Path Planning on the Warp Computer: Using a Linear Systolic Array in Dynamic Programming

F. Bitz and H. T. Kung

August 28, 1987
CMU-CS-87-180

Department of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

ABSTRACT

Given a map in which each position is associated with a traversability cost, the path planning problem is to find a minimum-cost path from a source position to every other position in the map. The paper proposes a dynamic programming algorithm to solve the problem, and analyzes the exact number of operations that the algorithm takes. The algorithm accesses the map in a highly regular way, so it is suitable for parallel implementation. The paper describes two general methods of mapping the dynamic programming algorithm onto the linear systolic array in the Warp machine developed by Carnegie Mellon. Both methods have led to efficient implementations on Warp. It is concluded that a linear systolic array of powerful cells like the one in Warp is effective in implementing the dynamic programming algorithm for solving the path planning problem.

The research was supported in part by Defense Advanced Research Projects Agency (DOD) monitored by the Space and Naval Warfare Systems Command under Contract N00039-87-C-0251, and in part by the Office of Naval Research under Contracts N00014-87-K-0385 and N00014-87-K-0533.

1. Introduction

During 1984-87, Carnegie Mellon has developed a programmable systolic array machine called Warp [2, 3]. Currently produced by GE, the machine has a linear systolic array of 10 cells, each capable of delivering 10 million floating-point operations per second (10 MFLOPS). One of the first applications of the machine is navigation for robot vehicles using vision and laser range-finder [4, 6, 8, 9].

Related to robot navigation is path planning. Given a map on which each position is associated with a traversability cost, the path planning problem is to find a minimum-cost path from a source position to every other position in the map. We use a dynamic programming algorithm to solve the problem. Two general methods of mapping the dynamic programming algorithm onto the linear systolic array in Warp have been considered. Both methods have led to efficient implementations of the dynamic algorithm on Warp.

In the next section we define the path planning problem in mathematical terms. Section 3 describes the dynamic programming algorithm for solving the problem, and analyzes the number of times that the algorithm requires to scan the map. The linear systolic array in the Warp machine is briefly described in Section 4. Section 5 describes two general approaches of mapping the dynamic programming algorithm onto Warp and compares their performance. Some concluding remarks are given in the last section.

2. Path Planning Problem

A *map* is an $n \times n$ grid of positions, for some positive integer n. Each position x in the map is associated with a non-negative real number $tc(x)$ corresponding to the traversability cost of the position. For simplicity, we assume that the function tc is directional independent, i.e., it is as expensive to cross a position in one direction as in another. (For applications where this assumption is not true, tc should be defined as a vector function consisting of scalar functions corresponding to various directions.) Figure 1(a) gives traversability costs of positions in a map with $n=6$. In general, n can be 512, 1024, or larger. Figure 2 gives traversability costs in a 400×480 map corresponding to an area near Denver. In the figure, a darker position has a higher traversability cost.

A position in the map has up to eight neighbors as shown in Figure 1(b). An *edge* (u,v), denoted by an arrow in Figure 3, is a directed line segment connecting two neighboring positions u and v in the map. The cost of edge (u,v) is

$$\frac{tc(u)+tc(v)}{2}, \quad \text{if the edge is horizontal or vertical,}$$

or

$$\left(\frac{tc(u)+tc(v)}{2}\right)\cdot\sqrt{2}, \quad \text{if the edge is diagonal,}$$

where $tc(u)$ and $tc(v)$ are traversability costs of u and v, respectively. The $\sqrt{2}$ multiplier reflects the added traveling distance due to the diagonal connection.

