# FPGA Implementation of an Optic Flow Sensor using the Image Interpolation Algorithm

**M.A.Garratt A.J.Lambert T.Guillemette**
University of New South Wales, Australia
m.garratt@adfa.edu.au

## Abstract

A small embedded vision sensor has been developed using Field Programmable Gate Arrays (FPGA) technology. The sensor computes optic flow using a streaming version of the image interpolation algorithm to provide flow vectors at 36 equally spaced locations in a 1024×768 pixel image. The sensor has been tested in flight on an unmanned helicopter and benchmarked against GPS measurements of velocity. A small form factor (70mm×55mm) and low-weight for the sensor makes it ideal for implementation on a small flight vehicle.

## 1 Introduction

FPGAs are integrated circuits containing numerous logic gates, which can be reconfigured through software to form any digital processing network desired. Due to their internal architecture, FPGA chips can process data in a massively parallel way rather than the sequential fashion normally present in computers, enabling real-time processing of high-bandwidth visual information. The convenient reprogrammable feature of FPGA makes them an excellent platform for development work. Modern FPGA may have millions of logic gates and contain dedicated elements for random access memory, multipliers, dividers etc. Intellectual property (IP) cores are available which provide templates for ready-to-use high level components such as microprocessors. Some FPGAs are available with DSP cores already built into them allowing a hybrid design based on combining easy to debug sequential processing with hardware processing. The usability of FPGA has increased significantly since the availability of graphics based tools such as Altium Designer which allow complicated digital circuits to be designed both in schematic form using a drag and drop approach, and with hardware description languages. Components based on IP cores can be assembled together with user made blocks based on logic elements or low level hardware descriptor languages. Designs are automatically compiled by the software into low-level logic formats and then downloaded into the FPGA.

In this application the restrictions of size, weight and power consumption are imposed by the need to fly on a micro-air vehicle (MAV) with minimal disruption to the aerodynamics of the platform. Furthermore the use of video information for guidance requires high speed manipulation of pixel data, hence a streaming architecture that minimises use of memory is desired. A FPGA is engineered to just perform the necessary functions, and doesn't include auxiliary or peripheral functions that are superfluous, as might be the case for a more general purpose processor solution. Finally the input-output flexibility on most of the FPGA pins means we can easily interface directly with streams of parallel pixel data from image sensors and supporting hardware.

We have determined that it would be possible to implement a complete autopilot comprising control and sensor fusion on the same FPGA which would run the vision processing algorithms. This enables a very low weight autonomy system to be realized.

## 2 Related Work

A number of researchers have proposed using custom made integrated circuits as autopilot devices with optic flow sensing built-in [Barrows,2000]. Ultimately, this approach would lead to a lightweight system, but the expense and long turn around time associated with developing integrated circuits is prohibitive in the short term. FPGAs provide an apparatus that can be used to rapidly prototype vision processing algorithms which can be later implemented on ASICs if mass-production is required.

Some other groups have attempted optic flow on FPGA. A biologically inspired FPGA optic flow sensor is described in [Aubépart,2007] based on a model of the *Elementary Motion Detector* (EMD) neuron used in an insect for computing flow. This design was able to measure 1D optic flow using a linear array of 12 photorecep-

tors. In [Arribas,2003] an implementation of the *Horn and Shunk* optic flow algorithm [Horn and Schunk,1979] is implemented on a Xilinx FPGA. In this work, two FPGAs working in parallel were used to process images of 50×50 pixels at up to 19 frames per second (fps). In [Martín,2005], another implementation of the Horn and Shunk optic flow algorithm is described which is capable of generating optic flow on a 256×256 pixel image at 60 FPS. In [Díaz,2006], an FPGA implementation of the *Lucas and Kanade* gradient optical flow algorithm [Lucas and Kanade, 1984] is described which shows results for images of 320×240 pixels and 30 fps. A scheme is described in [Wei et al, 2007] for calculating optic flow at 640×480 pixel images at 64 fps using a tensor based optic flow algorithm. In [Brown et al,2008], results for a gradient based scheme for calculating optic flow are provided but a frame rate of only 2 fps is achieved with a 120×120 pixel image.

We believe our work is the first attempt to implement the Image Interpolation Algorithm [Srinivasan,1994] for calculating optic flow on a Field Programmable Gate Array. The Image Interpolation algorithm, abbreviated $I^2A$, has many advantages owing to its simplicity. The technique is non-iterative, does not require identification or tracking of individual features in the image, and does not require the calculation of high order spatial or temporal derivatives. These features make it robust to noise and quick to execute. Of all the techniques reviewed, these properties make $I^2A$ most suited to use for flight control. We also believe this work to be the first presentation of flight test data using an FPGA based optic flow sensor.

