IMPLEMENTATION AND PERFORMANCE OF A COMPLEX VISION SYSTEM ON A SYSTOLIC ARRAY MACHINE

Ed CLUNE, Jill D. **CRISMAN, Gudrun** J. **KLINKER and** Jon. **A.** WEBB
Computer Science Department and The Robotics Institute, Carnegie Mellon University. Pittsburgh, PA 15213, U.S.A.

Complex vision system are usually quite slow, requiring tens of seconds or minutes of computer time for each image. As the complexity and experimental nature of the system increases, the speed is especially low, since all components of the system must be optimized if the system is to show good performance. The FIDO system, a stereo vision system for controlling a robot vehicle, has existed for a number of years and has been implemented on a number of different computers. These computers have ranged from a DEC KL10 to the current implementation on the Warp machine, a 100 Million Floating-point Operations Per Second (MFLOPS) systolic array machine. FIDO has shown an enormous range in speed; its ancestor took 15 minutes per step, while the Warp implementation takes less than 5 seconds per step. Moreover, while early versions of FIDO moved in slow, start-and-stop steps, FIDO now runs continuously at 100 mm/second. We review the history of the FIDO system, discuss its implementation on different computers, and concentrate on its current Warp implementation.

* **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 5352, 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.**

North-Holland
Future Generations Computer Systems 4 (1988) 15–29

Sophisticated vision algorithms are usually disappointingly slow when they are put together into a complete system. This is especially true of systems created for research purposes, since speed is usually not a primary concern and much effort must be spent on working out ideas in a user-friendly environment, which is not conductive to high speed algorithms. However, even in a research environment, reasonable speed is desirable. One reason for this is that it is difficult to debug an algorithm that takes minutes to compute one meaningful result; not many debugging runs can be made in a day. Another reason is the difficulty of using the algorithm in a non-simulated environment, for example to control an actual robot vehicle; if the algorithm takes minutes to run for each step, it is not practical to debug given time constraints imposed by the physical environment. Finally, it is not possible to integrate an algorithm into a working vision system, which may use multiple sources of inputs, if the speed of one algorithm is much lower than the others.

One sophisticated vision system that has received much attention over the years, and which continues to be developed, is the FIDO (Find Instead of Destroy Objects) vision system. FIDO is a stereo vision navigation system used for the control of robot vehicles; it includes a stereo vision module, a path planner, and a motion generator. This system is descended from work done by Moravec at Stanford [1]. After Moravec came to Carnegie Mellon in 1980, work was done by Thorpe and Matthies [2,3], who gave the system its name. More recently, work has been continued by the authors of the present paper, as well as others. This vision system is unusual in its longevity and in the range of speed over its span of development: Moravec's original algorithm, which was heavily optimized (though different in many important ways from the FIDO algorithm), took 15 minutes to make a single step while running on an unloaded DEC KL10; the Vax 780 implementation ran at 35 seconds per step; the Sun 3 implementation presently at Carnegie Mellon takes 8.5 seconds per step; and the current Warp implementation takes 4.8 seconds per step. This is a factor of 190 in improvement.

We will begin by describing the Warp system, on which the current implementation of FIDO runs, and how it is programmed. Then we will review the history of the system that became FIDO, and how the different parts of FIDO were implemented on Warp. Finally, an evaluation of the Warp/FIDO system is given and a brief description of future implementations.

2. The Warp System

2.1. *Warp Hardware*

A discussion of the Warp hardward is necessary in order to fully understand the Warp implementation of FIDO. This discussion is greatly abbreviated; more detail is available elsewhere [4]. The Warp machine has three components: the Warp processor array, or simply Warp, the interface unit, and the host, as depicted in Fig. 1.

All of the 'work on FIDO on Warp so far has been done using the demonstration and prototype Warp machines, which are wire-wrapped machines built according to Carnegie Mellon's design (the demonstration machine was built by Carnegie Mellon; prototypes were built by General Electric and Honeywell). The wirewrapped machine has been superseded by an improved production Warp machine [5], built using printed circuit boards by General Electric. The machine described here is the wire-wrapped machine.

The Warp processor array is a programmable. one-dimensional systolic array, in whch all the cells, called Warp cells, are identical. Each cell is a complete computer, with computational units and local data and program memory, except that address generation is normally done on the interface unit, so that addresses. along with data, flow through the array. Each cell contains two floating-point units; one multiplier and one ALU, each of which can deliver up to 5 MFLOPS. The peak processing rate is 10 MFLOPS per cell. A 4 k-word memory is provided for resident and temporary data storage.

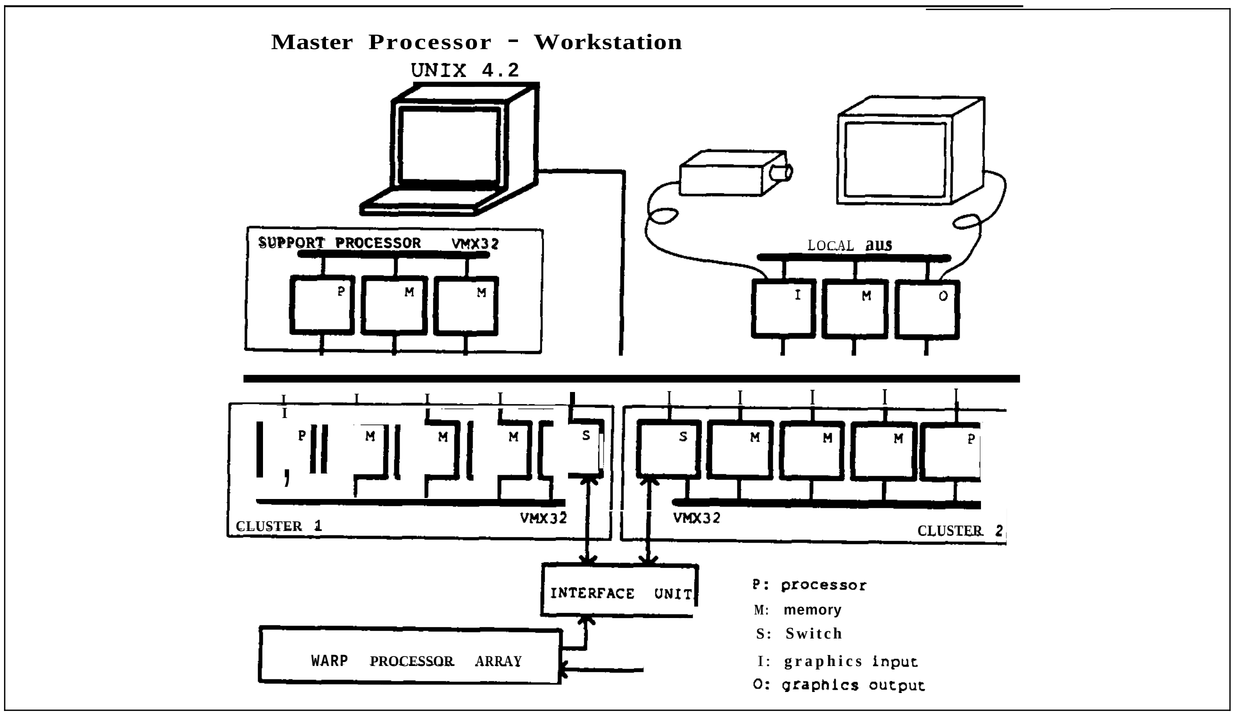

Fig. 1. Warp *host.*

As address patterns are typically data-independent and common to all the cells in low level algorithms, full address generation capability is factored out of 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 conversions in the process.

