MCGILL UNIVERSITY

DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING

ECSE 456 - FINAL REPORT

A Real-Time Object Tracking System

Authors:

Benjamin BROWN
benjamin.brown2@mail.mcgill.ca
260450182

Taylor DOTSIKAS
taylor.dotsikas@mail.mcgill.ca
260457719

Supervisors:

Warren GROSS, PROF.

Arash ARDAKANI

Abstract

Video processing represents an extremely relevant challenge as both the demand for intelligent, aware systems and the quality of modern video technology increases. This increase in quality comes at a cost of large amounts of data being handled under strict time constraints [3]. This project aimed to study how a common video processing algorithm such as object motion tracking can be accelerated using custom hardware. This report concludes on the background, design, and findings of Phase 1 of the project; where platform research, algorithm research, and software implementation was performed. It was clear that an FPGA would be the best solution for fast, low power hardware implementation and a Kalman filter based algorithm was the best solution for a predictive algorithm not prone to noise. Finally, the large amount of time the software needed to process relatively short videos proved that direct hardware implementation of the algorithm is essential for applications. Software implementation was performed in MATLAB, and hardware implementation will be done on an Altera Cyclone II FPGA during Phase 2 of the project.

Acknowledgments

The authors would like to thank Professor Warren Gross, for undertaking the responsibility of supervising this project. We would also like to thank Arash Ardakani for all of his advice, help, and time spent meeting with us throughout the semester. Finally, we would like to thank anyone who contributed to the wealth of online resources about object tracking that we hope to contribute to.

Contents

1 Abbreviations & Notation 3

2 Introduction 3
2.1 Motivation . . . 3
2.2 Applications . . . 3

3 Background 4
3.1 Video Processing . . . 4
3.2 Optical Flow . . . 5
3.3 Kalman Filter . . . 6
3.4 Fixed vs. Floating-Point . . . 7

4 Requirements 9
4.1 Time Frame . . . 10
4.2 Algorithm . . . 10
4.3 Software . . . 10
4.3.1 Floating-Point . . . 10
4.3.2 Fixed-Point . . . 10
4.4 Hardware . . . 10

5 Design 10
5.1 Platform . . . 10
5.2 Algorithm . . . 12
5.2.1 Delta Frame Generation . . . 12
5.2.2 Block Matching . . . 12
5.2.3 Kalman Filter . . . 13
5.2.4 Adaptive Filter . . . 13
5.2.5 Design Decision . . . 13
5.3 Software . . . 14

6 Future Work 16

7 Impact on Society 18

8 Allocation of Work 18

9 Conclusion 18

A Additional Figures 21

1 Abbreviations & Notation

FPGA - Field Programmable Gate Array

ASIC - Application Specific Integrated Circuit

CPU - Central Processing Unit

DFG - Delta Frame Generation

VGA - Video Graphics Array

I/O - Input/Output

VLSI - Very Large Scale Integration

2 Introduction

2.1 Motivation

The goal of this project is to develop a system that can perform real-time motion based object tracking. This is motivated by the fact that scene recreation and analysis is imperative in digital systems that must understand and react to events in their environment. Some typical examples of this include surveillance, robotics, and human-computer interaction. A variety of sensors can be employed for such a task including ultrasonic, radar, and passive infrared, but all of these sensors do not come close to modeling an environment as completely as a video camera. With the increase in image quality and device accessibility, the video camera seems like the obvious solution.

However, due to the vast amount of data and system imposed processing time constraints, video processing is a challenge. For instance, the transition to a high-definition video platform produces six times more data than the previous standard-definition one [3]. This project aims to study how a complex video processing algorithm such as real-time object tracking can be greatly accelerated when implemented directly in hardware. Object tracking represents an excellent example of the preceding challenges because it requires capturing an image of a scene, processing the image to locate the object in motion, and reconstructing said scene with emphasis placed on the motion, all in real-time.

2.2 Applications

The finished system will generate object localization data that can be used by other systems. Systems that require responsive and accurate data for real-time use are the main applications. These secondary systems can now react to moving objects in the selected environment.

Specifically, the algorithm implemented in the system tracks a single object from a camera that watches a stationary environment. This makes the design ideal for tracking a disturbance in the field of view. For example, the addition of the device to a camera surveillance system would provide data about where and when disturbances take place. This information could be used to quickly react to disturbances such as a trespassers on private property.