## 3 The Image Interpolation Algorithm

Various versions of the $I^2A$ exist for combinations of rotations, shear and translation. However, for small rotations between frames, it is possible to only treat the cases of translation as the other image transformations can be reconstructed based on the distribution of vectors in the flow field. A simple version of the algorithm is therefore presented which calculates the flow due to lateral and longitudinal translation only. A detailed derivation of the $I^2A$ algorithms can be found in [Srinivasan,1994].

The algorithm compares the current image with a set of four reference images which are translated from a previous frame. We define the pixel intensity functions at times $t_0$ and $t$ as $f_0(x, y)$ and $f(x, y)$ respectively where $x$ and $y$ are the image coordinates measured in pixels. The reference images $f_1$, $f_2$, $f_3$ and $f_4$ are formed from the first image by shifting them by reference shifts $\Delta x_{ref}$ and $\Delta y_{ref}$ pixels along the x and y axes, so that Equation (1) applies.

$$\begin{aligned} f_1(x,y) &= f_0(x+\Delta x_{ref}, y) \\ f_2(x,y) &= f_0(x-\Delta x_{ref}, y) \\ f_3(x,y) &= f_0(x, y+\Delta y_{ref}) \\ f_4(x,y) &= f_0(x, y-\Delta y_{ref}) \end{aligned} \tag{1}$$

The $I^2A$ algorithm relies on the assumption that the image at time $t$ can be linearly interpolated from $f_0$ and the reference images. With this assumption, the pixel coordinate translations $\Delta x$ and $\Delta y$ can be expressed as in Equation (2).

$$\hat{f} = f_0 + 0.5\left(\frac{\Delta x}{\Delta x_{ref}}\right)(f_2 - f_1) + 0.5\left(\frac{\Delta y}{\Delta y_{ref}}\right)(f_4 - f_3) \tag{2}$$

The objective is to calculate the translations $\Delta x$ and $\Delta y$ which give the best match between the actual image $f$ and the interpolated approximation $\hat{f}$. We can express this as a least squares problem over an image patch defined by the window function $\Psi$ by minimising the error $E$ where $E$ is defined by Equation (3). The two dimensional window function $\Psi$ acts to localize each motion computation. We have used a simple boxcar kernel as a window function.

$$E = \int\int \Psi \cdot [f - \hat{f}]^2 \, dx \, dy \tag{3}$$

The error may be minimised by taking the partial derivatives of $E$ with respect to $\Delta x$ and $\Delta y$ and setting them to zero. After simplification, the two resulting linear equations are at (4-5). After evaluating the integral terms to scalars, the two simultaneous equations are solved for $\Delta x$ and $\Delta y$ using trivial linear algebra (i.e. by inverting a 2x2 matrix).

$$\begin{aligned} \left(\frac{\Delta x}{\Delta x_{ref}}\right)\int\int \Psi \cdot (f_2 - f_1)^2 \, dx \, dy + \\ \left(\frac{\Delta y}{\Delta y_{ref}}\right)\int\int \Psi \cdot (f_4 - f_3)(f_2 - f_1) \, dx \, dy \\ = 2\int\int \Psi \cdot (f - f_0)(f_2 - f_1) \, dx \, dy \end{aligned} \tag{4}$$

$$\begin{aligned} \left(\frac{\Delta x}{\Delta x_{ref}}\right)\int\int \Psi \cdot (f_2 - f_1)(f_4 - f_3) \, dx \, dy + \\ \left(\frac{\Delta y}{\Delta y_{ref}}\right)\int\int \Psi \cdot (f_4 - f_3)^2 \, dx \, dy \\ = 2\int\int \Psi \cdot (f - f_0)(f_4 - f_3) \, dx \, dy \end{aligned} \tag{5}$$

## 4 Hardware Implementation

The FPGA used on the board is a Xilinx Spartan-3E. As the design requires a powerful, yet small FPGA to create a small and lightweight board, the Spartan-3E is the best choice within the low-cost Spartan family. It is a logic optimized version of the Spartan-3, designed for applications where logic densities matter more than input/output (I/O) pin count.

The video sensor used is a MT9M001 from Micron. It is a CMOS digital image sensor, with an output resolution up to 1280×1024 (at 30 frames per second), and an ADC resolution of 10-bit on-chip. It is well adapted for optic flow computation since we need a greyscale image with a rather high ADC resolution. The pixels are scanned from the CMOS array, clocked by an external 50 MHz clock. It is important to note that the sensor also outputs a pixel clock (of same frequency as the master clock), synchronising valid pixel data output with its falling edge. After each line and after each image, there is a blanking period during which nothing is output. The blanking periods are used for synchronization of a monitor (to be used for debugging). The two synchronization signals LINE_VALID and FRAME_VALID allow us to know whether we are receiving valid or blanking pixels. Using these signals and knowing the resolution of the image, we can easily create a counter that is going to give us the position of the pixel being sent by the sensor (by giving its row and column on the screen). This sensor has also a very important feature, which is an $I^2C$ interface bus. It allows real time control of the sensor by the FPGA, on parameters such as resolution, exposure, gain and blanking periods.

