Hardware Parallelization of the Scale Invariant Feature Transform Algorithm

Jasper Schneider, Skyler Schneider

***Abstract* —In robotics, problems of image recognition and robot localization can be solved with the help of the Scale Invariant Feature Transform (SIFT) algorithm. This algorithm can extract scale and rotation invariant features from a 2D image, but it is computationally intensive. In this paper, we discuss our attempts to increase the speed of the SIFT algorithm by parallelizing it in a very-high-speed integrated circuit hardware description language (VHSIC-HDL, or VHDL) module included in a LabVIEW project to be programmed onto a field-programmable gate array (FPGA) of an NI Robot. This parallelization would allow for fast image recognition on a robot through the use of reprogrammable hardware. Our results have revealed that a full-hardware implementation of the SIFT algorithm on this robot platform is less effective than a slower, software implementation, and that this platform is not well suited for image manipulation techniques.**

I. Introduction

THE SIFT algorithm, developed by David Lowe, is a good algorithm to use for feature recognition because it is both scale and rotation invariant. Objects that appear in one image can be recognized as the same objects in a second image, even if they are closer to or farther away from the camera and placed at a different angle. The key quality of SIFT features that enable this kind of recognition is that each feature contains descriptors of the surrounding environment, captured in the scale and orientation of the feature [1].

The SIFT algorithm has already been put to use in previous robot-related projects. Se, Lowe, and Little used SIFT features as landmarks to create a 3D model of surrounding objects in a room and navigate a robot to avoid obstacles [2]. Valgren and Lilienthal were able to extend the use of SIFT to an outdoor environment for long-term localization [3]. Lowe himself has suggested the use of SIFT features for image recognition, which would allow a robot to seek out or avoid specific objects [4].

Unfortunately, the SIFT algorithm itself is slow in that it requires many computations, including multiple Gaussian filters and gradient calculations. Most previous implementations of the SIFT algorithm have been on stationary computers for use in image processing. For example, Robert Hess implemented the SIFT algorithm in C code using the OpenCV library [5], which would only be practical to run on a larger commercial processor like those found in laptops and desktop computers, not a smaller microprocessor like those often found in small mobile devices. The previously mentioned robot projects were implementing the SIFT algorithm on mobile robots with

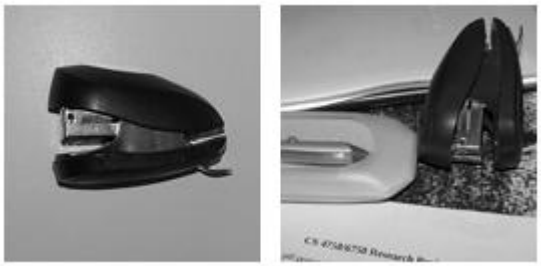

Fig. 1. Example of a case where SIFT feature recognition would be beneficial. An object of interest (stapler, left) is present in the right picture but smaller and rotated.

embedded computers (for example, running at 2Hz on a Pentium III 700MHz processor). This kind of system is relatively expensive and computationally intensive compared to robots running on small microcontrollers or FPGAs.

However, the algorithm lends itself well to parallelization. As an example, the Gaussian filters are applied to each pixel in an image, and these independent calculations can be performed in parallel to lessen the total processing time of the algorithm. Several small processing elements, or a Single-Instruction-Multiple-Data (SIMD) processor architecture could theoretically compute the filtered values of different pixels in parallel, and then later calculate quantified characteristics of different SIFT features in parallel.

In this paper, we focus on parallelization and other speed benefits through specialized low-level hardware. Our ultimate goal was to implement the SIFT algorithm with VHDL, which can be downloaded to an FPGA. The benefits of programming the hardware of an FPGA is that the properties for different pixels or features can be computed in parallel physically as signals on different wires, and specialized hardware setups can be configured to quickly compute common mathematical operations in the SIFT algorithm that would otherwise require several instructions on a generalized processor to compute.

