

Calculating Optical Flow using FPGAs

Michael Chisholm (u4212270)
Supervisor: Dr. Eric McCreath
Course: COMP3750 (6 units)

13th June 2008

Abstract

Optical flow is the velocity of the movement of individual pixels within a sequence of images. It gives information about the way both the camera and the objects it is viewing move. Adding optical flow detection capabilities to various devices can enhance their abilities in useful ways, for example a self-controlled car can use optical flow to avoid colliding with obstacles. Extracting optical flow from image sequences is a complex problem, and there are various methods that attempt to solve it based on different assumptions.

For optical flow to be usable in an embedded application the processor calculating it must be small and have low power consumption. FPGAs are ideal for this task. They consist of reprogrammable logic hardware in a single chip package. Their nature is well suited to highly pipelined and parallel operations, such as optical flow extraction.

I investigate, using well-defined criteria, various optical flow extraction algorithms and FPGA-based implementations. I then propose creating a new implementation designed for an FPGA, using a previously unused algorithm that has high output density and relatively low error rates and resource usage. I go on to prove that this algorithm can be implemented using an FPGA. I then create a functionally-equivalent PC-based simulation to investigate the performance and behaviour of the design. The accuracy of the implementation is found to be less than ideal, so I propose methods of improving the results.

I show that the algorithm can process 7600000 pixels per second on a Nexys-2 development board, which is enough to extract optical flow from QVGA video at over 98 frames per second. It does this with a latency of less than four frames, making it well suited to real-time environments.

Contents

1	Introduction	4
2	Background - Optical Flow	7
2.1	Visual Assumptions	7
2.2	Temporal Assumptions	8
2.3	Spatial Assumptions	8
2.4	The Aperture Problem	9
2.5	Optical Flow Calculation Techniques	10
2.5.1	Correlation: Region Matching	10
2.5.2	Energy in the Frequency Domain	10
2.5.3	Phase in the Frequency Domain	11
2.5.4	Image Gradients and Space-time Derivatives	11
2.6	Assessment of Optical Flow Techniques	12
2.6.1	Efficiency	12
2.6.2	Accuracy	13
2.6.3	Other Benchmarks	14
2.7	Algorithm Selection for Implementation	16
3	FPGAs	17
3.1	Other Implementation Options	17
3.1.1	Software on Conventional Processors	17
3.1.2	Software on Parallel Processors	18
3.1.3	Analog VLSI	18
3.1.4	Digital Signal Processors	18
3.2	Previous Optical Flow Implementations on FPGA	18
3.2.1	Horn & Schunck's algorithm implemented in 1998	19
3.2.2	Camus Correlation implemented in 2001	19
3.2.3	Lucas and Kanade's algorithm implemented in 2004	19
3.2.4	Horn and Schunck's algorithm re-implemented in 2004	20
3.2.5	Tensor-based method implemented in 2007	20
4	Design	21
4.1	Requirements	21
4.2	Algorithm used in the implementation: Uras et al.	22
4.3	Pipeline design for algorithm implementation	23
4.3.1	Spatial convolution	23
4.3.2	Temporal filtering	25
4.3.3	Velocity calculation	27

5	Implementation	30
5.1	FPGA Development System	30
5.1.1	Nexys-2 development board	30
5.1.2	Software tools	31
5.2	Blocks implemented on FPGA	32
5.2.1	Gaussian spatial blur	32
5.2.2	External memory access	33
5.2.3	PC communications	34
5.2.4	Resources consumed	34
5.3	Functionally equivalent simulation	35
5.3.1	Temporal filter data format optimisation	35
5.3.2	Velocity post-processing techniques	36
6	Results and evaluation	38
6.1	Test sequences	38
6.1.1	Sinusoids	38
6.1.2	Squares	39
6.1.3	Trees	39
6.1.4	Yosemite	39
6.2	Resulting test sequence flow fields	41
6.2.1	Sinusoid-1	41
6.2.2	Sinusoid-2	41
6.2.3	Squares	42
6.2.4	Trees	43
6.2.5	Yosemite	44
6.3	Numerical error results	44
7	Conclusions and future work	47
A	Timetable	49
	Bibliography	51

Chapter 1

Introduction

For computer vision related problems, optical flow provides valuable information on the way objects move within a scene. This review will discuss what optical flow is and techniques for calculating it. The focus will then be on recent work using Field Programmable Gate Arrays (FPGAs) to calculate optical flow and how that work can be extended to provide a better accuracy versus efficiency trade-off.

Optical flow is the representation of the motion of objects in a visual scene. Motion is typically represented as a vector field, with each vector approximating the apparent motion of a pixel or group of pixels within an image. Figure 1.1 shows three frames of an image sequence with a square moving to the right. Optical flow is extracted from a sequence of images to produce a vector field of the velocity flows. Figure 1.2 shows the estimated optical flow at the time of the second frame.

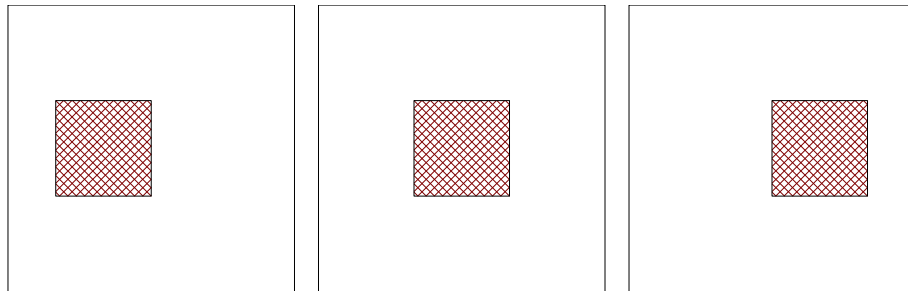


Figure 1.1: Three frames of an image sequence showing a square moving to the right

From the flow vectors, further processing can find regions of the image moving at different speeds relative to each other. These regions typically correspond to separate objects, and so optical flow can be used to separate the pixels representing the objects from the rest of the scene. The overall motion of the vectors provides information about the movement of the camera relative to the scene and can be used for judging the speed of the camera relative to, for example, the ground, a wall or even another moving object. This information can be used for collision avoidance by calculating time until impact [31]. Since the camera

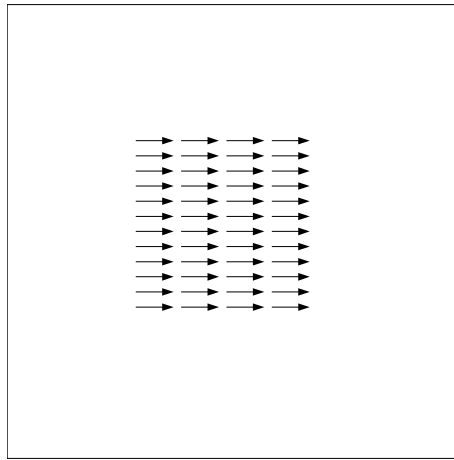


Figure 1.2: Optical flow estimate at time of second frame

can be mounted on a vehicle, optical flow can be used to assist in the unmanned control of the vehicle.

This is just one area in which optical flow is useful. Other tasks where it may be used include, but is not limited to: tracking people in security camera footage, determining the three-dimensional position of objects within a scene, motion capture for computer-generated imagery, and aiding other visual processing tasks, such as gesture recognition.

For many such tasks, the camera and the associated processing system needs to be completely mobile. This implies that the processing system must be small, light, robust, and draw minimum amounts of power. Clearly, personal computers are not ideally suited to the task given these size and power constraints. In addition, the optical flow calculations must be done in real-time to be useful. For a typical camera with a resolution of 320 by 240 pixels, running at 30 frames per second (FPS), over 2.3 million pixels must be processed each second. Optical flow calculation may be pipelined, so that each stage of the calculation feeds directly into the next stage and more data may be input into earlier stages while later stages are still processing. For such tasks FPGAs are ideal. They have low power requirements and have a highly parallel design. This makes them well suited to calculating optical flow in an embedded system.

After reviewing various optical flow algorithms, I have chosen an algorithm created by Uras et al. for creating the design. It offers a high flow density and a relatively low error rate. The algorithm is well suited to implementation on an FPGA, as it has a very linear nature that can be transferred into a pipeline form. To prove that the algorithm will work on an FPGA, I have taken two approaches.

First, parts of the pipeline are implemented on the FPGA to determine the resources consumed and processing capabilities of the final design. This was done using a Nexys-2 development board containing a Spartan-3E 1200K FPGA. I have successfully shown that the design will fit on the FPGA and is able to process 7600kpx/s.

Secondly, the entire algorithm was implemented on a PC in a functionally-equivalent simulation of the FPGA's design. This means all parameters, data

formats and processing steps are the same as would be used on the FPGA. This approach allows the design to be easily adjusted for fine-tuning. The simulation was trialled with a variety of test sequences. This revealed that the design can successfully extract optical flow from image data, however the error rate is higher than desired. I therefore suggest several ways that the results can be improved.

In Chapter 2 I discuss the problem of extracting optical flow, assumptions made, ways to measure and assess the performance of techniques used, and specific algorithms and implementations used to extract optical flow. Chapter 3 will describe what an FPGA is, other hardware options for implementing optical flow algorithms, and previous implementations of optical flow extraction algorithms that use FPGAs. In Chapter 4 I propose a new design that uses an FPGA, then I describe its implementation in Chapter 5. Finally, I assess the results of this implementation in Chapter 6, and discuss ways of improving its error rate. This leads into the conclusions that can be drawn from this implementation, and what work can be done in the future towards creating a final design for use in embedded devices.

Chapter 2

Background - Optical Flow

To understand how optical flow extraction algorithms work, we must first know what problems they are solving. These problems are a result of the way image data are captured and the limitations in capturing that data.

To calculate the optical flow in a scene a view of the scene must be created or captured. This view is a single 2D representation of the 3D scene. It is taken from one position, often in the form of a video stream from a camera. A view is only an approximation of the scene, and this approximation can cause errors when calculating the optical flow. The video is quantised in three ways: spatially, temporally, and digitally. When an image is taken, it is divided into a grid of pixels, with each pixel representing a point in the image. Details that are smaller than a pixel in size are lost. The images themselves are only snapshots of the scene at a particular time, and any movement in between two images has to be inferred from the initial position in the first image and the final position in the second image. The pixels themselves are a digital approximation of the true intensity. For optical flow calculation, in most cases only the luminosity of each pixel is used and colour is ignored. Despite these losses in information, optical flow can still be calculated provided some assumptions about the images are made.

To extract meaningful optical flow information from a series of images, certain assumptions about the images must be made, visually, temporally, and spatially. There exist various algorithms for calculating optical flow, each based on a unique combination of assumptions. These assumptions and their violations are now stated and explained.

2.1 Visual Assumptions

Visually, surfaces should have a noticeable gradient, texture, or pattern, or at least have noticeable edges. This is known as non-uniformity, and is required so that changes in the position of the surface are noticeable. If this assumption is violated, differences in the position of surfaces will not be noticed and hence optical flow can not be extracted. Some algorithms place less requirement on the surface being non-uniform and are able to interpolate motion from the edges of it.

An often used assumption, as stated by Black [12, pg 7], is that of intensity

constancy or data conservation:

Image measurements (for example, image intensity) corresponding to a small image region remain the same, although the location of the region may change over time.

This is required for gradient-based, correlation-based, and filtering-based methods [12]. This assumption is often violated by the lighting in parts of a scene. If an object crosses into shadow then the overall intensity of the surface of that object will vary, causing problems for algorithms expecting it to remain constant. Lighting can also cause reflections on the object that move independently of the object itself, creating flow discontinuities. In most cases the image will retain enough data from the previous image for optical flow to be calculated.

2.2 Temporal Assumptions

Temporally, the motion of groups of pixels should not be too large between consecutive frames. That is, changes should be gradual. If this assumption is violated, optical flow may not be perceived correctly, if at all. The acceptable limits on motion varies between algorithms and even between the same algorithm using different parameters. To increase the upper limit on motion, some algorithms determine gross optical flow at a very coarse scale and then fine tune the flow at finer scales. Alternatively, the frame rate may be increased at the cost of computational time.

Another temporal problem occurs when, although an object itself hasn't moved faster than can be detected, its surface pattern is changing faster than the Nyquist limit imposed by the frame rate. This will cause the perceived flow to be slower than it is, or even in the opposite direction. The effect of sampling too slowly can be observed in the spinning of wagon wheels in movies where, unless motion blurring has been applied, the wheel may appear to be spinning backwards. To counteract this effect, the image may be filtered temporally to remove high frequency components of motion. Although these high frequency components won't be perceived, they also won't be perceived incorrectly, which can be worse.

2.3 Spatial Assumptions

Spatially, local groups of pixels should all belong to the same surface and flow coherently together. This assumption is clearly violated at the boundaries of objects, where neighbouring pixels may move in different directions, but it is often necessary for the pixels making up an object to move together for their motion to be detected correctly.

The boundaries of objects present many problems for optical flow algorithms. They not only introduce spatial discontinuities but temporal ones too. When an object is moving in front of a background, pixels in the background are obscured and revealed. These are sudden changes that occur between frames, and often cause spurious results at the boundaries of objects for certain algorithms.