The streaming paradigm [Bowman, 2008] employed in this design seeks to remove the need for random access to pixel data in a frame, rather creating delayed streams of pixel data using circular buffers or first-in first-out (FIFO) memories. In this way computation nodes are presented with current, pixel or line or frame delayed streams of pixels. This has a number of advantages - the first meaning that DMA memory access engines are not required, the second meaning a new brand of self-refreshing dynamic RAM memories may be used which reduce power consumption over technologies such as static RAM, and the FIFO nature does not require the extra address line connections as would be the case for random-access memory. The dual-ported and hence independent read and write operations of the FIFO, may be conducted at the fast access times required for the pixel data.

Small FIFOs are created within the FPGA using the block and distributed memory, providing line and pixel delays to the streams, while large FIFOs for full frame delays are external to the FPGA. The FIFOs used on the design are the MS81V32322 1100k×32 bit word memory from OKI Semiconductor, which have been arranged as 2200k×16 bits by feeding the upper two bytes of the output, back to the lower two bytes of the input. These lower two bytes at the output are then provided back to the FPGA. Two such separate configurations are used in our design, but these are easily wired together within the FPGA to double the length FIFO, resulting in a maximum of 4M×16 bit words. The image sensor provides 10 bits per pixel, so we have the option to pad these to 16 bits per access, or truncate to 8 bits and again double the effective length of the FIFO. We decided to compute the algorithm on a 1024×768 pixel image, which involves storing 1268×784 pixels (including the vertical and horizontal blanking) in the FIFO, requiring a memory depth of 1268×784 = 994k words, or slightly less with more sophisticated clock manipulation of the vertical blanking period. Each word hosts a pixel of depth 10 bits.

The FIFO access time of 6 ns, allows more than one set of pixel data to be interleaved at word rates of 150 MHz, but our current design exercises these at the pixel clock of 50 MHz. Read and Write clocks may be different rates providing for burst processing, but again this is currently not utilised. The FIFO internal addressing can be reset at any stage for either read or write addressing, and allows for inclusion of a software defined starting address. In this way we can create a circular frame buffer that can be a full frame or multiple lines in length. The former results in a stream of pixels that is one frame delayed to that coming from the sensor. The size of the image to process can be adapted; however we wish to have a rather high resolution for more accuracy.

The video digital-to-analog converter (DAC) generates VGA signals displaying the image sensor output with additional debugging information superimposed. The DAC used on this design is the ADV7123 from Analog Devices. It consists of three high-speed, 10-bit, video D/A converters with complementary outputs, a standard TTL input interface and a high impedance, analog output current source. The video signal resolution is set by $I^2C$ on the video sensor, and some rules have to be respected since the frame rate depends on the resolution, and a monitor has restricted locking range (in resolution and frequency). For this project, we have chosen a resolution of 1024×768, which leads to a frame rate of 50 Hz. For testing on our larger helicopters, the VGA output can be converted to PAL or NTSC composite video, so that the image can be monitored on the ground using a 2.4 GHz downlink and receiver.

The configuration data for the boot of the FPGA is contained in a PROM, but during debugging a JTAG interface is used for reconfiguration. When powering up the board, the configuration file will be loaded into the FPGA. The PROM used in this design is the XCF04S from Xilinx.

The main schematic, illustrated in figure 1 shows the architecture of the optic flow sensor. It contains the following subsystems:

