

Calculating Optical Flow using FPGAs

Michael Chisholm

Supervisor: Eric McCreath

Outline

- What is Optical Flow?
 - Quick example
- What are FPGAs?
- Why use FPGAs to calculate Optical Flow?
- Will an Optical Flow calculation algorithm fit onto an FPGA?
- How well does it perform?
 - Simulating the algorithm on a PC
 - Comparing the results
- Conclusions and future work

2

First you'll need to know what optical flow is and what FPGAs are

Then I'll detail why we want to use FPGAs to calculate optical flow. What use does optical flow have and why is doing it on an FPGA better than doing it on a PC?

Finally I'll go into the part of the problem that I worked on, what I achieved, and further research that can be done

Motivation

- Calculate optical flow without a PC
- FPGAs:
 - are smaller and lighter
 - use far less power
 - Can be used in embedded devices
- Goal of this project:



To show that useful optical flow can be calculated, using an FPGA, in real-time

3

We want to calculate optical flow without using a PC

By embedding optical flow detection capabilities in everyday things, they can act smarter:

For example planes and cars can determine when they're about to hit something, or even just figure out their speed and the speed of any obstacle in their environment

The goal of this project is to show that useful optical flow can be calculated using an FPGA, in real-time

Optical Flow

- Input: Sequence of images
 - From camera, movie, etc.
- Output: Vector field of motion at each pixel
- Information extracted:
 - Motion of objects in the visual scene
 - Motion of the camera

4

Optical flow information tells you the way that each pixel is moving in a video

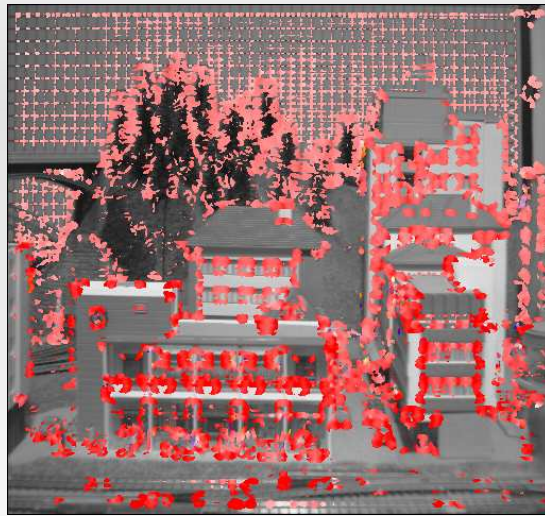
The world is sampled both spatially with pixels and temporally with image frames

Those samples are fed into an optical flow algorithm and you end up with a vector field representing the motion

The motion of the pixels tells you the motion of the objects that they represent, as well as the motion of the camera observing the scene

{Next 20 slides show moving scene}

Flow Field – Lucas & Kanade



25

Flow field calculated using FlowJ, written by M. D. Abramoff, W. J. Niessen and M. A. Viergever

Here you can see the resulting optical flow after using the Lucas & Kanade algorithm

Lighter pink indicates slower moving pixels, like in the background

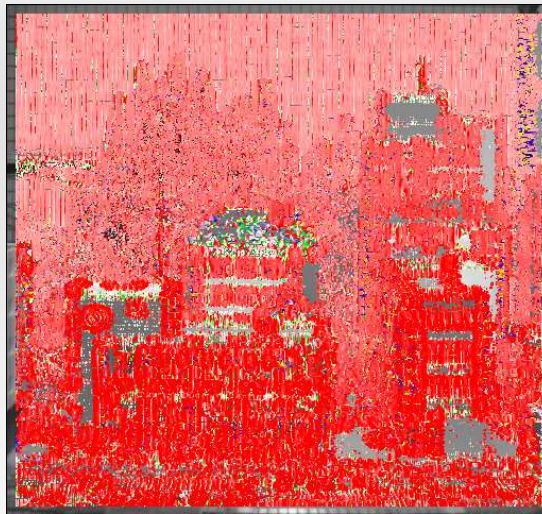
Dark red indicates fast moving pixels, like the buildings in the front

This algorithm is based on first derivatives of the image with respect to time and space.

It applies local smoothness constraints, meaning they consider velocities to be smooth within a local region, but it can't fill in the gaps from surrounding areas

Even though everything was moving, this algorithm only detects movement of strong contrast changes
The flat shades of the roofs and road don't appear to be moving

Flow Field – Uras et al.



26

Flow field calculated using FlowJ, written by M. D. Abramoff, W. J. Niessen and M. A. Viergever

This is the result produced by the Uras et al. algorithm

It provides denser flow fields, meaning it will pick up more of the movement

This comes at the cost of slightly more inaccuracy compared to the previous algorithm

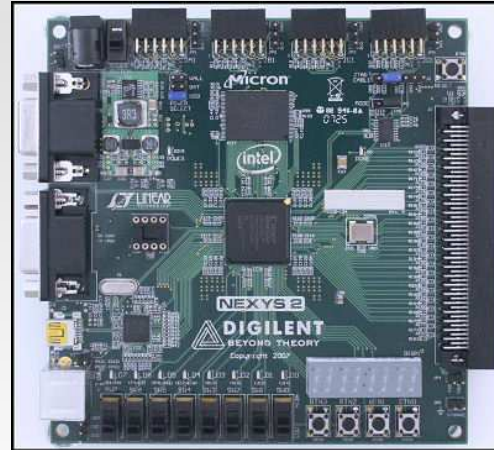
This is the algorithm that I've been implementing and testing on an FPGA

I'll explain the details of how it works a bit later

Field Programmable Gate Arrays

“FPGAs”

- Reprogrammable digital logic
- 100 000s of logic gates all on one chip
- Implement algorithms in hardware:
 - cryptography
 - speech recognition
 - video processing
 - many more



27

An FPGA is a Field Programmable Gate Array

Field programmable means you can reprogram it even when it is inside some larger device

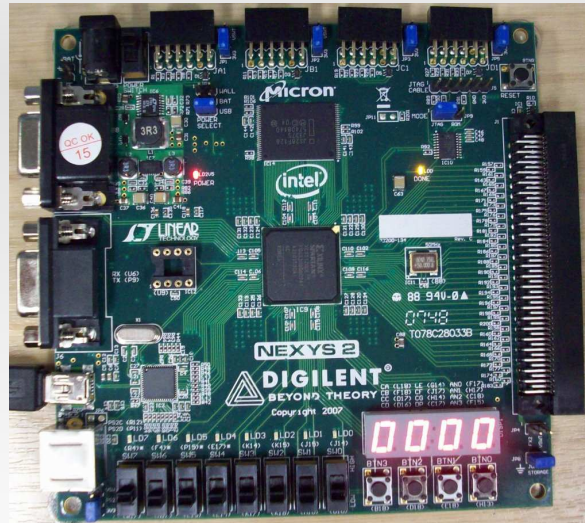
Gate Array refers to the fact that it has 100 000s of logic gates all arrayed together in the one chip

At the hardware level, programming it means changing the way the gates are connected, and what specific function they perform, like AND or NOR

They are very good for implementing pipelined or parallel algorithms in hardware, such as {see slide}

Development Board

- Spartan-3E FPGA
 - 17344 Flip-flops
 - 17344 Lookup tables
 - 640K Bits of RAM
 - 28 Dedicated multipliers
- Nexys 2 development board
 - USB interface
 - 16MB of PSRAM
 - Buttons and LEDs for simple interaction