Dedicated hardware allows for simpler connectivity to secondary systems. It presents a modular design which is ideal for testing purposes and debugging. If object tracking needs to be performed on higher definition video, dedicated hardware has its own memory which can be specifically allocated for handling larger image frames. Performance depends on the technical specifications of the hardware being used. The current hardware is limited due to budget constraints, however it would be possible to upgrade the hardware to match the needs of the desired application.

The Kalman filter in the design tracks the object more smoothly and can also predict the location of the object if it disappears from the view temporarily. This further enhances the performance of secondary systems by providing them with more accurate and useful object tracking data, which is the main goal of the device.

3 Background

3.1 Video Processing

The most fundamental way of understanding video data is to consider it as a collection (static) or stream (real-time) of discrete images called frames. A frame is an MxN matrix of pixels. Each pixel is the smallest discrete element of the image, and store intensity data about the image. There are many different ways of representing intensity in a pixel. In the case of a color image, the pixel contains multiple values that describe the color space. Common representations of the color space are RGB or YCbCr [1]. In both of these models, each pixel has 3 values. Since MATLAB's `VideoReader` class uses RGB, for convenience, this color space is used for the remainder of the project. A single color RGB frame is mathematically described as an $MxNx3$ matrix with the form

$$F_k = \begin{bmatrix} (R_{11}, G_{11}, B_{11}) & ... & (R_{1N}, G_{1N}, B_{1N}) \\ ... & ... & ... \\ (R_{M1}, G_{M1}, B_{M1}) & ... & (R_{MN}, G_{MN}, B_{MN}) \end{bmatrix}. \quad (1)$$

Where

$$k = 1, 2, ..., n.$$

$$0 \leq R_{ij}, G_{i,j}, B_{i,j} \leq 256$$

The first frame, F_1, is defined as the *base frame*, and the remaining frames are defined as the *current frame* for processing. If a video is t seconds long, the *frame rate* is defined as

$$f = \frac{t}{n}. \quad (2)$$

In the case of a black and white image, the pixel contains a single value that represents the grayscale intensity. Similar to F_k, the grayscale frame also contains n frames, and its elements, Y_{ij}, are limited between 0 and 256. However, it is an MXN matrix only. It will become apparent later that conversion between RGB and grayscale is imperative for many video processing algorithms. Unsurprisingly, there are multiple ways to do this. The colorimetric conversion principle converts RGB to grayscale using the following weighted sum:

$$Y_{ij} = .2126 \cdot R_{ij} + .7152 \cdot G_{i,j} + .0722 \cdot B_{i,j}. \quad (3)$$

Finally it should be mentioned that video data can be streamed in either progressive or interlaced format. Progressive format is one frame at a time while interlaced divides the frames in half into fields. Each field contains either odd or even rows of it's corresponding frame. While this does requiring processing more frames, it gives a clearer, smoother picture as there are less scene changes between frames [1]. This design choice plays a larger role in Phase 2 of the project, since software implementation uses static (pre-recorded) videos.

3.2 Optical Flow

Optical flow addresses the idea of determining apparent motion based on changes in image intensity (i.e. brightness) over space and time [4]. All of the proposed differential and matching (feature-based) techniques proposed in Trucco & Verri [4] go beyond the scope of this project. The simplest of these algorithms involve derivatives of the brightness constancy equation and a least squares solution at each pixel. Not only would they be difficult to implement in both software and hardware, but they produce what is known as the motion field of the image. This is more information than what is needed for the Kalman filter algorithm that is discussed next, which requires simply knowing the (x, y) coordinate of the object in each frame.

A far simpler, less robust approach is used to locate the object in motion for each frame, known as Delta Frame Generation (DFG) [8]. This method makes the following assumptions about the image:

1. The object of interest is in motion.
2. The object of interest is the only part of the scene in motion.
3. The first frame of the sequence does not contain the object.
4. The lighting and background of the scene does not change between frames.

The delta frame is then calculated as

$$\Delta_i = |F_i - F_1|. \tag{4}$$

Where F_i is the current grayscale frame and F_1 is the base grayscale frame discussed in the previous section. Assuming the above assumptions hold, the delta frame, Δ, will be non-zero for only the object. Knowing that Δ contains only the object, drawing a line from the top/bottom most points, and finding its intersection with another line from left/right most points produces the center point of the object. The mathematical background for this is as follows. By means of simple search algorithms, the top, bottom, left, and right most points are located as

$$\begin{aligned} p_t &= (x_1, y_1), \\ p_b &= (x_2, y_2), \\ p_l &= (x_3, y_3), \\ p_r &= (x_4, y_4). \end{aligned}$$