- The *Clock Unit* receives the master clock from the FPGA (which receives a clock signal from a 50 MHz crystal oscillator). It then generates different clocks for the other modules: a slow 400 kHz clock for the $I^2C$ module, and a 5 MHz clock for the RS232 module. It also uses the pixel clock (PIXCLK) from the video sensor to generate via a DCM a 90° phase-shifted version. If need arises, a locked harmonic or slightly different frequency may be created using the DCM. Such would prove useful if electromagnetic interference from the FPGA fouls other radio navigation signals.
- The *Grid Unit* generates the coordinates of the current pixel from the pixel clock and the synchronization signals (horizontal and vertical: HREF and VREF). Other units use these coordinates to know where the current pixel is located in the frame.
- The *External FIFOs Control Unit* controls the write and read resets of the external FIFOs. By cascading two of the FIFOs and resetting the write and read pointers in the middle of the frame, we create a one-frame delay used to feed the algorithm. As the clocks are programmable we can facilitate scaling the FIFO for a change in region of interest of the sensor.
- The *Averaging Filter Unit* is a filter that may perform a moving average on the image. It is used to blur the image, as a pre-processing before calculating optic flow. Of course, similar effects can be obtained optically prior to the sensor with an appropriate aperture to the lens, or within the capability of the sensor programming, but aperture control of exposure can now be decoupled from the optical resolution, and the length of the moving average filter may be altered in soft- or firmware.
- The *Optic Flow Pixels Unit* outputs on every clock event, the pixel operands as streams that are needed by the algorithm for the computation. From the current pixel and the pixel received exactly one frame before, it creates delays to send simultaneously $f$, $f_0$, $f_1$, $f_2$, $f_3$ and $f_4$ to the processing unit.
- The *Optic Flow Processing Unit* receives the pixel streams from the Optic Flow Pixels unit and computes the optic flow $I^2A$ algorithm. It then outputs two 8-bit bytes, X_OF and Y_OF, corresponding respectively to the value of optic flow on the X-axis and Y-axis, for subsequent patches of the image.
- A *Transmit Unit* is connected to a dual port memory implemented as an IP core within the FPGA. Its function is to save the computed values in the memory to read them out at a much slower speed to fit the data rate of the RS232 standard (in our case, 115,200 baud). There are alternative, faster serial links in the design for future connectivity with other systems.
- The *Communication Unit* manages the $I^2C$ and RS232 communications.

### 4.1 Optic Flow Processing Unit

To compute optic flow, two steps are necessary. They are represented by two different blocks in figure 1: one block receives the pixel stream coming out from the video sensor and the pixel stream sent by the video sensor one frame earlier, and outputs the operands $f$, $f_0$, $f_1$, $f_2$, $f_3$, $f_4$ used to feed the algorithm; and the second block contains all the processing units to calculate the 2D optic flow vectors. The two blocks refine to lower level blocks detailed below.

The computation of optic flow requires six different pixel streams at the same time: the current pixel on which we want to compute optic flow, the one that was sent one frame before, and four pixels that correspond to shifted versions of this same previous frame. The current pixel $f$ is obviously obtained at the output of the video sensor, the one delayed by one frame $f_0$ is retrieved at the output of the external FIFO creating the one frame delay, and the four pixels corresponding to shifted version of $f_0$ are simply obtained by creating delays by implementing FIFOs in the FPGA. However, there is a problem by doing so: two of those pixels are outputted after $f_0$. So, if we want to use only one big external FIFO, optic flow will not be computed exactly on the pixel coming out of the video sensor, but on one that has be outputted earlier. The shift reference has been set to 8 pixels, so when receiving a pixel, the optic flow will be calculated on the pixel that was received 8 lines and 8 pixels before. This way, we just need to create three 8 pixels delay and two 8 lines delay after the one frame delay to obtain all the pixels. The latency for optic flow vectors from the current pixel stream is eight lines and eight pixels.

The delays are created by FIFOs, which can be external or implemented in the FPGA. The 8 pixel delay FIFOs have been coded in VHDL since Xilinx do not provide IP cores for FIFOs of this size. The 8 lines delay are created using internal block RAM that are managed like FIFOs by some VHDL code, because we need 10,144 words of storage for 8 lines (the blanking pixels have to be included, so a line is actually 1268 pixels wide) and the available IP cores sport power of two sizes and hence are unsuitable. We then configure read-before-write mode, and the address is incremented on every rising edge of the pixel clock, and reset to zero when the