28

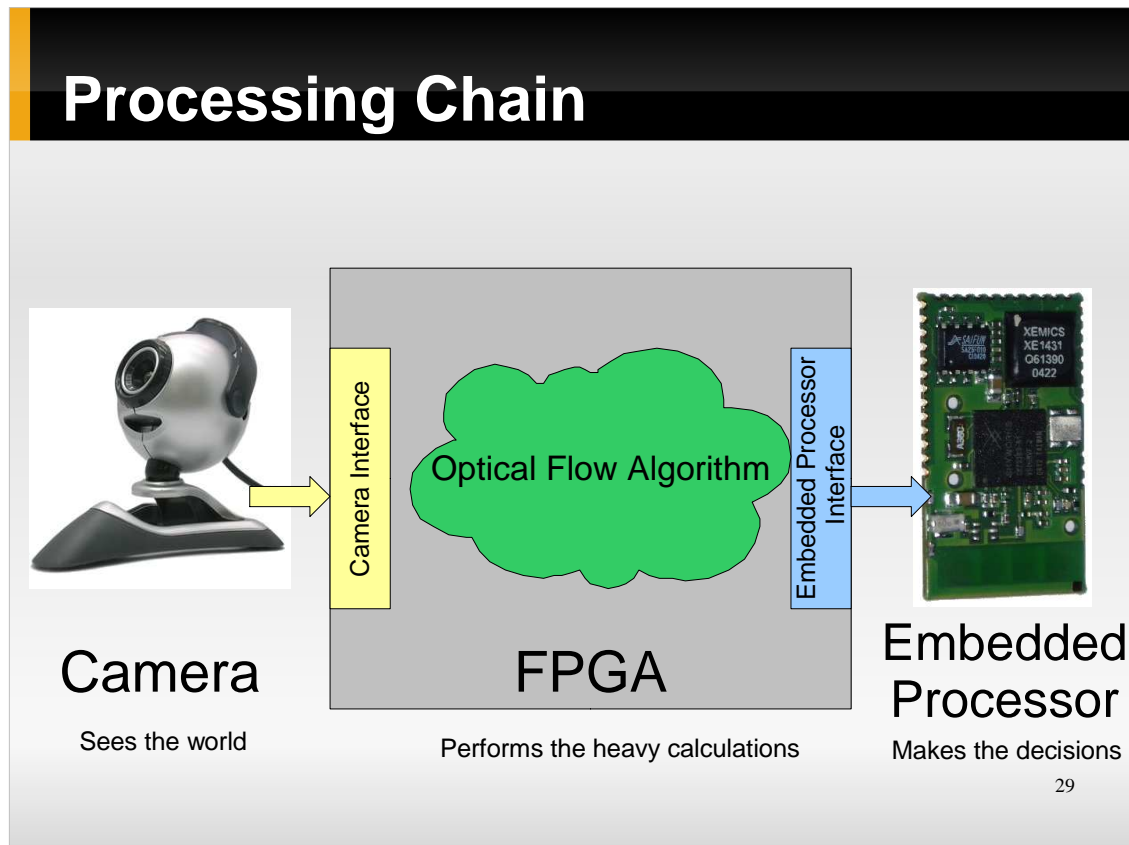
Flip flops and lookup tables used to implement logic by changing the contents of the LUTs and the connections between them and the various Ffs
RAM bits used for efficient storage, not enough to store a full frame of data

Dedicated multipliers can multiply two 18 bit integers to get a 36 bit result

USB interface provides for programming of the FPGA and for data transfer to/from the PC

The PSRAM - "Pseudo Static RAM" is used for storing large amounts of data, supplementing FPGA's RAM at the cost of speed

Buttons and LEDs are used for debugging



This is the basic processing chain

The camera takes pictures of the world at 25fps, 30fps, 60fps or even more

The images are fed into the FPGA, which performs all the demanding optical flow calculations

That data is then fed into the embedded processor, which extracts the information it needs, such as:

- Speed of the camera
- Motion of objects that the camera sees
- Predictions of the object's trajectories – collision avoidance

Uras et al.'s Algorithm

- The part that transforms image data into velocity vector fields
- Created by S. Uras, F. Girosi, A. Verri, and V. Torre
- Uses second-order spatio-temporal derivatives
 - Derivatives with respect to both position and time
- Assumes the image intensity gradient moves with a constant velocity
- To simplify the calculations, assumes that the flow is relatively smooth over large regions

30

After evaluating many types of algorithms, I settled on one created by S. Uras, F. Girosi, A. Verri, and V. Torre.

It relies on second order spatio-temporal derivatives. They assume that the image intensity gradient moves with a constant velocity.

They assume that flows are uniform over large regions, to simplify the formula

This also acts to fill in the gaps in the flow with velocities from surrounding regions, giving a higher flow density

This assumption is violated for occlusion boundaries and other sharp changes in velocity

So they apply some postconditioning to the velocities to smooth out any errors

Uras' Algorithm

$$\begin{bmatrix} V_x \\ V_y \end{bmatrix} = \frac{1}{\frac{\partial^2 I}{\partial x^2} \frac{\partial^2 I}{\partial y^2} - \frac{\partial^2 I}{\partial x \partial y} \frac{\partial^2 I}{\partial x \partial y}} \times \begin{bmatrix} \frac{\partial^2 I}{\partial y \partial t} \frac{\partial^2 I}{\partial x \partial y} - \frac{\partial^2 I}{\partial x \partial t} \frac{\partial^2 I}{\partial y \partial y} \\ \frac{\partial^2 I}{\partial x \partial t} \frac{\partial^2 I}{\partial x \partial y} - \frac{\partial^2 I}{\partial y \partial t} \frac{\partial^2 I}{\partial x \partial x} \end{bmatrix}$$

31

This is the central formula of Uras's algorithm

It returns a velocity vector based on 2nd order temporal and spatial derivatives of the image

Uras' Algorithm

$$\begin{bmatrix} V_x \\ V_y \end{bmatrix} = \frac{1}{\frac{\partial^2 I}{\partial x^2} \frac{\partial^2 I}{\partial y^2} - \frac{\partial^2 I}{\partial x \partial y} \frac{\partial^2 I}{\partial x \partial y}} \times \begin{bmatrix} \frac{\partial^2 I}{\partial y \partial t} \frac{\partial^2 I}{\partial x \partial y} - \frac{\partial^2 I}{\partial x \partial t} \frac{\partial^2 I}{\partial y \partial y} \\ \frac{\partial^2 I}{\partial x \partial t} \frac{\partial^2 I}{\partial x \partial y} - \frac{\partial^2 I}{\partial y \partial t} \frac{\partial^2 I}{\partial x \partial x} \end{bmatrix}$$

- Gives the x and y components of velocity

These are the velocity components

Uras' Algorithm

$$\begin{bmatrix} V_x \\ V_y \end{bmatrix} = \frac{1}{\frac{\partial^2 I}{\partial x^2} \frac{\partial^2 I}{\partial y^2} - \frac{\partial^2 I}{\partial x \partial y} \frac{\partial^2 I}{\partial x \partial y}} \times \begin{bmatrix} \frac{\partial^2 I}{\partial y \partial t} \frac{\partial^2 I}{\partial x \partial y} - \frac{\partial^2 I}{\partial x \partial t} \frac{\partial^2 I}{\partial y \partial y} \\ \frac{\partial^2 I}{\partial x \partial t} \frac{\partial^2 I}{\partial x \partial y} - \frac{\partial^2 I}{\partial y \partial t} \frac{\partial^2 I}{\partial x \partial x} \end{bmatrix}$$

- Gives the x and y components of velocity
- Requires
 - 2nd order spatial derivatives

These are the 2nd order spatial derivatives in the x and y directions

Uras' Algorithm

$$\begin{bmatrix} V_x \\ V_y \end{bmatrix} = \frac{1}{\frac{\partial^2 I}{\partial x^2} \frac{\partial^2 I}{\partial y^2} - \frac{\partial^2 I}{\partial x \partial y} \frac{\partial^2 I}{\partial x \partial y}} \times \begin{bmatrix} \frac{\partial^2 I}{\partial y \partial t} \frac{\partial^2 I}{\partial x \partial y} - \frac{\partial^2 I}{\partial x \partial t} \frac{\partial^2 I}{\partial y \partial y} \\ \frac{\partial^2 I}{\partial x \partial t} \frac{\partial^2 I}{\partial x \partial y} - \frac{\partial^2 I}{\partial y \partial t} \frac{\partial^2 I}{\partial x \partial x} \end{bmatrix}$$