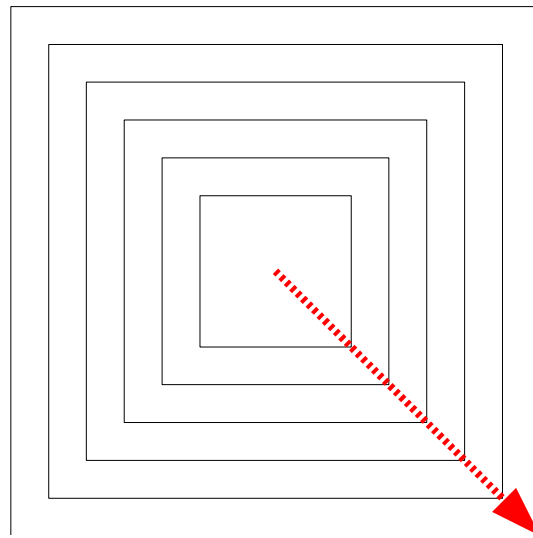


Figure 2.1: Actual motion of a textured surface

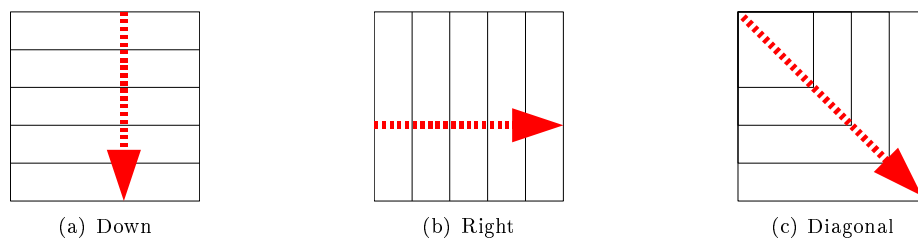


Figure 2.2: Apparent motion of textured surface as viewed through an aperture

2.4 The Aperture Problem

When a texture that varies only on one axis moves, only the flow parallel to the texture variation can be perceived [22]. Even if the texture varies along two axes at the large scale, when viewed through a small aperture it may still only appear to vary along one axis. For example, the texture in Figure 2.1 is moving diagonally down and to the right. However, when seen through different apertures, the perceived motion is only in the direction that the texture appears to change in. In Figure 2.2(a) the texture appears to be going down only and in 2.2(b) it appears to be going right only. It is only in 2.2(c), where the texture visibly varies in both directions, that its true motion can be perceived.

The assumption of non-uniformity will be violated at the local level, for example individual pixels, in these cases. To rectify this further assumptions must be applied, such as the spatial coherency assumption. It tells us that neighbouring pixels should be moving together, and so we can use the flow information of neighbouring pixels to improve the estimate of flow of the observed pixels.

2.5 Optical Flow Calculation Techniques

Most optical flow extraction algorithms use one of only a few general techniques, with the different algorithms derived by applying different constraints and assumptions to a given problem. This gives rise to a large variety of algorithms to choose from. Each one has benefits and trade-offs, and no single algorithm is suited to every possible application. A variety of algorithms must therefore be considered before choosing the best one to implement using an FPGA. Not every algorithm is described here, but the ones that have been well researched with known accuracy and speed. The approach I implemented is the last of the ones described. It was proposed by Uras et al. [40] and uses the spatial and temporal second derivatives of the image texture.

2.5.1 Correlation: Region Matching

Correlation techniques work on the basis of matching regions of the current image to regions in previous images. For each region in the first image, the second image is searched for a matching region. A match is determined by maximising a correlation function or, equivalently, minimising a distance function. The velocity vectors are then defined as the shift in the regions over time [9]. Correlation techniques are useful when only two frames are available, or when there is significant noise in the images [10].

One such technique, created by Anandan [6], uses the sum of squared differences (SSD) to compute correlation. Window (region) sizes are fixed at 5x5 pixels, and matches are searched for in a 3x3 space. This is done starting at a very coarse scale by effectively reducing the resolution of the image before performing a motion search. These results are then used as conditioning parameters at increasingly finer scales until flow vectors are calculated at the individual pixel level. Confidence parameters are calculated from the local minima of the SSD values, but they are unreliable [9].

Singh [35] has described a technique similar to Anandan's, except three frames are used in the computation, the previous, current and next frames. An SSD surface is calculated from the sum of the two SSDs obtained between the previous and current, and current and next frames [10]. This is converted into a 2D probability distribution at each pixel. The mean of this distribution in the x and y directions then gives the flow vectors. Like Anandan's technique, the flow vectors are first calculated coarsely then progressively refined. This technique also provides confidence parameters, calculated from the probability distribution, which prove more reliable than Anandan's [9].

2.5.2 Energy in the Frequency Domain

Another technique is based on the output of velocity-tuned filters. These filters are sensitive to particular orientations and frequencies in the Fourier domain [10]. They also perform better than correlation-based methods when complex patterns are being observed, such as a random dot pattern translating across the field of view [3].

A technique using the combined output of twelve filters was developed by Heeger [20]. He uses Gabor filters that are constructed in the Fourier domain to give a peak response at a centre frequency in the space-time domain. Eight

of the filters are tuned to certain patterns moving in certain orientations. For example, one filter is tuned to vertical patterns moving rightward. The other four filters are tuned to stationary images with patterns in various directions. The filters are not tuned for particular velocities. Instead, the image is resized to various levels before being filtered, giving different output responses for different sizes, but there is a correspondence between the size of the image, the velocity, and the output of the filters.

These filters are convolved with the image sequence to produce a set of output energies. To get the velocity from the filter outputs, a least-squares method is used to minimise the predicted energy and actual filter output. This gives a single velocity for each image region. However, Fleet and Jepson [18] state that this “does not reliably resolve different velocities on the same spatial patch”. That is, only one flow velocity will be calculated for each image region even if there are actually multiple velocities there, such as occurs at the edge of an object moving in front of another object.

2.5.3 Phase in the Frequency Domain

Phase-based techniques are related to energy-based techniques in that the motion is analysed in the Fourier domain, however the phase of the output of velocity-tuned filters is used instead of the energy output at particular frequencies. Fleet and Jepson [18] describe a technique similar to Heeger’s, with the use of multiple Gabor filters tuned in different directions, however they use the phase output of the filters instead of the energy output to determine the image flow velocities.

To obtain the phase information, they use a more generic form of the Gabor filter equation with output in the complex domain with an amplitude and phase component. By taking the derivative of the phase a contour map is generated, from which the original image flow velocities can be determined [9].

Fleet and Jepson [18] claim, and Barron et al. [9] experimentally prove that phase-based techniques provide higher velocity resolution, accuracy and robustness compared to energy-based techniques. However, this comes at an even higher computational cost [24].

2.5.4 Image Gradients and Space-time Derivatives

Gradient-based techniques work on the principle of image differentiation with respect to space and time. Since the velocity of an object can be calculated from the derivative of its position with respect to time, it stands to reason that the velocity of components of an image can be calculated from its time derivative. Gradient-based techniques are more likely to work in real-time than other techniques because of their efficiency and low computational costs [19, 9, 24].

Horn and Schunck Horn and Schunck [23] were the first to describe this technique directly. By considering each pixel in the image to be the result of a function of space and time, $E(x, y, t)$, they were able to take the first-order numerical derivative of images to produce a flow velocity vector field. The derivative was approximated to the local gradient at each pixel by taking the difference between adjacent pixels in both space and time. The velocity

estimates are obtained by iteratively solving a set of simultaneous equations. More iterations gives more accurate results at the cost of longer computation time and latency. As a compromise, it is possible to perform only one iteration per frame, with the initial estimates taken from the results of the previous frame. The algorithm will then converge to a reasonably accurate result after a number of frames, provided that velocity does not vary too much between frames. This technique is sensitive to noise, and subsequent implementations smooth the images (using a Gaussian blur) before calculating the derivatives [9].

As discussed previously, the aperture problem means that the velocity may not be accurately determined at the local level for regions that aren't strongly textured. Horn and Schunck apply a global smoothness constraint, assuming that neighbouring pixels have similar velocities. This deals well with smoothly textured regions, filling in the velocities at their centres with the velocities from their edges and giving higher flow densities. However, this causes problems in regions with naturally different velocities, such as object edges.

Lucas and Kanade Lucas and Kanade [26] also use a first-order numerical derivative, but they apply a local smoothness constraint instead. As described by Beauchemin et al. [10], the velocities for each pixel are obtained by finding the least-squares error value within a local region. This technique is more accurate in regions containing different velocities. However, it doesn't fill in the centre of smoothly textured regions and produces much lower flow densities as a result. Unlike Horn and Schunck's algorithm, this one also provides confidence parameters for pixel velocities.

Uras et al. Uras, Giroi, Verri and Torre [40] use second-order spatio-temporal derivatives of the image sequence to calculate flow velocities. Uras' technique gives a higher density than Lucas and Kanade's and is more accurate than Horn and Schunck's [9]. This is the algorithm that I will implement using an FPGA. Their method will be discussed further in subsection 4.2 on page 22.

2.6 Assessment of Optical Flow Techniques

Before discussing why the Uras et al. algorithm was chosen, it is useful to have a set of common criteria to measure the various algorithms against. There are two broad areas that we are interested in: efficiency and accuracy. The chosen algorithm will be the most accurate possible while still giving real-time results when implemented on an FPGA.

2.6.1 Efficiency

The efficiency of optical flow extraction algorithms is generally determined through experimental measurement. To compare the efficiency of various algorithms they are all implemented on the same platform. For software implementations, test image sequences are processed by the algorithms and the time taken is measured. For FPGAs, the resources consumed by the algorithm and the latency of each stage is more important. There is a correlation between

Table 2.1: Efficiency results reported by McCarthy & Barnes for 192x144 pixel images on an Intel PIII 866MHz [30]

Method	Camus 2	Camus 5	Lucas	Horn	Nagel
Time (ms)	460	1770	263	105	630
Speed (kpx/s)	60.1	15.6	105.1	263.3	43.9

Table 2.2: Efficiency results reported by Liu et al. and Bober & Kittler for 150x150 pixel images [24, 13]

Method	Horn	Uras	Anandan	Lucas	Fleet	Bober
Time (min:sec)	8:00	0:38	8:12	0:23	30:02	8:10
Speed (px/s) (NB: units changed)	47	592	46	978	12	46
Time (min:sec) on HyperSparc 10	2:00	0:10	2:03	0:06	6:00	2:03
Speed (px/s) (NB: units changed)	188	2250	183	3750	63	183

speed in software and resources used on an FPGA so results from software tests can predict efficiency of FPGA implementations.

Various authors have implemented a range of algorithms in software to measure their speed [30, 13, 24, 37]. Unfortunately, their implementations and test data are varied, making it difficult to directly compare their results. Tables 2.1, 2.2 and 2.3 show the time results they obtained. To ease comparison, I converted the times into a speed measured in kilo-pixels per second (kpx/s). This gives a value that is independent of image size but is still determined by the test platform.

As can be seen from the tables, gradient-based methods, such as Horn and Schunck or Uras et al., are the most efficient. Correlation methods, such as Camus correlation, are less efficient. Frequency filter methods, such as Fleet and Jepson, have the lowest efficiency overall. Correlation methods also require searches to be performed between images. Although searches can be run in parallel, the build up of costs associated with each test will decrease their efficiency on FPGAs.

The efficiency can also depend on the complexity of the scene being examined, as shown by Sosa et al. in Table 2.3. An FPGA implementation will be designed to give a constant processing speed, independent of scene complexity.

2.6.2 Accuracy

The most common measurement of accuracy was established by Fleet and Jepson [18]. They use a set of synthetic video sequences with known optical flow fields. These test sequences are fed into implementations of the algorithms and the output is compared to the reference fields. The average speed and angular error of the flow vectors produced by the algorithm give an indication of how accurate the algorithm is. McCane et al. [28] created additional synthetic sequences and also developed a method for extracting true optical flow data from simple real sequences. They showed that benchmarking against synthetic

Table 2.3: Efficiency results reported by Sosa et al. for various sequences of 256x256 pixel images on an Intel Xeon 3GHz [37]

Method	Horn	Change-driven Horn
Time for Indoor sequence (ms)	90	80
Speed (kpx/s)	728.2	819.2
Time for Outdoor sequence (ms)	141	94
Speed (kpx/s)	464.8	697.2
Time for Artificial sequence (ms)	94	31
Speed (kpx/s)	697.2	2114.1

video sequences gives a good indication of how an algorithm will perform on real sequences.

Many algorithms also output a confidence measure for each vector. By only using vectors with a high confidence, the accuracy of the results is improved at the cost of the density of the flow field. The density is the percentage of flow vectors that are retained after rejecting the low-confidence vectors.

The accuracy may be affected by parameters other than the technique used, such as the frame rate, the numerical accuracy of calculation results, the level of spatial and temporal filtering applied to the input data before optical flow extraction, and the level of thresholding on spurious values. This must be kept in mind when comparing the results presented in different papers.

Tables 2.4 and 2.5 show the accuracy results obtained by Barron et al. and Bober & Kittler. The results are the angular error in comparison to known optical flow vectors for a video of trees diverging as the camera approaches. It should be noted that algorithms that do not provide confidence measures tend to produce the most inaccurate results, even though they have the advantage of providing 100% density. Of those algorithms that do provide confidence measurements, the correlation-based approaches produce consistently less accurate results than the other techniques.

In addition to quantitative results, accuracy can be judged qualitatively by looking at a visual representation of the output of an algorithm. This will often reveal where an algorithm is producing spurious results, such as at the edges of objects, and indicates where the denser regions are. For the best indication of accuracy, however, angular and speed error still give better results.