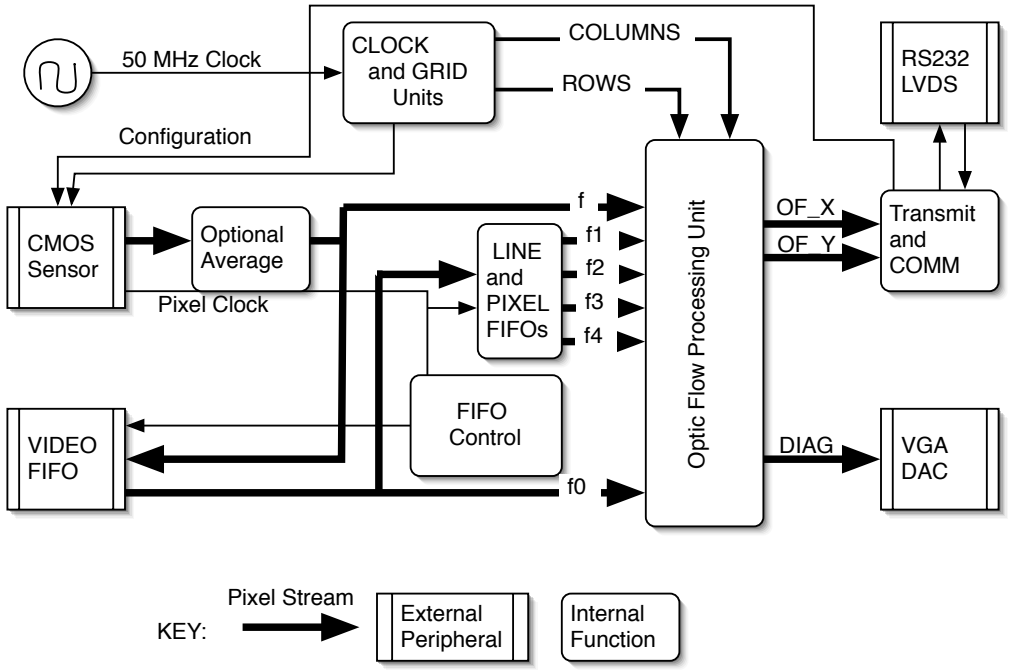


Figure 1: Block diagram of optic flow sensor architecture.

address reaches 10,143.

All six pixel streams exit the first block to be used for computation. It is the most consuming part of the design, since it takes most of the resources of the FPGA. The calculated optic flow values could be positive or negative, depending on the moving direction. For that reason, we first need to add a sign bit and use two's complement notation. The video sensor has an ADC resolution of 10 bits, but 8 bits are used to code the gray value of a pixel (from 0 to 255). This results in some loss of precision but this is still good enough to provide reliable flow vectors in most circumstances. A larger FPGA may accommodate the full 10 bit words.

The first step of the computation is to have the result of the following three subtractions: $f - f_0$, $f_2 - f_1$ and $f_4 - f_3$. To do so, we first send $f$, $f_0$, $f_1$, $f_2$, $f_3$, $f_4$ into a block that is going to sign-extend their size to 9 bits. Instead of subtracting those numbers, the two's complements of $f_0$, $f_1$ and $f_3$ are taken. Thus, the subtractors can be replaced by adders to obtain the same result. The second step involves five multipliers (the multipliers are all using the dedicated multipliers present in the FPGA), using two 9 bits inputs and outputting a 18 bit number, to give: $(f_2 - f_1)^2$, $(f_4 - f_3)^2$, $(f - f_0)(f_2 - f_1)$, $(f - f_0)(f_4 - f_3)$ and $(f_2 - f_1)(f_4 - f_3)$.

Now, the integral over x and y has to be performed, i.e. the five multiplications have to be summed over a window. We want to compute optic flow on a certain number of patches regularly spaced on the image, since there is a lot more interest in having localized values of optic flow than a global value for an image (if we want to determine where an obstacle is located to avoid it). The number of patches is an important parameter, and another one is their size. They have to be big enough to give more accurate results by averaging the values over an area, but not too big so that they obscure detail in the image such as small obstacles. Simulations done with MATLAB have shown that a 4×4 matrix of patches start being very satisfactory, and the best size for each patch is 64×64 pixels even if results are not so different with values slightly smaller, like 48×48. With the resources of the Spartan-3E, it has been possible to calculate the optic flow on 6×6 patches of 64×64 pixels each.

Considering how the pixels are coming out from the video sensor, when in the region of the first patch, we will sum over the first 64 pixels of the 64 lines of this patch and then move to the second patch, and so on. Then, the first patch will be completed after incorporating its last line of 64 pixels. It means that when calculating the optic flow for one patch, we have to sum the values obtained on its first line, store them, and re-access them later when starting its second line. And of course, the storage has to be done for the six patches. We just need to store six values, because when a line of patches is complete, the stored numbers don't need to be kept and we can reuse the same variables to store the values of the next line.

We have to store five numbers for each of the six

patches, which are coded on 30 bits (to match the size of the biggest number, 255×64×64). However, because of resource limitations in slices and multipliers, and that the next step of the processing is another multiplication, we have to cut the LSBs to reduce the 30 bits numbers to 18 bits numbers. This suits the dedicated multipliers. Once the sum has been calculated, we have for every patch:

$$\begin{aligned} A &= \Sigma(f_2 - f_1)^2 \\ B &= \Sigma(f_4 - f_3)(f_2 - f_1) \\ C &= \Sigma(f - f_0)(f_2 - f_1) \\ D &= \Sigma(f_4 - f_3)^2 \\ E &= \Sigma(f - f_0)(f_4 - f_3) \end{aligned} \quad (6)$$