- Gives the x and y components of velocity
- Requires
 - 2nd order spatial derivatives
 - Derivative of image gradient with respect to time
 - 2nd order spatio-temporal derivatives

34

These are the spatio-temporal derivatives

That is, the way the image gradient changes with respect to time

Or, put another way, the 1st order spatial derivatives of the 1st order temporal derivative

Uras' Algorithm

$$\begin{bmatrix} V_x \\ V_y \end{bmatrix} = \frac{1}{\frac{\partial^2 I}{\partial x^2} \frac{\partial^2 I}{\partial y^2} - \frac{\partial^2 I}{\partial x \partial y} \frac{\partial^2 I}{\partial x \partial y}} \times \begin{bmatrix} \frac{\partial^2 I}{\partial y \partial t} \frac{\partial^2 I}{\partial x \partial y} - \frac{\partial^2 I}{\partial x \partial t} \frac{\partial^2 I}{\partial y \partial y} \\ \frac{\partial^2 I}{\partial x \partial t} \frac{\partial^2 I}{\partial x \partial y} - \frac{\partial^2 I}{\partial y \partial t} \frac{\partial^2 I}{\partial x \partial x} \end{bmatrix}$$

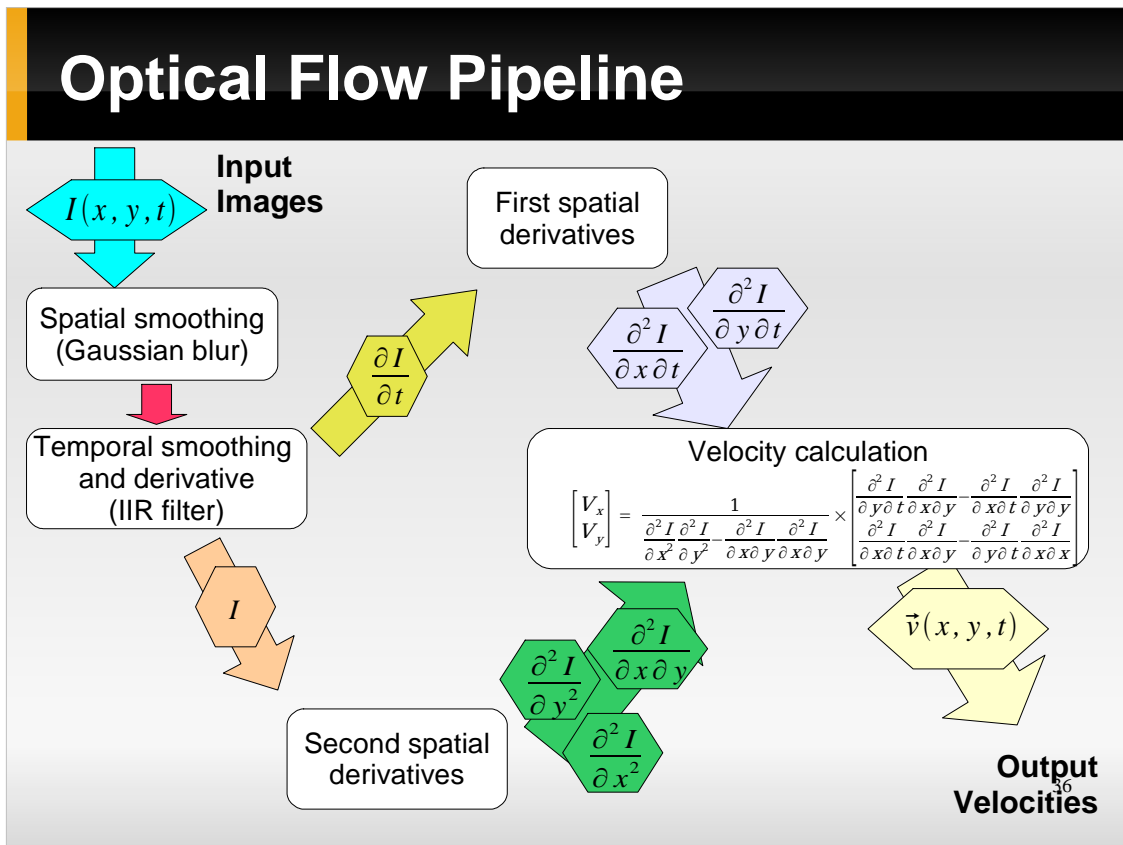
- Also need to remove high frequency components of image
 - Use a Gaussian blur for spatial components
 - Smooths out sharp changes in contrast
 - Use a type of motion blur for temporal components
 - Smooths out sharp changes in position with respect to time

35

The 2nd order derivatives use numerical approximations to true derivatives, and are very sensitive to problems with the input data

As such, we smooth the input images in space using a Gaussian filter and smooth the images in time using an infinite impulse response filter.

This has the effect of removing most of the noise and high frequency components of the image that will have an adverse impact on our results



This is the basic overview of how my implementation of Uras' algorithm works

Uses a processing pipeline where each stage is constantly processing new data as it comes in and passes its results onto the next stage.

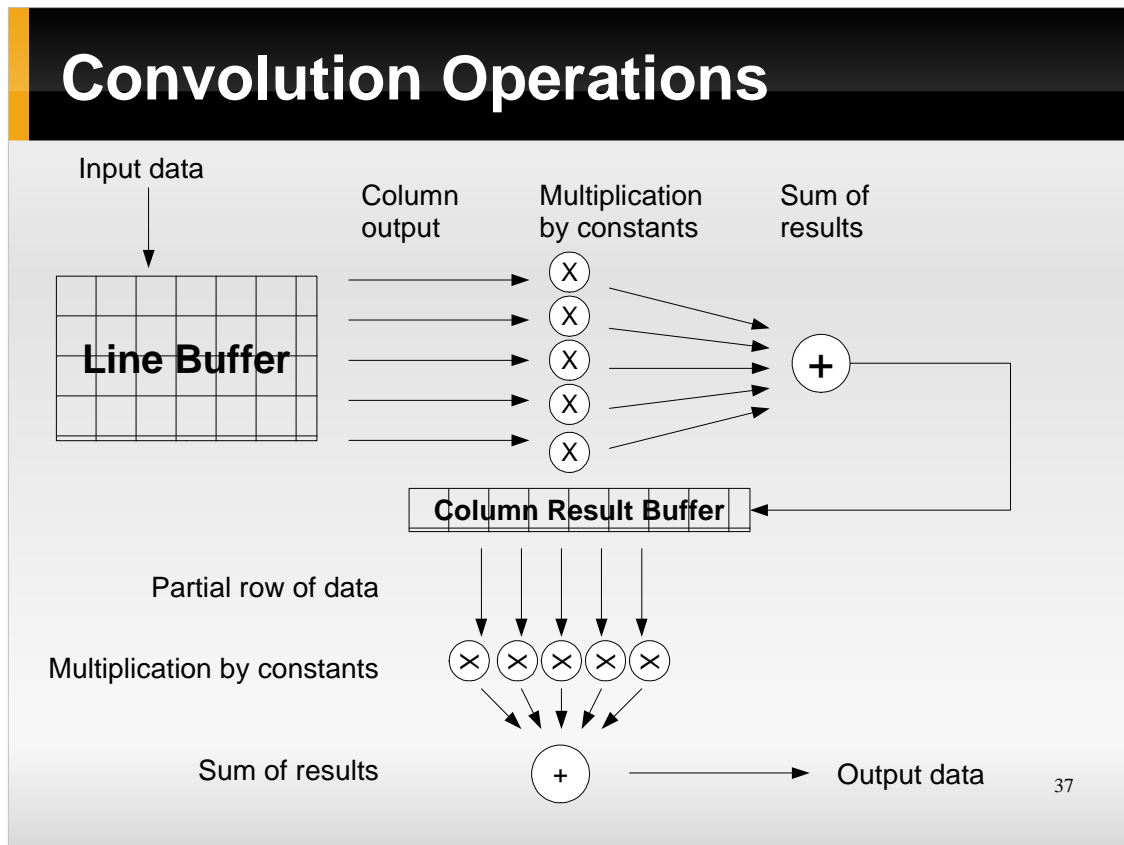
Image data comes in one end a pixel at a time, and velocity data comes out the other end, also one element at a time