2.6.3 Other Benchmarks

Although efficiency is the best indicator of the overall performance of an algorithm, there are other things to consider in an embedded environment. Even if results are calculated within the same time period as data arrives, fulfilling the real-time requirement, those results still need to be delivered within a certain delay to be useful. The latency of the results will affect such things as collision avoidance, navigation, and object tracking. There is a trade-off between latency and accuracy, with different tasks putting more emphasis on one or the other. One way of balancing this is with the selection of temporal filters. Fleet and Langley describe a way to use Infinite Impulse Response (IIR) instead of Finite Impulse Response (FIR) temporal filters, reducing storage requirements and further improving the efficiency of gradient-based methods [19]. IIR filters also

Table 2.4: Accuracy reported by Barron et al. for the Diverging Tree sequence [9]

Method	Average Error (°)	Standard Deviation (°)	Density (%)
Horn & Schunck	12.02	11.72	100
Lucas & Kanade	1.94	2.06	48.2
Uras et al. (unthresholded)	4.64	3.48	100
Uras et al. (thresholded)	3.83	2.19	60.2
Nagel	3.21	3.43	53.5
Anandan	7.64	4.96	100
Singh	8.40	4.78	99.0
Heeger	4.49	3.10	74.2
Fleet & Jepson	0.99	0.78	61.0

Table 2.5: Accuracy reported by Bober & Kittler for the Diverging Tree sequence [13]

Method	Average Error (°)	Standard Deviation (°)	Density (%)
Horn & Schunck	9.84	8.86	100
Anandan	8.23	6.17	100
Uras (unthresholded)	6.51	7.00	100
Heeger	4.95	3.09	73.8
Lucas & Kanade	3.05	2.53	49.4
Fleet & Jepson	1.08	0.52	64.3
Bober	3.69	4.39	100

reduce delays between inputting a new image and getting the flow corresponding to it, but this comes at the cost of accuracy [29].

Both performance and accuracy will be affected by the complexity of motions within a scene. For an algorithm to be useful in a range of environments, it must be consistent in its performance and accuracy. The robustness will be affected by the basic assumptions made in the design of the algorithm, such as global or local smoothness constraints [24], as well as statistical methods used to calculate resulting velocities [12]. A higher degree of accuracy can be maintained by spending more computational time on complex scenes [15]. However, an unpredictable computational cost is not suited to a real-time environment.

2.7 Algorithm Selection for Implementation

As discussed previously, gradient-based methods are the most computationally efficient and give good levels of accuracy. Although Fleet and Jepson's method had superior accuracy, it has very low efficiency due to the number of computationally expensive filters required to implement it [9]. Therefore, a gradient-based technique was chosen.

In their 1998 paper, Liu et al. [24] predicted that conventional processors available in 2005 would be able to implement Lucas and Kanade's algorithm, Uras' would be viable in 2006, Horn and Schunck's in 2010 and other algorithms in later years. This prediction was for software-based approaches, and hardware-based approaches are able to achieve better results sooner. Lucas and Kanade's algorithm has already been implemented in hardware, with real-time results achieved in 2004 [16, 17]. Horn and Schunck's algorithm has also been implemented in hardware multiple times, with real-time results achieved in 2005 [7, 27]. An approach that only calculated new flow velocities for frames that had noticeably changed since the previous frame was implemented in 2007, with a further speed improvement for low-motion scenes, but not high-motion scenes [37].

After reviewing the literature, it was found that Uras' algorithm has not previously been implemented in hardware. As already stated, their algorithm produces denser output than Lucas and Kanade's with similar efficiency and only a slight reduction in accuracy. Higher density is important when finding and tracking objects within a scene. For this reason, the feasibility of implementing Uras' algorithm using FPGAs will be investigated, with the goal of producing a design that works in real-time.

Chapter 3

FPGAs

As defined by Brown and Rose, an FPGA is a “*type of integrated circuit used for implementing digital hardware, where the chip can be configured by the end user to realize different designs*” [14]. They describe an FPGA as composed of a large array of logic blocks that are joined via interconnect channels, which are often arranged in a grid. The logic blocks consist of inputs fed into look-up tables (LUTs) that are then connected via logic gates to the block’s outputs. Logic blocks may also contain flip-flops, or even hardware dedicated to a particular task, such as multipliers.

This structure allows for a wide variety of complex logic to be implemented, including memory controllers and even CPUs. They are well suited to operations that can be pipelined or run in parallel. Other tasks, such as division, consume large amounts of space and time and should be used as sparingly as possible [25]. Uras’ algorithm can be implemented using mostly addition and multiplication, with very little division [2], making it well suited to FPGAs.

3.1 Other Implementation Options

There are other options for implementing optical flow detection. They each have benefits and downsides, but FPGAs are the best overall option for an embedded environment.

3.1.1 Software on Conventional Processors

There are numerous implementations of optical flow algorithms using conventional CPUs but it is only now that they are capable of approaching real-time performance [24]. An implementation of the Horn and Schunck algorithm running on a 3GHz Intel Xeon CPU managed approximately one quarter of the speed of an FPGA implementation [37]. It has also been found that, for FPGAs and CPUs released at the same time, FPGAs have higher floating-point performance [39].

The Xeon processor consumes large amounts of power and requires active cooling. Processors designed for embedded applications, such as the Intel XS-scale, are slower than the Xeon and will give even less performance.

3.1.2 Software on Parallel Processors

Optical flow computation is a highly parallel operation well suited to parallel processors. One such processor, the Cell Broadband Engine, performs at 204.8 GFlops, compared to 3.6 GFlops for an Intel Itanium 2 [44]. Likewise, the GeForce 7800 GTX can reach 165 GFlops [33]. Unfortunately, the Cell draws 40W of power [44] and the GeForce 7800 draws up to 100W [1]. This is far too much for an embedded system, especially mobile ones reliant on batteries for power.

3.1.3 Analog VLSI

Another approach to getting optical flow estimation into an embedded design is to use analog circuitry in a Very Large Scale Integration (aVLSI) package. So far, however, the maximum resolution achieved has been 30x30 pixels [38]. This is too low for object tracking applications, although it may be sufficient for overall motion estimation of a vehicle.

3.1.4 Digital Signal Processors

Digital Signal Processors (DSPs) are specialised microprocessors designed specifically for processing signals in real-time. They often have higher data bandwidth than conventional processors and are capable of some degree of parallel processing [36], although this is in a more limited form compared to FPGAs.

Compared to FPGAs, most DSPs have major bottle-necks when it comes to image processing tasks. They may need to transfer data between memory and the processor multiple times for each pixel. For operations dealing with many pixels at a time, such as convolution, they require several clock cycles and operate sequentially, compared to the pipelined nature of an FPGA [43].

In a direct comparison between Application Specific Integrated Circuits (ASICs) and DSPs for optical flow extraction, it was found that an ASIC implementation had at least ten times the throughput and one tenth the latency of a single-DSP implementation [34]. Although not a direct comparison between FPGAs and DSPs, ASICs are designed in the same way as FPGAs, showing that the highly pipelined and parallel nature has merit.

DSPs have been used in the past for optical flow extraction and object tracking, but five boards containing two DSPs each were required to achieve real-time performance on 160x120 pixel images [49]. For an embedded system it is more desirable to have all the processing performed on a single chip.

DSPs may prove viable for optical flow estimation, but it was decided to implement the algorithm using FPGAs due to their greater flexibility and intrinsically pipeline-able nature.

3.2 Previous Optical Flow Implementations on FPGA

There have been a number of optical flow extraction implementations using FPGAs. The first attempts were made over ten years ago. As FPGA technology

has improved, so has the performance of the implementations. All of the investigated designs broke the algorithm down into stages, then implemented these stages as separate blocks that were linked together. This is a natural way of pipelining the algorithms and allows the blocks to be tested separately. Performance measurements were often expressed in terms of frames per second with a certain frame size. As suggested before, these values are converted to kilo-pixels per second (kpx/s) for comparison.

3.2.1 Horn & Schunck's algorithm implemented in 1998

One of the first FPGA-based designs was implemented in 1998 by Cobos and Monasterio [7]. It was based on the Horn and Schunck algorithm with three iterations per frame. It used two FPGAs in a pipelined arrangement, running at an unspecified clock speed. The first, a Xilinx XC4005H containing only 5000 gates [45], was used to read the input data and calculate the gradients. The second, a Xilinx XC4020E containing 20 000 gates [46], was used for the remaining calculations including iterative solving and finding the final velocity values. This design managed only 19 FPS on 50x50 pixel images, giving a rate of 47.5kpx/s.

3.2.2 Camus Correlation implemented in 2001

The same authors created a second design in 2001, based on Camus correlation [8]. Since it was based on a correlation-based technique it has lower accuracy than other designs. Motion has to be tracked for more than one frame to get finer accuracy than 1 pixel per second. The design was implemented on an Altera EPF10K50RC240-3, which contains 50 000 gates [5], running at 15MHz. This design achieved 22.5 FPS on 96x96 pixel images, a rate of 207kpx/s. Clearly, neither this implementation nor the previous one are suitable to real-time optical flow extraction.

3.2.3 Lucas and Kanade's algorithm implemented in 2004

In 2004, Díaz et al. [16, 17] implemented a design based on the Lucas and Kanade algorithm. They did not use error thresholding, so although the output had 100% density, it also had an average error of 18° compared to 4.3° for a thresholded software implementation tested on the same image sequence. Their design used fixed-point numbers up until the final velocity calculation, which had to be done using floating-point due to the large dynamic range. This saved room on the FPGA with only a small increase in average error compared to their all floating-point, unthresholded, software implementation, which had 15° of error. Another modification they made was to use an IIR temporal filter instead of an FIR temporal filter, as previously suggested, decreasing latency and storage requirements. Using a Xilinx Virtex XCV2000E containing over 500000 gates [47], they achieved 24 FPS on 320x240 images, a rate of 1843kpx/s. This is suitable for real-time use, but the average error may be too high for some applications.

3.2.4 Horn and Schunck’s algorithm re-implemented in 2004

Later in 2004 Martín et al. [27] implemented another design based on the Horn and Schunck algorithm, although they did it on a more advanced FPGA than that used by Cobos and Monasterio. Their design used fixed-point values, which gave only 2° more error than an equivalent floating-point implementation. While implementing their design, they tested individual blocks from the pipeline separately to ensure they were working before integrating them into the final design. They implemented their final design on an Altera EP20K300EQC240-2, which contains 300 000 gates [4]. This particular FPGA didn’t have enough memory to store an entire frame of data, which is required for numerical derivative calculation, so they used external hardware FIFOs instead. Adding more external hardware increases the cost of a design in terms of power used and money spent, so it may have been better to choose a device with more internal memory, as are typically available now. Their design achieved 60 FPS on 256×256 images, a rate of 3932kpx/s.

3.2.5 Tensor-based method implemented in 2007

In 2007 Wei et al. [42] implemented a design based on 3D tensors. Tensors are “a compact representation of local orientation” [42], which means this design uses a subcategory of gradient methods. It was implemented on a Xilinx Virtex-II Pro XC2VP30 running at 100MHz. This device contains 3 million gates and 136 dedicated multipliers [48]. Using this FPGA, the design achieved 64 FPS on 640×480 images, a rate of 19661kpx/s. It also had a lower average error than previous designs, with 12.9° and 100% density on the same sequence as Díaz et al. used. The authors later showed that by using an FPGA with more resources they would be able to improve the accuracy even further for real-life sequences [41].

Chapter 4

Design

4.1 Requirements

The first aim is to prove that an optical flow algorithm can be implemented using an FPGA. As other groups have successfully done this[42, 17], our aim is to improve upon their results with the following criteria in mind.

Maximise flow density For object tracking it is desirable to have velocity estimates that cover the entire object and not just the object's boundaries. This is reflected in having a high flow density. Although 100% flow density guarantees a flow estimate at every pixel, we have to allow for some velocity estimates to be excluded due to inaccuracies or stillness in the viewed scene.

Minimise error Any velocity estimate errors will degrade the accuracy of any systems that use the estimates. It is therefore important to keep the errors as low as possible. How much error can be tolerated depends on the application, so we don't have an exact error level goal. However, other groups have achieved 18.30° of error in the Yosemite sequence test, so we want to at least match that.

Minimise latency McCarthy and Barnes have shown that, for real-time vehicle control, lower latency is preferable to lower velocity estimation errors [29]. As such, we want to minimise latency as much as possible, with the goal of having no more than 200ms delay between the video input and the motion vector output.

Maximise pixel through-put Many low-cost image sensors produce video with quarter VGA (QVGA) resolutions of 320x240 pixels, at a minimum of 25 frames per second. To process such a video stream in real-time requires the ability to handle at least 1920kpx/s. An increase in processing speed allows higher resolutions and frame rates.

Minimise hardware requirements For a design that is likely to become part of embedded systems, it is important to use as little hardware as possible. This is for both power and money considerations. We will aim to implement

the entire algorithm using only one FPGA and as little support hardware, such as external RAM chips, as possible.

4.2 Algorithm used in the implementation: Uras et al.

Considering the above criteria, we decided on using the algorithm created by Uras et al. This algorithm offers denser results than comparable algorithms, with only a small increase in computational cost and estimate error [9].

As mentioned previously, Uras, Giroi, Verri and Torre [40] use second-order spatio-temporal derivatives of the image sequence to calculate flow velocities. To do so, they assume that the image intensity gradient ∇I moves with a constant velocity. Accounting for sharp changes in flow velocity, they show that the following identity holds for each point in the image:

$$\mathbf{H}\mathbf{v} = -\nabla I_t - \mathbf{M}^T \nabla I \quad (4.1)$$

where \mathbf{H} is the Hessian of the image intensity with respect to x and y coordinates, \mathbf{v} is the velocity vector, ∇I_t is the derivative of the image intensity gradient with respect to time, ∇I is the image intensity gradient and \mathbf{M}^T is the transpose of the matrix of partial derivatives of the velocity,

$$\mathbf{M} = \begin{pmatrix} \frac{\partial v_x}{\partial x} & \frac{\partial v_x}{\partial y} \\ \frac{\partial v_y}{\partial x} & \frac{\partial v_y}{\partial y} \end{pmatrix} \quad (4.2)$$

This says that the velocity of the image gradient, $\mathbf{H}\mathbf{v}$, is equal to the derivative of the image gradient with respect to time, $-\nabla I_t$, after accounting for sharp changes in velocity, $\mathbf{M}^T \nabla I$.