Then, other multiplications have to be calculated between 18 bits numbers, resulting in these 36 bits numbers: $C \times D$, $B \times E$, $A \times E$, $C \times B$, $A \times D$, $B \times B$. Then, we just need to do three subtractions to obtain: $CD - BE$, $AE - CB$, $AD - B^2$. Since the results are coded on 37 bits, including a sign bit, and the dividers created with the Xilinx CORE Generator can take up to 32 bits in input, we need to drop 5 bits to both the dividend and divisor. It is done by a VHDL block, which also computes the multiplication by 16 (two times the shift reference) and multiply by 8 (to 39 bits obtaining an 1/8th of a pixel precision: the algorithm will give results ranging from -64 to +64, with 8 corresponding to a 1 pixel shift). To do so, the dividend is arithmetic shifted right by 5 bits (the arithmetic shift keeps the sign bit), and the divisor by 5+4+3 bits (4 corresponds to the multiplication by 16, and 3 corresponds to the multiplication by 8). Finally, we divide a 32 bit number by a 25 bit number, resulting in a 32 bit number corresponding to this operation:

$$\Delta x = 2\Delta x_{ref} \frac{CD - BE}{AD - B^2} \quad (7)$$

$$\Delta y = 2\Delta y_{ref} \frac{AE - CB}{AD - B^2} \quad (8)$$

The result is given in 32 bits, but the result will at maximum represent an 8 pixel shift, which means that the maximum result will be 64 in absolute value (since we multiply by 8 to have a 1/8th pixel precision). This precision has been chosen because it has to be a multiple of two (so it can be obtained by simply shifting the result), and a higher pixel precision would require more than one byte (8 bits) per flow vector component to be transmitted.

The processing block outputs the x and y axis optic flow results for 6×6 patches of 64×64 pixels. It also outputs an image for display of the patch locations overlaid on the camera image (see figure 3). The block uses almost half of the slices of the FPGA, because as described before, to compute 2D optic flow on 6×6 patches, 5×6=30 numbers coded on 30 bits have to be saved using slices. A solution to this problem would be to avoid the use of variables, and buffer using block RAM. The rest of the design (without the moving average filter) is using 16 of the 20 available block RAMs in the Spartan 3E FPGA. Better management of partitioning for the available block RAM will reduce this number. Storing the variables in block RAM instead of slices would require 5 more block RAMs, while without this partitioning only 4 are available.

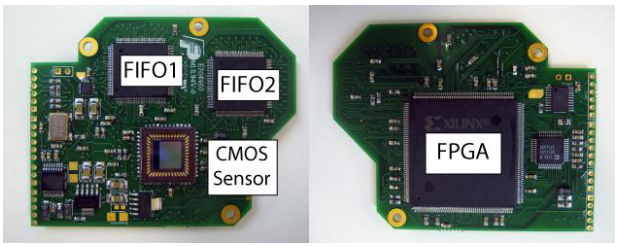


Figure 2: Printed circuit board design. FIFOs and camera sensor (left). Spartan III FPGA on opposite side of circuit board (right).

Once the optic flow has been computed, the flow vectors must be sent to a computer for use in a control loop or to be logged. For 6×6 patches, 72 results are obtained for each frame (optic flow values on x and y for 36 patches), which means that 72 results are calculated and have to be sent every 20 ms (since the frame rate is 50 Hz for a 1024×768 image resolution).

### 4.2 Printed Circuit Board

The final working PCB design comprises a six-layer 70×55mm board. In this design, the routing has been complicated by the two FIFOs, which both have 128 pins, all of which are required for this design. However, use of six layers has made this possible. The FPGA is placed on one side with the DAC and the PROM, while the video sensor is on the other side with the two FIFOs, the RS232 transceiver, and the power circuit. A lightweight box provides EMI shielding, mounting to the optical lens, and so the whole unit with connectors weighs 140 grams. The final printed circuit board design is shown in figure 2.

## 5 Calibration

The treadmill arrangement shown in figure 3 was constructed to enable indoor testing of the sensor. An electrically driven pair of rollers enables a belt to move horizontally under the sensor to simulate translational mo-

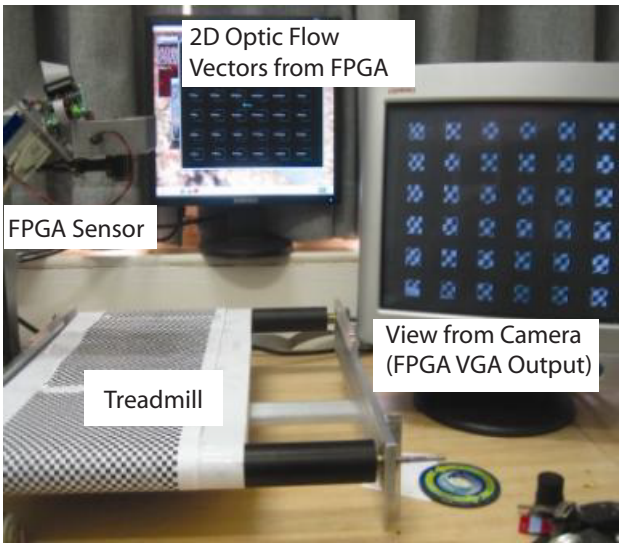


Figure 3: Treadmill apparatus used for verifying and calibrating FPGA flow sensor.

