the same step of the task, as was shown in Figure 3-2. All cells in a given section run copies of the same program, and operate using the same communication model. The output messages of one section of the array are used as the input messages for the next section of the array. These different sub-programs could be combined in arbitrary ways based on the task to be completed.

It is assumed that all the data structures corresponding to the static object model are already resident on the array. This includes the 3D object model data structures, as well as the 2D views of the object model. This requires a pre-processing step to load the array with those data structures. Exactly how they are stored on the array will be described with the individual steps of the program.

The image data were fairly small, as shown in Table 2-1. There were approximately 40 lines in an image, and about 25 parallel line pairs. We do not consider in this mapping the computation of the line segment data or the computation of the parallel line pairs. With this size data set this is a small fraction of the total computation time. Important factors in the mapping were:

- *Computational needs*. The part of the computation that is the most time consuming on a sequential processor is the testing of all the possible 2D match directions (the step called "Pre-match")—this is even *after* we have reduced the number of directions to consider using the distance and angle constraints. Therefore this step is split over several cells.
- *Memory needs*. The data structure that takes up the most space is the table with the 2D appearance of the 3D object at all attitudes from a tesselation of the unit sphere. Because this data structure was too big to fit in one cell's memory, it is split over a group of cells.
- *Simplicity of communication*. As we discussed in Section 4, we want a single input data type and output data type flowing in a forward direction between steps. Communication between steps follows the general pattern of one message per current match hypothesis.

The following is a detailed description of the mapping of the algorithm:

1. **Parallel line pairs:** Find all pairs of parallel lines from the image such that the distance between the lines in a pair are within a certain range.

   This was such a small percentage of the computation time for the images we tested that we assume that these values are computed on the first cell and broadcast to all the later cells before the rest of the computation starts. This way they can be referenced by a common identification in any subsequent messages. The attributes of the image line segments are also broadcast in this manner.

2. **Distance-length condition:** Find all pairs consisting of a parallel line pair in the image and a parallel line pair in the 3D model which satisfy the following conditions: (1) the distance between the image parallel line pair is less than the distance between the model parallel line pair; (2) the lengths of the lines in the image line pair is less than the lengths of the lines in the model line pair. This stage of the computation uses the broadcast input method (method 1 from Section 4.1) for computing all pairs in $A \times B$. In this mapping all parallel line pairs from the object model are the A's already resident on the cells; they were divided evenly between all the participating cells for this stage during the pre-processing step. The parallel line pairs from the image are the B's broadcast over the cells.

   The output messages are *image-model pairs*, i.e., pairs of parallel line pairs which satisfied the condition from the preceding paragraph. In an image-model pair one member of the pair is an image pair and the other is a model pair.

   As will be seen in the next section, this step is run on only *one* cell of the array. A one-cell version of this parallel algorithm is used in that case.

3. **Orientation condition:** Find all *pairs* of image-model pairs such that their component lines have different orientations in the image. A second constraint applied here is that the distance between the 2D image pairs must be smaller than the 3D model pairs. This is an example of the broadcast input model for testing all pairs from $A \times A$.

4. **Verify:** Based on the angle between pairs and distance between them look up valid viewing directions. For each image-model pair in the input message (which contains two of them), (a) compute the valid viewing directions based on the *distance* in image coordinates between the two parallel pairs in the image, and on the hypothesized corresponding model lines; (b) compute the valid viewing directions based on the *angle* in image coordinates between the two parallel pairs in the image, and on the hypothesized corresponding model lines. This is actually done by a table-lookup, since it was part of the information about the model that we pre-compiled. The valid viewing directions are the intersection between those from constraints (a) and (b).

   This is an example of the input data partitioned model of computation. Each cell has an identical program and data structures stored on it. We would like the current input message to go to the next free cell (it only needs to be processed at one of the ones in this section). This could be accomplished by sending the data to each of the cells in a group, until one accepts it. Alternatively, sending it to one of the cells at random might give a good enough distribution of the computation. It is difficult to predict how well this will behave dynamically, since the input messages for this step come from multiple cells from the last step.

   One message is output for each valid viewing direction within each pair of image-model pairs hypothesized. These are the only ones that we will attempt to do further matching on.