The host consists of several processors: The 'master' controlling the Warp array is a Sun workstation which can run Unix to provide a convenient programming environment and provide compatibility with other Carnegie Mellon vision research programs. Data transfer to and from the Warp array, and control of vision peripherals such as cameras and frame buffers, is provided by an 'external host' (physically external to the Sun) which communicates with the Sun through a VME bus repeater. The external host consists of three 'standalone processors': two of them, called 'cluster processors', are used for sending and receiving data during a computation (they can exchange roles as needed) and the third one, called 'support processor,' is used for controlling peripherals and executing a runtime library. The runtime library allows event sequencing and processing of interrupts from Warp.

2.2 Warp Software

The development of programming tools that manage the Warp system efficiently requires the integration and extension of state-of-the-art techniques and further advances in compiler technology.

A basic runtime library has been written for the host as a first attempt at handling the system level issues [6]. The library has mechanisms for allocating memory on the external host and sequencing events on the Master, standalone processors, and Warp array as well as mechanisms for running the Warp array and using peripheral devices.

A compiler exists for a language called 'W2', a Pascal-like language in which all the underlying parallelism in the host and within each cell in hidden [7,8]. Only the parallelism between the cells is seen by the user and must be managed explicitly. An asynchronous send/receive protocol is supported for communication between the cells.

3.1. History of FIDO

The predecessor to FIDO was the Stanford Cart developed by Moravec at Stanford University starting in 1973. The Cart was a vision-guided vehicle that could navigate through a world containing obstacles. The vehicle moved in a series of one meter steps. At the end of each step. pictures were taken to determine the Cart's position by tracking known obstacles in the scene.

The Stanford Cart algorithm and the FIDO algorithm contain the same steps. First obstacles are identified in one image, using a feature detector. Then obstacles in one image are matched with obstacles in the other images. The three-dimensional positions of the obstacles can be now be calculated by using a triangulation technique. Next the motion of the vehicle is planned arid executed. After obstacles are found in the new set of images, they are compared to the previous set of obstacles to determine the relative vehicle motion.

The performance of the Stanford Cart algorithm was limited largely by unreliable hardware. For example. each image was digitized 24 times in order to obtain at least one good image. and binocular vision was abandoned in favor of a stereo algorithm that used nine images for greater robustness. The motion of the cart was not accurate. so much time had to be spent determining the cart's exact position after a move. Even with careful optimization. the prograni needed 15 minutes per step on an unloaded DEC KL10. Despite its slow speed. the Standford Cart had several successful indoor runs and a successful outdoor run in 1980.

Thorpe and Matthies developed FIDO. starting with simplifications to the Stanford Cart algorithm due to better hardware. Thorpe and Matthies also experimented with the various modules to give a scientific basis to some heuristically chosen parameters. In addition, some modules (for example, the path planner) were re-implemented using faster algorithms. With all of these changes, FIDO was able.to run in the laboratory. on the Neptune robot on a Vax/780, at a speed of 30 seconds per half meter step. a speedup of 30 over the Stanford Cart. More than 80% of this time was spent running basic vision routines such as image reduction. correlation. and the interest operator.

Crisman began work on FIDO in May 1985. As part of the Carnegie Mellon Strategic Vision Project [9] FIDO was moved to a Sun Workstation (currently a Sun 3). In addition, Crisman began a general cleaning of the FIDO code, which had become quite cluttered with unused and experimental code. Several improvements were implemented to increase the speed of FIDO, including optimizing the display routines and using arcs in the motion generator. The code was also adjusted to run on the new outdoor robot, Terragator. In an attempt to take advantage of much more accurate motion expected from Terragator, the module in FIDO that determined position by obstacle tracking was removed. This also increased the speed of the algorithm.

3.2. FIDO algorithm

FIDO performs the following steps. First, it takes two pictures of its environment. It then tries to locate all of the obstacles that it knows about in the new pictures. Then it checks to see if there are any other obstacles present that it had not been able to see before. It plans a path to its goal around all of the old and new obstacles. Then it moves one meter along the planned path and stops to re-investigate its environment. This repeats until the goal position is reached.

To clarify the algorithm, the following terminology will be used. An *obstacle* is an identified location in the three-dimensional world where the vehicle is unable to pass. A *feature* is the two-dimensional appearance of the obstacle in the image. FIDO considers only point features. This assumes that physical objects will have enough features in the image to be correctly bounded in three-dimensional space.

Below is a more detailed description of the FIDO algorithm.

- ***Digitize:*** Two 512 x 512 images, called the left and right images, are taken from identical