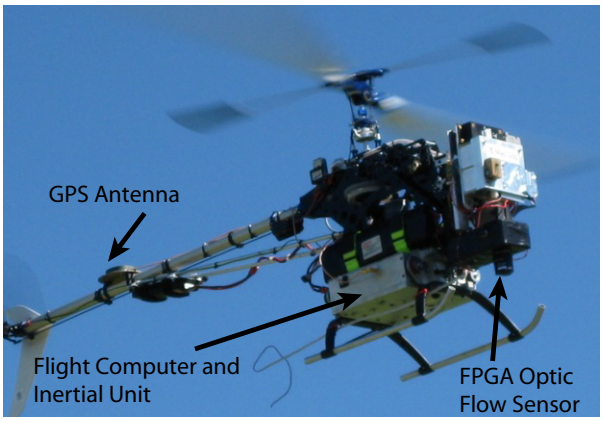


Figure 4: Eagle helicopter used for flight tests.

tion. The speed of the belt can be varied by changing the voltage supplied to the motor. Various textures can be simulated by changing the pictures affixed to the belt. By comparing the magnitude of the optic flow output from the sensor with the known speed of the belt, it is possible to determine a scale factor which when applied to the sensor output yields an optic flow in radians per second, rather than pixel shift per frame.

## 6 Flight Test

The sensor was fitted to an 8kg Hirobo 'Eagle' electric helicopter shown in figure 4. The Eagle helicopter is a test bed for control and sensor research at UNSW@ADFA. The Eagle avionics comprises an in-house designed Inertial Measurement Unit, MPC555 microcontroller based autopilot, PC104 flight computer, NovAtel GPS, Honeywell magnetometer and a bluetooth modem connection to a ground control station. The Eagle platform is described in detail in [Garratt et al, 2006].

The prototype optic flow sensor was fitted to the front of the Eagle helicopter in a downwards looking orientation. The sensor was interfaced to the flight computer using RS232 communications. The sensor was configured to transmit the average lateral and longitudinal optical flow components calculated at 36 equally spaced points in the image to the flight computer. The optic flow data was transmitted to the flight computer at 50 fps. The average of the flow vectors for each frame was recorded with synchronized GPS and inertial data using the flight computer logging system.

For the purposes of this experiment, highly accurate carrier phase DGPS measurements were available to provide a benchmark. We used a NovAtel OEM4-G2L GPS card which was mounted adjacent to the flight computer. The OEM4 card was used with differential corrections from a nearby base station fitted with another NovAtel OEM4 card. In the real-time kinematic positioning mode, the card provides us with 20Hz position and velocity with an accuracy of 1-2cm Circular Error Probability (CEP).

The Eagle was flown backwards and forwards at up to 4m/s at approximately constant altitude. The fused GPS/inertial longitudinal velocity and the average longitudinal optic flow for all flow vectors was recorded. The results of the flight test are shown in figure 5. In these results, the optic flow was multiplied by a scale factor to match it to the GPS velocity. The scale factor depends upon the height of the helicopter above the ground and the angular resolutiuon of each pixel. The scale factor can be determined from calibration using the treadmill arrangement. The results show a clear correlation between the optic flow due to the translation of the helicopter and the measured GPS velocity. Discrepancies can be attributed to a number of factors which include the unevenness and slope of the ground and the variation in height above ground during the flight.

Optic flow is created by both translations and rotations (pitch, roll and yaw). An additional source of error therefore arises from not correcting for the pitch rate of the helicopter as the helicopter has to be pitched to change from backwards to forwards motion. If the optic flow sensor was to be used for flight control, the rate gyroscopes in the helicopter can be used to correct for the effects of rotation rates. This has been done in previous experiments where optic flow has been used for terrain following [Garratt and Chahl, 2007] and control of hover [Garratt and Chahl, 2008].