By assuming the flows are uniform over a large region, (4.2) vanishes and from (4.1) they get the simplified equation

$$\mathbf{H}\mathbf{v} = -\nabla I_t \quad (4.3)$$

In expanded form, this is

$$\begin{pmatrix} \frac{\partial^2 I}{\partial x^2} & \frac{\partial^2 I}{\partial x \partial y} \\ \frac{\partial^2 I}{\partial x \partial y} & \frac{\partial^2 I}{\partial y^2} \end{pmatrix} \begin{bmatrix} v_x \\ v_y \end{bmatrix} = \begin{pmatrix} \frac{\partial^2 I}{\partial x \partial t} \\ \frac{\partial^2 I}{\partial y \partial t} \end{pmatrix} \quad (4.4)$$

This is then solved for \mathbf{v} giving

$$\begin{bmatrix} v_x \\ v_y \end{bmatrix} = \frac{1}{\frac{\partial^2 I}{\partial x \partial x} \times \frac{\partial^2 I}{\partial y \partial y} - \frac{\partial^2 I}{\partial x \partial y} \times \frac{\partial^2 I}{\partial x \partial y}} \begin{bmatrix} \frac{\partial^2 I}{\partial y \partial t} \times \frac{\partial^2 I}{\partial x \partial y} - \frac{\partial^2 I}{\partial x \partial t} \times \frac{\partial^2 I}{\partial y \partial y} \\ \frac{\partial^2 I}{\partial x \partial t} \times \frac{\partial^2 I}{\partial x \partial y} - \frac{\partial^2 I}{\partial y \partial t} \times \frac{\partial^2 I}{\partial x \partial x} \end{bmatrix} \quad (4.5)$$

The derivatives are calculated using numerical approximations from the difference of nearby pixels, as in other gradient-based methods. The second derivative is especially prone to noise [21], so they apply Gaussian smoothing in both space and time.

There are a few ways of deriving accuracy estimates. Uras et al. suggested using $\mathbf{M}^T \nabla I$ as a way of eliminating bad estimates, since a large ratio of $\mathbf{M}^T \nabla I$ to $-\nabla I_t$ indicates that the assumption made for (4.3) is false. They

also suggested using the ratio of the maximum and minimum eigenvectors of \mathbf{H} , following a paper from Bertero et al. [11]. Barron et al. suggest using the determinant of \mathbf{H} instead, with results suggesting that it gives a more accurate estimate of reliability [9]. The determinant is found while calculating the velocity so it comes at no extra computational cost. This improves the performance of any implementation using the determinant instead of the other two accuracy estimates.

4.3 Pipeline design for algorithm implementation

The central focus of Uras' algorithm is the velocity vector calculation in equation (4.5). To calculate this requires the calculation of second-order spatio-temporal derivatives of the video input. These derivatives are calculated numerically and are sensitive to noise, so we first apply spatial and temporal filters to improve the results.

Since this design is targeting hardware, numbers are formatted as fixed-point where possible. They consist of an integer and a fractional component. The number of bits assigned to each component remains fixed unless explicitly changed, as may occur between processing stages. Floating-point numbers offer a greater range, but mathematical operations using them are costly in terms of FPGA resources consumed. They are therefore only used where necessary, such as in the result of a division operation.

Each step of the process - filtering the images, calculating derivatives, then calculating the velocity - can be separated into a stand-alone block. These blocks are connected together in the correct sequence, or pipeline, to get the desired result. Figure 4.1 on the next page shows an overview of this pipeline, including data paths and formats. The fixed-point formats specified are those used in the functional simulation used to get the results in Chapter 6

To simplify the connections, a standard interface has been created for the blocks. Each block takes one pixel at a time as input, then outputs the processed results, also one pixel at a time. Since the data stream is not continuous, each block signals to the next when data is being transferred. This extends to the top level, with the FPGA receiving a pixel stream from the camera one pixel at a time and then outputting the velocity estimates one point at a time.

The benefit of a pipeline approach is that each new pixel of data can be processed without waiting for processing to finish on previous pixels. This design allows the greatest throughput, by having each stage working on data as soon as it is received, instead of waiting for the next stage to be ready for more data input. It also minimises memory use, by not requiring data buffers between stages. However, it relies on image data arriving from the camera no faster than the slowest pipeline stage can process it. The overall pixel through-put is therefore determined by the speed of the slowest block.

There are only a few types of block used in the design. These are: spatial convolution, time filtering, and velocity calculation.

4.3.1 Spatial convolution

Spatial convolution is used for both the spatial filtering and derivatives. In each case a different convolution kernel was used. For filtering, a 1D Gaussian kernel

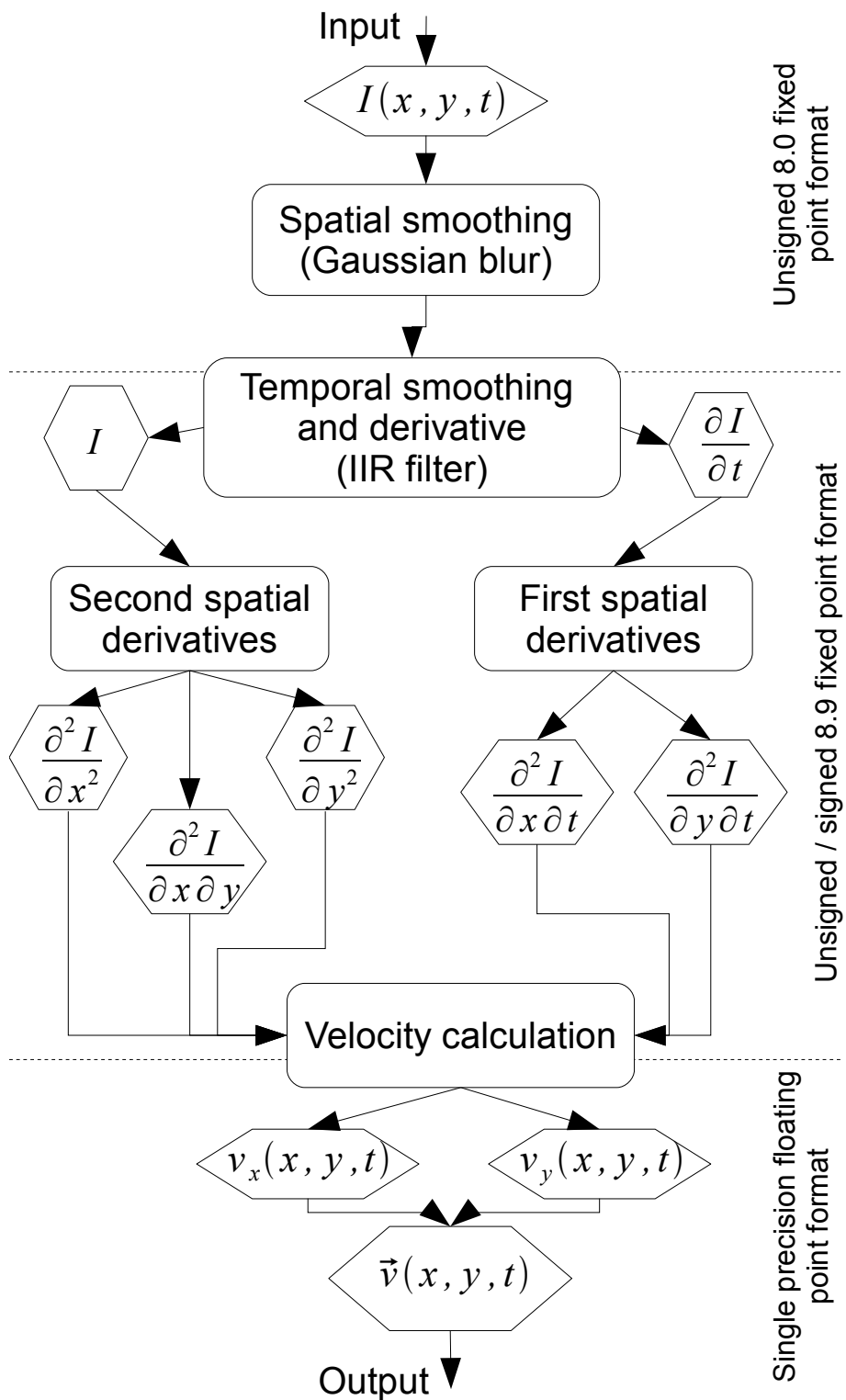


Figure 4.1: Pipeline over view - Pixels are pushed into the top, one at a time, and velocity estimates are pushed out the bottom, one pixel's worth at a time.

Table 4.1: Gaussian kernel

1	2	4	7	11	14	16	17	16	14	11	7	4	2	1
---	---	---	---	----	----	----	----	----	----	----	---	---	---	---

Table 4.2: First-order central difference

-1	8	0	-8	1
----	---	---	----	---

Table 4.3: Second-order central difference

-1	16	-30	16	-1
----	----	-----	----	----

is applied in the vertical and horizontal directions. It has a standard deviation of three pixels, and is stored in 0.7 fixed-point format. Table 4.1 on page 25 shows integer constants which are divided by 128 to obtain the kernel.

For derivatives, five-point central difference kernels are used. For a true central difference, the values in tables 4.2 and 4.3 would be divided by 12. However, the factor of 12 cancels out in the division in the velocity calculation. Resources can be saved by not having to divide during the derivative calculation.

Figure 4.2 on page 26 shows how the convolution block operates. The convolution block applies a vertical convolution then a horizontal convolution. To do the vertical convolution, pixels are read sequentially into a line buffer. The line buffer is wide enough to hold a full row's width of pixels, and is as tall as the length of the convolution kernel. As each row is filled, it is shifted down, and the next row is read in. The active column, where the new pixel is stored, is advanced with each pixel inputted. Figure 4.3 on page 27 shows this process.

As each new pixel comes into the line buffer, the entire active column is read out. Each pixel from this column is multiplied by the corresponding constant in the vertical convolution kernel then those results are summed and stored in a short column result buffer. This buffer is as wide as the horizontal convolution kernel, but only one row tall. Each new pixel stored shifts the other pixels along. Once this buffer is filled, the horizontal convolution takes in the same manner as the vertical convolution. The results are then output to the next stage.

The line buffer is reset at the start of each frame, and the column result buffer is reset at the start of each row. This ensures that pixels from previous frames and rows do not affect the results when it is not desired. Resulting values are kept in the same format as the input values. Output frames are shorter and narrower than the input, since only the valid region of the result from the convolution is taken. They are reduced by the kernel size - 1 in each direction. For example, a 320x240 input frame convoluted with a 15-long kernel in each direction would result in a 306x226 output frame.

4.3.2 Temporal filtering

Temporal filtering is done using an Infinite Impulse Response (IIR) filter that was described by Fleet and Langley [19]. Figure 4.4 on page 28 shows the structure of the filter, which is made from a combination of adders, constant multipliers and delay blocks. The equation for this filter in the z-domain is given by [19]:

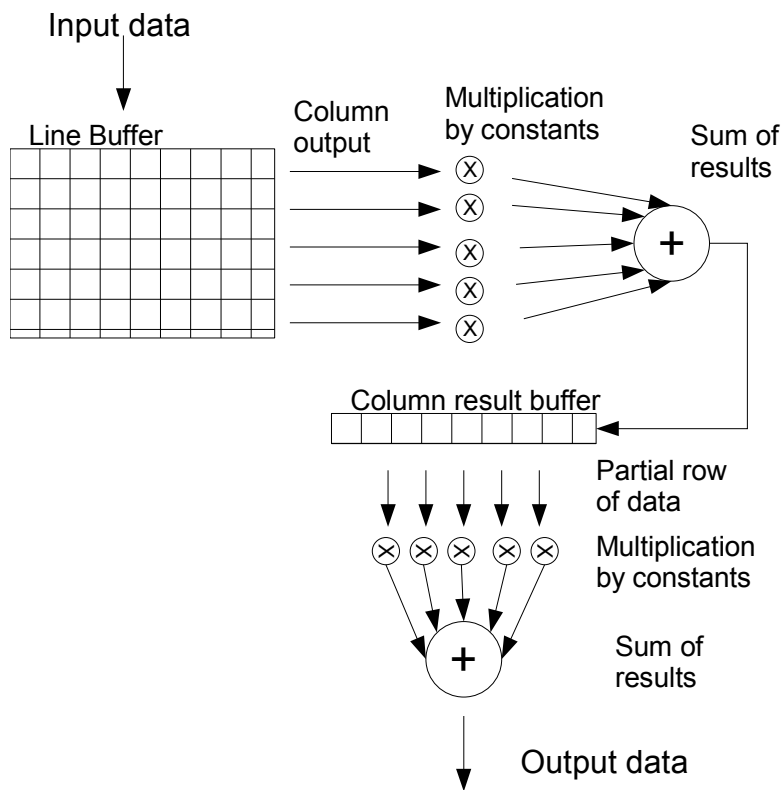


Figure 4.2: Convolution block operation: Data is read into a line buffer. Each pixel from the active column in the line buffer is multiplied by a constant factor from the vertical convolution mask, then the results are summed and stored into a shift register buffer. Each cell in the shift register is simultaneously multiplied by a constant factor from the horizontal convolution mask. The results are then summed and outputted from the block.