(a)

6	1	2	2	2	2
2	2	1	3	2	5
3	4	6	2	7	6
4	1	1	4	1	4
9	2	5	8	1	5
6	7	4	3	1	1

(b)

n1	*n2*	*n3*
n8	*x*	*n4*
n7	*n6*	*n5*

Figure 1. (a) Traversability costs in a 6×6 map, and (b) eight neighbors, $n1$ . . . , $n8$, of position x

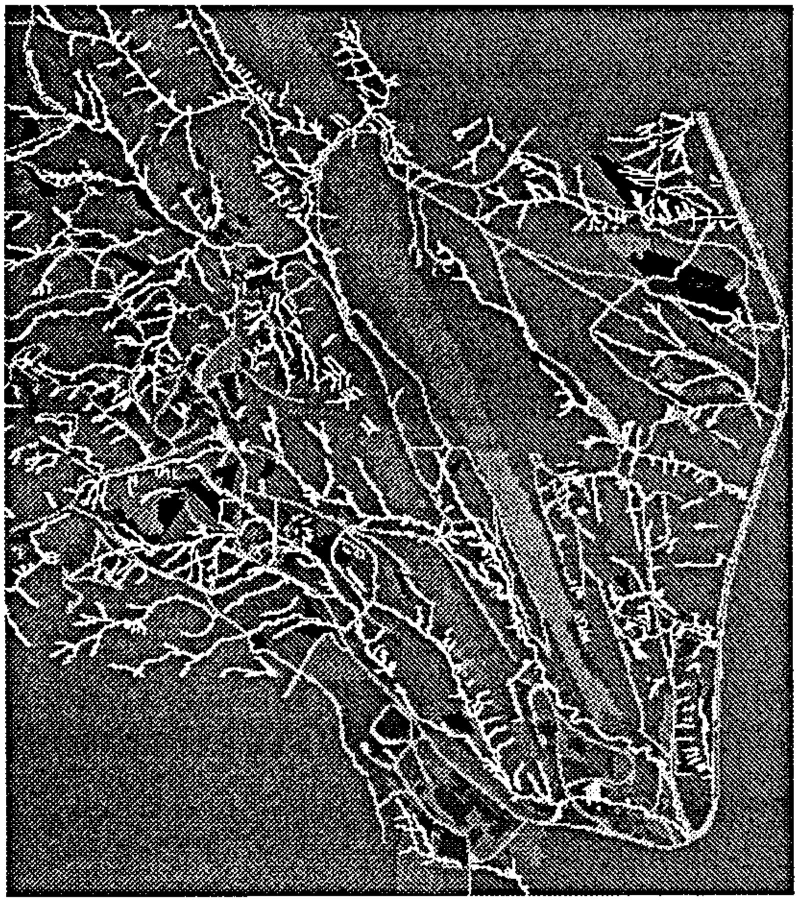

Figure 2. Traversability costs in a 400×480 map

A *path* from position A to position B is a sequence of edges $(s_1,s_2),(s_2,s_3), \ldots ,(s_{p-1},s_p)$ for some positive integer p, where $s_1=A$ and $s_p=B$. The cost of a path is the sum of costs of all its edges. For example, the cost of the path shown in Figure 3 is

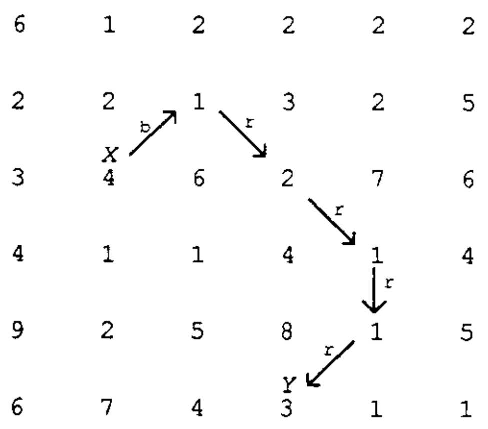

Figure 3. Path from X to Y, where edges labelled by r and b are red and blue edges, respectively

$$(\frac{4+1}{2})\cdot\sqrt{2}+(\frac{1+2}{2})\cdot\sqrt{2}+(\frac{2+1}{2})\cdot\sqrt{2}+(\frac{1+1}{2})+(\frac{1+3}{2})\cdot\sqrt{2}.$$

The color labels of the edges in Figure 3 will be explained in Section 3.3.