In order to transfer a dominantly software-implemented algorithm like SIFT into a low-level hardware implementation, we needed to break down the algorithm into a series of operations that could be performed in hardware. We first wrote our own software implementation of the SIFT algorithm at a basic level (e.g. no functions, just instructions that have an easily translated equivalent in hardware). From there, we reorganized the code into a state machine format to emulate a sequential hardware implementation and facilitate translation into VHDL. Then,

we actually performed the translation into a VHDL module.

The remainder of the paper is organized as follows. In Section II we discuss the robot we used and the programmable interface for configuring the robot. In Section III, we discuss the SIFT algorithm and our approach to parallelizing the algorithm. In Section IV we discuss the results of the project. This includes the preliminary results of the intermediate low-level software SIFT implementation, as well as the results when downloading the VHDL hardware implementation onto the robot, including problems that arose and our solutions to those problems. Section V summarizes our results and draws from them conclusions about the project. It also offers suggestions for future work with regards to the algorithm and robot.

II. Robot

The robot used in this project is the NI sbRIO-9631 included in the NI LabVIEW Robotics Starter Kit for Education. This robot is pictured in Figure 2. It includes an ultrasonic sensor, two single-direction wheel motors (for the left and right wheels of the robot), four LEDs, and a Xilinx Spartan-3 FPGA, which is programmable through an Ethernet connection to a computer running LabVIEW Robotics software.

After creating our VHDL implementation of the SIFT algorithm, we program it onto the NI robot's FPGA through the LabVIEW software. This software allows the insertion of VHDL modules to control the hardware design on the FPGA by use of "HDL Nodes" in a block-level design of the system. Our VHDL module can therefore compute whether or not each wheel of the robot should spin depending on where an object of interest is in an image fed to the robot.

The original goal of the project was to have the robot identify and move toward an object of interest, initially chosen to be a tennis ball. A series of training images would be supplied to the robot in order for it to learn the SIFT features that characterize the ball and then have the robot acquire test images via a connected camera and decide whether to move left, right, or forward in order to approach the ball after detecting it. However, conversations with NI technical support convinced us that it would not be possible to attach a camera in this manner. Similarly, it would not be possible to connect a wireless router to the robot's Ethernet port and have image data from a camera streamed wirelessly to the robot. This approach in infeasible because the robot only uses its Ethernet connection to communicate with the computer running LabVIEW for the sake of downloading designs and exchanging debug information.

With no better option for acquiring real-time image data, we decided to directly program image data into a hardware ROM when downloading our design onto the robot and have the VHDL access this ROM instead of actual camera input. Both the training images and test images must be downloaded in this manner, and so our project became a proof-of-concept experiment for the application of a hardware-implemented SIFT algorithm to be extended by further research with a setup that allows real-time image analysis.

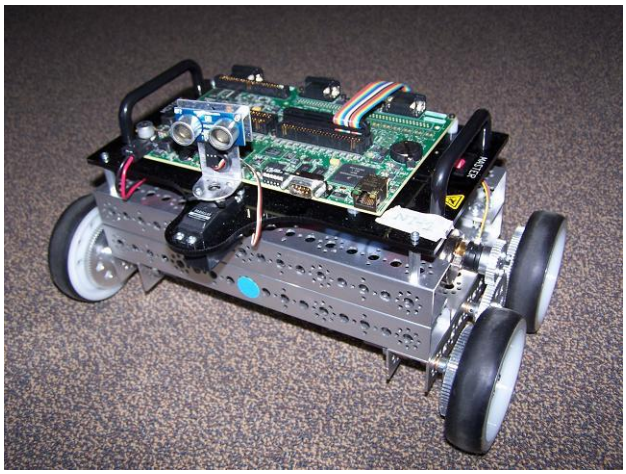

Fig. 2. The NI robot used in our project, including FPGA board, wheels, LEDs, and ultrasound sensor.