5. **Pre-match:** Now we have a hypothesized viewing direction, and model to image correspondence for four lines. Based on this viewing direction, we look up the appearance of the other lines in the model. This is the addressed input model of computation: we send the input message only to the cell which has the correct part of the table stored there. We can do this with an arbitrary number of cells by using a hash function which maps from the array indices of the table to the possible cell id's. This table of the 2D shape data for the model is one of the ones that were computed and stored on the array in the pre-processing step. In this step the translation and rotation within the image plane are computed. The match test is applied between a line in the image and a line in the projected and transformed model at this viewing direction for each of the four corresponding line pairs we have been using from the image and the model. Only the ones that pass this test will be sent on to the next group of cells.

   For the data that satisfy the test the output message will consist of the pair of image-model pairs from

the input message, plus the 2D projections of the other lines in the model at this viewing direction (this is what we looked up before).

6. Match: In this step the other image lines that match aie found, given this hypothesized correspondence and transformation of the model. The match test is applied between each 2D model line in the input message and all image lines at the same orientatioa This step uses the "broadcast input, with the A's in buckets'* (method 2) model for testing pairs in *AxB*. In this case the *A* objects are the image line segments that were stored here in step one. The data is split among the cells, but an index into it by (rho,theta) is kept at each step. The *B* objects are the model line segments.

   The output of this step *is* one output message for each input message. All hypothesized matches are output at this point, for a final decision at the host Alternatively we could make some threshold (on the number of model lines with corresponding image lines) on what constitutes a plausible match; and only output those above the threshold.

6. Simulation Results

We did some simulation experiments to test the performance of Koezuka and Kanade's algorithm on an iWarp array. We estimated the number of iWarp instructions that would be required for the straight-line C code from the system running on the VAX. We then ran the C program on several segmented images to gather statistics on the dynamic behavior. For each ran we determined the number of operations required for each of the steps described in the preceding section. We used the averages over all these runs to determine how many cells to allocate to each step for an iWarp implementation. The iWarp times were then computed from these operation counts, assuming this static allocation of cells.

For this analysis we made the following assumptions about the iWarp cell:

- 125 ns cycle time for floating point operations.
- 64K words of memory per cell.
- Delays due to cache misses were ignored.
- Adds and multiplies on the cell were not overlapped, even though the processor has both units. The parallel utilization of these units depends cm compiler optimizations.
- Code size was ignored in determining memory requirements. This mapping doesn't come close to the boundaries of memory, so there would be more than enough room for the application code, and any system routines.
- The following cycle times were assumed for each of these unary operations: sine, cosine: 50 cycles each; inverse: 10 cycles each; square root: 30 cycles each.

The foEowing assumptions were made about the array overall:

- The overhead to process a message is ignored. The iWarp array has special hardware to handle messages.
- Message conflicts *m* the pathway are ignored. There is only limited bandwidth cm the pathways, and the messages come out at arbitrary times from multiple cells, so there is the possibility of delays from conflicts. Actual simulations of the array will be necessary to test this.
- It is assumed that the pipeline is always moving smoothly. Since the ctiffenem sections of the array arc consuming awl producing messages asyocbxttoousiy this could cause sane delays *in* an actual nut That is, messages destined for a particular cei might sit around in buffers, and even block previous ceils, because the destination ceil is swamped with eaiiier data to process. Although we attempted to even ait the average data-rales between sections in the mapping, it is difficult to predict this type of delay without doing array simulations.

The results axe presented In tables 6-1 and 6-2. In cable 6-1 the first two columns -number of ceils, and memory requirements -axe the same for all the *runs,* because we used static load balancing an! static data structures. The

other columns show information pertaining to one particular run. The consume rate refers to how often on the average (in cycles) an input message will be requested by a section of the array; and the produce rate is how often (in cycles) an output message is produced from a section of the array.

Table 6-2 shows the comparison times between the C program running on a VAX 8650 and the estimated run times for the same images on the iWarp array. These results are for the same set of images of L-shaped blocks that were used in table 2-1. The speedup varies from a factor of 100 to 256 over these runs. The wide range in speedup comes from using static load-balancing on varying data set sizes. Dynamic load balancing is quite difficult on a linear array. The overhead involved in re-allocating cells, including completely copying memory from one cell to another, is too big. But for domains where the complexity of images is well known or where the user is willing to accept widely varying processing rates, static allocation seems adequate.