Then the two lines are given as

$$\begin{aligned} l_1(t, b) &: y = m_1 x + b_1, \\ l_2(l, r) &: y = m_2 x + b_2. \end{aligned}$$

The intersection of these two lines, denoted (x^*, y^*) is given as

$$x^* = \frac{b_1 - b_2}{m_2 - m_1} = \frac{((x_1 y_2) - (y_1 x_2))(x_3 - x_4) - (x_1 - x_2)((x_3 y_4) - (y_3 x_4))}{D},$$

$$y^* = \frac{m_2 b_1 - m_1 b_2}{m_2 - m_1} = \frac{((x_1 y_2) - (y_1 x_2))(y_3 - y_4) - (y_1 - y_2)((x_3 y_4) - (y_3 x_4)}{D}.$$

The common denominator in both results is denoted D, and can be expanded as

$$D = (x_1 - x_2)(y_3 - y_4) - (y_1 - y_2)(x_1 - x_2).$$

When $D = 0$, the case of parallel lines occurs. In this case, the middle of the object is simply taken to be the average of the top and bottom points.

3.3 Kalman Filter

A Kalman filter is used to combine continuous predictions from a theoretical system model with continuous measurements from a real implementation of the same system. This is done to help reduce the effects of noise on system measurements, and can provide a prediction of the system's next state if for some reason the measurements fail or contain lots of noise. Implementation of a Kalman filter is highly intuitive for a sensor that has been characterized with testing. But for object tracking, it is much more difficult to understand as all of the Kalman filter jargon is focused around ”sensors” and ”measurements” that sound odd when referring to an object tracking system.

The Kalman filter equations can be divided into two key sets: the predication equations and the update equations [4], [12].

Prediction Equations:

$$\vec{x_{k+1}} = F \cdot \vec{x_k} + B \cdot \vec{u_k} \tag{5}$$

$$P_{k+1} = F \cdot P_k \cdot F^T + Q \tag{6}$$

Intermediate Calculations:

$$\vec{y_{k+1}} = \vec{z_k} - H \cdot \vec{x_{k+1}} \tag{7}$$

$$S_{k+1} = H \cdot P_{k+1} \cdot H^T + R \tag{8}$$

$$K_{k+1} = P_{k+1} \cdot H^T \cdot S_{k+1}^{-1} \tag{9}$$

Update Equations:

$$\hat{\vec{x_{k+1}}} = \vec{x_{k+1}} + K_{k+1} \cdot \vec{y_{k+1}} \tag{10}$$

$$\hat{P_{k+1}} = (I - K_{k+1} \cdot H) \cdot P_{k+1} \tag{11}$$

There are a few key observations that can be immediately made about this set of equations. Note that the distinction between intermediate calculations and predication equations is one that is not usually made in descriptions of Kalman filtering like the one by Trucco & Verri [4]. However, from an implementation perspective this distinction makes sense, as the system in which the Kalman filter is placed in (or main algorithm) is blind to these equations. Also notice that matrices and vectors without subscripts are ones that do not change each iteration, and remain constant from initialization to completion.

While all elements in this set of equations have physical meanings and distinct characteristics, for the sake of simplicity the only ones that need to be discussed in detail are the inputs, outputs, and constant variables. The two quantities already discussed are the measurement vector $\vec{z}$ of length m and the state vector $\vec{x}$ of length n. For this application, $m = 2$ since by methods of optical flow discussed previously we are able to get a 2D Cartesian coordinate for the object.

$$\vec{z} = \begin{bmatrix} x \\ y \end{bmatrix} \tag{12}$$

The length of the state vector depends on the model that has been chosen for the system. Two common models for object tracking applications are the constant velocity model and the constant acceleration model. The assumptions made in these models are self explanatory; $n = 4$ for constant velocity and $n = 6$ for constant acceleration. In this system, the constant velocity model is used.

$$\vec{x} = \begin{bmatrix} x_p \\ y_p \\ v_x \\ v_y \end{bmatrix} \tag{13}$$

The next system dependent variable is the state transition matrix F which is nxn. As equation 5 indicates, it describes how the theoretical, predicted behavior of the system changes with each iteration [12]. For a 2D constant velocity model, this matrix is just implementing kinematic equations:

$$x_{new} = x_{old} + t * v_x \tag{14}$$

With a similar equation existing for y. In matrix form this is

$$F = \begin{bmatrix} 1 & 0 & f & 0 \\ 0 & 1 & 0 & f \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{15}$$