In between are various blocks that act to smooth the images, calculate spatial and temporal derivatives, then use those derivatives to estimate pixel velocities

Spatial Blur and Spatial Derivatives both use the same technique – convolution

Temporal smoothing and derivative calculation uses an infinite impulse response filter, which I'll show you in two slides time

Data is represented using fixed point numbers up until the very last part of the velocity calculation, where it is converted to floating point to give a larger range of magnitudes for the division results



How convolution works

Apply a convolution kernel to a group of input pixels to get an output pixel

Data is stored in a line buffer, then taken out column by column. This is convoluted with the vertical kernel, and the result stored in another buffer

Since the kernel constants are fixed, the multiplication can be reduced to a series of bit shifts and additions

This buffer is convoluted with a horizontal kernel to get the final result

This works because all of the 2D convolutions used can be broken down into a combination of 1D vertical and horizontal convolutions

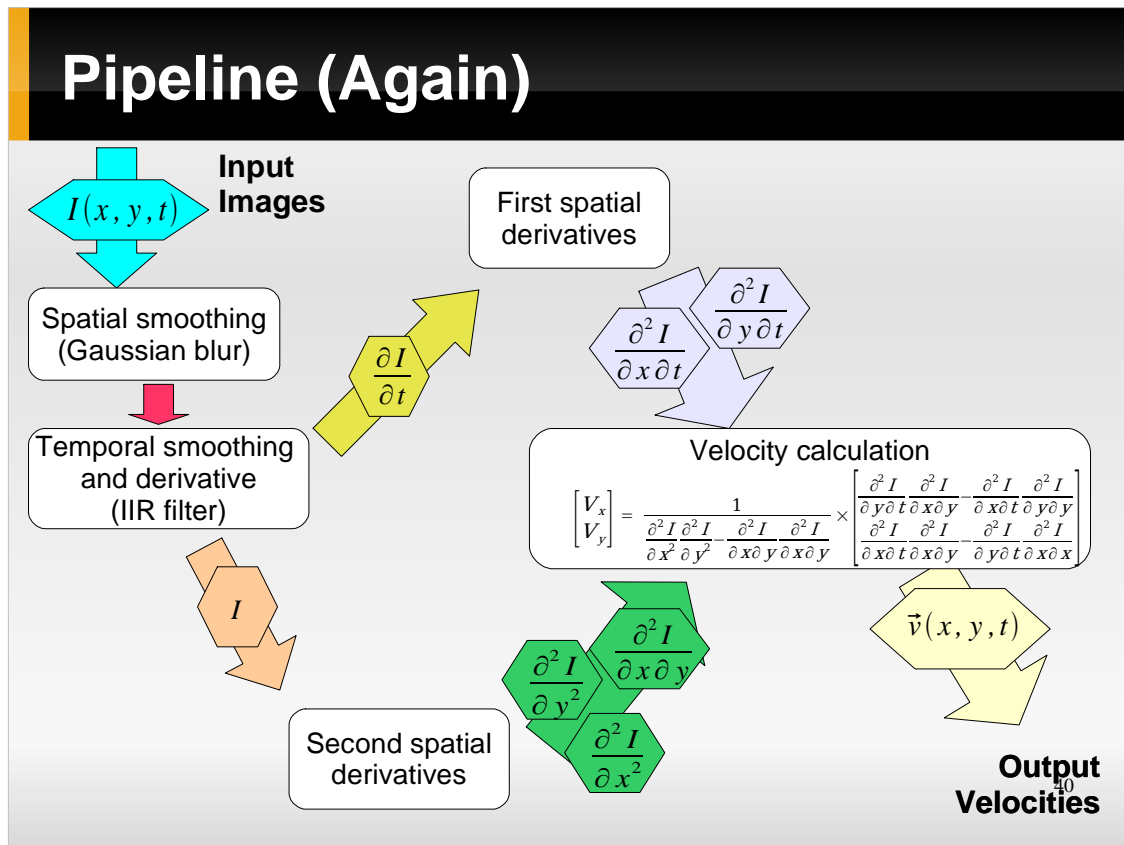
Convolution Kernels

- Gaussian blur with a standard deviation of 3, with the kernel stored in 0.7 fixed point format:
 $\{1, 2, 4, 7, 11, 14, 16, 17, 16, 14, 11, 7, 4, 2, 1\} * 1/128$
- First derivative using a 4-point central difference operation:
 $\{-1, 8, 0, -8, 1\} * 1/12$
- Second derivative using a 4-point central difference operation:
 $\{-1, 16, -30, 16, -1\} * 1/12$

38

These are the convolution kernels applied at the various stages

They're all 1D kernels, which makes the convolution relatively straight-forward



Implemented some components to make sure it was feasible to use an FPGA before I did the entire thing

Implemented a spatial blur to ensure convolution was possible

Tested memory bandwidth to make sure I could get data in and out as needed, which is important for the IIR filter stage

Determined area needed for the most space consuming components to ensure it would all fit

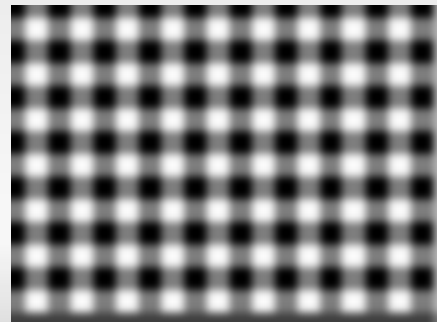
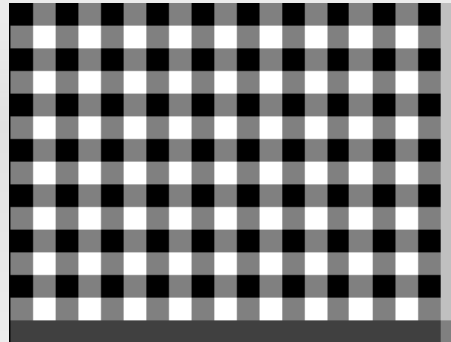
I've also determined what the limiting factor on the processing speed is:

It's limited by the memory bandwidth to 2.5 million pixels per second

The latency is 3 frames, introduced by the IIR filter

FPGA Feasibility – Spatial Blur

- Applies a Gaussian blur to the image using convolution
- Kernel was:
 $\{1, 2, 4, 7, 11, 14, 16, 17, 16, 14, 11, 7, 4, 2, 1\} * 1/128$
- Can process one input pixel per clock cycle
- Example shows a 320x240 input image blurred into a 206x226 result



The kernel is a 1D Gaussian applied both horizontally and vertically. It has a standard deviation of 3 pixels, and is in 0.7 fixed point format.

For this block, it can process 1 pixel per cycle on a 50MHz clock, or 50 million pixels per second.