Since we do not have a camera, and the images we provide may not be of the actual environment the robot is in, the response of our robot has been modified. We do not want our robot to crash into things, so we have instead disallowed the robot to move forward. It can rotate left or right in circles, but instead of moving forward it merely signals with an LED that it theoretically would like to move forward if the image being shown to it were an actual input image from a camera. That is, from the test images, the robot either turns left if the target object is on the left of the image, turns right if the target object is on the right of the image, lights an LED if the target object is near the center of the image, and does nothing if the target object is not detected.

III. Our Approach

A. Preliminary Low-Level Software

The SIFT algorithm is used to identify features in an image, regardless of scale and orientation. We began with a 128x128 grayscale image.

There are four main stages to identifying SIFT features. In order, they are scale-space extrema detection, keypoint localization, orientation assignment, and keypoint descriptors [6].

First, we discuss our scale-space extrema detection. Scale-space is a continuous function of scale, the only possible kernel of which is a Gaussian function. Therefore, we construct a Gauss pyramid (shown in Figure 3) by first creating five 2D symmetric Gaussian filters with increasing standard deviations σ_1 through σ_5. There are six versions of the original 128x128 image, called the six intervals, where each interval is the result of applying one of the Gaussian filters to the previous interval. Collectively, the six intervals

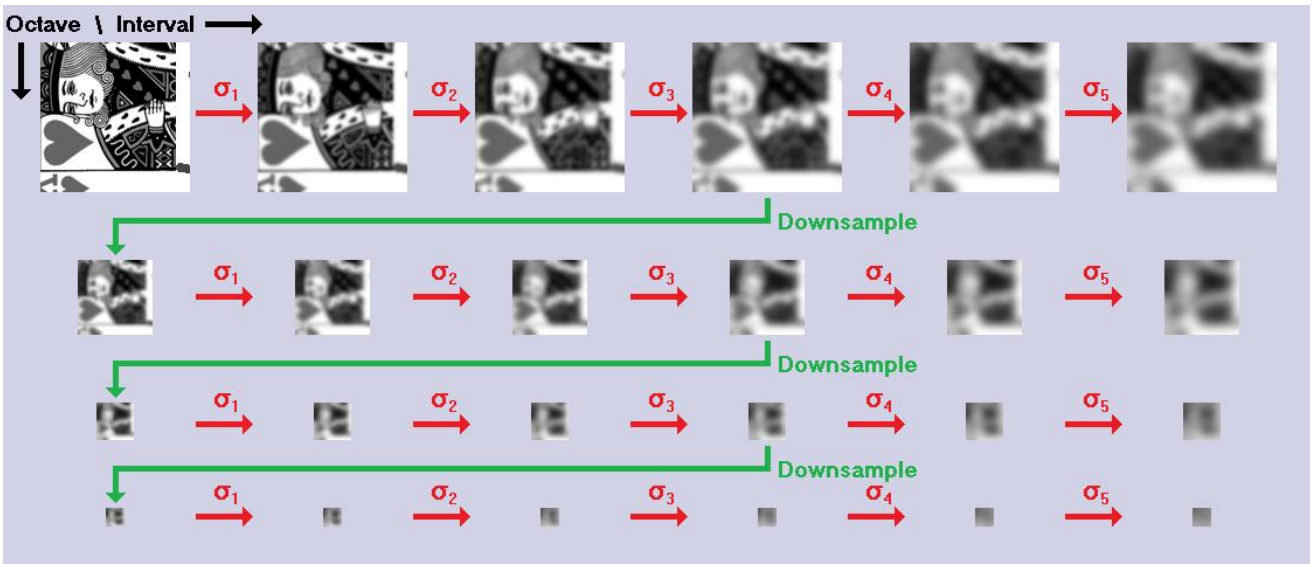

Fig. 3. Gauss Pyramid, in which each row is an octave consisting of six interval images. New intervals are computed by applying Gauss filters, and new octaves are computed by downsampling the fourth interval of the previous octave.

form an octave. The first image of the second octave is the 64x64 image that results from downsampling the fourth image of the previous octave by a factor of 2. This process of applying Gaussian filters and downsampling is repeated to create a Gauss pyramid consisting of four octaves with six intervals each.