A *shortest path* (or minimum-cost path) from position A to position B has the minimum cost over all paths from A to B. The *path planning problem* considered in this paper is that given a position, called the *source*, we want to find a shortest path from it to *every* other position in the map. This is in contrast to another possible definition for the problem that calls for finding a shortest path from the source to only one other position.

3. Dynamic Programming Algorithm

It is easy to see that every subpath of a shortest path is itself a shortest path. Therefore we can use the principle of dynamic programming [1]. That is, to find a shortest path from the source to a position we need only consider shortest paths from the source to the neighbors of the position. This motivates the dynamic programming algorithm described below.

Before the algorithm starts, the best known cost for every position in the map and its surroundings is assigned a value, 0 at the source and infinity at all other positions. The algorithm alternately performs the so-called *red sweep* and *blue sweep*, described below. In each sweep the value of each position in the map is updated. At any time during the algorithm execution, the value of a position represents the cost of a current ''shortest path'', which the algorithm has recognized up to this moment, from the source to the position.

3.1. Red Sweep

The red sweep scans the map in the row-major ordering as shown in Figure 4 (a). During the sweep, a new value (subscripted by *new* in the equations below) is computed for every position in the map in the order of the sweep, using the old values (subscripted by *old*). That is, the new value of the current position of the sweeping, $current_{new}$, is computed by:

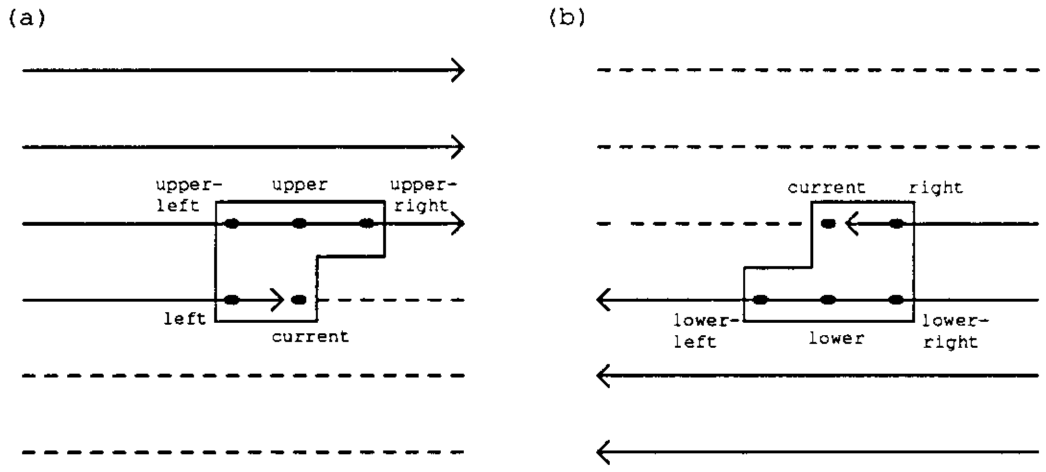

Figure 4. (a) Red sweep and (b) blue sweep, and the associated masks

$$current_{new} = \min\left(current_{old},\ upper\text{-}left_{new} + \frac{tc(upper\text{-}left)+tc(current)}{2}\cdot\sqrt{2},\ upper_{new} + \frac{tc(upper)+tc(current)}{2},\right. \\ \left. upper\text{-}right_{new} + \frac{tc(upper\text{-}right)+tc(current)}{2}\cdot\sqrt{2},\ left_{new} + \frac{tc(left)+tc(current)}{2}\right), \tag{1}$$

where quantities in the right hand side are associated with positions in the mask shown in Figure 4 (a).