For comparison, a VGA video at 60FPS has 18 and a half million pixels per second

This is the throughput, though. There is still a latency caused by the data flowing through the various stages

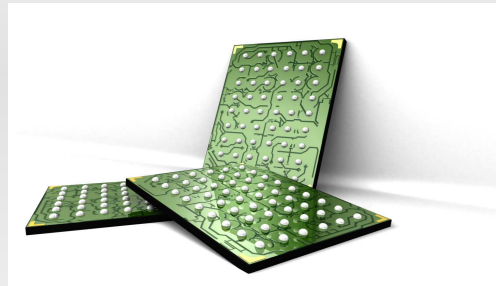
The output image is smaller because it uses the valid result from the convolution operation.

It requires the data to be sent and received in two parts, as the FPGA doesn't have enough in-built RAM to buffer the full resulting image

This wouldn't be a problem for the final implementation, as data would be coming in one port of the FPGA and output from another, and the FPGA would be able to use the external PSRAM as a buffer

FPGA Feasibility – Memory Bandwidth

- Can transfer 16 bits every 3 clock cycles
- Temporal filter requires 102 bits transferred per pixel
 - PSRAM speed becomes limiting factor in design



Needs 20 clock cycles per pixel for data transfers
Can still process around 2,500,000 pixels per second

42

If the low bandwidth becomes too problematic, can switch to burst memory transfers which achieve around 16 bits every 1.5 cycles
Can also use multiple PSRAM chips in parallel to increase the data transfer width
So it's not too big of a problem

FPGA Feasibility – Area consumed

- Largest component is floating point divider
 - Needs 3421 FFs and 2311 LUTs:
 - ~20% of the FPGA's logic resources
- Spatial blur needs 722 FFs, 864 LUTs & 35K RAM bits: ~5% logic and ~6% on-chip RAM
 - Derivative needs less space
- Temporal filter mostly needs storage space
 - Supplied by external RAM
 - Interface needs 128 FFs and 311 LUTs: ~2% logic
- Around 40% of logic resources and 20% RAM resources used in total

43

Largest area is floating point divider

Spatial blur requires around 5% of the logic and 6% of the on chip RAM

The two derivatives need less space each

Temporal filter mostly needs storage, which is supplied by external RAM

Predicted total area consumed is 40% logic, 20% RAM

Actual total area consumed will be less than the sum of the parts, due to optimisations made by the compiler

FPGA Code Difficulties

- Compared to a PC application, it is difficult to change FPGA code once it has been written
 - Parameters, such as convolution kernels, are “baked” into the design
- It is difficult to debug embedded code
 - Everything runs at the same time
 - You can't see into the FPGA when it's running
- For these reasons, I wrote a functional simulation for the PC using C

44

Compared to a PC application, it is difficult to change FPGA code once it has been written

Parameters, such as convolution kernels, are “baked” into the design

It is also difficult to debug the code

Everything runs at the same time and timing is very important

For example, a one cycle delay in registering a line for the spatial blur results in a skewed image

You also can't see into the FPGA when it's running, only in hardware simulations running on a PC

But these simulations can't cover everything

For these reasons, I wrote a functionally equivalent simulation for the PC using C

Functional Simulation

- All pixel data and maths is done as it would be on the FPGA
 - Values are represented in fixed point
 - The order of operations are kept the same
- Benefits of a functional simulation:
 - I can quickly change the value representation
 - I can quickly change parameters
 - I can rapidly prototype changes before committing to 2 weeks of writing and debugging FPGA modules

45

All pixel data and maths is done as it would be on the FPGA

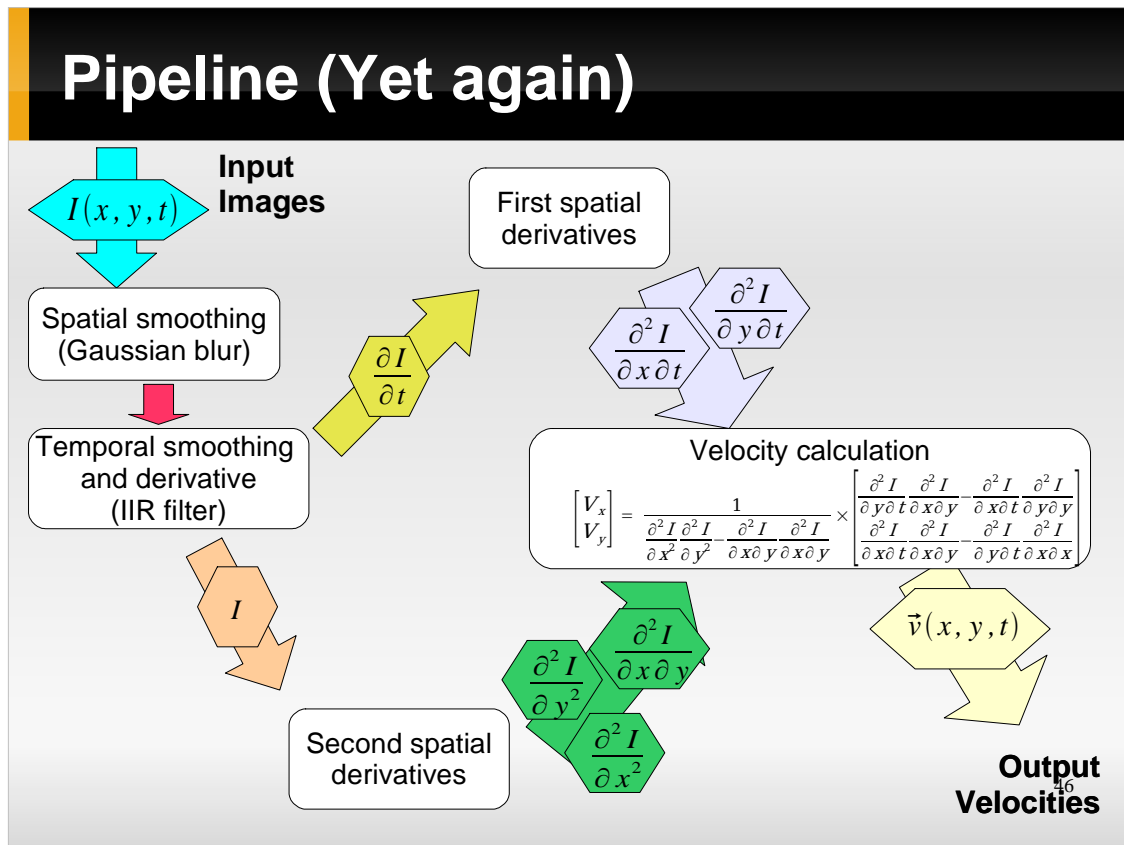
Values are represented in fixed point

The order of operations are kept the same

Benefits of a functional simulation:

- I can quickly change the value representation
 - For example, I can increase the number of bits assigned to the fractional component of values used in certain stages
- I can quickly change parameters
 - For example, I can change the standard deviation used in the gaussian blur

Most importantly, I can rapidly prototype changes before committing to 2 weeks of writing and debugging FPGA modules



To make sure the algorithm would actually work, all of this is implemented in a C program

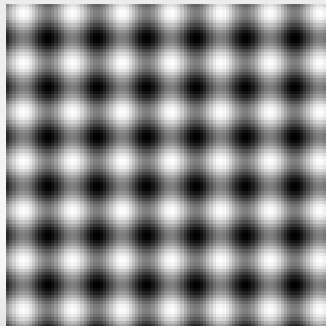
I was easily able to tweak parameters to see their effects

For instance, I started off with using 8.8 fixed point numbers for the temporal filter processing, but found this produced large numeric errors in the output. Simply increasing the number of fractional bits to 9 solved this problem

The output from the program is formatted so that it can be compared to the results from other implementations