Next, we subtract each interval's image from the previous interval's image to obtain a series of Difference of Gauss images. There are 5 such images in each octave, and along with x and y spatial dimensions, the interval of the Difference of Gauss octave serves as a third dimension. Therefore, each octave in the Difference of Gauss pyramid can be thought of as a 3D image. For each octave, we compute the 3D local intensity extrema, where an extremum is defined as being a pixel with either greatest or least intensity among its eight neighbors in a 3x3x3 cube in the three dimensions of x, y, and interval. Each extremum is a feature candidate.

In the keypoint localization step, we narrow down the number of feature candidates. We apply upper and lower bounds on the contrasts of the extrema and perform gradient calculations to eliminate features that are too "edge-like" because they are vulnerable to noise effects, which differ depending on the scale of the image. After these candidates are eliminated, we are able to determine which of the candidates are actually SIFT features of interest.

During the orientation assignment step, the SIFT features are decorated with information about their surrounding environment. This information consists of a scale, which can be informally defined as the size of the feature (which is relative to the size of the object of interest and therefore is resistant to scale transformations), and an orientation, which can be informally defined as a 2D direction of the feature that will remain the same after rotation. The scale of a feature is determined from that feature's octave and interval. The orientation of a feature is computed by examining the 2D spatial neighborhood of the feature within its interval and performing histogram analysis. Collectively, the location, scale, and 2D orientation define the feature.

The final stage of the SIFT algorithm, keypoint descriptors, is the decoration of features with extra information that makes the feature invariant to additional parameters beyond scale and rotation, such as illumination and 3D point of view. Our algorithm does not perform this fourth stage, as our intent to provide the robot with multiple training images compensates for this information and can therefore save computational effort on a per-image basis (although this is not so much an optimization for our image ROM-based system as it is for an actual real-time image-acquiring system).

The SIFT algorithm described above was first implemented as a preliminary MATLAB script. We kept the code at a low level so that we could easily translate it to hardware.

B. Reorganized State-Machine Implementation

In preparation for a VHDL hardware implementation of the SIFT algorithm, we reorganized our preliminary MATLAB code into a second MATLAB script organized into a large state-machine. In hardware, all synchronous (clock-driven) sequential logic can be thought of as a state machine. Registers hold their values and update only on clock ticks. The value that the registers are updated to may be a different function of other signals dependent on the current state of the state machine. By default, registers hold their own value from state to state, since there are only a select few times when each signal needs to change. States transitions also occur upon clock ticks.

To model this behavior in MATLAB, we can consider a register represented by variable R to have a next value

represented by variable R_next. When in a state S, R_next is a function of S and all other register variables, but it defaults to R unless a change is required. After all _next variables have been calculated, a clock tick can be modeled by all registers R (including S) being updated to their respective newly calculated R_next values. The algorithm begins at a beginning state at which image data is received, and the cycle of state transitions is repeated until an end state is reached, at which point the SIFT features have been calculated.

The importance of this intermediate MATLAB script is then the division of computation into states. We partition to each state a computation that can be completed based on the available results of previous state computations, but not too much computation that the resulting hardware design would not be able to meet timing constraints. In a standard processor, no more than one arithmetic operation can be completed per clock cycle, but since we have designed this hardware specifically for our SIFT algorithm implementation, we have grouped in some states several commonly-used arithmetic operations that the SIFT algorithm would benefit from being able to compute quickly.

C. VHDL Hardware Implementation

The final step to creating a hardware description that could be downloaded to the NI robot's FPGA was translating from the state machine software simulation to an actual hardware module. This translation was greatly facilitated by the fact that we had broken the tasks down into ones that had intuitive hardware equivalents.