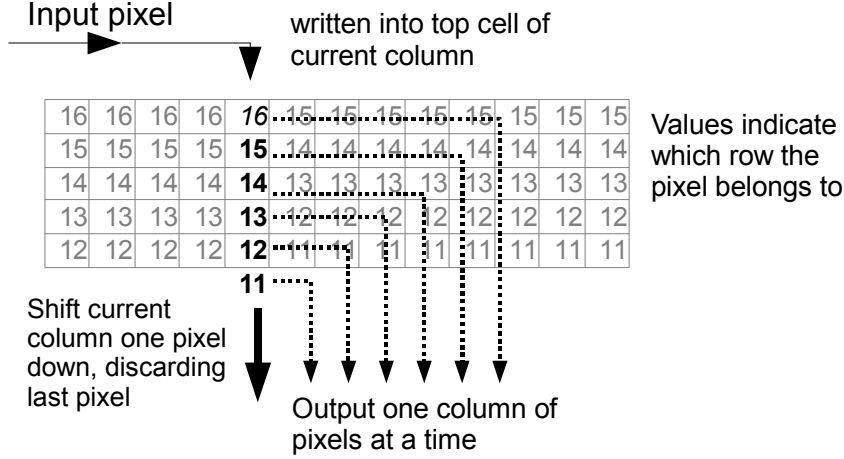


Figure 4.3: Operation of line buffer memory: Pixels are read into the top row, moving across by one column each time. Every time a new pixel is added to a column, the pixels already there are shifted down one row. All pixels in the active column are outputted simultaneously.

$$H_3(z) = q^3 \left(\frac{1 + 2z^{-1} + z^{-2}}{1 + 2rz^{-1} + r^2z^{-2}} \right) \left(\frac{1 + z^{-1}}{1 + rz^{-1}} \right)$$

I have chosen a filter with a time constant of $\tau^{-1} = 1.25$. The other parameters are calculated from this value: $q = \frac{\tau}{\tau+2}$ and $r = \frac{\tau-2}{\tau+2}$. These constants were chosen because they offer the best filtering results with only three delay stages.

The format of the values stored in the delay stages and those output to the next block are kept the same, for simplicity and accuracy. More fractional bits are used than in previous blocks because numerical errors can accumulate in the delay stages.

Each delay requires the storage of a full frame's worth of data and adds one frame of latency to the pipeline. It is for this reason that the number of delays was kept to a minimum. A finite impulse response filter consisting of a Gaussian blur and five-point central difference can add a latency of more than twelve frames. This is another reason that an IIR filter is desirable.

4.3.3 Velocity calculation

Velocity calculation is done using (4.5), with the input derivatives taken from the previous pipeline blocks. The multiplications and subtractions are done in fixed-point, with double the precision of the input data. This ensures that no truncation occurs. The divisions are performed in floating-point, to allow for the larger range in magnitudes of the results. Two divisions are required, one each for the horizontal and vertical velocity components.

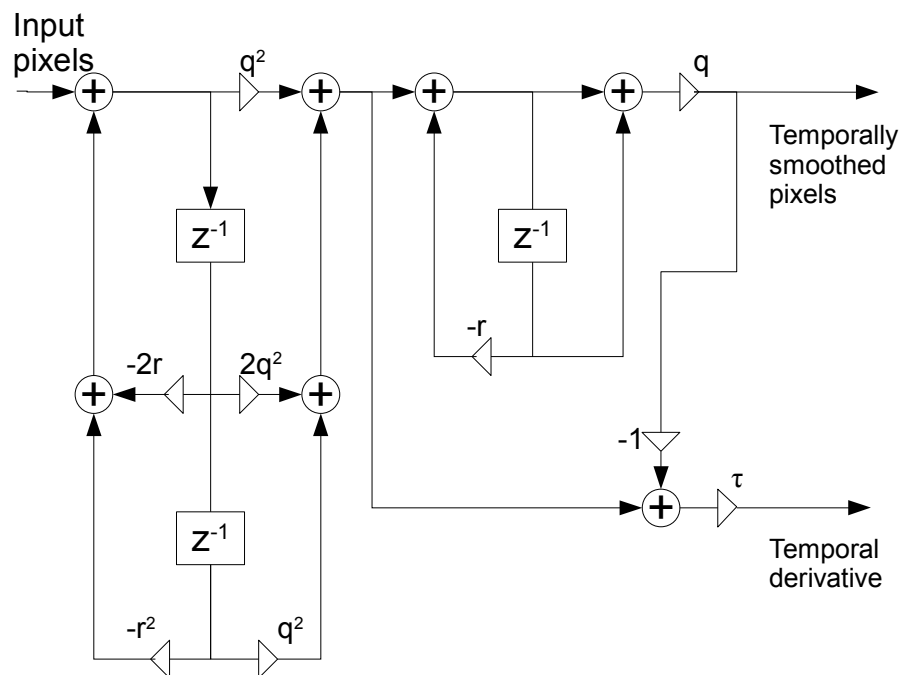


Figure 4.4: Temporal filter structure, adapted from a paper by Fleet & Langley[19]: One pixels is inputted at a time, with the temporal derivative and smoothed pixels being outputted three frames later. The filter is constructed from a network of adders, multipliers and frame delays.

Once the divisions are complete, the velocity has been fully calculated. It can either be processed further to minimise errors in the flow field estimates, or it can be output as-is, giving a 100% flow field density. This design will output as-is, reducing computational complexity. The output of the velocity therefore marks the end of the pipeline.

Chapter 5

Implementation

There were two main goals for the implementation: to prove that the algorithm worked, and to show that it would work on an FPGA. Proving the algorithm works was done by creating a functional simulation on a PC. Various components of the pipeline were implemented using an FPGA, to show that it would work in hardware.

5.1 FPGA Development System

The FPGA development system consisted of software used to write and debug the FPGA code, and a hardware platform to test it on.

5.1.1 Nexys-2 development board

The hardware platform was a Nexys-2 development board from Digilent Inc., shown in Figure 5.1. It has:

- A Xilinx Spartan-3E 1200K FPGA,
- 16MB of pseudo static RAM (PSRAM),
- 16MB of flash memory,
- a USB interface used for programming and data communications, and
- various buttons and LEDs for interaction with a human.

The FPGA contains 17344 look-up tables (LUTs), 17344 flip-flops (FFs), 28 dedicated 18x18 integer multipliers, and 640K bits of memory. The LUTs and FFs are used to form the device's logic, with the multipliers supplementing them for multiplication tasks. The memory is divided into various blocks, and some blocks share interfaces with some multipliers. In each of these cases, either a multiplier or a memory block can be used, but not both simultaneously. The FPGA runs at a default clock rate of 50MHz.

The USB interface takes the place of a JTAG connection for uploading programs to the FPGA or flash memory. It also allows transfer of data one byte at a time to and from the logic running on the FPGA. The data transfer is useful for debugging purposes.

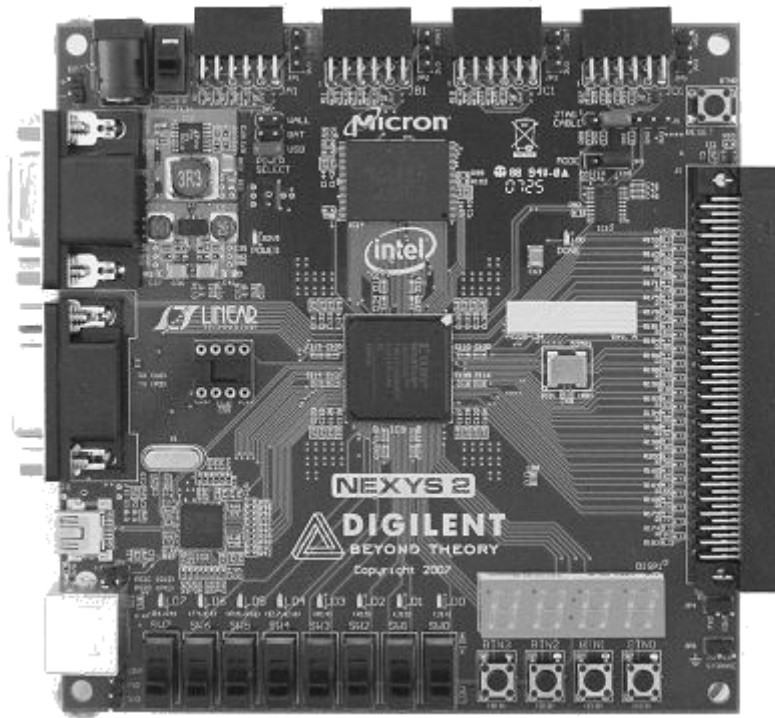


Figure 5.1: Nexys-2 Development board

The flash memory can be used to store programming files that get loaded into the FPGA every time it is turned on, removing the need to program it via USB. It can also be read and written directly from the FPGA, to act as long-term data storage.

The PSRAM is accessed via a 16-bit data bus and a 20-bit address bus. It supplements the FPGA's limited internal memory, but at the cost of speed.

5.1.2 Software tools

FPGA code was written in Verilog. It is a high level language that is synthesised into programming files used to reconfigure the FPGA. **Xilinx ISE 9.2** was used as the IDE. It provides editors, compilers and simulators to assist in creating the FPGA code.

Verilog modules were simulated using **ISE Simulator**. This involves writing a Verilog test fixture to wrap around the logic module. The test fixtures are written to provide input and record the output from the module. The resulting waveforms are displayed after the simulation is complete, providing visual feedback for debugging purposes. Figure 5.2 shows an example waveform simulation.

The **Digilent Adept** suite was used for all USB communications with the FPGA. Programming the FPGA involved using the **Adept ExPort** tool to send .bit files to the board. Test data was sent and received using the **Adept TransPort** tool. It allows individual bytes and complete or partial files to be

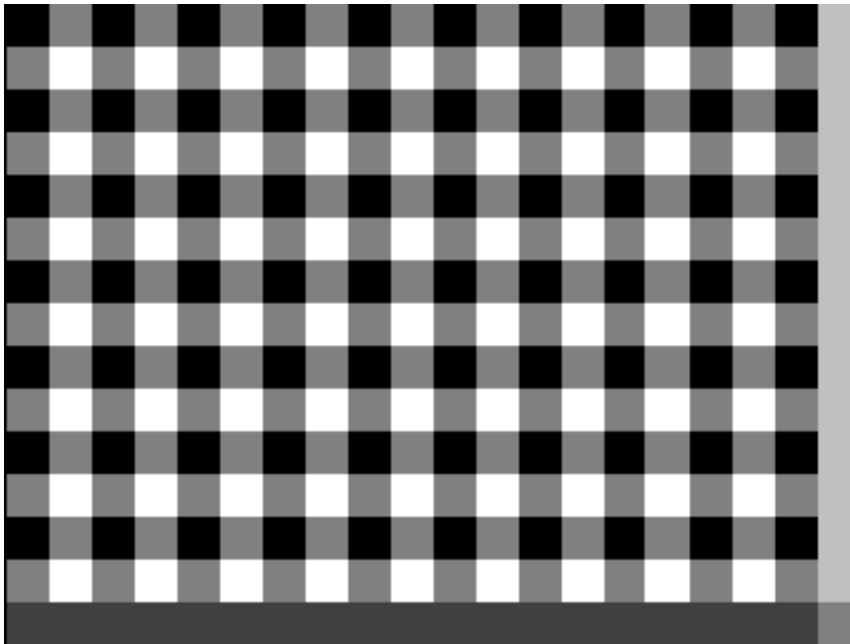


Figure 5.3: Example test image for Gaussian spatial blur function

For input and output to the block, the USB connection is used. Input data is passed immediately into the spatial blur processing block. Results are stored in a FIFO buffer until they are read back by the PC. The FPGA only has enough internal memory to buffer half of the results at a time, so half of the source image is sent to the FPGA, the results are read back, then the other half is sent to the FPGA and the rest of the results are read. Figure 5.4 on page 34 shows the output from the convolution block when Figure 5.3 on page 33 is used as input.

5.2.2 External memory access

In the final hardware implementation, external memory will only be needed for the delay stages of the temporal filter. However, its speed could become a bottle-neck for the entire system. To test its speed, a small test module was written.

Data is transferred either to or from external memory 16 bits at a time. Using asynchronous accesses, one 16-bit transfer can be completed successfully without corruption every 4 clock cycles. Using synchronous burst mode, 128 16-bit words can be transferred over the course of 132 cycles. This is the maximum speed that the external memory can handle, as confirmed by its datasheet [32].

The IIR temporal filter requires 17 bits to be transferred in and out per pixel stored, for each pixel processed. Since there are three delay stages, the filter requires 102 bits of data to be transferred for each new pixel processed. Taking into account the memory transfer speeds, the IIR filter has a theoretical maximum throughput of approximately 7600kpx/s.