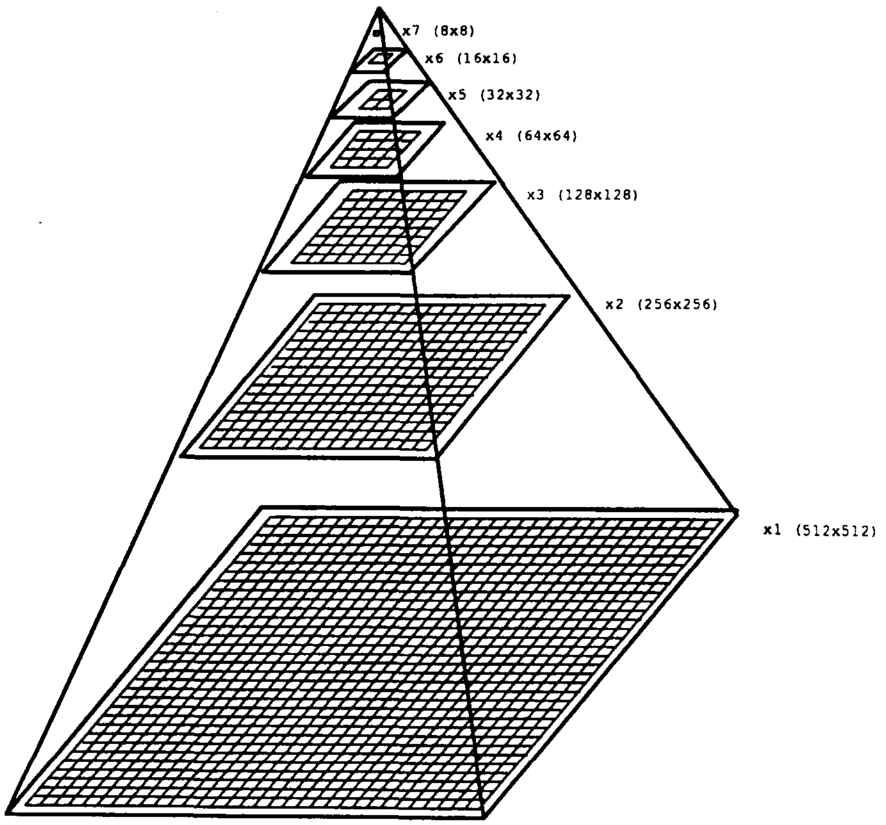

Fig. 2. Image pyramid.

cameras with known relative positions. The right image from the previous image pair is saved for the reacquire step.

- ***Reduce:*** Reduce the two 512x512 images to create an image pyramid as shown in Fig. 2.
- ***Reacquire:*** Find the currently known features from the old right image pyramid in the new right image pyramid, using pyramid correlation as explained in section 5.3.
- ***Match Features:*** Match features from the previous step between the new right and new left image pyramid, using pyramid correlation. The three-dimensional position of the features are then calculated.
- ***Catalog:*** The catalog is a list of obstacles known to FIDO. This forms a simple map describing the environment surrounding the vehicle. The new objects that are discovered by the reacquire and matching steps are added to the map. Previously found obstacles that should be seen but are not are discarded, to eliminate spurious obstacles.
- ***Plan a path:*** Plan a path from the vehicle's current position to its destination that avoids all of the obstacles.
- ***Move***: Move one step toward the destination, along the path.
- ***Pick new features***: Use the interest operator to find new 'interesting' points in the image. These features are added to the features matched in the ***Match Features*** step.
- ***Match new features:*** Find the new features from the new right image in the new left images using pyramid correlation.

In the summer of 1984, Dew. Chang, Matthies and Thorpe designed a new version of the FIDO system to run on Warp, which was then in its initial design phase. They identified the most time consuming parts of FIDO to be the major vision algorithms (correlation, interest operator, and reduction), which were considered to be suitable for implementation on a systolic array such as Warp. They redesigned FIDO to run on a Warp multiprocessor system, based on an early design that used the Aptec bus [10] as the external host [11]. The external host was later changed to be the one described in section 2.1, largely for reasons of programmability.

Once the initial design was done, implementation of FIDO on Warp proceeded in three steps:

Step 1 Implement the three vision modules of FIDO for the Warp array, providing a simple demonstration program to show that the implementation works. This was done by Klinker using Warp microcode on a demonstration Warp system.

Step 2 Reimplement the vision modules on the prototype Warp system. This was done by Clune using W2. At the same time, exploit the external host to overlap computation on the Warp array with preparation and post-processing of the data in the cluster processors, achieving more parallelism.

The steps above use the most powerful part of Warp — the Warp processor array — to speed up the FIDO system. However, it is also possible to use Warp's external host to exploit algorithm-level parallelism and get further speedup. This approach can sometimes give a large speedup in the system, since in some cases significant portions of the code can be run in parallel with the rest of the system, effectively executing that code for free. Therefore. we formulated:

Step 3. Make efficient use of the multiprocessor host system.

To some extent, this step has already been accomplished, but there is still room for exploiting algorithm-level parallelism in FIDO. as we shall discuss later.

Section 5 describes the initial implementation of the vision modules on Warp using microcode (Step 1). Section 6 describes the current implementation of FIDO on Warp in W2 (Steps 7 and 3).

FIDO was first implemented on the Warp demonstration system, which included a Sun 2. an interface unit, and two Warp cells. In addition to being the first work on FIDO on Warp. this step showed in the demonstration system that Warp can be used for vision applications. Moreover. it allowed us to test algorithm decomposition techniques and to test the Warp hardware with realistic algorithms. Once the initial implementation was done, it was ported to the ten-cell prototype system and modified to use the cluster processors for input and output. This implementation was then superseded by new software, as discussed in the next section. Here we describe the vision modules in FIDO in detail and give their timings on the demonstration system and first prototype implementation.

5.1. Image Pyramid Generation

The image pyramid used in FIDO consists of 7 levels, starting with a **512 × 512** image at level $x1$ and ending with an 8×8 image at level $x7$, as shown in Fig. **2.** Areas **of 2 × 2** pixels are replaced by one pixel in the next higher level of the pyramid. The new pixel value is computed by averaging over a window in the higher resolution image to produce a one pixel result in the lower resolution image. The simplest averaging is to take a **2 × 2** pixel area and average it to one pixel. The initial implementation on the Warp array used overlapping **4 × 4** windows, which gave slightly better results than **2 × 2** windows.

5.1.1. Mapping the Algorithm onto Warp

The pyramid generation algorithm has been implemented in Warp microcode in a systolic scheme, as suggested by Kung for convolution-type algorithms **[12].**

The pyramid generation algorithm was planned to fit the ten cell Warp array. The algorithm required the processors to accumulate and normalize 16 pixels in a **4 × 4** window to produce one reduced pixel value. This was mapped onto the Warp array as nine modules, with the first eight each adding two new pixel values to the accumulated partial sum, and the ninth module normalizing the result. The second, fourth and sixth module also stored the partial results until the necessary pixels from the next row underlying the **4 × 4** window had arrived at the module. The new data and the partial results were then sent together to the next module.

Because the demonstration Warp system consisted of only two cells when the micro-programmed version of the pyramid generation algorithm was implemented and tested, we had to face the problem of mapping nine conceptual, algorithmic modules onto two physical cells. This was achieved by having the first cell run the first four modules and the second cell run the remaining five modules. Each cell switched between consecutive modules whenever eight new partial results were produced. Thus, the pixels were processed in batches of sixteen at a time. Two extra pixels had to be sent with each hatch. due to the overlapping kernel.