3.2. Blue Sweep

The blue sweep scans the map in the reversed row-major ordering as shown in Figure 4 (b). The new value of the current position of the sweeping, $current_{new}$, is computed by:

$$current_{new} = \min\left(current_{old},\ lower\text{-}left_{new} + \frac{tc(lower\text{-}left)+tc(current)}{2}\cdot\sqrt{2},\ lower_{new} + \frac{tc(lower)+tc(current)}{2},\right. \\ \left. lower\text{-}right_{new} + \frac{tc(lower\text{-}right)+tc(current)}{2}\cdot\sqrt{2},\ right_{new} + \frac{tc(right)+tc(current)}{2}\right), \tag{2}$$

where quantities in the right hand side are associated with positions in the mask shown in Figure 4 (b).

Note that the combination of the two masks associated with the red and blue sweeps allows the value of the current position to be updated using the values of all the eight neighboring positions. Therefore if a sufficient number of red and blue sweeps are performed, the final value of every position, which is the cost of a shortest path from the source to the position, will be obtained.

3.3. Number of Required Sweeps

The red and blue sweeps are performed alternatively until no values are changed in one sweep. The following analysis shows how the number of required sweeps depends on the given map and the source position.

Consider a shortest path from the source to any other position. We color its edges in red or blue according to their directions. That is, edges pointing to left, upper-left, upper, and upper-right directions are colored blue, whereas edges pointing to right, lower-right, lower, and lower-left directions are colored red. The colors of the edges shown in Figure 3 follow this coloring scheme.

Consider a subpath whose edges all have the same color, say, the red color. Suppose that when a red sweep is about to start, the head of the first edge in the subpath has already received its final value. Then it is easy to check that after the red sweep *all* positions on the subpath will receive their final values. This implies that following result:

> **Theorem:** Suppose that a shortest path from the source to any other position changes colors c times. Then if the red and blue sweeps are performed alternately, the final values of all positions on the shortest path will be obtained exactly after c or c+1 sweeps.

Therefore the number of required sweeps is C+1 or C+2, where C is maximum number of color changes in a shortest path from the source to any other position. The total number of operations that the dynamic programming algorithm will take is $O(C{\cdot}n^2)$, since each sweep performs the computation corresponding to equation (1) or (2) n^2 times. In the worse case C can be as large as $O(n^2)$. However, for real terrain maps C is expected to be much smaller than n. For the 400×480 map in Figure 2, C is about 40.

3.4. Obtaining a Shortest Path

When the algorithm is terminated, all positions in the map have received their final values. Using these values, it is straightforward to trace a shortest path to the source from any other position. That is, the next position on the path is a neighboring position whose value leads to the value of the current position using equation (1) or (2). This procedure continues until the source is reached. The total number of operations is $O(n)$.

One possible way to speed up the procedure is to maintain a back pointer in each position during red and blue sweeps. Every time when the value of the current position is updated, we also update the back pointer to point to the neighboring position whose value leads to the new value for the current position. Following these back pointers, a shortest path to the source from any other position can be easily obtained. However, using back pointers may have two drawbacks. First, storing the pointers consumes memory space, which may be a critical resource for some computers. Second, a back pointer in a position may be updated several times by different sweeps, and therefore extra computations may be performed. This may not increase the computation time significantly when a sweep is carried out by a number of processors in parallel, as to be described in the rest of the paper.

4. Linear Systolic Array in Warp

Systolic arrays have high communication bandwidth between neighboring cells, which are defined by a simple topology. As shown in Figure 5, in the case of the Warp processor array, this is the simplest possible topology, namely a linear array. The linear configuration was chosen for several reasons. First, a linear array is easier to program than other interconnection topologies; it is relatively simple for a programmer to envision a linear array and map his or her computation onto it.