Table 6-1: Operations, Memory, and I/O for one run of Koezuka and Kanade's Algorithm

Stage in computation	# cells	memory reqts (words per cell)	# total operations (thousands per cell)	consume rate	produce rate
Distance-length condition	1	200	19	N/A	25
Angle-condition	7	2700	1060	1429	73
Verify	7	9100	1247	86	21
Pre-match	47	3400	1038	18	1182
Match	10	1500	1046	1192	24

Table 6-2: Recognition times for several images (in milliseconds)

Time on VAX 8650	Estimated time on iWarp array	Speedup
34000	157	217 X
5300	36	147 X
4300	22	195 X
3600	36	100 X
4700	45	104 X
15100	59	256 X
400	4	100 X

7. Conclusions

In this work we show one can successfully do object recognition in parallel on a systolic array. We have presented the following:

- We demonstrated the ability to map multiple steps of a task with varying data-sets to make a coarse-grained pipeline on a systolic array. This type of mapping is a good match between this type of problem, with successive generation, and filtering of combinations of objects for the search, and the large linear array, which is best utilized in a heterogeneous manner.

- We described some general models of computation on the iWarp array. Some of these are currently in use on the Waip array, working on fixed-sized data sets such as images. The flexible communication mechanisms of the iWarp airay enable us to use the models in a more general way. In particular, we use the dynamic deteimination of destination in the addressed input style of computation, and the ability to communicate with non-neighboring cells in the multi-function pipeline.
- We presented some techniques for doing general join and select operations on the array. We used these models as components in mapping an object recognition system to the array. These models are general components which we hope to be able to compose in novel ways in the future for various image understanding search tasks.
- We showed results of an analysis of our mapping based on runs of the sequential program on real images. The speed of this system indicates an increased ability to routinely use *relationships* between image features from which to start a search. As we saw, relationships are a good way to initially constrain the search. They also are more stable than other features in the face of imperfect segmentations.

Although our results directly pertain to a single, fairly simple object recognition system, this type of parallel architecture- non-shared memory, a relatively small number of high-powered processors, linear array-looks promising for intermediate-level vision processing. We did not have to resort to a brute-force search to use parallelism. We were able to take a fast sequential algorithm running on a fast processor, and show considerable speedup on the iWarp array.

In future work, this system could be scaled up by storing multiple object models in iWarp memory. The search could involve multiple passes through the anay, one per object model Another way to extend this system would be to construct it such that it could utilize other types of image features such as curves, regions, or color.

The ability to replace one module with another in the multi-function pipeiine is important for the generality of this method. We would like to use the general feature searching methods presented as a basis for a language and compiler for a system that could take a high level description of the search operations required for the task, and would generate the parallel program to run on all or part of the array.

8. Acknowledgement

We wcwld like to thank Takeo Kanade for helpful advice from discussions of this work, including eootributioos to the idea of efficiently mapping multiple feature search operations onto a systolic aixay.

References

[1] 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.

[2] Annaratone, M., Bitz, F., Deutch, J., Hamey, L., Kung, H. T., Maulik, P., Ribas, H., Tseng, P. and Webb, J.
Applications Experience on Warp.
In *Proceedings of the 1987 National Computer Conference.* AFIPS, 1987.

[3] 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.

[4] Date, C. J.
An Introduction to Database Systems.
Addison-Wesley Publishing Company, 1981.

[5] Flynn, A. M. and Harris, J. G.
Recognition Algorithms for the Connection Machine.
In *Proceedings IJCAI.* Los Angeles, California, August, 1985.

[6] Goad, C.
Special Purpose Automatic Programming for 3D Model-Based Vision.
In *Proceedings DARPA IUS Workshop.* 1983.

[7] Hamey, L. G. C., Webb, J. A., and Wu, I-C.
Low-level Vision on Warp and the Apply Programming Model.
Parallel Computation and Computers for Artificial Intelligence.
Kluwer Academic Publishers, 1987.
Edited by Janusz Kowalik.

[8] Koezuka, T. and Kanade T.
A Technique of Pre-compiling Relationships between Lines for 3D Object Recognition.
In *Proceedings International Workshop on Industrial Applications of Machine Vision and Machine Intelligence.* Tokyo, Japan, February, 1987.

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

[10] Roberts, L. G.
Machine perception of three-dimensional solids.
Optical and electro-optical information processing.
MIT Press, 1965, pages 159-197.