5.1.2. Microcode Timing of the Pyramid Generation Algorithm

Since the entire processing in each module consisted of adding two numbers and passing them on, each module needed two processing cycles to perform its task on a given pair of input pixels. The modules could thus start to operate on a new window position every two cycles. Accordingly, the modules could perform their tasks as fast as the data could be sent to the cells. On an image with $m \times n$ pixels, the nine modules thus needed essentially $m \times n$ cycles to reduce an $m \times n$ image into an $\frac{1}{2}m \times \frac{1}{2}n$ image. plus a few initial cycles for startup. To generate the described image pyramid, consisting of seven levels, this evolved to

$$\sum_{l=1}^{6} \frac{512}{2^{l-1}} \times \frac{512}{2^{l-1}} \text{ cycles.}$$

which is

341401 cycles **-** **0.07** seconds on 9 cells.

1365358 cycles **-** 0.**27** seconds on 2 cells.

A simpler sequential algorithm (with non-overlapping reduction windows) took about one second on a Vax/780. Nine Warp cells thus provided a speed-up of 14. which is relatively small. The Warp implementation of the pyramid generation algorithm was communication intensive: it used the adder effectively only half of the time (in every other row). It did not use the multipliers at all (except for a normalization). Each Warp cell was used as a **2.5 MFLOPS** machine, for 25 MFLOPS from the array. This explains the relatively small speed-up of the pyramid generation algorithm. Note that adding more cells would not increase the speed since this would not reduce the communication requirement.

5.2. Interest Operator

'Interesting points' are those points which can be localized in different- pictures (for example

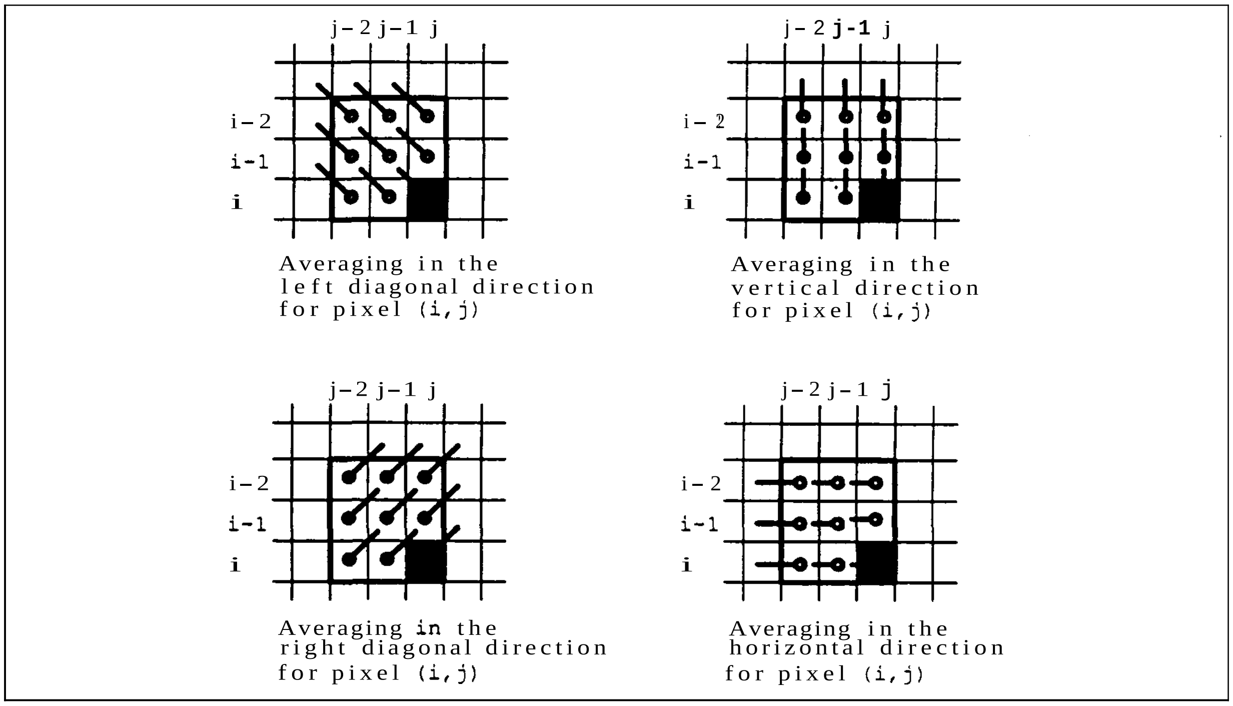
j-2 j-1 j
i-2
i-1
i
Averaging in the left diagonal direction for pixel (i,j)
Averaging in the vertical direction for pixel (i,j)
Averaging in the right diagonal direction for pixel (i,j)
Averaging in the horizontal direction for pixel (i,j)

Fig. 3. Calculation of the interest operator.

comers). The image intensities change rapidly in all directions for a point that is 'interesting'.

Interesting points are found using an interest operator, which is largerly unchanged from Moravec's Stanford Cart work. The interest operator takes squared pixel differences around a pixel in the vertical, horizontal and both diagonal directions and accumulates them (separately for each direction) in the 3×3 neighborhood of the current pixel [2; 11], as shown in Fig. 3. For the current pixel to be an interesting point, all four accumulated differences must be large. Their minimum gives the interest value of the current pixel. The interest value are locally maximized in one hundred subimages that are arranged in a 10×10 grid. The maxima of all subimages are stored in a list of decreasing interest values. This gives a set of points, distributed across the image, which can be localized in the image for matching.

Only the first part of this algorithm, computing the accumulated squared pixel differences in all four directions for every pixel, was implemented in microcode on the Warp array. In the demonstration system processing stopped here. Later, we implemented the minimization, maximization and list processing on the cluster processor.

5.2.1. Mapping the Algorithm on Warp

The interest operator does not have a good partitioning into modules with similar timings. We thus did not try to implement it in a systolic scheme, as was described for the pyramid generation algorithm in the previous section. Instead. we used the ***input partitioning*** model [13] in which not the algorithm. but the data is divided into equally sized parts. In this scheme. each cell performs the complete algorithm on a portion of the data. An $m \times n$ image is devided into c vertical stripes to be processed on c different cells. For the interest operator, the stripes had to overlap by 4 pixels. due to the width of the operator window. Thus. every cell ran on $m \cdot (\lceil n/c \rceil + 4)$ pixels. The systolic communication facilities were then used like a 'bus': each cell received data from the previous cell and sent it to the next cell. The host sent the data interleaved such that each cell could use every cth pixel for itself. At the beginning of every new iteration, c new pixels were sent over the 'bus'. The offset between programs that ran on neighboring cells was two cycles such that each cell started a new iteration exactly when a new pixel arrived.