Additions in software were implemented as adders in hardware using the (+) VHDL operator. Subtractions are likewise simple to implement in hardware using subtracters (really, adders with negated input) via the (–) VHDL operator. Multiplications and divisions were similarly implemented as dedicated arithmetic blocks, although there is no equivalent operator in VHDL (prior to the new EDA VHDL-2008 standards, which were not supported by the LabVIEW HDL node environment). There are bit check equivalents for comparisons such as equality. Absolute values for contrast analysis in Difference-of-Gauss images were implemented by multiplexing the positive and negative values of the variable, conditional on the top bit (because negative numbers start with a 1, and non-negative numbers start with a 0). All fractional numbers required for Gauss filtering, gradient analysis, and, in general, precision for pixel representation, were implemented as 16-bit fixed point numbers with an 8-bit integer component and an 8-bit fractional component. Similar to the setup of the intermediate MATLAB script, registers are updated simultaneously during state transitions at each clock tick. For each register R, a temporary value R_next is calculated in combinational logic depending on the current state, and R is updated to R_next at the next clock tick. By default, R_next is R, unless the state machine is in some state that is specifically calculating or otherwise modifying register R.

IV. Results

A. MATLAB Implementation Results

We have run our SIFT algorithm on a variety of images. We first tested with computer-generated images, two examples of which can be seen at the left of Figure 4. Feature locations are identified by blue circles, and the scale and orientation of each feature is identified by the red arrows (length shows scale, and direction shows orientation). We test the accuracy of our SIFT algorithm, we also downloaded a C implementation of the SIFT algorithm by Rob Hess [7] to verify our results. This implementation uses the OpenCV library. It is therefore more complicated than ours, in that it uses many function calls, imports many libraries and headers from OpenCV, and uses a large variety of different data structures. These characteristics make this C implementation impossible to translate directly into VHDL. We ran this C implementation using the same images that we used with our MATLAB implementation for comparison. The output of the C program can be seen at the right of Figure 4.

There is some difference in certain features from our results. Minor differences can be attributed to round-off in boundary cases or differences in implementations of certain components of the program (like how the Gaussian filter handles edge-of-image behavior). Overall, many of our features are the same. The largest differences can be attributed to simplifying approximations of certain functions such as tangents, exponentials, and square roots that are required to calculate the orientation of SIFT features. Because there is no simple hardware equivalent to these functions, we implemented them as polynomial approximations, which can be calculated as a series of multiplications and additions. Even though the match between features between our algorithm and Rob Hess's algorithm is not perfect, we still label our MATLAB code as "working" because it calculated a set of SIFT-like features that can be used to identify an object in a test image.

We also compared our MATLAB SIFT implementation (functionally equivalent to the VHDL hardware implementation) to Rob Hess's code for images taken from an actual camera, which we had intended to use as test images for locating a tennis ball. Two examples are shown in Figure 5. Our program tends to ignore smaller features resulting from noise more often than the C code does.

B. VHDL implementation Results

After creating a fully synthesizable VHDL hardware description for the SIFT algorithm, we encountered several problems when trying to route it to a Spartan-3 FPGA, which we were required to fix by further modifying the VHDL.

An overall summary of our problems and solutions are shown in Table 1. This table also summarizes the tradeoff of each design decision.