Where f is the frame rate (i.e. the time step) described in equation 2. Also stemming from equation 5, the vector $\vec{u}$ describes any external inputs applied to the system between iterations. For a simple, constant velocity object tracking system, there are none. This makes the vector irrelevant as well the associated matrix B. Getting back to the measurements, the next matrix of interest is the measurement model H [4]. This is an mxn matrix that relates the predicated state to the measured value. For this application, it should take the vector x of length 4 and place it in the same vector space as z of length 2. Thus,

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}. \tag{16}$$

The final two constant matrices that must be initialized prior to starting are Q (nxn) and R (mxm). These matrices are related to the error in the theoretical model (i.e. air resistance, friction) and the error in the measurement model (i.e. your "sensors", in this case optical flow) respectively [12]. In theory, if there are vectors $\vec{w}$ and $\vec{v}$ that characterize the error in the model and measurement respectively, then Q and R should be diagonal matrices containing the variance of these vectors [4].

The final, and perhaps most important, value that should be touched on is the co-variance matrix P (nxn) which changes with each iteration. In simple terms, this matrix is a measure of how well the measurements follow the model, and thus how accurate the filter is [12]. Unlike the previously discussed values, P will eventually converge to certain values due to the iterative nature of the algorithm.

3.4 Fixed vs. Floating-Point

Fractional portions of numbers can be represented in either fixed or floating-point in hardware. The essential difference between these two data representations is that in fixed-point the decimal point (i.e. dividing the integer and fractional portions) does not really exist from a hardware perspective. It is the job of the programmer to know where the decimal point is for each binary string, and then use the data under that interpretation. Most modern computing solutions use floating-point numbers, as it allows the user to implement real numbers without constraint. For example, by default all numbers in MATLAB are 64-bit double precision [10]. A 64-bit architecture (i.e. all numbers are 64-bits long) as well as a floating-point unit take up considerable space in hardware [10], and embedded hardware solutions have limited resources and power. It is clear that floating-point data representation will not be possible for an application such as this, and fixed-point conversion of the software is necessary.

Fixed-point conversion works by scaling all real (non-integer) numbers by a large number and then rounding to make it an integer. The idea is that if the number is scaled large enough, rounding will only remove a very small portion of the number, if any. The scale factor is generally used as a power of 2 or power of 10. Powers of 10 are much more intuitive from a programming sense, as humans use base-10, but powers of 2 are much more useful for hardware implementation, as multiplication by a power of 2 represents a shift along a binary string. The object tracking algorithm discussed in the previous sections only uses addition, subtraction, multiplication, and division operations. There is one matrix inverse operation, but since it is a $2x2$ matrix it is treated as swapping entries and dividing. When all division operations (there are only two) are treated as multiplication by an inverse and subtraction as addition by a negative number, the algorithm can be converted from floating-point to fixed-point by implementing the algorithms for conversion, addition, and multiplication discussed in [10].

MATLAB implementation of the three algorithms just mentioned are shown below. Note that every fixed-point number has an associated word length W and a fractional portion F. W denotes how long the binary string used to represent the number is, and F denotes the location of the decimal point from the least-significant (rightmost) bit. These two values are measured in number of bits.

Fixed-Point Conversion:

```
function [fixed] = floatToFix(float, F)

    global archW;

    d = float .* 2^(F);
    fixed = round(d);

    %Verify that enough fractional bits were used to represent this number
    if (float(1,1) ~= 0 && fixed(1,1) == 0 && F > 0)
        disp('ERROR: The chosen F is too small to represent this value.');
    end

    %Verify that the value is not too large for the desired architecture
    binaryString = length(dec2bin(abs(fixed(1,1))));
    if (binaryString > archW)
        disp('ERROR: Value larger than desired architecture.');
    end
end
```

Fixed-Point Addition:

```
function [result, F] = fixedAdd(fix1, F1, fix2, F2)

    global archW;

    if(F1 > F2)
```

```
        fix2 = fix2.* 2^(F1-F2);
        F = F1;
    else
        fix1 = fix1.* 2^(F2-F1);
        F = F2;
    end

    result = fix1 + fix2;

    %Verify that the value is not too large for the desired architecture
    binaryString = length(dec2bin(abs(result(1,1))));
    if (binaryString > archW)
        disp('ERROR: Value larger than desired architecture.');
    end

end
```