Note that. since the programming scheme of the interest operator was organized around partitioning the data and not the algorithm, this algorithm could be easily adapted to run on any number of cells. In the demonstration system, the algorithm ran on two Warp cells, computing the interest value for every pixel of a 256×256-sized image (level $x2$ of the right image pyramid) in two parallel vertical stripes each consisting of 256×132 pixels. It was a matter of changing a few constants that indicated the width of the vertical stripes to provide a version that ran on ten cells, when the ten cell Warp became available. The ten cell version divided the image into ten vertical stripes, each consisting of 256×30 pixels.

The algorithm iterated on a sequence of two steps:

(1) Get the next pixel, compute the difference between the new pixel and its neighbors in four directions, and square the differences.

(2) Add the squared differences to the partially computed interest values of the nine pixels whose 3×3 windows overlap at the current pixel.

Within each of the two steps, the code was optimized using software pipelining to use all facilities, such as the multiplier and adder, of the cell simultaneously. For ease of programming, no optimization was organized between the steps. This made the innermost loop 65 cycles long, whereas the ideal algorithm would have needed only 40 cycles.

The algorithm must store the most recent two rows of pixels and the most recent three rows of partial interest values. Thus, $(2+3\cdot 4)\cdot\lceil n/c\rceil+4$ pixels were stored in the local memory of each Warp cell. For the given memory size of 4096 k words, a maximal row width of $n=256$ columns per cell could be allowed.

5.2.2. Microcode Timing of the interest Operator

The interest operator ran in

$$(m-1)\cdot\left(\left(\left\lceil\frac{n}{c}\right\rceil+4\right)\cdot 65+21\right)+30$$
$$+c\cdot\left(\left\lceil\frac{n}{c}\right\rceil+4\right)\text{ cycles.}$$

For a 256×256 image, the algorithm takes

502935 cycles - 0.10 seconds on 10 cells,
2193549 cycles - 0.44 seconds on 2 cellls.

The sequential algorithm ran in about 2.65 seconds on a Vax/780. Ten Warp cells provided a speed-up factor of 26.5. The adder was the most used resource of the interest operator. It was used in 40 out of 65 cycles of the innermost loop. The multiplier was barely used (4 multiplications in 65 cycles). The algorithm thus used each cell as a 3.4 MFLOP machine. The addition of more cells would greatly improve the speed. In the described implementation, each cell needed a new pixel every 65 cycles. Thus, maximally 65 cells could have been used in parallel before the interest operator had become 1/0 limited. It would then have needed $O(m \times n)$ cycles to compute the results.

5.3. image Pyramid Correlation

For a given pair of images and a given list of interesting points in one image. the correlation algorithm is used to find the corresponding points in the other image. The search for the most likely corresponding points of the interesting points is performed on the image pyramids, starting at the highest level ($x7$:8×8 image). At each level. a 4×4 template around the current interesting point is taken and correlated with an 8×8 search area in the other image pyramid at the same level. The best matching position of the template in the search area determines the position of the search area in the next lower, more detailed, pyramid level [1], as shown in Fig. 4.

Pseudo-normalized correlation is used. as given by this formula.[11]:

$$\mathrm{CORR}_{lm}=\frac{S_t-t_{\text{mean}}S_1}{t_{\text{var}}+\left(S_2-S_1^2\right)/16}$$

with

$$S_1=\sum_{i,j=0}^{3} I_{i+l,j+m},\qquad S_2=\sum_{i,j=0}^{3}\left(I_{i+l,j+m}\right)^2,$$

$$S_t=\sum_{i,j=0}^{3} t_{ij}I_{i+l,j+m}$$

where t_{ij} denotes the template element at position (i, j),and $I_{i+l,j+m}$ denotes pixel at position $(i+l, j+m)$ in the image.

In the micro-programmed version of the algorithm, Warp was used to find the optimal correlations of all features for one given pyramid level at a time. First, all templates of a pyramid level were sent. The cells stored the templates and computed their means and variances. Then the

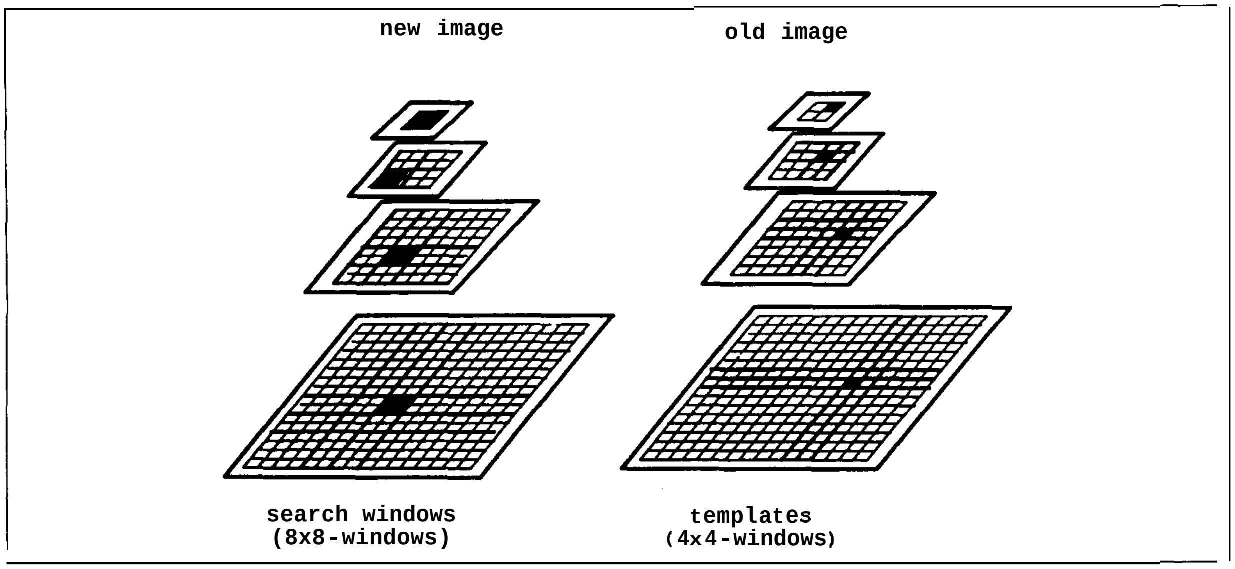

Fig. 4. Correlation over an image pyramid.

search areas of that level were given to the Warp array in the same sequence as the templates. The cells correlated the current template with the current search area and sent the correlation results for every template position to the output cluster. The cluster processor then found the optimal position of each template within its search windown and determined the search for the next lower level. The process was then repeated for the next lower pyramid level [11].