Firstly, the FPGA size was limited to 17280 logic elements and 432kbit of RAM. With this amount of RAM,

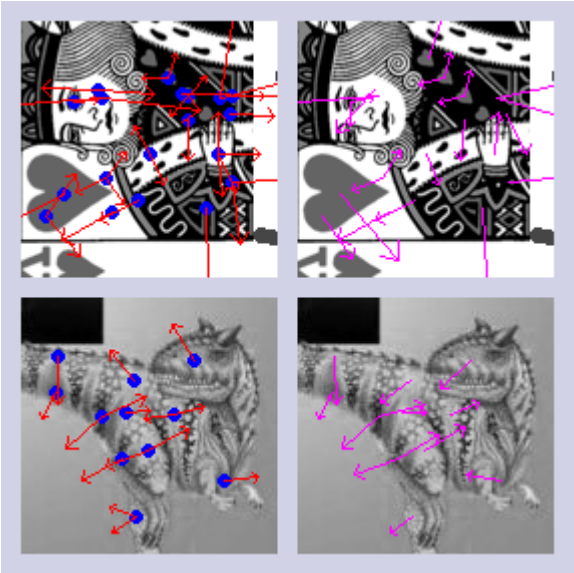

Fig. 4. SIFT features recognized by our MATLAB code (left) and Rob Hess's C/OpenCV code (right) for computer-created images.

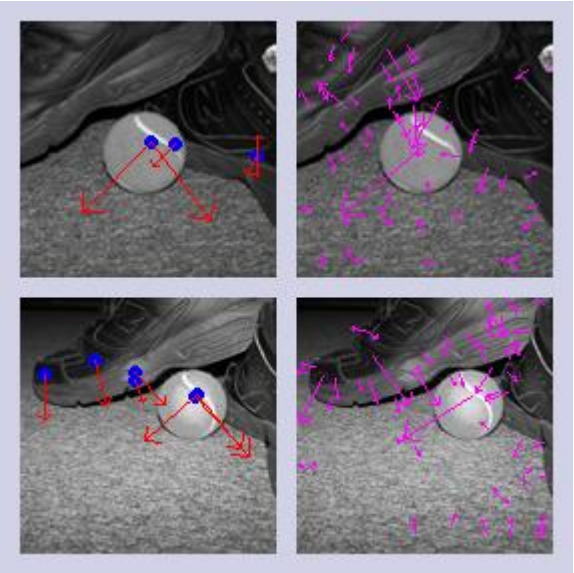

Fig. 5. SIFT features recognized by our MATLAB code (left) and Rob Hess's C/OpenCV code (right) for actual photographs.

we were unable to fit images of size 128x128 pixels directly on the board (which would require 262kbit per image, and there are at least six 128x128 images for the first octave). We solved this problem by reworking our module to handle 32x32 images for the first octave, 16x16 for the second octave, and so on down to the fourth octave. This reduction cut our total RAM usage by a factor of 16, which allowed all images in the Gauss pyramid to barely fit in RAM. However, only one image can fit in RAM at a time, so only one image can be analyzed at a time. If this image is a test image, then its features need to be recorded after the SIFT algorithm completes, and then the image must be flushed from memory. If it is a test image, then it must be the only test image being processed at a time, which prevents the analysis of several 32x32 images simultaneously to simulate a larger image. A second reason why dividing a larger image into 32x32 sub-blocks for serial analysis will not work is because the SIFT algorithm could only find features assuming the 32x32 image is the entire picture, which eliminates the possibility for features derived from larger objects in the image that span multiple 32x32 sub-blocks. Essentially, this situation is the equivalent of having a camera that only captures 32x32 pixel images. Unfortunately, the number of SIFT features that can be extracted from such an image size is often less than five, and so there is a great loss of precision compared to a 128x128 image.

Secondly, we encountered area and timing errors caused by combinational logic in our state machine. The greatest problems arose from our dividers. In hardware, division is a very complicated operation that generally lasts several clock cycles in a fast processor. In the SIFT algorithm, division is required when calculating gradients during the keypoint localization and orientation assignment stages. When trying to fit the design to a Spartan-3 FPGA using Xilinx ISE, the divider modules we used were too large and were unable to complete in the specified clock cycle. Because those divisions are required by the algorithm, the only option was to use different dividers. We switched to using the EDA VHDL-2008 support library's fixed point package, which included a fixed point divider.