Fixed-Point Multiplication:

```
function [result, F] = fixedMult(fix1, F1, fix2, F2)

    global divisorF;
    global archW;

    result = fix1*fix2;
    F = F1 + F2;

    %Normalize any values that need more than the max fractional bits
    X = divisorF;
    if (F > X)
      norm = F - X;
      result = floatToFix(result, -norm);
      F = X;
    end

    %Verify that the value is not too large for the desired architecture
    binaryString = length(dec2bin(abs(result(1,1))));
    if (binaryString > archW)
        disp('ERROR: Value larger than desired architecture.');
    end

end
```

Going through the floating-point software and replacing all non-integer numbers with their converted value as well as replacing all operations with these algorithms converted the software to fixed-point. Finally, it is important to note that using these algorithms recursively or repeatedly causes the fractional portion F to grow significantly. Due to architecture constraints described above, it is necessary to normalize the fixed-point numbers by ”chopping” extra bits when there are too many. This is done in MATLAB by calling `floatToFix.m` with a negative fractional component, and would be done in VHDL or other hardware description languages by simply removing bits since they are accessible binary strings.

4 Requirements

Derived from the background research performed in section 3 and the purview of the project discussed in section 2, the completed system shall meet the following requirements.

4.1 Time Frame

The project shall be completed by December 2015. The project will be broken into two phases. Phase 1 of the project will be January - April 2015, and Phase 2 of the project will be September - December 2015.

4.2 Algorithm

An algorithm shall be chosen such that the system is capable of tracking an object that is in motion, in real-time. The algorithm will be simple enough such that it can be implemented in hardware within the project's duration and within the scope of the author's skills. To ensure this, the chosen algorithm will contain basic operations in order to ease hardware implementation. The algorithm will be chosen during Phase 1 of the project.

4.3 Software

Software implementation of the algorithm shall be performed during Phase 1 of the project to act as a proof of concept for the algorithm, and to understand the strengths and weaknesses of the algorithm. Software implementation shall be performed in MATLAB. Upon completion, the software will be able to track an object in motion with a clear and visible cursor, for an input video that satisfies all assumptions and prerequisite conditions the algorithm may have.

4.3.1 Floating-Point

The first iteration of the software implementation shall be in floating-point to provide a high-level structure of the algorithm. The software shall not have any dependencies on MATLAB (i.e. use built in functions) with the exception of simple helper functions such as `eye`, `zeros`, `round`, and `fix`. MATLAB's `VideoReader` and `VideoWriter` class may be used for I/O.

4.3.2 Fixed-Point

The second iteration of the software implementation shall be in fixed-point to make hardware implementation easier and more efficient. No values in the fixed-point software shall have fractional portions (e.g. reals, floats, or doubles) with the exception of constants declared directly before conversion and preparing data for MATLAB's `VideoWriter` class as output.

4.4 Hardware

Hardware implementation of the algorithm shall be performed during Phase 2 of the project. The completed hardware implementation of the algorithm shall be the final system. The system will contain embedded hardware which the algorithm has been implemented on. The system will include a video camera providing a live feed of the chosen scene. Upon completion, the system will be able to track an object in motion in real-time and identify the object in motion on a display with a clear and visible cursor. All components of the system shall be as low priced as possible.

5 Design

5.1 Platform

During the beginning of the project, rapid research was performed in order to determine what the best embedded hardware solution would be to implement the object tracking algorithm on. This design choice needed to incorporate findings from similar projects in order to avoid common failures, as well as factors specific to this project.

The main embedded hardware solutions seen today for applications that require dedicated computing are microprocessors, ASICs, and FPGAs. Prior to weighing the best solutions, it is useful to identity the challenges that come with video processing in real-time. Challenges that are paramount in embedded video processing are algorithm complexity, timing requirements, and the need for high processing power (fast clock cycles) [2]. An FPGA surpasses microprocessors and ASICs in a number of ways for video processing solutions [1], [2]:

- Full control over the internal workings of the FPGA allows for a parallel architecture unlike microprocessors.
- Configuration after completion allows for bug fixes and hardware modifications.
- Internal logic of microprocessors is not custom, and thus not application specific.
- ASIC development cost is extremely high.
- Video signals have moved to digital formats (some exceptions such as VGA), making programmable logic solutions more relevant.

In addition to these findings, an Altera White Paper [3] on video processing describes the ideal architecture video processing hardware should have as high performance, flexible, easy to upgrade, and low priced to develop. From these past projects and studies it is clear that the best hardware solution for algorithm implementation is an FPGA.