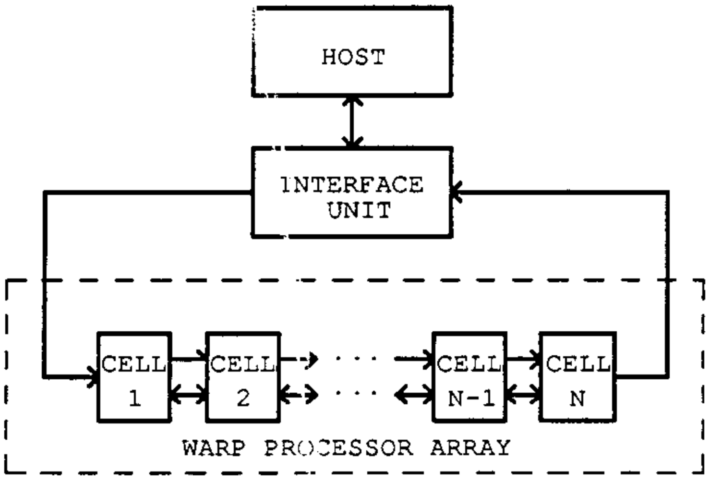

Figure 5. Warp system

Second, the linear interconnection is easy for hardware implementation. Because of its simplicity, the Warp array has a very clean backplane, capable of providing 40 Mbytes per second data bandwidth between every pair of neighboring 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 parallel processing. The linear interconnection also makes it 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. Therefore there is no need to have an exotic host to provide very high bandwidth I/O with the array.

In contrast with some parallel arrays such as MPP [5] and the Connection Machine [10] that have 2-dimensional arrays of simple processors, each Warp cell in the Warp array is a powerful processor with the following features:

- P1. Pipelined floating-point adder and multiplier, giving a total peak computation rate of 10 MFLOPS
- P2. High data I/O bandwidth of 80 Mbytes per second
- P3. 32K-word data memory
- P4. Crossbar interconnection between data memory, input queues, register files, etc.
- P5. Horizontally microcoded (>250 bits)
- P6. Sequencer and 8K-word program memory
- P7. Hardware flow control for inter-cell communication

These features plus the optimizing Warp compiler [7] help to increase the Warp cell's performance and programmability.

5. Two Mapping Methods

We consider two general methods for mapping the dynamic programming algorithm in Section 3 onto a linear systolic array such as the one in Warp. Suppose that the linear array has k cells, with $k \leq n$.

5.1. Vertical Partitioning Method

Recall that a red or blue sweep scans the map in the row-major or reversed row-major ordering. The vertical partitioning method assigns an equal number of consecutive columns of the map to each cell. This is depicted in Figure 6.

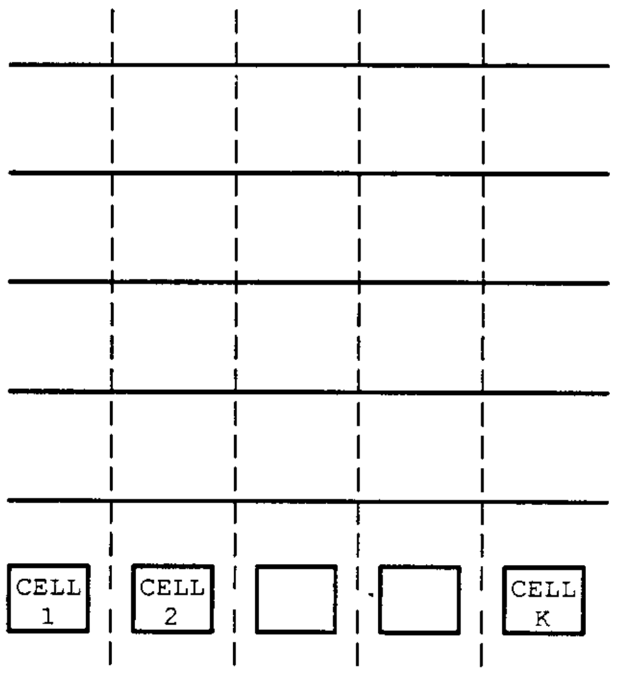

Figure 6. Vertical partitioning, where the horizontal solid lines are scan lines

Consider first the red sweep. Cell 1, the left-most cell, starts computing on the first portion of the first scan line. Immediately after the value of the last position in this portion is computed, cell 1 passes the value to cell 2. Cell 2 can then start computing on the second portion of the first scan line, while cell 1 starts computing on the first portion of the second scan line. The computed value of the first position in the second portion of the first scan line needs to be passed from cell 2 to cell 1. This value is needed for cell 1 to compute the value of the last position in the first portion of the second scan line. In this way every cell will eventually start its computation. Cell i+1 starts later than cell i by roughly the time needed to compute the values of positions in a portion of a scan line. The computed value for each position is stored at the cell that performs the computation.