5.3.1. Mapping the Algorithm onto Warp

The correlation algorithm was implemented in a systolic programming scheme, as for the pyramid generation algorithm. It was designed as nine modules. Each of the first modules covered two template elements. The algorithm was designed so that initially, each module received the template elements and stored the respective template elements of each template. The mean and the variance of all templates were computed and stored in the ninth module. Then, in the correlation phase, each module got the pixels of the search areas and the partial sums S_1, S_2, and S_t from its left neightbor and updated the partial sums before it sent them to its right neighbor with the next pair of pixels. As in the case of data pyramid generation, the second, fourth and sixth module stored the derived partial results until the pixels of the next row, underlying the current window position. arrived. The ninth module combined the partial sums and the mean and variance of the current template into a correlation value that denoted how well the template fitted the data in the search area at the current position.

In the demonstration system, the correlation algorithm ran on two cells. The first four modules ran on the first cell, the other five modules ran on the second cell. For every pair of pixels. six additions and four multiplications had to be performed in the first eight modules. In the ninth module, one addition, two subtractions, three multiplications. and two divisions had to be executed. Thus, the adder was the most used resource for the first eight modules (used six times per module run), whereas the multiplier was used most in the last module (five times, so that it was not the bottleneck). It was possible to achieve the optimal speed for the first eight modules, i.e., start a new module run every six cycles, keeping the adder busy all the time. When the modules were run on two cells, however, the ninth module had to share the resources of the second cell with the fifth through eighth module. Since the microcode of the ninth module was very different from the microcode of the other modules (heterogeneous modules), it was impossible to schedule the facilities of the second cell such that one resource was used in every cycle: the ninth module required that the innermost correlation loop be augmented by twelve cycles (i.e., two extra module runs).

5.3.2. Microcode Timing of the Correlation Algorithm

The nine modules needed $r \cdot [(6m^2 + 18) \cdot n + 113]$ cycles for n templates and search areas, with each search area being of size $m \times m$, on r resolutions. In FIDO's case, $n = 50$, $m = 8$, and $r = 7$. The algorithm thus ran in

141491 cycles $\approx$ **0.03** seconds on *9* cells,

848946 cycles $\approx$ 0.16 seconds on 2 cells.

The sequential algorithm took about 2.3 second on a Vax/780. Nine cells thus provided a speed-up factor of *78*. This was a much higher speed-up than that acheived by the pyramid generation algorithm and the interest operator because the multiplier was used in every cycle and the adder was used in every other cycle. Each cell thus ran here as 7.5 MFLOP a machine. The communication facilities were also used in every other cycle. Therefore, the correlation algorithm was a fairly well balanced algorithm. The maximum speed-up would have been reached if *18* cells had been used (due to communication requirements).

5.4. Performance of the Vision Modules

Times for the three vision modules are shown in Table 1. The speedup (optimal and actual) of a Sun 2 with Warp over a Sun 2 without Warp is also shown for each module. As mentioned above, only the two cell system was available at this time. In addition, the system software was not fully tuned so that it took approximately **0.3** seconds to call each Warp module.

Most interesting is the speedup of the pyramid generation module on Warp. It actually takes longer to run on the Warp than on the Sun alone. This is because the data flow between the clusters and Warp is unbalanced. Very time consuming manipulations were required to order the data correclty for Warp in this implementation, but the actual pyramid generation on the Warp array is not computationally intensive. The array is virtually starved for data. This is a case where the ordering of data is too complex for Warp (specifically the clusters). A more efficient implementation is for the cluster processors to send the pixels in the order it is stored in memory so that data can flow rapidly into the array, and have the Warp array handle the data reordering itself.

The interest operator and correlation functions did not perform at the optimal speeds either, although they are faster than the comparable Sun functions. If the startup times for the Warp implementations is subtracted (the overhead for startup is much less in the prototype and production Warp machines), the actual times are close to optimal times as shown in Table 1.

The hardware completely changed from the demonstration system to the prototype system described in Section 2. The number of cells in the Warp array increased from two to ten. Just as important for system performance, the master was upgraded from a Sun 2 to a Sun 3 with the MC68881 floating point hardware. and the external host became available. These changes markedly improved the performance of the system.

In addition. the software environment changed radically. Previously. all of the vision modules for the demonstration system were coded directlv in microcode, a tedious task. With the prototype system, the W2 compiler became available, greatly simplifying the programmer's task.

Table 1
Demonstration timings

Function	System					Speedup	
	Vax/780	Sun 2	Clusters	Warp: Optimal	Warp: Actual	Maximum (Warp/Sun2)	Actual
1 pyramid generation	0.5 sec	3.2	0.7	0.3	18.0	11	0.2
2 pyramid generations	1.0 sec	6.4	0.8	0.6	36.0	11	0.2
Interest op.	2.7 sec	13.0	-	0.4	0.9	33	14.0
Correlation	2.3 sec	14.0	-	0.2	3.0	70	4.7

We completely reimplemented the vision modules as a result of these changes:

Pyramid generation. This module was reimplemented as a C program to be run on the clusters, since very little computation is done here. This made it possible to do the two pyramid generations in parallel using the two cluster processors. In this implementation non-overlapping 2 x 2 windows were used instead of the overlapping **4 × 4** windows in the Warp implementation. to simplify the computation.

Interest operator. This module was reimplemented in W2, without significant change in the algorithm.

Correlation. This module was originally written as a systolic program, but could not be reimplemented in this way because the prototype W2 compiler allowed only homogeneous code. Instead, it was implemented using input partitioning, as in the interest operator.

In addition, preprocessing and postprocessing of the data in modules that send data to the Warp array was implemented as C programs to be run on the clusters, which send and receive data directly to the Warp array. These C programs replaced the standard compiler-generated modules, which transferred data to the Warp array from memory. Use of the clusters in this way exploited some of the parallelism available in the Warp system.

6.1. Analysis of FIDO performance on the Prototype

The reimplementation of FIDO led to a total system time for one step of **4.8** seconds, which is a large speedup over the original time, but still relatively small compared to the time on a Sun 3 alone (8.5 seconds). Here we analyze how we got this speedup and identify the parts for the system that allow a further increase in performance (Table 2).

6.1.1. Pick New Features

'Pick new features' includes the interest operator function. The actual time for this function to execute is about 0.1 seconds, the same as estimated in Section **4** for the ten cell implementation. Some of the rest of time is spent starting the Warp array (about 25 milliseconds). However, most of the time is spent in post processing. After the interest operator has been run, some sorting and selecting is done from the resulting data. This is done on one of the cluster processors. which is about 28% slower than the Sun 3 processor. because of a slower clock rate. The effect of this post-processing is shown in Table 3.

The interest operator is sped up bv a factor of 10, from one second to 0.1 seconds. This leaves only the sorting and selecting, which was about one-quarter of the time of the original function. but which is 80% of the time in the Warp implementation. The total speedup is approximately three.