The next key topic for research was choosing the specific FPGA manufacturer to use. Choosing the right manufacturer is important because the quality of the tools, resources, and support they provide will have a major impact on the ease of development. Starting with the research of similar projects, Roth [1] chose to use an FPGA from the Altera Cyclone family for a real-time video application over Lattice of Xilinx due to the fact that Lattice had poor online support an Altera was lower cost than Xilinx. It was also found that Altera Quartus II was superior to Xilinx ISE Design Suite because it encapsulated the optimization and configuration process into one piece of software (i.e. Quartus II can perfrom compile, simulation, and programming) [1]. In a similar project, Saeed et al. [2] took advantage of the Altera DE2 breakout board by using it's on-board composite video input port as well as the video decoder module.

Based on these two projects, it was very convincing that Altera should be the manufacturer of choice. However as mentioned before, it is equally important to account for the requirements of this project specifically when choosing the platform for development. Independent of [1], [2], and [3], an Altera breakout board such as the DE1 or DE2 seemed to be the best solution for multiple reasons.

Familiarity: Both members of this project have experience using Altera Quartus II with VHDL on a Cyclone II FPGA. Using familiar hardware and development tools will make implementation in Phase 2 easier.

Avoiding Fabrication: Due to the constraints on time and money in this project, it will be best to avoid having to fabricate a custom breakout board for the chosen FPGA and any necessary I/O or peripherals. Due to this, the major manufacturers such as Altera, Xilinx, and Lattice should be assessed on their breakout boards, and not just their FPGAs.

Cost: This project has no funding, and the hardware platform will not be expanded past a single working prototype for this project. Due to this, a low-cost (ideally free), educational development board is the ideal solution. McGill has Altera DE1 and DE2 boards that may be available for use during Phase 2 of this project. If not, the price of the Altera DE1 board is only $127 for academic use ($150 commercial) and the DE2 board is only $284 for academic ($495 commercial) as well. Investigating comparable solutions by Xilinx for video and image processing applications found the Spartan-6 FPGA Embedded Kit which costs $695.

Peripherals: In order to give the system real-time video capabilities, the breakout board would ideally have some hardware for video decoding and display generation. This is not a necessity, as the live video input feed could be streamed digitally from a computer using a serial connection like USB. The live video output feed could be streamed back to the computer to be displayed. The scenario just described would use the FPGA only for processing, making the full system require a CPU. However, if the entire system was to entirely autonomous from a laptop of other CPU, the breakout board would need to be able to handle incoming video data (analog or digital) and then drive a display. Examining the schematics of the Altera DE1 and DE2 boards in Figures 4 and 5 located in Appendix A shows that the DE2 is capable of being a fully functional system because of the 10-bit high speed video DAC (ADV7123) between the FPGA and VGA output port while the DE1 board only has a 4-bit RGB block that is speculated to not provide the needed output picture. It should also be noted that the DE2 board has a video decoder block (ADV7181) placed between the composite video (analog) input port and the FPGA while the DE1 does not.

In summary, it was decided based on the research of similar projects, as well as taking into account certain project specific parameters like cost, time, and familiarity that an Altera DE1 or DE2 board would be the best FPGA breakout board solution. The DE2 board comes equipped with peripherals and I/O that are speculated to make it a fully featured system if analog video input is used,. However a fully featured system not a requirement, as the goal of this project is to implement just the algorithm in hardware.

5.2 Algorithm

After choosing the platform, the next step was to choose the best algorithm to meet the requirements and goals defined for this project. There are a number of popular motion based object tracking algorithms for both software and hardware implementation. The top four algorithms were chosen and compared for a final decision. This section will outline these top four algorithms and discuss there merits, concluding with the reasoning behind the final design decision.

5.2.1 Delta Frame Generation

Shrikanth and Subramanian [8] outline a high-level algorithm for motion base object tracking that uses DFG described in section 3.2. The algorithm also applies a mean filter and median filter to the delta frame to reduce noise. The main advantages behind using this algorithm include the fact that [8] outlines pseudocode that would help during implementation, only basic operations are used (addition, multiplication, etc.), and the algorithm is extremely simple to understand. The main disadvantages of this algorithm stem from the fact that is not predictive, and just makes a measurement based on the current frame. This could be major issue for real-time implementation. The algorithm simplicity also means that it requires that all the assumptions outlined in section 3.2 must all be satisfied, making it not very robust.