I can also use the results to visualise what's happening

Test data – Translating Sine



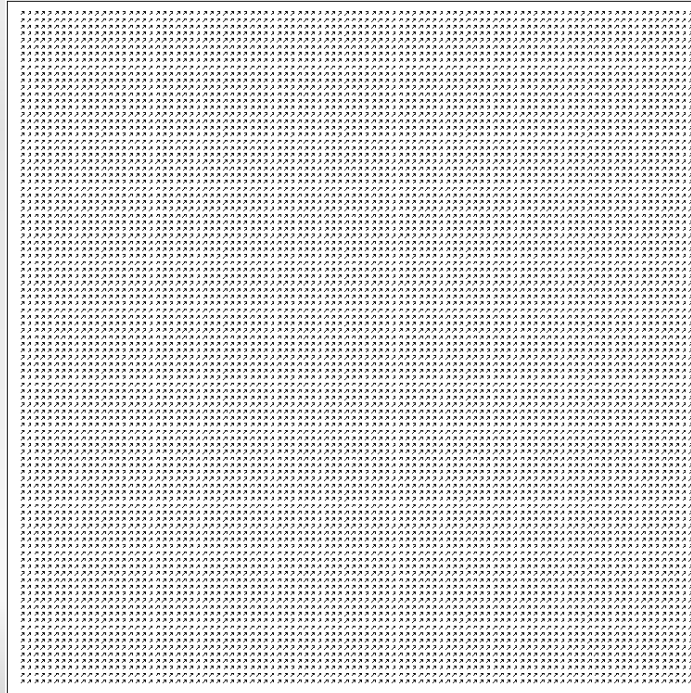
47

This is a very basic test of the algorithm

It is a uniform combination of vertical and horizontal motion

The image has no sharp contrast changes and no jumps in position

Translating Sine - expected result

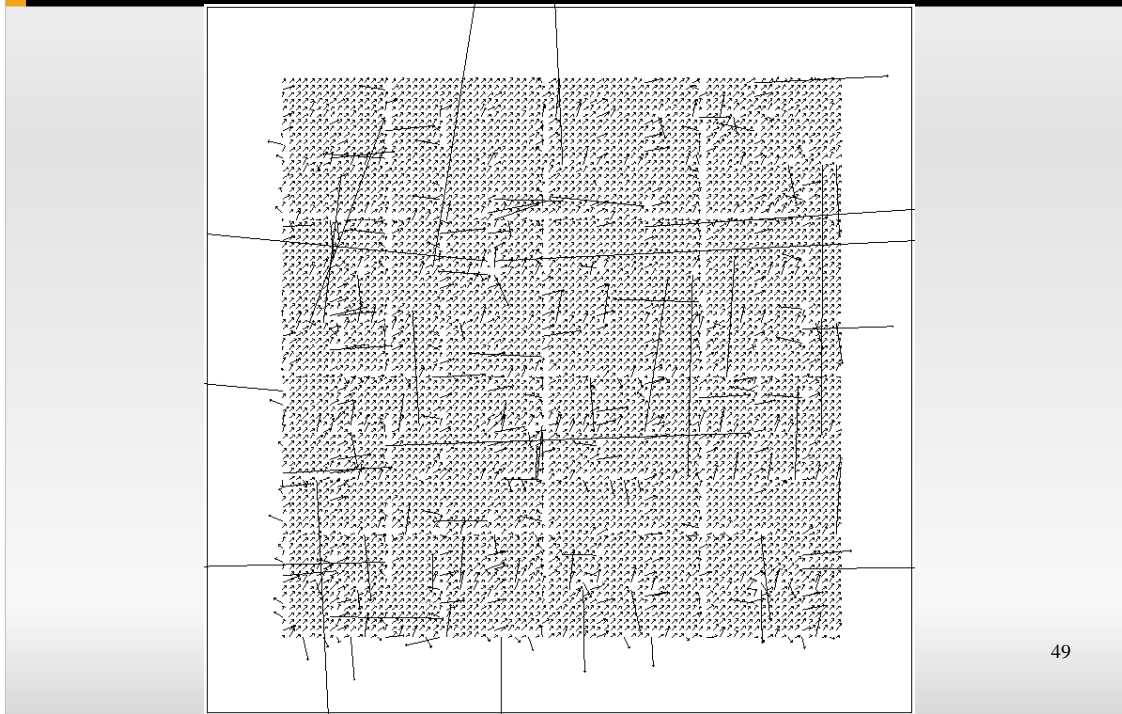


48

This is result we expect to see

You'll notice that all the arrows are the same size and they're all pointing to the top right of the square, in the same direction that the image was moving

Translating Sine – actual result



49

This is what I actually got

Most of the arrows are indeed pointing to the top right of the screen, and most are the same length

There are some that point in random directions with a random length

These are errors in the result, and can be caused by a number of things

- Calculations not accurate enough – not enough fraction bits or maybe I need to use floating point at an earlier stage
- The high frequency filtering was not aggressive enough – that's not a big problem here, since the image is already smooth, but it may be a problem for other images, such as the next one I'll show you