6.1.2. Correlation Modules

All of the modules that use the correlation function (e.g. 'Match Features') have less thnn a factor of three speedup, compared to a Sun 3 alone. **A** breakdown of the times for Warp is shown in Table 4. As with the interest operator. the implementation of the correlation function on the Warp array reduces its execution time. formerly a large component of the total. to a small factor. and the small amount of time needed to process data for the Warp array on the cluster processors

Table 2
FIDO module times

Module Function	System		
	Vax/780 (sec)	Sun3 (sec)	Sun 3 w/Warp (sec)
Reduce images	4.8	0.9	0.8
Reacquire features	13	1.6	0.6
Match features	7.8	1.6	0.6
Catalog features	0.1	0.1	0.1
Plan a path	3.1	0.7	0.7
Pick new features	2.1	1.6	0.5
Match new features	7.8	1.6	0.6
3-D calculation	N/A	0.4	0.9
Total time	39	8.5	4.8

Table 3
'Pick new features' times

Function	System	
	Sun 3 (sec)	Warp
Interst op.	1.3	0.1
Sorting	0.3	0.4
Total time	1.6	0.5

Table 4
Times for correlation modules

Function	Time (sec)
setup	0.3
Startup	0.2
Correlation	0.1
Total time	0.6

dominates the total. This time included the following:

- Warp startup overhead of 25 ms. In one step, correlation is called seven times, for a total overhead of approximately 0.2 seconds.
- Rearranging data for Warp. Complex addressing is needed to send the image patches from the different pyramid levels to Warp.
- Because of restrictions in the prototype W2 compiler, a fixed number of features must be processed in every correlation, which is set to fifty in this case, although the average number of features in a correlation is approximately half that.

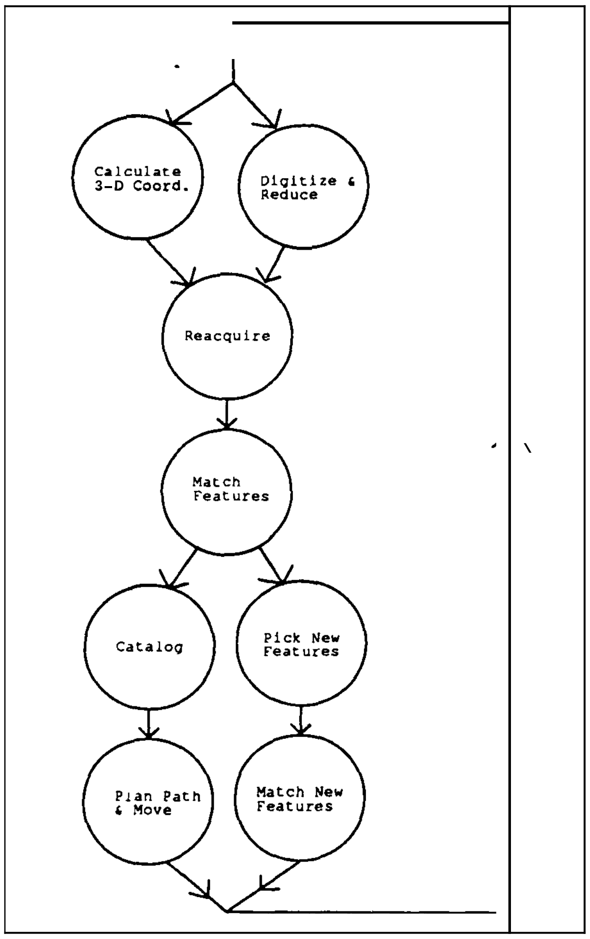

Fig. 5. Planned FIDO loop with parallelism

Table 5
FIDO module times

Module Function	System	
	Sun 3 w/Warp (sec)	Sun 3 w/Warp parallel (est) (sec)
Reduce images	0.8	-
Reacquire features	0.6	0.6
Match features	0.6	0.6
Catalog features	0.1	-
Plan a path	0.7	-
Pick new features	0.5	0.5
Match new features	0.6	0.6
3-0 calculation	0.9	0.9
Total time	4.8	3.2

It is possible to exploit algorithm-level parallelism in FIDO. as shown in Fig. 5. Pyramid generation, which takes a large amount of time since it must access every pixel of the original image. can be executed in parallel with calculation of the three-dimensional coordinates of the image features. Similarly, finding new features can be done in parallel with cataloging and path planning execution. FIDO does not yet take advantage of these sources of parallelism, but our analysis of FIDO so far allows us to make predictions of performance once these approaches have been exploited.

These predictions are shown in Table 5. compared with the times on the current FIDO system. We see that pyramid generation and cataloging and path planning and execution are done for free. This reduces the total time for FIDO from 4.8 seconds to 3.2 seconds, which is a speedup of 2.6 over the Sun 3 version.

General Electric has now manufactured a production version of the Warp machine. as mentioned earlier. This machine design was influenced by our experiences with the prototype system, including our experiences with FIDO. We have made a number of modifications to this system, which will improve the performance on FIDO:

System overheads reduced. Overhead for startup of a Warp program is now 5 milliseconds, down from ***25*** milliseconds on the prototype. This will substantially reduce the overhead of calling a Warp program with a fast execution time. For example, this should reduce the startup overhead in correlation from approximately 0.2 seconds to 35 ms.

Heterogeneous code. In the course of implementing FIDO, we originally implemented the pyramidgeneration and correlations functions systolically, then reimplemented them using input partitioning when we reprogrammed them using W2. This was due to a restriction in the prototype machine that made it impossible for W2 to support heterogeneous code in a general way. This restriction has now been removed, and we are free to reimplement these modules systolically.

Variable loop bounds. The prototype W2 compiler required that all loop bounds be known at execution time. In FIDO, this meant that the correlation function always processed fifty features, although the average useful number was twenty-five. This implied a doubling of the execution time on Warp, and, even more significatnly, a doubling of the time for the host to prepare data to be sent to Warp. Allowing variable loop bounds in the W2 compiler should therefore eliminate approximately 0.2 seconds.

More flexible programming model. The production Warp machine supports programs with variable execution time, and makes use of the reverse path in the Warp array (therefore, global communication among the cells that does not pass through the host) more efficient. This means the FIDO can be reorganized to do even more computation on Warp, eliminating host overhead altogether.

Hardware improvements. The hardware of the production Warp machine includes these enhancements:

- Direct framebuffer communication with the Warp array. A special purpose board has been constructed that allows direct transfer of data from the framebuffer to the Warp array, bypassing the cluster processors. This will allow reimplementation of the pyramid generation step on the Warp array, with each pyramid generation taking approximately 60 ms.
- Faster cluster processors. The clusters are being upgraded to use microprocessors with a faster clock.
- DMA from the host. The new processors also support DMA from the host to the Warp array, eliminating a significant bottleneck in feeding data from the host to the Warp array.