The maximum throughput can be increased by increasing the

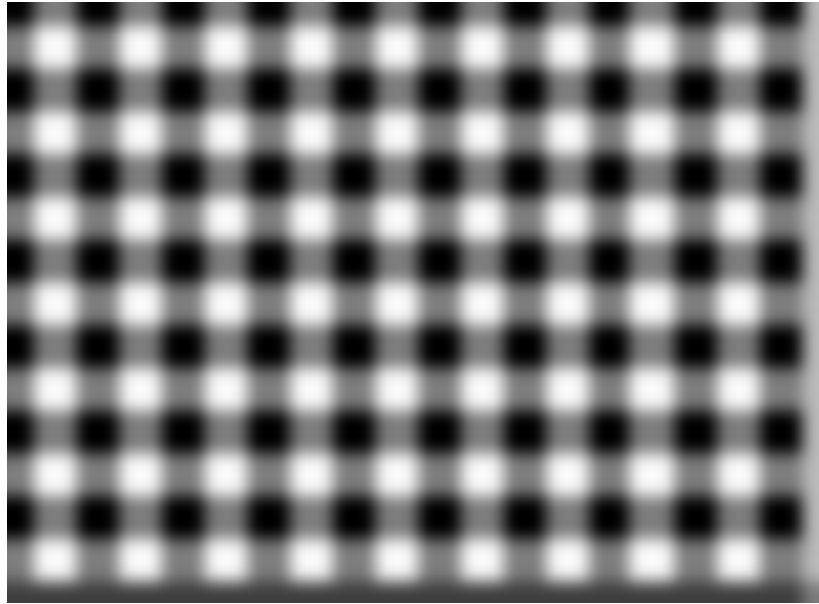


Figure 5.4: Example output from Gaussian spatial blur

width of the external memory bus. This would require a new board layout with multiple PSRAM chips running in parallel or a different RAM chip with a higher bandwidth. Designing a new board is very time-consuming, so this would only be done if the current throughput is too low.

5.2.3 PC communications

A generic PC communication block was also made. Although this is not needed for the final design, it is used while testing other blocks in the pipeline. The USB connection is made to the FPGA using an 8-bit data bus with additional lines for handshaking. The communication protocol is very sensitive to timing, so great care had to be taken when writing the communication block. Eventually a finite-state machine was created to handle the PC/FPGA communications. This block was used when testing the external memory access and the spatial blur block.

5.2.4 Resources consumed

The largest component of the entire pipeline is the floating-point divider. The velocity calculation requires three conversions from fixed-point to floating-point, then two floating-point divisions. This takes a total of 3421 FFs and 2311 LUTs, which is approximately 20% of the FPGA's resources.

The spatial blur needs 722 FFs, 864 LUTs and 35Kbits of memory. Even if each of the five spatial derivatives requires its own convolution block with the same kernel size as the spatial blur, that is a total of 4332 FFs, 5184 LUTs, and 210Kbits of memory consumed.

The external RAM interface needs 128 FFs and 311 LUTs. The IIR filter

can't use the FPGA's memory for the delay stages because at least three 306x226 frames of 17 bits per pixel are required for the delay stages. This would require 3444Kbits of memory, which the FPGA does not have.

The velocity calculation requires 6 multipliers. The rest of the IIR filter and the velocity blocks will mostly be adders, which consume only a small number of LUTs and FFs.

In total, around 8000 FFs and an equal number of LUTs will be used, along with 210Kbits of memory and 6 multipliers. This is less than half of the total resources available on the FPGA, so Uras' optical flow algorithm will fit.

The largest bottle-neck is caused by the memory transfers in the IIR filter. This limits the processing speed to approximately 7600kpx/s. QVGA video at 60fps needs a throughput of 4608kpx/s, so the implementation will be able to handle QVGA video without problems.

The IIR filter also introduces the largest latency, with three frames of storage causing three frames of latency. The convolution blocks each only contribute a few lines worth of delay, while the division contributes 28 pixels of latency. Therefore there will be less than four frames of latency between input and output.

5.3 Functionally equivalent simulation

A functionally equivalent PC-based simulation was created to determine whether the pipeline design of Uras' algorithm would work. The simulation was written in C as a command line program. Running it on the PC had a number of benefits.

Debugging is greatly simplified by the availability of tools such as **gdb**. The logic in an FPGA can only be debugged using the inputs and outputs of the module that is running. On a PC, **gdb** allows the setting of code break points, probing of internal memory, tracing of program execution, and various other debugging techniques. This allowed data within the pipeline to be examined during execution, and not just the output after it had completed.

C code is more flexible than Verilog. For example, in the C code, the kernel used for the Gaussian blur can be modified simply by changing the constants stored in an array. In the Verilog code, the series of bit-shift and addition operations has to be rewritten to change each value. The PC-based simulation was used to test various parameters before committing to a final, fixed design for the FPGA.

The PC program was configured to give output in a variety of formats. Some were used to visualise the flows for a human to qualitatively check the results. Other formats were used to compare against the expected results from a variety of test cases and quantify the errors in the velocity flows produced.

5.3.1 Temporal filter data format optimisation

Using 8.9 fixed-point format in the IIR temporal filter is a result of testing the output for various fraction bit widths. A static video frame was passed through the filter for a number of processing steps, until the filter results converged. The video frame has half black and half white. This tested two things: that the filter

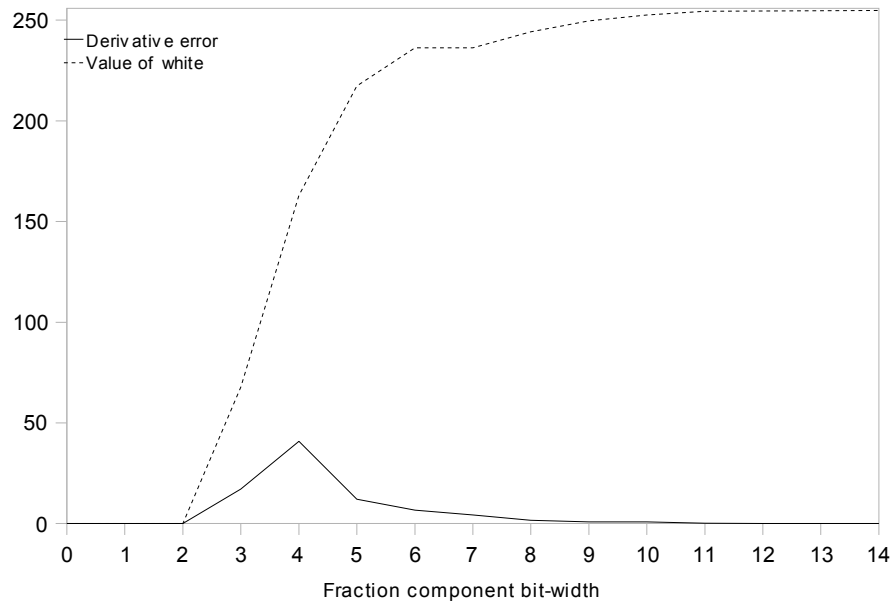


Figure 5.5: The output value of white and the error in the temporal derivative versus the IIR filter's value representation's fractional component bit-width

would eventually converge, and that the time derivative produced by the filter was zero at convergence.

Various bit widths were tested for the filter. Figure 5.5 on page 36 shows the outcome of these tests. As can be seen, the derivative error drops below 1 when at least nine fractional bits are used, and the value of white converges towards 255. Further testing of velocity calculation errors showed that there was no benefit gained by using more than 9 fractional bits.

5.3.2 Velocity post-processing techniques

Various velocity post-processing techniques were trialled using the PC simulation, with the aim of improving the velocity error. Three techniques were tried: thresholding the determinant of the Hessian matrix above one (that is, $\det(\mathbf{H}) > 1$), minimising the ratio of the eigenvalues of the Hessian matrix, and checking the change in velocity to discharge the assumption of a smooth flow field. Trialling them using the PC simulation allowed the use of square root and division operations without concern for the hardware resources consumed.

Thresholding the determinant can be done with only a small computational cost. The determinant is already calculated during the velocity calculations, and becomes the divisor in the divisions. When the determinant is very small, the divisions will give a very large, incorrect result. Barron et al. have also suggested that it is more reliable than the ratio of eigenvalues [9]. Testing this, however, showed almost no reduction in the velocity error and resulted in a halving of the flow field density.

Uras et al. have suggested minimising the ratio of the eigenvalues of the Hessian matrix. This technique was previously discussed by Bertero, Poggio

and Torre [11]. The ratio of eigenvalues is known as the condition number of the matrix, and a value close to one indicates that the matrix is well conditioned. For the two-by-two matrix, the eigenvalues are calculated by:

$$\lambda_{1,2} = \frac{1}{2} * \left(\frac{\partial^2 I}{\partial x \partial x} + \frac{\partial^2 I}{\partial y \partial y} \pm \sqrt{\left(\frac{\partial^2 I}{\partial x \partial x} - \frac{\partial^2 I}{\partial y \partial y} \right)^2 + 4 \frac{\partial^2 I}{\partial x \partial y}} \right)$$

This calculation requires a square-root operation, which is costly in terms of FPGA area consumed. This technique eliminated some erroneous velocity values, but left others. It also resulted in a patchy flow-field, due to many good velocity values being eliminated too. Decreasing the threshold for the eigenvalue ratio increases the number of erroneous velocity values eliminated, but also increases the number of good vectors eliminated. Overall, this technique had little effect on average angular error.

Finally, Uras et al. also suggested checking the calculated velocity against the original images, to ensure that the predicted position of the pixels is accurate. This is done by calculating the ratio:

$$\Delta = \frac{\|\mathbf{M}^T \nabla \mathbf{I}\|}{\|\nabla \frac{\partial \mathbf{I}}{\partial \mathbf{t}}\|}$$

where \mathbf{M} is the Jacobian matrix of the velocity. This ratio evaluates to:

$$\Delta = \sqrt{\frac{\left(\frac{\partial v_x}{\partial x} \times \frac{\partial I}{\partial x} + \frac{\partial v_y}{\partial x} \times \frac{\partial I}{\partial y} \right)^2 + \left(\frac{\partial v_x}{\partial y} \times \frac{\partial I}{\partial x} + \frac{\partial v_y}{\partial y} \times \frac{\partial I}{\partial y} \right)^2}{\frac{\partial^2 I}{\partial x \partial t}^2 + \frac{\partial^2 I}{\partial y \partial t}^2}}$$

The square root can be eliminated by squaring both sides, but this calculation is still costly. It requires the first derivatives of the velocity to be calculated, and also requires a division. Varying the threshold on Δ had no effect on the average angular error.

None of the techniques successfully reduced the average angular error by a useful margin. Therefore the pipeline is kept as it is, and will produce output with 100% density.

Chapter 6

Results and evaluation

The PC simulation produces results that can be analysed and compared against other implementations. To do this, a standard set of test image-sequences were used. They are the same sequences used by Barron et al. in their comparison of various optical flow techniques, and have also been used to test various other implementations.

6.1 Test sequences

The test sequences range from a very simple translating sinusoid pattern to a complicated fly-through of a virtual valley. Note that all of the velocity vector (flow) fields have been sub-sampled to make them easier to see in print.

6.1.1 Sinusoids

The sinusoids are the least complicated sequences to extract motion from. The images have no sharp changes in brightness, no occlusion boundaries, and a constant velocity in space and time. A 2-dimensional sinusoid texture translates with constant velocity to the top right of the screen. Figure 6.1 shows the first sinusoid texture used and flow field expected. Figure 6.2 shows the same for the second sinusoid.

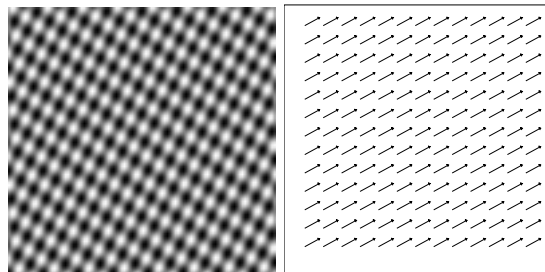


Figure 6.1: Sinusoid-1 texture and expected velocity vector field

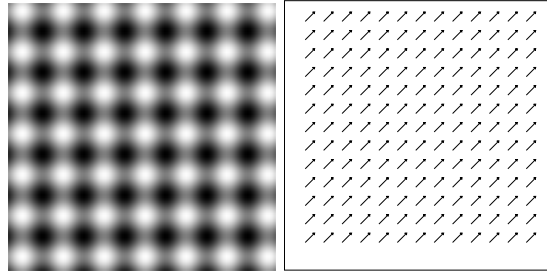


Figure 6.2: Sinusoid-2 texture and expected velocity vector field

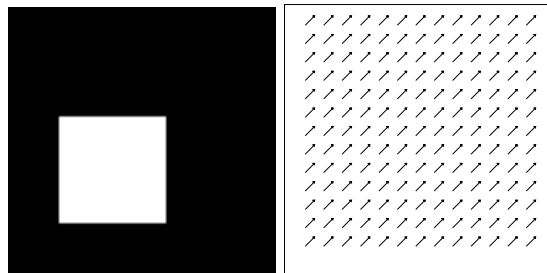


Figure 6.3: Square-1 texture and expected velocity vector field

6.1.2 Squares

The square test cases increase the complexity of the scene with sharp changes in brightness and untextured surfaces. Figure 6.3 shows one of the square test cases. Although the source flow is given as constant across the entire image, the perceived flow will only appear around the edges of the square, where there is enough detail to differentiate between pixels that have changed position.

6.1.3 Trees

The tree test cases use a texture from an actual photograph. There are no repeating patterns in the image, and it has a complicated texture that aids with determining flow. The texture is used for two different flows. One is a diverging flow where the texture comes towards the camera. The second is a translating flow where the texture moves to the right. Figure 6.5 shows the texture and the two flows. Note that it is only the 2D texture that zooms or moves, the visible objects do not move relative to each other.

6.1.4 Yosemite

The Yosemite sequence consists of a fly-through of a virtual valley with a cloudy sky. It is the most complicated scene of all the test cases, since it not only has detailed textures, non-constant flow and sharp changes in brightness, but it also has occlusion between the top of the valley and the sky.