After the red sweep is completed, the blue sweep starts similarly from cell k, the right-most cell. Red and blue sweeps are performed totally inside the linear array in an alternate manner, without host interaction. Note that bi-directional communications are needed between adjacent cells for each sweep.

5.2. Horizontal Partitioning Method

With the horizontal partitioning method each row of the map is assigned to a cell. Figure 7 illustrates this partitioning method for $k=n$. In this case cell i gets row i.

For the red sweep, immediately after cell i has computed the values of two positions, it will pass the values to cell $i+1$ to get it started. Therefore the latency between the startup times of two adjacent cells is very small.

Suppose that k is smaller than n. Then a cell will have to compute values of positions in multiple rows. That is, cell i will be assigned to row $i+j \cdot k$ for $j=0,1,2,\ldots$. In this case cell 1 will need to receive computed values from cell k. For the Warp implementation this feed-back loop is accomplished through the host. Since cell k produces its first results before cell 1 finishes all its computations, the feedback time through the host is totally overlapped with the computation time of cell 1.

After the red sweep is completed, the blue sweep starts from cell k in a similar way, but the intermediate values will flow on the array in the opposite direction.

Note that the horizontal partitioning has a small startup latency between adjacent cells and only uni-directional communications are needed for each sweep. This partitioning method, however, has the complication of requiring the feed-back loop from cell k to cell 1, and vice versa.

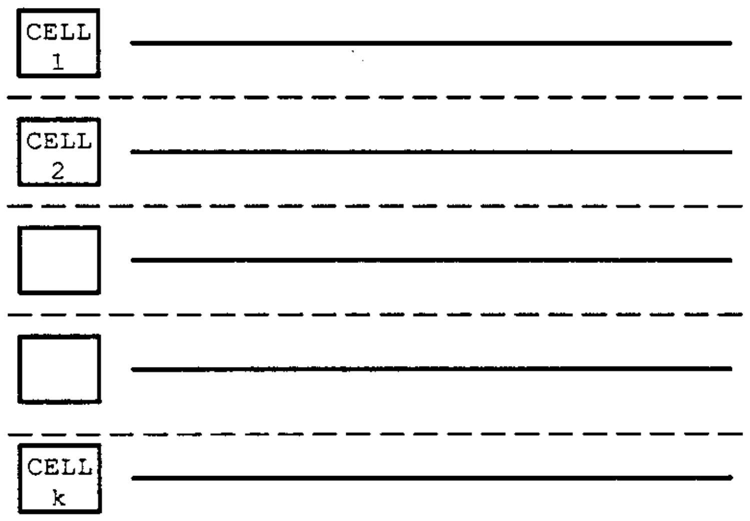

Figure 7. Horizontal partitioning, where the horizontal solid lines are scan lines

5.3. Performance Analysis

Suppose that a cell takes a unit time to perform the computation corresponding to equation (1) or (2). Then for both the mapping methods, the execution time for either the red or blue sweep is:

$$T=(n^2/k)+\text{ overhead},$$

where the overhead is n or k for the vertical or horizontal partitioning method, respectively. The overhead is due to the latency of startup times of cells. We are interested in the ratio of the overhead to the total execution time. Using some algebra, we have:

$$\frac{\text{overhead}}{T}=\frac{\alpha}{1+\alpha}$$ for the vertical partitioning method, and

$$\frac{\text{overhead}}{T}=\frac{\alpha^2}{1+\alpha^2}$$ for the horizontal partitioning method,

where α is k/n. For both cases the ratio approaches 1/2 as $\alpha(=k/n)$ approaches unity. However, when α is small, i.e., when k is a small fraction of n, we see that the horizontal partitioning method is more efficient than the vertical partitioning method.