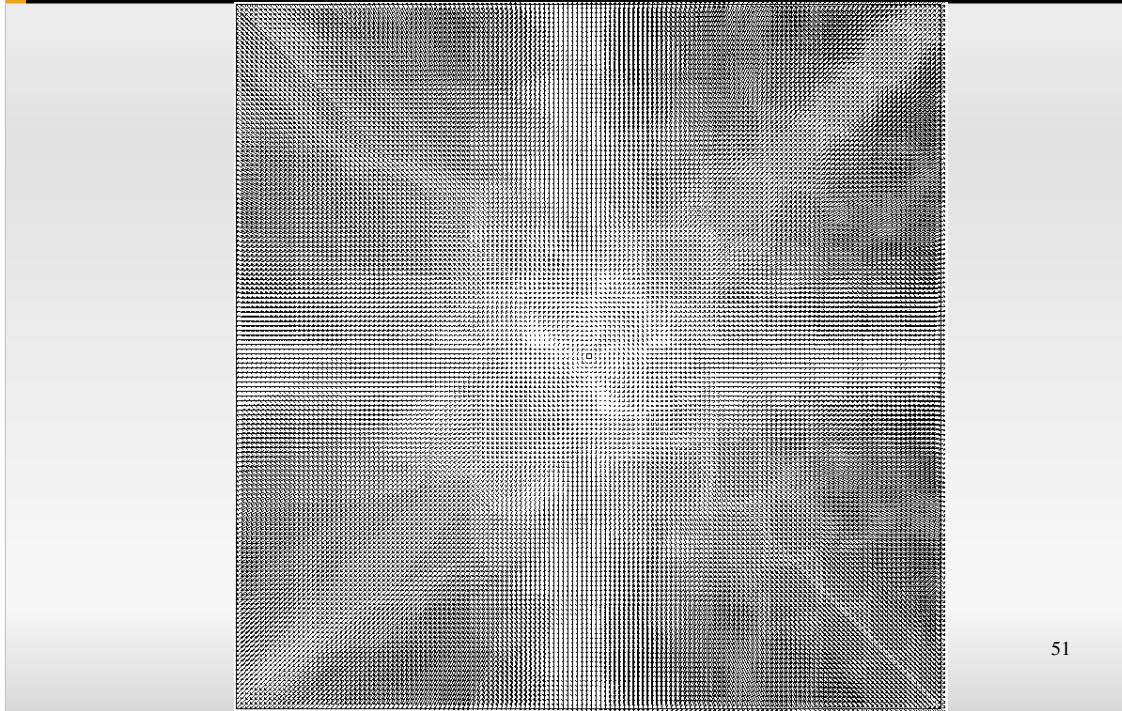
Diverging Tree



This is a simple zoom on an image

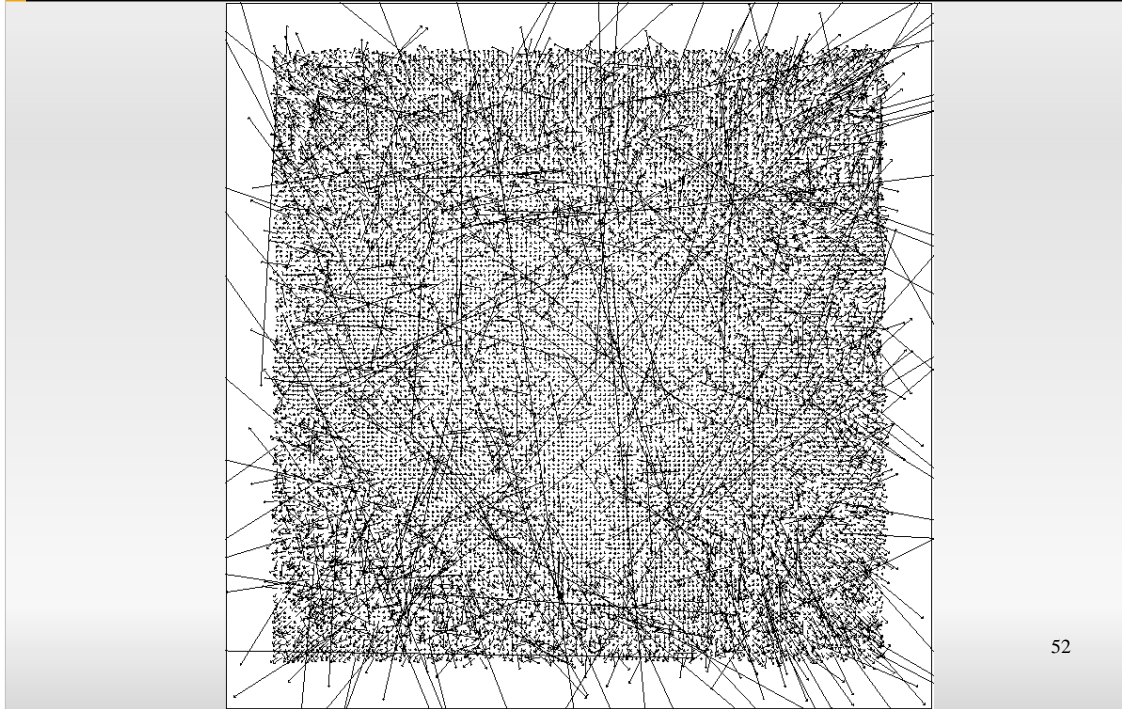
It tests that the algorithm can handle motion that isn't all in the same direction with the same speed, and that it can deal with sharp contrast changes

Diverging Tree – exected result



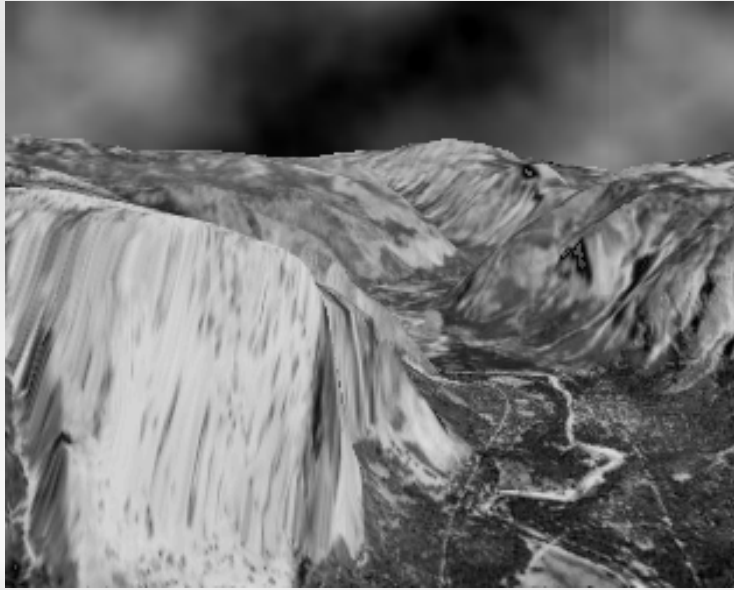
Here you'll notice the arrows all point outwards, with arrows further from the centre being larger than the ones in the middle

Diverging Tree – actual result



Again we get mainly correct results, but there are quite a few errors too, mostly with the length of the arrows, indicating that the speed wasn't calculated accurately – This could be a problem with the division in the velocity calculation

Yosemite



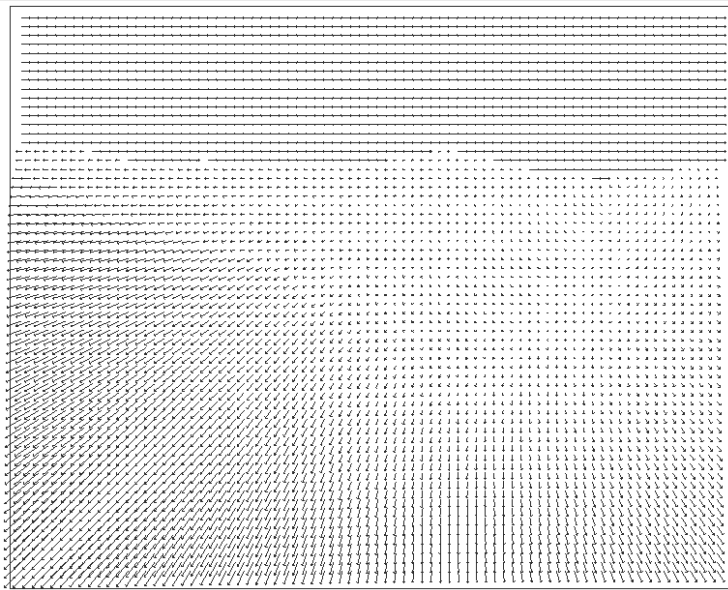
53

This is a computer generated sequence of a fly through a canyon

This is the most difficult image sequence to deal with

The textures are complicated and have sharp edges

Yosemite – expected result

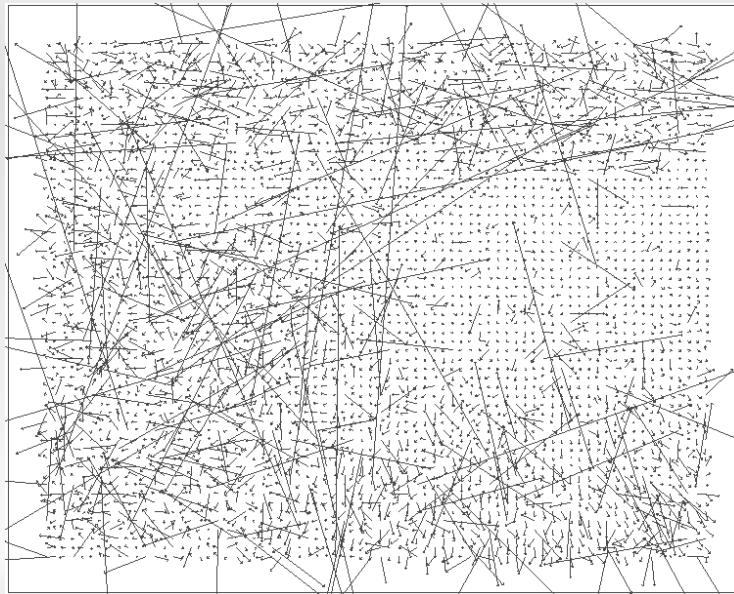


54

If you look at the expected motion results, you'll notice there's constant horizontal motion in the clouds, and a zooming motion in the valley

You can also see an occlusion boundary between the clouds and the valley

Yosemite – actual result



55

Here you can see that the horizontal and zooming motions were indeed picked up, but there's also a lot of errors in the lower left wall and the valley itself, as well as in the clouds