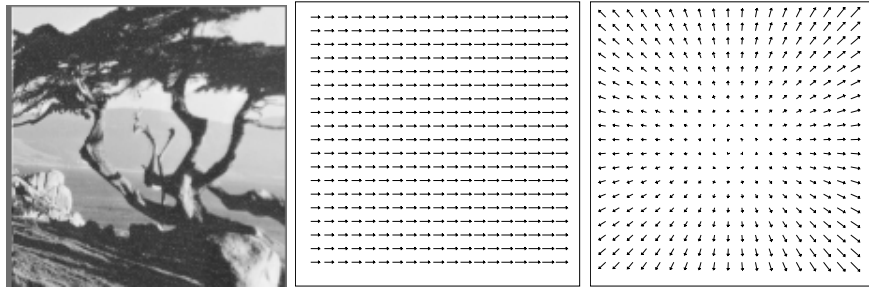


Figure 6.4: Tree texture and expected velocity vector fields for translation and divergence

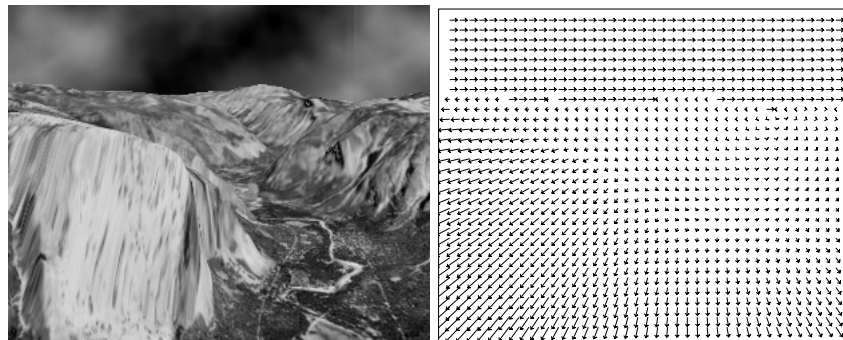


Figure 6.5: The 16th image in the Yosemite sequence and its expected flow

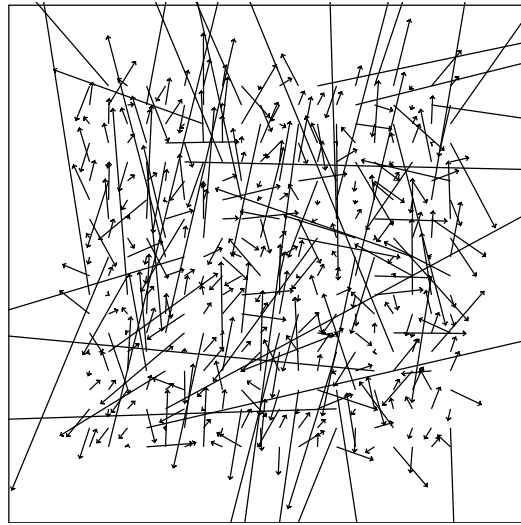


Figure 6.6: Sinusoid-1 result

6.2 Resulting test sequence flow fields

These are the velocity vector (flow) fields produced from the various test sequences. They have 100% density because no velocity post-processing techniques were used.

6.2.1 Sinusoid-1

Figure 6.6 shows the resulting flow field when the Sinusoid-1 test sequence is passed through the PC implementation of the optical flow pipeline. As can be seen, the error in the vectors is very large, with the results having no relation to the expected flow field. There is an average angular error of 66° in this flow field, confirming that it is a poor result.

The poor result is due to the small spacing of the peaks in the sinusoidal texture which, when spatially and temporally filtered, gives a flat texture. This is seen in Figure 6.7(b), which shows the image after spatial and temporal filtering. It is completely smooth, and so all second derivatives taken from that image will be zero. However, the time derivative image is not completely smooth, so the first-order spacial derivatives taken on it will be non-zero. A combination of incorrectly zero spatial derivatives combined with correctly non-zero spatio-temporal derivatives gives the poor result seen in the flow field. To fix this, the standard deviation of the Gaussian kernel used for spatial filtering would need to be reduced, but this can cause errors for textures with larger spacing but sharper changes in brightness.

6.2.2 Sinusoid-2

Figure 6.8 shows the resulting flow field with the Sinusoid-2 test sequence. It is mostly correct, with an average angular error of only 12.4° . This improvement over Sinusoid-1 is due to the contrast that still remains in the spatio-temporally

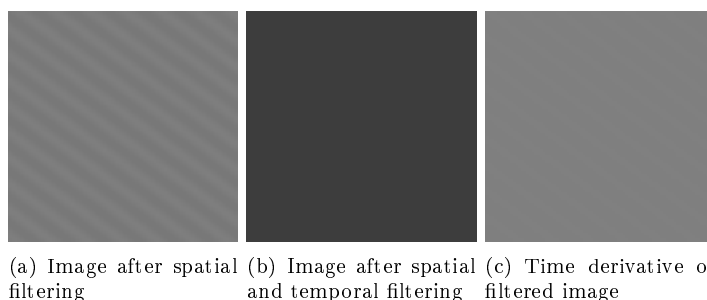


Figure 6.7: Sinusoid-1 results from intermediate stages of pipeline

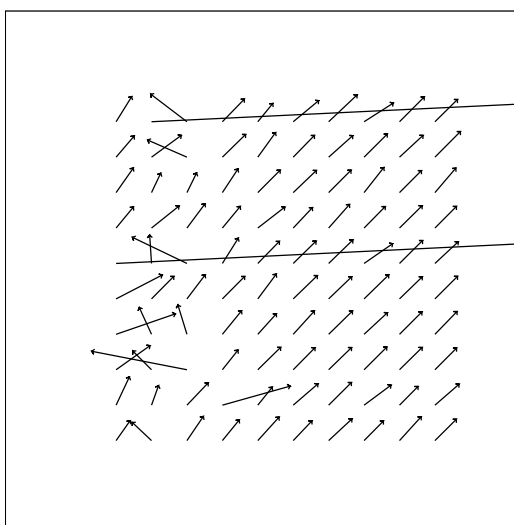


Figure 6.8: Sinusoid-2 result

filtered image, which is shown in Figure 6.9(b). The remaining errors occur along the low-brightness troughs of the image. This cannot be seen in Figure 6.8 due to the effects of sub-sampling, but the errors in the flow are arranged like the troughs of the original image.

6.2.3 Squares

The flow resulting from the Square-1 sequence, shown in Figure 6.10, is a good demonstration of the aperture problem. Only flows around the corners are able to be calculated, with the rest of the vector field being zero. The centre of the square and the background has no contrast to extract flow from. The edges of the square only have contrast along one axis, so accurate flow cannot be extracted there either.

There is an average angular error of 21.5° in this result. Most of the error occurs in the regions at the boundary of the well-defined corners and the contrast-less background. The flow is picked up by the algorithm, but there is not enough information for it to calculate the flow correctly. Applying any of the velocity post-processing techniques did not improve this. In a real-life

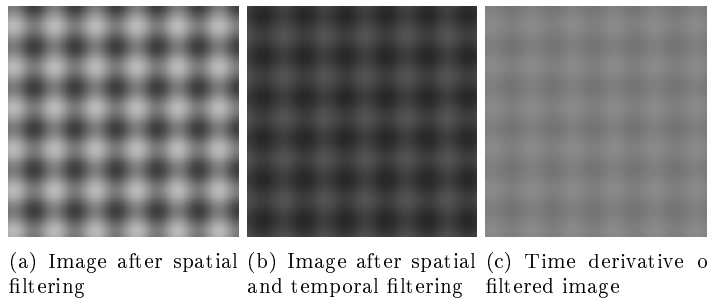


Figure 6.9: Sinusoid-2 results from intermediate stages of pipeline

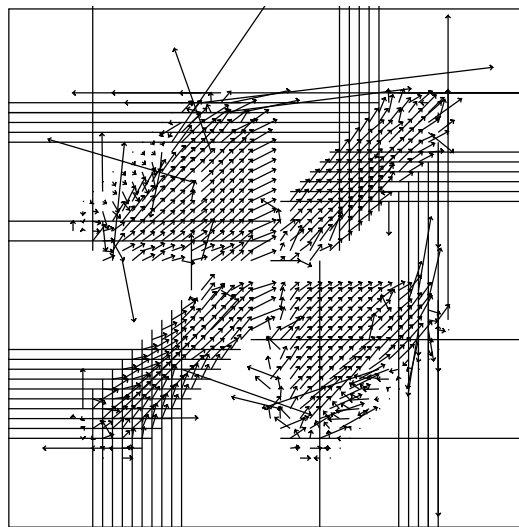


Figure 6.10: Square-1 result

scene there would normally be enough texture on objects for this problem not to occur, so it is not completely detrimental to what is required of the final implementation.

6.2.4 Trees

The results for the tree texture are shown in Figure 6.11. The translating sequence had significantly more error with 39.9° , than the diverging sequence with 26.1° . That this occurs despite the same texture being used for both sequences shows that the algorithm works better in some situations than others. Many of the errors in the translating sequence are simply vectors pointing backwards, indicating a reversal in sign at some stage of the pipeline, probably due to the sensitivity of the five-point central difference used to calculate the derivative. That, combined with the motion blur effects of the temporal filter, can have the effect seen.

The diverging sequence has mostly magnitude errors, indicating the denominator in the division stage is smaller than expected. This may be improved by increasing the number bits assigned to the fractional component of values

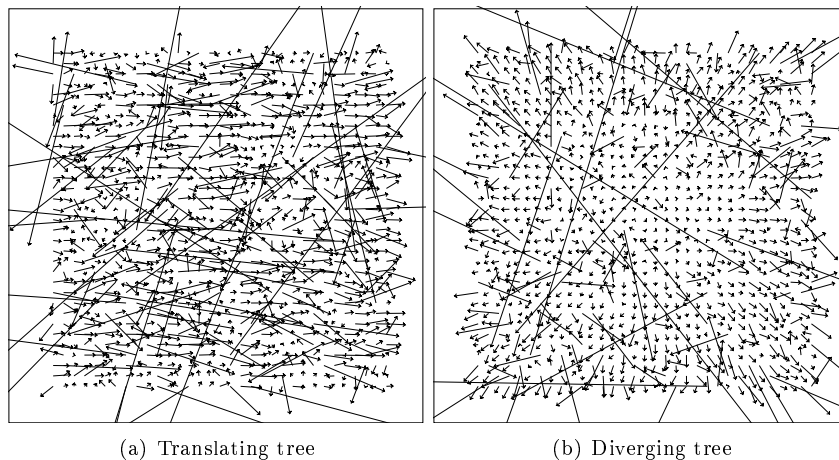


Figure 6.11: Tree results

throughout the pipeline, increase the accuracy of values that have a small magnitude.

6.2.5 Yosemite

As seen in Figure 6.12, the Yosemite sequence results have many errors. The average angular error is 47.7° . Much of this error occurs in the sky and in the left side of the valley. Notice that in Figure 6.13 the texture on the left of the valley in the spatio-temporal filtered image has almost no contrast. This is the same problem encountered with Sinusoid-1.

The sky is at an occlusion boundary, and the results show the same effects as occur in Square-1. In addition to this, the texture has a low spatial frequency, so the second-order spatial derivatives are close to zero in many patches of the clouds.

The valley in the middle and right side has a good texture and almost no occlusion, giving the good results seen.

6.3 Numerical error results

Table 6.1 shows the error results obtained for each of the test sequences, and the results obtained by Barron et al. for their purely floating-point implementations [9]. As can be seen, our implementation does poorly by comparison.

The Yosemite sequence has been used by other authors to test their FPGA-based implementations of optical flow algorithms, as previously discussed in Section 3.2. Wei et al. achieved an average angular error of 12.9° with a standard deviation of 17.6° using a 3D tensor approach [42]. Díaz et al. achieved an average angular error of 18.30° with a standard deviation of 15.8° using an implementation of Lucas and Kanade's algorithm [16, 17].

This shows that our implementation needs improving. The area where it differs most from other implementations is in the use of the Infinite Impulse Response (IIR) filter. The IIR temporal filter can be replaced with a finite impulse

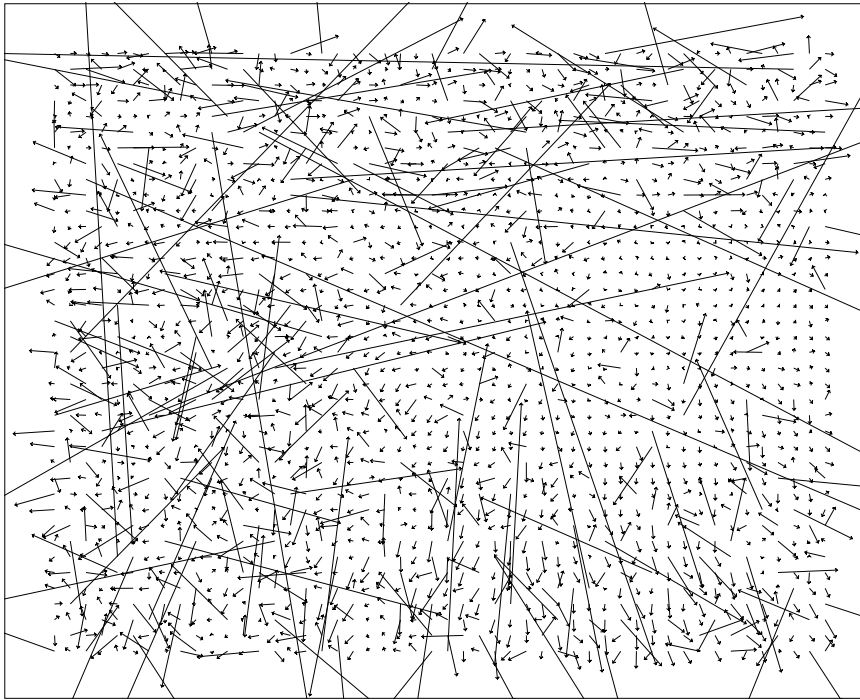


Figure 6.12: Yosemite result



Figure 6.13: Frame 16 of the Yosemite sequence after spatial and temporal filtering

Table 6.1: Angular error for various sequences, and the results obtained by Barron et al.[9] Some results were unavailable.