This change introduced a third problem. When trying to switch from ISE tools, which could predict synthesis and fitting results for the Spartan-3, to the LabVIEW software, which is required to actually program the FPGA on the NI robot, there were compatibility issues with the transition. Mainly, it is difficult in the LabVIEW software to use multi-file VHDL designs or external libraries. Although there is an option for VHDL modules with external dependencies, there are limitations when the external dependency is non-standard (e.g. not in the ieee library). Because the fixed point package was only added to the ieee library in 2008, the LabVIEW software did not support it by default. When trying to link to the external files containing the fixed point code, the software was unable to recognize the division routines. There was no solution for this problem to get the fixed point package dividers working. The only alternative would be to redesign the dividers to be pipelined and take multiple clock cycles, and that would introduce a larger RAM use in a system that is already memory-constrained.

Finally, there was the issue of the LabVIEW compiler not treating combinational logic the same way as the Xilinx ISE compiler. In hardware, combinational logic is expected to compute continuously. However, because of the interactions between the Spartan-3 FPGA and the rest of the sbRIO-9631 system on the NI robot, inputs to the FPGA are only toggled every so often at the frequency of the rest of the NI robot, and the outputs are only read periodically in the same manner. This changes the nature of

TABLE I
ISSUES AND SOLUTIONS RELATING TO VHDL IMPLEMENTATION

Issue	Solution	Tradeoff
128x128 image uses too much RAM	Use 32x32 images	Resolution, number of extractable features
Custom divider module requires too much area/time	Use EDA VHDL-2008 fixed point library	VHDL module depends on external package for division
External package not recognized by LabVIEW	Proposed: multi-stage (pipelined) divider module	RAM overheads, must redesign a new divider module
LabVIEW HDL node executes combinational logic sequentially	Place within single-cycle loop	Must meet LabVIEW input/output protocol constraints
General lack of available memory for image-intensive design	Proposed: coarser parallelization, system-level hardware design	Limits benefits of parallelization

combinational logic into something that is more sequential. The documentation with regard to this matter has stated that the best use for inserted HDL is for small modules that do not use internal registers, and that the behavior should differ depending on whether the HDL node is located schematically within a single-cycle loop in the larger design of the robot system. A hardware implementation of the SIFT algorithm clearly requires internal registers and multiple stages to implement a state machine. The solution to this issue is to redesign the VHDL to conform to the input and output requirements of the inserted HDL component, including specifically selecting to allow single-cycle timed loops (which is not the default option) and then controlling internal registers with the LabVIEW-generated enable_in signal and controlling the enable_out signal to lag the enable_in signal until the computations in the state machine are complete.

V. CONCLUSION

Our results have revealed that trying to implement a full-hardware SIFT algorithm on the NI robot introduces several problems caused by the system's memory, area, and timing limits. The board has only enough RAM to analyze one input image at a time, and this image must be sized at approximately 32x32 pixels, which drastically limits the resolution and number of extractable SIFT features. It is especially difficult to implement fixed-point division on the FPGA of the NI robot, and inputs and outputs of the FPGA must be carefully regulated to meet the timing of the external system. Our results lead us to conclude that a full-hardware implementation of SIFT is not well-suited for the NI robot platform.

Previous implementations of the SIFT algorithm have used powerful serial processors to compute and match features on large images with dimensions of a few hundred pixels. If the algorithm should be parallelized, the parallelization should be at a coarser level than a complete VHDL implementation of the algorithm. Below we provide our suggestions for future research in this field, which includes several alternatives that would be better-suited for SIFT parallelization, and suggestions for better use of the NI robot platform.

Multiprocessors: The SIFT algorithm has already been implemented in software on powerful serial processors. The algorithm could be parallelized by having several processing elements simply divide up the instructions that could be considered independent of one another. Programming for parallel implementation with pthreads, OpenMP, or OpenMPI could divide the Gauss filters and per-feature analysis equally amongst processors while keeping the level of abstraction the same as in previous implementations of SIFT. There would be a cost overhead from using several large processing elements.