5.2.2 Block Matching

The basic idea for this algorithm presented by El-Azim et al. [5] is that the all frames are divided into blocks, and the current frame is compared to the previous frame in order to find the best match for each block between the two frames. A mean absolute difference is used to quantify the best matches between blocks. The displacement vector between the two matching blocks is taken as the ”motion vector”. The larger motion vectors should correspond to the object while smaller motion vectors would correspond to the background. Some advantages to this algorithm is that is seems more robust then DFG by allowing some background motion to occur (i.e. changes in lighting) during the video. Pseudo-code is provided for a high-level outline of the algorithm which is helpful for implementation. The main concern about this algorithm is that [5] never discusses hardware implementation in any detail. VLSI implementation is mentioned, but there is no in depth study about the successes and failures of hardware implementation for the algorithm.

5.2.3 Kalman Filter

The use of a Kalman filter for object tracking appeared to be the most common algorithm based on its use in [4], [7], and [9]. The background behind using a Kalman filter for object tracking has been discussed in great detail in section 3.3. The biggest advantage to using this algorithm is the fact that it is predictive. For instance, if the object was to disappear briefly behind a portion of the background the algorithm could continue to track its motion. Upon initial study, a Kalman filter was very difficult to understand and the mathematical background was well beyond any other algorithms discussed. Hardware implementation also appeared to be quite difficult considering the recursive nature of the algorithm, and the fact that it requires a matrix inverse.

5.2.4 Adaptive Filter

The final algorithm considered to be a top prospect was an adaptive filter presented by Caner et al. [6]. In summary, the background for this algorithm is that there exists some system response matrix $h_o(x, y; x_0, y_0)$ which can map the current image I_1 to the next image I_2,

$$I_2(x_0, y_0) = \sum_{x,y} h_o(x, y; x_0, y_0) I_1(x, y) + e(x_0, y_0).$$

The purpose of the adaptive filter is to estimate h_o using a negative feedback structure. An advantage to using this algorithm is the predictive nature of it, which could be very useful for real-time implementation. The biggest disadvantage was, again, that a very complex mathematical background needed to be understand to implement an adaptive filter, including Hilbert curves and finite impulse response (FIR) filtering. The complicated background produced many questions about both high-level (software) and low-level (hardware) implementation, which ultimately caused this algorithm to not be chosen.

5.2.5 Design Decision

After weighing the advantages and disadvantages of the four algorithms just discussed, it was decided the that Delta Frame Generation (DFG) would be the best algorithm to implement due to the simple nature of it. The source provided clear insight on how to implement the algorithm in software, and the success of the project proved that hardware implementation would certainly be possible. All operations in this algorithm were basic, and the algorithm was so linear it would be easy to add or remove filters as needed.

After discussing this design choice with the project supervisors, it was decided that DFG would not be a sufficient algorithm to meet the goals and requirements defined for this project. The non-predictive nature of the algorithm could present issues for providing real-time capabilities in the system. It was decided that a Kalman filter would also be implemented, and the algorithm would use DFG to produce measurements for the Kalman filter, which would produce a better estimate of the position and velocity of the object.

5.3 Software

The next step in the design, and where the majority of the time was spent during Phase 1, was developing floating-point software, and then converting this software to fixed-point. The software implementation was done in MATLAB due to the convenient I/O classes for video files as well as the software's portability and common use in the academic community. The high-level design of the software in displayed in figure 1, and the main script is located in `ObjectTracking_FP_v3.m`. The main script starts by loading in the video file using MATLAB's `VideoReader` and extracting important constants from it such as the number of frames, rows, and columns as well as the video's duration. Then the script iterates through each video frame and calls functions corresponding to blocks in figure 1. Below is a brief description of what each function does in reference to the theory discussed in this report.

`ObjectTracking_FP_v3.m`: Main script. Reads and writes video files, calls all other functions, and draws a red box around the location of the center of the object.

`RGB2GRAY.m`: Performs grayscale conversion described in section 3.1.

`deltaFrame.m`: Computes the delta frame described in section 3.2. Implements threshold filter where the threshold is computed based on the mean of the delta frame. Calls `myMean.m` which computes the mean of a matrix excluding zero values.

`medianFilter.m`: Performs median filtering by replacing each non-zero pixel of the post mean filtered delta frame with the median of its neighbors. Calls `myMedian.m` which computes the median of the input vector containing the neighbors using a k-th smallest sorting algorithm presented in [11].

`measure.m`: Determines the center of the object using the filtered delta frame based on the theory described in section 3.2.

`applyKalman.m`: Implements the Kalman filter equations presented in section 3.3.