5.4. Performance on Warp

We have implemented both partitioning methods on the Warp machine at Carnegie Mellon. On a 10-cell Warp, a single red or blue sweep for a 512×512 map takes 250 ms or 163 ms using the vertical or horizontal partitioning method, respectively. (We expect that these times can be reduced to 80 ms or less, by using more optimized code.)

6. Concluding Remarks

We have described a dynamic programming algorithm for solving the path planning problem. The algorithm accesses the map in a highly regular way, so it is suited for parallel implementation.

A linear array interconnection fits naturally for the parallel implementation of the algorithm. It does not seem that any other interconnection of higher dimensionality such as a 2-dimensional processor array can be more efficient. Both the vertical and horizontal partitioning methods have yielded efficient implementations of the algorithm on the linear systolic array in Warp. Basically, all the cells can do useful work most of the time during the algorithm execution. The only inefficiency is due to the latency in cell startup times. As pointed out in Section 5.3, this latency is at a minimum level for the horizontal partitioning method.

Several other architectural features of Warp have contributed to its effectiveness in the implementations.

- The bi-directional flow is useful for the vertical partitioning method, whereas the ring structure via the host supports the horizontal partitioning method.
- High-bandwidth inter-cell communication allows that intermediate values to be passed between adjacent cells efficiently.
- Floating-point operations in each cell provide the dynamic range needed to accumulate the costs of a large number of edges on a shortest path.
- Large local memory at each cell can store values that it computes in a sweep and the required traversability costs. Therefore cost and computed values need not be input to the cells for each new sweep, in order to reduce the I/O time.
- High degree of programmability of the cells allow rapid experiments of various partitioning methods, and the optimized Warp compiler produces efficient code for the Warp machine.

Results of this paper are not restricted to 2-dimensional, square maps. Generalizations to other computations that involve rectangular or 3-dimensional grids and use masks similar to those used in the paper are straightforward.

Acknowledgments

The authors wish to thank Chuck Thorpe for his comments on a draft of the paper.

References

1. Aho, A., Hopcroft, J.E. and Ullman, J.D.. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Massachusetts, 1975.

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

3. 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. Proceedings of the 1987 National Computer Conference, AFIPS, 1987, pp. 133-140.

4. Annaratone, M., Bitz, F., Deutch, J., Hamey, L., Kung, H. T., Maulik, P., Ribas, H., Tseng, P. and Webb, J. Applications Experience on Warp. Proceedings of the 1987 National Computer Conference, AFIPS, 1987, pp. 149-158.

5. Batcher, K.E. "Bit-serial Parallel Processing Systems". *IEEE Trans. Computer C-31*, 5 (May 1982), 377-384.

6. Clune, E., Crisman, J. D., Klinker, G. J., and Webb, J. A. Implementation and Performance of a Complex Vision System on a Systolic Array Machine. Tech. Rept. CMU-RI-TR-87-16, Robotics Institute, Carnegie Mellon University, 1987. To appear in *Proc. of Conference on Frontiers in Computing*, Amsterdam, The Netherlans, December 1987..

7. Gross, T. and Lam, M. Compilation for a High-performance Systolic Array. Proceedings of the SIGPLAN 86 Symposium on Compiler Construction, ACM SIGPLAN, June, 1986, pp. 27-38.

8. Kanade, T., and Webb, J. A. End of Year Report for Parallel Vision Algorithm Design and Implementation. Robotics Institute, Carnegie Mellon University, 1987.

9. Thorpe, C., Hebert, M., Kanade, T. and Shafer, S. Vision and Navigation for the CMU Navlab. Annual Reviews, Palo Alto, California, November, 1987, pp. 521-556.

10. Waltz, D. L. "Applications of the Connection Machine". *IEEE Computer 20*, 1 (January 1987), 85-97.