Parallelized processors: These include Very Large Instruction Word (VLIW) processors, DSP processors, and GPUs. Each of these options is an SIMD architecture designed to perform the same operation on multiple pieces of data simultaneously. They are especially optimized for performing digital filters such as the ones required to build the Difference-of-Gauss pyramid in the scale-space extrema detection portion of the SIFT algorithm. Each processing element of any of these architectures is generally cheaper than a single processing element in a standard multiprocessor, but there is less support for compilers for these architectures.

Microcontrollers: A microcontroller is also capable of executing compiled C code, and like larger processors, it has the benefit of built-in memory. Accessing memory will not be as quick as on a full-hardware FPGA because of the serial nature of access, but several microcontrollers working in parallel could speed up the process. It would be relatively easy to hook up a camera to a multi-microcontroller system compared to the NI robot.

FPGA with system-level programming: It would still be possible to take advantage of the FPGA's full-hardware parallelization even with certain sequential segments of the SIFT algorithm. A memory controller would allow access to an FPGA board's on-chip SDRAM so that images need not be stored in full hardware in the SRAM. An FPGA that has been programmed at a system level would bypass the issues of having to write relatively low-level modules like dividers. Se, Ng, Jasiobedzki, and Moyung [8] have used an FPGA programmed with System Generator to successfully speed up some of the vision processing functions in the SIFT algorithm for use on a planetary rover. The speedup was approximately 10x compared to a Pentium III 700MHz processor implementation, but the implementation was not full-hardware and still required a separate processor for

most computation aside from the vision functions. It is actually possible to program a small embedded processor onto an FPGA (for example, a Leon3 or Nios II processor) that could be modified to run the main program of the robot while exporting parallelizable operations such as the Gauss filters to the rest of the FPGA for fast hardware implementation. These processors are already configured to have memory handlers and dividers, so they are preferable to a full-hardware implementation.

Suggestions for future uses of the NI robot are as follows. The platform is most suited for use of the mounted ultrasound sensor. Based on the difficulties we had planning to attach an external camera to the robot, we conclude that it is not well-suited for algorithms that require image-based processing. Based on the difficulties interfacing between the on-board FPGA, the SDRAM, and the rest of the sbRIO-9631, we conclude that the FPGA should be limited to speeding up small operations that could execute within one or two clock cycles and could benefit from hardware parallelization. Using the FPGA for larger hardware designs with many internal registers is not recommended. Trying to access SDRAM and other external memory from within the NI robot's FPGA is also not recommended. The primary programming method for the NI robot should be through LabVIEW software, and not through VHDL hardware modules.

REFERENCES

[1] D. G. Lowe, "Distinctive Image Features from Scale-Invariant Keypoints," *International Journal of Computer Vision*, vol. 60, no. 2, pp. 91–110, November 2004.

[2] S. Se, D. Lowe, and J. Little, "Vision-based Mobile Robot Localization And Mapping using Scale-Invariant Features," *The International Journal of Robotics Research*, vol. 21, no. 8, pp. 735–758, August 2002.

[3] C. Valgren and A. Lilienthal, "SIFT, SURF and Seasons: Long-term Outdoor Localization Using Local Features," in *Proc. European Conference on Mobile Robots (ECMR)*, Freiburg, 2007, pp. 253–258.

[4] D. G. Lowe.

[5] R. Hess, "SIFT Feature Detector," unpublished. Retrieved February 15, 2010 from http://web.engr.oregonstate.edu/~hess/.

[6] D. G. Lowe.

[7] R. Hess.

[8] S. Se, H.-K. Ng, P. Jasiobedzki, and T.-J. Moyung, "Vision Based Modeling and Localization for Planetary Exploration Rovers," in *55th International Astronautical Congress*, Vancouver, 2004, pp. 1–11.