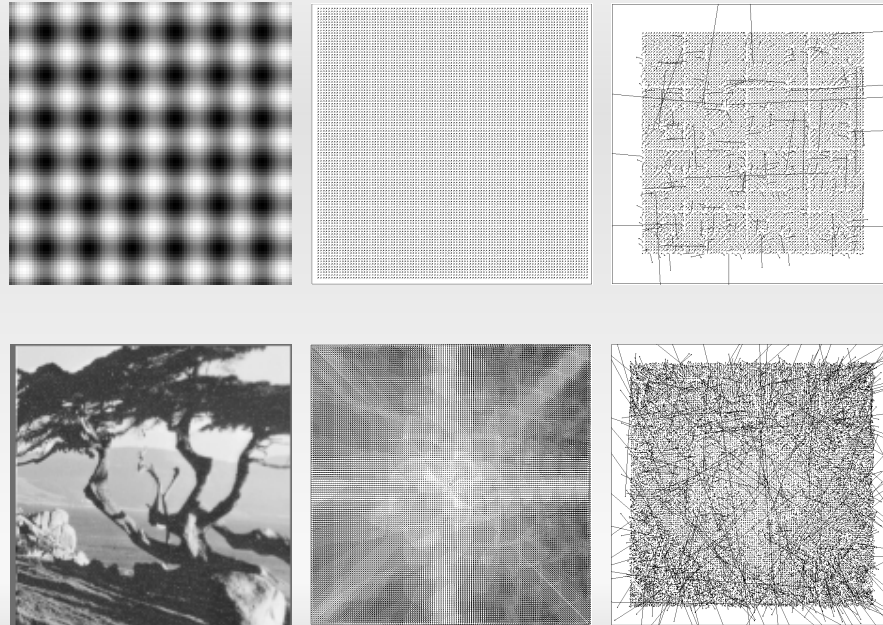
This time the errors may result not only from inaccurate calculations and not enough filtering, but it can also be down to one of the assumptions made in the algorithm – that the motion is uniform over large regions

Uras et al. discussed ways to rectify this error by postconditioning the vectors.

(either averaging the results or comparing the obtained velocities back against the original data)

This will involve adding further stages to the pipeline.

Test results



So these results indicate that the implementation actually works, but it needs improvement for more complex scenes

Simulation results

- For the Diverging Tree sequence, there is an average angular error of 26°
 - Barron, Fleet and Beuchemin obtained 4.64° using a PC-based floating point implementation of the algorithm
- For the Yosemite sequence, there is an average angular error of 49° .
 - Barron et al. got 9°
 - Díaz et al., using an implementation of the Lucas & Kanade algorithm on an FPGA, got 18°

57

These are some of my results compared to what other people have achieved

Barron, Fleet and Beauchemin wrote implementations of various optical flow algorithms and compared the results. They used floating point numbers on a PC for their implementations

They also wrote the software which I used to produce the flow field diagrams

Diaz implemented the Lucas & Kanade algorithm on an FPGA using mostly fixed point numbers. They didn't specify the exact format of their numbers, and they used floating point at an earlier stage to me. They also used a specially designed board for their implementation, with dedicated FIFO buffers for each stage

Improving the results

- Use more bits to represent values
- Use floating point at an earlier stage for more dynamic range
- Replace the IIR filter with a simple FIR filter
- Postcondition the output velocities using methods discussed in Uras et al.'s and various other papers:
 - Selecting the best result in each small patch and rejecting the rest
 - Averaging out the results

58

Using more fractional bits reduces numerical error

Using floating point gives a larger range of possible values, and will have more fractional bits available compared to the currently used 8.9 format.

The infinite impulse response filter can act to compound numerical errors. A Finite Impulse Response filter does not recursively store intermediate calculations, so it won't have this problem

This comes at the cost of a much higher memory bandwidth requirement, around 350 bits transferred per pixel processed

Methods for selecting the best result include: setting a minimum on the magnitude of the divisor in the velocity calculation; having a maximum difference between the spatial derivatives; and applying the calculated velocity back to the original image to check the error in the result

Conclusions

- It is definitely possible to use FPGAs to calculate optical flow in real time
- Uras' algorithm can work when implemented on an FPGA
- How well it works depends on:
 - The parameters chosen for the Gaussian and motion blur
 - The accuracy of the intermediate calculations
 - The hardware's ability to keep up with demands, like memory transfer bandwidth

59

The goal of this research was to see if it is possible to implement an optical flow detector using an FPGA. I've done this by:

- 1) Showing that an implementation of Uras' algorithm will fit on a device and
- 2) Showing that the implementation will give us optical flow data when restricted to the restraints of the hardware

There are still some problems with the resulting flows, but these can be improved by tweaking the various stages of the pipeline

Future work

- Improve the accuracy of the results obtained from the simulation
- Implement the entire pipeline on the development board
- Design a final board that:
 - Doesn't rely on a PC for data input and output
 - Has a small footprint that can fit in an embedded device

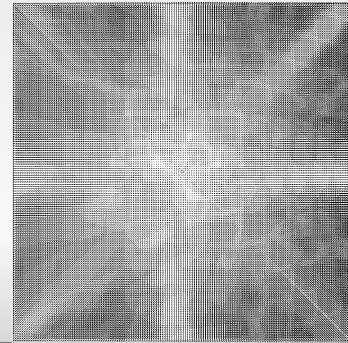
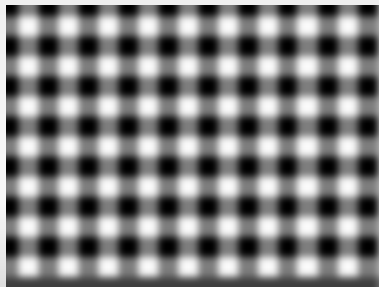
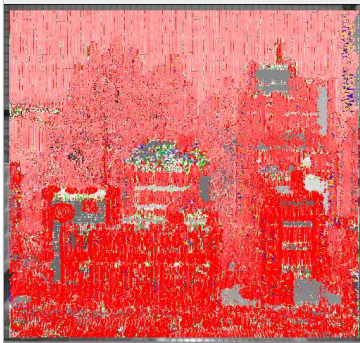
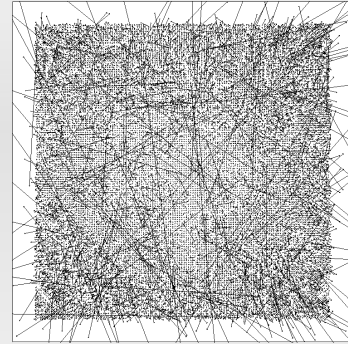
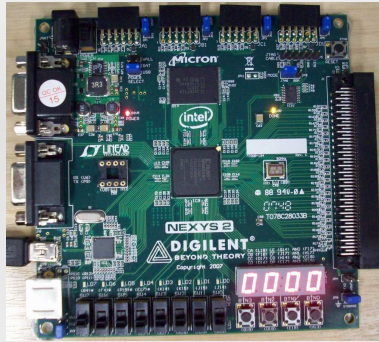
60

Now that I've shown it's possible, the next step is to improve the accuracy of the results

Once that's done, the entire pipeline can be implemented on the Nexys-2 development board

Then finally the hardware can be shrunk down so that it can be incorporated into various embedded devices

Questions?



Any questions?

Planned Time Line

Weeks 1-3	Background research into various optical flow algorithms Requirements specification
Week 4	Initial presentation Learn a language used to program FPGAs
Weeks 5-7	Write literature review draft Begin design implementation
Mid-semester Break	Write final copy of literature review Further implementation
Weeks 8-11	Refine implementation Test various optimisations
Week 9	Write outline of final report
Weeks 12-13	Finalise design and results Write draft report
Week 13	This presentation
Week 14	Finish and submit report

This is my planned time line

The actual time line was fairly similar, although I've been playing with implementations up until the weekend just been