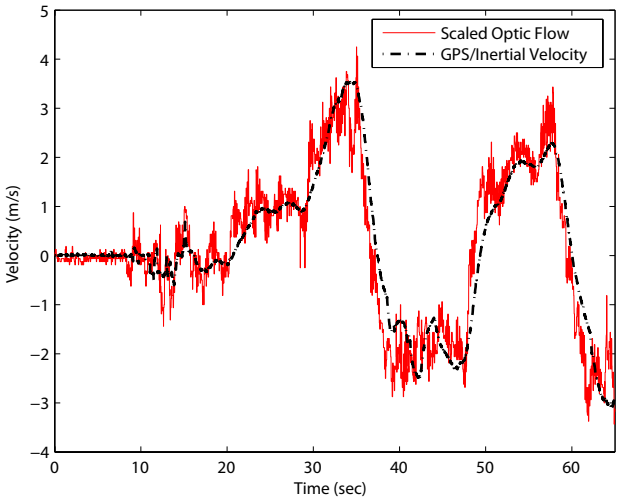


Figure 5: Benchmarking of optic flow versus GPS during flight test.

## 7 Conclusion

In this project, a small embedded vision sensor has been developed using Field Programmable Gate Array (FPGA) technology. The sensor computes optic flow using the image interpolation algorithm to provide flow vectors at 36 equi-spaced locations in a 1024×768 pixel image. The sensor has been tested in flight and benchmarked against GPS measurements of velocity. A small form factor (70mm×55mm) and low-weight for the sensor makes it ideal for implementation on a small flight vehicle.

## Acknowledgments

This work has been partly supported by seed funding from the US Army International Technology Center Pacific (ITC-PAC).

## References

[Aubépart,2007] F.Aubépart and N.Franceschini Bio-inspired optic flow sensors based on FPGA: Application to Micro-Air-Vehicles *Microprocessors and Microsystems*, 31:408–419,2007

[Arribas,2003] P.C. Arribas and F.M.H. Maciá. FPGA implementation of Santos-Victor optical flow algorithm for real-time image processing: an useful attempt. *Proceedings of SPIE*, 5117:23–32, 2003.

[Barrows,2000] G.L. Barrows and C.Neely. Mixed-mode VLSI optic flow sensors for in-flight control of a micro air vehicle. In *Proceedings of the 45th SPIE Annual Meeting*, San Diego, USA, July 31 - August 4 2000.

[Barrows,2002] G.L.Barrows. Future visual microsensors for mini/micro-UAV applications. In *Proceedings of the 2002 7th IEEE International Workshop on Cellular Neural Networks and Their Applications*, pages 498–506, 2002.

[Bowman, 2008] D. Bowman, A. Lambert, D. Fraser, and L. Swierkowski. Seeing through the haze. *ICHSIP28, 28th International Congress on High Speed Imaging and Photonics, Canberra, Australia, SPIE* , 7126:HS101-63, 9pp, November 2008.

[Brown et al,2008] T.A.Browne, J.V.Condell, G.Prasad and T.M.McGinnity. An Investigation inot Optical Flow Computation on FPGA Hardware *Proceedings of the International Machine Vision and Image Processing Conference*, pages 176–181, 3-5 Sep 2008.

[Díaz,2006] J. Díaz, E. Ros, F. Pelayo, E.M Ortigosa, and S. Mota. FPGA-based real-time optical-flow system. *IEEE Transactions on Circuits and Systems for Video Technology*, 16(2):274–279, 2006.

[Garratt et al, 2006] M.A.Garratt, B.Ahmed and H.Pota. Platform enhancements and system identification for control of an unmanned helicopter. In *IEEE Conference on Control, Automation, Robotics and Vision - ICARV 2006*, pages 1981-1986, 5-8 Dec 2006.

[Garratt and Chahl, 2007] M.A.Garratt and J.S.Chahl. An optic flow damped hover controller for an autonomous helicopter. In *Proceedings of the 22nd International UAV Systems Conference*, Bristol, UK.

[Garratt and Chahl, 2008] M.A.Garratt and J.S.Chahl. Vision-Based Terrain Following for an Unmanned Rotorcraft Journal of Field Robotics 25(4-5):284–301, 10 Apr 2008.

[Horn and Schunk,1979] B.K.P. Horn and B.G. Schunk Determining Optical Flow. *Artificial Intelligence*, 17:185–203, 1979.

[Lucas and Kanade, 1984] B. Lucas and T. Kanade An iterative image registration technique with an application to stereo vision In *Proc. DARPA Image Understanding Workshop*, pp.121–130, 1984.

[Martín,2005] J.L.Martín, A.Zuloaga, C.Cuadrado, J.Lazaro, and U.Bidarte. Hardware implementation of optical flow constraint equation using FPGAs. *Computer Vision and Image Understanding*, 98(3):462–490,2005.

[Srinivasan,1994] Srinivasan, M. (1994). An image interpolation technique for the computation of optic flow and egomotion. *Biological Cybernetics*, 71:401–415.

[Wei et al, 2007] Z. Wei, D.Lee and B.E.Nelson FPGA-based Real-time Optical Flow Algorithm Design and Implementation *Journal of Multimedia*, 2(5):38–45.