Sequence	Average error (°)	Standard deviation (°)	Barron et al. - Average error (°)	Barron et al. - Standard deviation (°)
Sinusoid-1	66.03	31.27	2.59	0.71
Sinusoid-2	12.38	16.25	—	—
Square-1	21.52	31.18	—	—
Square-2	21.85	30.94	0.15	0.10
Translating tree	39.92	38.55	0.62	0.52
Diverging tree	26.75	26.11	4.64	3.48
Yosemite	49.41	39.98	8.94	15.61

response filter based on a Gaussian blur and five-point central difference. This comes at the cost of increased FPGA resources used, but it may be necessary to achieve the desired accuracy.

Other areas where our implementation differs is in the use and precision of fixed-point numbers, the standard deviation used in the spatial blur, and in the period used in the IIR filter. The next step is to therefore increase the precision of the fixed-point numbers used and try floating-point numbers at an earlier stage of the pipeline.

All of these parameters can be altered in the PC-based simulation. Each variation will need an appropriate amount of time to ensure it is properly tested. This time must be invested, however, to ensure that the final implementation gives the best accuracy that it possibly can.

Chapter 7

Conclusions and future work

Uras' algorithm is a good choice for implementation on an FPGA. The velocity calculation can be broken down into pipeline stages that are implementable in hardware without consuming too many resources. By implementing a convolution block, an external memory block and a division block, I have shown that the entire algorithm will fit within the constraints of the Spartan-3E 1200K FPGA.

I have also shown that the algorithm can be fully pipelined, with the external memory being the limiting factor on speed. Using the PSRAM chip on the Nexys-2 development board limits the pipeline's processing speed to 7600kpx/s. This bottleneck can be removed by using a different board design that allows a greater transfer rate between external memory and the FPGA. The only other limiting factor is the FPGA's clock rate. The Spartan-3E on the Nexys-2 board is clocked at 50MHz. This gives a potential processing rate of 50000px/s, which is enough to process VGA resolution video at over 160 frames per second. With careful design, the FPGA's clock rate can be further increased as necessary.

There is less than four frames of latency between input and output. This improves the reaction time of the device making use of optical flow as compared to some designs that can introduce a latency of twelve frames or more.

The functionally-equivalent PC implementation shows that the algorithm does work, producing a velocity vector field that is related to the input image sequence. The field has 100% density, giving velocity estimates for every pixel within the valid region of the frame. This gives the downstream processor as much data as possible about the scene, so that it can extract elements of the flow as it requires. However, the accuracy can be greatly improved, as shown by Barron's floating-point-based implementation. This can be done by varying the spatial and temporal filters' parameters, increasing the bit-width of the fixed-point variables used in calculations or replacing them with floating-point numbers, and replacing the infinite impulse response filter with a finite impulse response temporal filter.

Once the functionally-equivalent PC implementation produces output with an acceptable error rate, the next step is to fully implement the pipeline using the development board. The development board allows debugging through its USB interface, and the Verilog code can easily be transferred to a different hardware platform when it is complete. A camera can be attached to one of the many ports available on the development board, and results output through

another.

The last step is to design a PCB that contains only the parts required: an FPGA, additional memory, a ROM to store the code, and interfaces to the camera and downstream processors. At this stage, it will be ready to embed into a range of devices, such as unmanned aerial vehicles, cars, and robots. We will thereby achieve the goal of providing them with extra data about their environment that can be used to behave more intelligently.

Uras' algorithm will work well on an FPGA, and with further improvement it will provide dense, accurate velocity vector fields that can be used to improve the capabilities of embedded devices.

Appendix A

Timetable

Table A.1: Planned Timetable

Weeks	Activities
1-3	Background research Requirements specification
4	Initial 5 minute presentation
5-7	Write draft literature review Begin design implementation
Mid-semester Break	Finalise literature review Further implementation
8-11	Refine implementation Test various optimisations
9	Write outline of final report
12-13	Final design and results Write draft report
13	Final 25 minute presentation
14	Finish and submit report

Table A.2: Actual Timetable

Weeks	Activities
1-6	Background research
3	Requirements specification
4	Initial 5 minute presentation
5-7	Write draft literature review
6-8	Design optical flow pipeline and sub-blocks
Mid-semester Break	Finalise literature review
7-11	FPGA implementation of selected sub-blocks
12-13	PC implementation
13-14	Write draft report
13	Final 25 minute presentation
14	Finish and submit report

Bibliography

- [1] Fuad Abazovic. G70 Geforce 7800 GTX final specs out. <http://www.theinquirer.net/en/inquirer/news/2005/06/15/g70-geforce-7800-gtx-final-specs-out>, June 15 2005.
- [2] M. D. Abràmoff, W. J. Niessen, and M. A. Viergever. Objective quantification of the motion of soft tissues: an application to orbital soft tissue motion. *IEEE Transactions on Medical Imaging*, 19(10):986–995, 2000.
- [3] Edward H. Adelson and James R. Bergen. Spatiotemporal energy models for the perception of motion. *J. of the Optical Society of America A*, 2(2):284–299, 1985.
- [4] Altera. APEX 20K programmable logic device family. Datasheet, February 2002.
- [5] Altera. FLEX 10K embedded programmable logic device family data sheet. Datasheet, January 2003.
- [6] P Anandan. A computational framework and an algorithm for the measurement of visual motion. *International Journal of Computer Vision*, 2:283–310, 1989.
- [7] P. C. Arribas and F. M. H. Maciá. FPGA implementation of the Horn & Shunk Optical Flow Algorithm for Motion Detection in real time Images. *Proceedings of the XIII Design of Circuits and Integrated Systems Conference*, pages 616–621, 1998.
- [8] P. C. Arribas and F. M. H. Maciá. FPGA implementation of camus correlation optical flow algorithm for Real Time Images. *14th International Conference on Vision Interface*, pages 33–38, 2001.
- [9] J.L. Barron, D.J. Fleet, and S.S. Beauchemin. Performance of Optical Flow Techniques - Systems and Experiment. *International Journal of Computer Vision*, 12(1):43–77, 1994.
- [10] S. S. Beauchemin and J. L. Barron. The computation of optical flow. *ACM Computing Surveys*, 27(3):433–467, 1995.
- [11] M. Bertero, T.A. Poggio, and V. Torre. Ill-posed problems in early vision. *Proceedings of the IEEE*, 76(8):869–889, Aug 1988.
- [12] M. J. Black. *Robust Incremental Optical Flow*. PhD thesis, Yale University, 1992.

- [13] M. Bober and J. Kittler. Robust motion analysis. *Computer Vision and Pattern Recognition, 1994. Proceedings CVPR '94., 1994 IEEE Computer Society Conference on*, pages 947–952, 21-23 Jun 1994.
- [14] Stephen Brown and Jonathan Rose. Architecture of fpgas and cplds: A tutorial. *IEEE Design and Test of Computers*, 13(2):42–57, 1996.
- [15] T. Camus and H. Bulthoff. Space-time tradeoffs for adaptive real-time tracking. In *Mobile Robots VI, Proceedings of the Society of Photo-Optical Instrumentation Engineers*, pages 268–279, 1991.
- [16] Javier Díaz, Eduardo Ros, Sonia Mota, Richard R. Carrillo, and Rodrigo Agís. Real time optical flow processing system. *Field-Programmable Logic and Applications*, 3203/2004:617–626, 2004.
- [17] J. Diaz, E. Ros, F. Pelayo, E. M. Ortigossa, and S. Mota. FPGA-based real-time optical-flow system. *IEEE Transactions on Circuits and Systems for Video Technology*, 16(2):274–279, February 2006.
- [18] D. J. Fleet and A. D. Jepson. Computation of Component Image Velocity from Local Phase Information. *International Journal of Computer Vision*, 5(1):77–104, 1990.
- [19] D. J. Fleet and K. Langley. Recursive filters for Optical Flow. *IEEE Transactions PAMI*, 17(1):61–67, 1995.
- [20] D.J. Heeger. Model for the extraction of image flow. *Journal of the Optical Society of America A*, 4:1455–1471, Aug 1987.
- [21] L. Hepplewhite and T.J. Stonham. Image representation, texture and the fuzzy n-tuple. *Pattern Recognition, 1998. Proceedings. Fourteenth International Conference on*, 2:1028–1030 vol.2, 16-20 Aug 1998.
- [22] Berthold K.P. Horn. *Robot Vision*. The MIT Electrical Engineering and Computer Science Series. MIT Press, Cambridge, Massachusetts, 1986.
- [23] Berthold K.P. Horn and Brian G. Schunck. Determining optical flow. *Artificial Intelligence*, 17:185–203, 1981.
- [24] Hongche Liu, Tsai-Hong Hong, Martin Herman, Ted Camus, and Rama Chellappa. Accuracy vs efficiency trade-offs in optical flow algorithms. *Computer Vision and Image Understanding: CVIU*, 72(3):271–286, 1998.
- [25] Jianhua Liu, Michael Chang, and Chung-Kuan Cheng. An iterative division algorithm for fpgas. In *FPGA '06: Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*, pages 83–89, New York, NY, USA, 2006. ACM.
- [26] Bruce. D. Lucas and Takeo. Kanade. An iterative image registration technique with an application to stereo vision. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence*, pages 674–679, 1981.
- [27] José L. Martín, Aitzol Zuloaga, Carlos Cuadrado, Jesús Láizaro, and Unai Bidarte. Hardware implementation of optical flow constraint equation using fpgas. *Computer Vision and Image Understanding*, 98(3):462–490, 2005.

- [28] B. McCane, K. Novins, D. Crannitch, and B. Galvin. On benchmarking optical flow. *Computer Vision and Image Understanding*, 84(1):126–143, October 2001.
- [29] Chris McCarthy and Nick Barnes. Performance of temporal filters for optical flow estimation in mobile robot corridor centring and visual odometry. In *Proceedings of the Australian Conference on Robotics and Automation*, December 2003.
- [30] Chris McCarthy and Nick Barnes. Performance of optical flow techniques for indoor navigation with a mobile robot. *Robotics and Automation, 2004. Proceedings. ICRA '04. 2004 IEEE International Conference on*, 5:5093–5098 Vol.5, 26 April-1 May 2004.
- [31] Enrico De Micheli, Vincent Torre, and Sergio Uras. The accuracy of the computation of optical flow and of the recovery of motion parameters. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(5):434–447, May 1993.
- [32] Inc. Micron Technology. Async/page/burst cellullarram 1.5 data sheet. Datasheet.
- [33] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.
- [34] T. Rowekamp, M. Platzner, and L. Peters. Specialized architectures for optical flow computation: A performance comparison of asic, dsp, and multi-dsp. In *Proceedings of the 8th International Conference on Signal Processing Applications & Technology*, September 1997.
- [35] A. Singh. An estimation-theoretic framework for image-flow computation. *Computer Vision, 1990. Proceedings, Third International Conference on*, pages 168–177, 4-7 Dec 1990.
- [36] Steven W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing, 1997.
- [37] Julio C. Sosa, Jose A. Boluda, Fernando Pardo, and Rocío Gómez-Fabela. Change-driven data flow image processing architecture for optical flow computation. *Real-Time Image Processing, Journal of*, 2(4):259–270, December 2007.
- [38] Alan A. Stocker. Analog integrated 2-d optical flow sensor. *Analog Integr. Circuits Signal Process.*, 46(2):121–138, 2006.
- [39] Keith Underwood. FPGAs vs. CPUs: trends in peak floating-point performance. In *FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, pages 171–180, New York, NY, USA, 2004. ACM.
- [40] S. Uras, F. Girosi, A. Verri, and V. Torre. A Computational Approach to Motion Perception. *Biological Cybernetics*, 60:79–87, 1988.

- [41] Z. Wei, D. J. Lee, and B. E. Nelson. FPGA-based Real-time Optical Flow Algorithm Design and Implementation. *Journal of Multimedia*, 2(5):38–45, September 2007.
- [42] Zhaoyi Wei, Dah-Jye Lee, Brent Nelson, and Michael Martineau. A fast and accurate tensor-based optical flow algorithm implemented in fpga. In *WACV '07: Proceedings of the Eighth IEEE Workshop on Applications of Computer Vision*, page 18, Washington, DC, USA, 2007. IEEE Computer Society.
- [43] R Williams. Using fpgas for dsp image processing. http://www.fpgajournal.com/articles/imaging_hunt.htm, 2006.
- [44] Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, and Katherine Yelick. The potential of the cell processor for scientific computing. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 9–20, New York, NY, USA, 2006. ACM.
- [45] Xilinx. XC4000, XC4000A, XC4000H logic cell array families product description. Datasheet.
- [46] Xilinx. XC4000E and XC4000X series field programmable gate arrays product specification. Datasheet, May 14 1999.
- [47] Xilinx. Virtex-E field programmable gate arrays preliminary product specification. Datasheet, November 9 2001.
- [48] Xilinx. Virtex-II Pro platform fpgas: Complete data sheet. Datasheet, November 11 2003.
- [49] S. Yamamoto, Y. Mae, Y. Shirai, and J. Miura. Realtime multiple object tracking based on optical flows. *Robotics and Automation, 1995. Proceedings., 1995 IEEE International Conference on*, 3:2328–2333 vol.3, 21-27 May 1995.