We have discussed the history of the FIDO algorithm, and its gradual increase in speed over a period of some 13 years, starting from Moravec's work on its predecessor at Stanford, through the current implementation on Warp. Over this period of time, three things have influenced its speed:

The reliability of the sensing and motor control devices available. With better digitizers, the matching was more reliable, which helped Thorpe reduce the number of images needed for reliable matching. With better motor control, Crisman could completely eliminate the motion parallax step from FIDO. This accounts for about one order of magnitude increase in speed, but much more than this in terms of programming simplicity and ease of use.

The speed of the computer hardware. This effect has been masked by the willingness of different researchers to carefully optimize code for greatest speed: Moravec unwound loops into arrays for maximum speed, for example, and Klinker programmed by hand a machine for which compiling was considered a significant research project. Also. the execution times for early versions of FIDO were never obtained routinely, but only in demonstration runs when everyone else could be removed from the system, which was a significant computing resource for many people. Execution times for more recent versions of FIDO are from more or less routine runs, when only a few if any people were inconvenienced by the desire to get the best time possible. Moreover, the effect of different improvements in hardware varies: the most sudden speedup in FIDO was due not to the introduction of Warp, which required recoding programs, but due to the replacement on a Sun 2 by a Sun 3 workstation.

The resourcefulness of the researcher. A researcher can get more use out of a computer system by placing constraints on calculations to reduce processing. For example, Thorpe and Matthies were able to increase the reliability of FIDO, while reducting the number of images needed, by adding more constraints in the stereo matching. This has accounted for about one order of magnitude improvement in speed. Note that the usefulness of constraints depends on the reliability of the underlying hardware (e.g. sensors) and that any improvement in computational speed of a program can be used to perform more experiments or incorporate other functions into the program.

Warp's potential in the implementation of FIDO is due to several factors, which reflect not only on the design of Warp but also on other special-purpose machines:

(1) The Warp array works well for the majority of the computation in FIDO, namely low-level vision computations. Working either in microcode or W2, we never had problems with the Warp array not having enough effective computation power. However, while low-level vision computations form the majority of the computation of FIDO, simply speeding them up is not enough for good speedup of the FIDO system as a whole.

(2) The external host is the weakest part of the Warp system. This was known when the host was designed; it was determined once we decided to use industry-standard processors and buses, instead of building our own. In our early versions of FIDO on Warp, this kept us from realizing full use of the Warp array, because of the constraints in rearranging data on the external host.

(3) The programmability of the Warp array allowed us to modify our algorithms and programmings models to accomodate a regular data patterns from the host. This is important even in the latest versions of the host, which have faster processors and higher data rate, but which can use DMA, which requires a regular address pattern. This flexibility is the main reason we have been able to observe the predicted performance of algorithm in actual Warp runs.

(4) W2 makes it possible to experiment with different algorithms, in the context of a research system such as FIDO, while getting good use of the powerful Warp array. As we program more and more of FIDO on the production Warp machine, programmability is essential, especially as it allows us to make use of more complex programming models that use the powerful Warp array more and require less intervention by the relatively weak host.

(5) Although the computing power of the external host is small compared to the Warp array, its programmability, and its close integration with the master and the Warp array, makes it an important part of the Warp system. Irregular operations can be mapped onto it as part of pre- and post-processing of data from Warp. Also. it can sometimes perform memory access-intensive but not compute intensive computations as well as or better than the Warp array, which can also allow the Warp array to be used for something else in the meantime.

The Warp group, a large and growing group at Carnegie Mellon and General Electric, has contributed to this work by the design. implementation, and maintenance of the Warp machine and associated software. Research in robot locomotion cannot progress without reliable mobile robots, the design, implementation, and maintenance of which is a difficult problem in itself; we are therefore indebted to the Mobile Robots Lab at Carnegie Mellon for Neptune. and the Field Robotics Center at Carnegie Mellon for Terragator, both of which ran FIDO. We have also benefited from the research contributions of the Image Understanding System group at Carnegie Mellon.

[1] H. Moravec. Obstacle avoidance and navigation in the real world by a seeing robot rover. Technical Report CMU-RI-TR-3, Carnegie-Mellon University Robotics Institute, September. 1980

[2] C.E. Thorpe. FIDO: vision and navigation for a robot rover, PhD thesis, Carnegie-Mellon University. December, 1984.

[3] L.H. Matthies and C.E. Thrope, Experience with visual robot navigation. Proc. IEEE OCEANS' 84 Confer. September, 1984. pp. 594–597.

[4] H.T. Kung and O. Menzilcioglu. Warp: a programmable systolic array processor, Proc. SPIE Symposium 495. Real-Time Signal Processing VII. August 1984, pp. 130–136.

[5] M. Annaratone, E. Arnould, R. Cohn, T. Gross, H.T. Kung, M. Lam, O. Menzilcioglu, K. Sarocky, J. Senko and J. Webb, Warp architecture: from prototype to production. in: ***Proceedings of the 1987 National Computer Conference*** (AFIPS Press. Reston, VA, 1987).

[6] M. Anneratone, Warp host software requirements and deliverables, Carnegie Mellon Department of Computer Science. 1985.

[7] T. Gross and M. Lam, Compilation for a high-performance systolic array, Proceedings of the SIGPLAN 86 Symposium on Compiler Construction (ACM, New York, 1986).

[8] B. Bruegge, C. Chang, R. Cohn. T. Gross, M. Lam. P. Lieu, A. Noaman and D. Yam, The Warp programming environment, in: ***Proceedings of the 1987 National Compurer Conference***. AFIPS,

[9] T. Kanade and C. Thrope (with contributions from CMU SCVision Project Staff), CMU Strategic Computing Vision Project Report: 1984 to 1985. Technical Report CMU-RI-TR-86-2. Carnegie-Mellon University, The Robotics Institute. Pittsburgh. PA. November, 1985.

[10] G. McApline, W.J. McLain and G.B. Feldkamp, Controller smooths data flow through multiprocessor systems. *Electronic* ***Design*** **45–49** (1982).

[11] P. Dew and C.H. Chang, Passive navigation by a robot on the CMU Warp machine, Internal report. Department of Computer Science, Camegie-Mellon University, August 1984.

[12] H.T. Kung, Systolic algorithms for the CMU Warp Processor. in: Proc. 7th Internat. Confer. Pattern Recognition, International Association for Pattern Recognition. 1984. pp. 570–577.

[13] H.T. Kung and J.A. Webb. Mapping image processing operations onto a linear systolic machine. *Distributed* ***Computing*** **1** (4) (1986) 246–257.