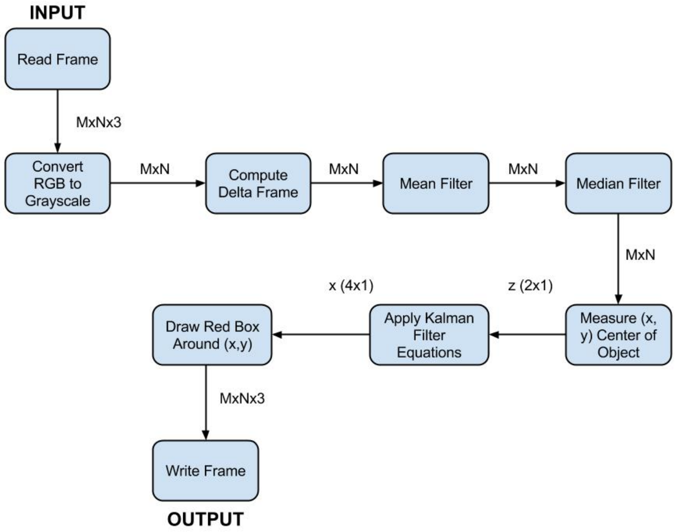

Figure 1: Software Flowchart

The major design challenges faced while writing the floating-point code included filtering and removing all MATLAB dependencies. It was important to not use any built in MATLAB functions in the algorithm as they will obviously not be available when doing hardware implementation. This is why mean, median, sorting, and grayscale conversion functions had to be written. The only MATLAB dependencies remaining in the code are the `VideoReader` and `VideoWriter` I/O classes, and simple helper functions like `fix`, `round`, and `eye` which can all be easily be replaced in hardware implementation. Initially a median filter was not used and the results were significantly worse, however the algorithm was much faster since a sort was not needed for each pixel in the delta frame. Tests were run to determined the performance and speed of the floating-point software with and without the median filter, and comparing MATLAB median functions against ours. The results of the these tests can be seen in Table 1.

Table 1: Floating-Point Software Median Filter Testing (20 second input video)

Trial #	Implementation	Execution Time (s)	Results
1	No median filter	14.875	Red dot deviates at noise, never recovers
2	`medfilt2`	19.430	Red dot deviates at noise, but recovers fast
3	`medianFilter.m` with `median`	161.043	Same as trial #2
4	`medianFilter.m` with `myMedian.m`	151.3311	Same as trial #2

From these results, it was clear that a median filter would be necessary due to the improved results. It was also clear that there was good motive for implementing this algorithm in hardware, as the run-time for a 20 second input video was very long as seen from trial #4 in Table 1 when all of our functions were used.

The next step in software development was converting the stable floating-point code to fixed-point. This process was described in section 3.4, and the converted main script is located in the file `ObjectTracking_FI.m`. The helper functions used in fixed-point conversion `floatToFix.m`, `fixedMult.m`, and `fixedAdd.m` are also presented in section 3.4. It was important to perform constant error checking and normalization during this process. All numbers that are converted from floating-point to fixed-point must have enough bits allocated to their fractional portions such that they are non-zero after conversion. For instance, here is an example of what a failed conversion would look like. Say,

$$X_{float} = .0453$$

Now applying `floatToFix.m` defined in section 3.4 with $F = 3$,

$$X_{fixed} = X_{float} * 2^3 = .3624,$$

$$X_{fixed} = round(X_{fixed}) = 0.$$

Clearly, $F = 3$ is not enough fractional bits to represent this number, as the original number cannot be recovered from zero. This is how the minimum F was determined for a properly functioning algorithm. A global variable is set for the F that is needed for all the very small numbers, and it is increased until the algorithm did not produce error messages. In a similar fashion, it was also important to determine the minimum W that was needed for the algorithm to work properly. After every floating point conversion or operation, the number of bits needed to represent the number is counted and must be less than another global variable.

Normalization was also very important in the conversion process. Fractional portions grow during fixed-point conversion due to the fact that the two fractional portions of the operands add during a multiplication operation. A normalization routine was embedded into the multiplication function `fixedMult.m` to ensure all results are under the global variable for the F just described. Normalization is also performed at the end of all functions where fixed-point conversion occurs, so all numbers in the main script have $F = 0$. A normalization is also performed in the middle of the Kalman filter `applyKalman.m` since so many multiplication operations occur. In MATLAB, normalization is performed by using `floatToFix.m` with a negative fractional portion. However, in VHDL this will be as simple as removing all bits to the left of the imaginary decimal point since all numbers are binary strings.

The initial design hoped to use 10 bits for integer portion as the largest number in the floating-point code was 1000, and then use the remaining 6 bits for the fractional portion. By the testing and error checking just described, it was quickly seen that this would not work. The two division operations (approached as multiplication of the inverse) that occur once per frame produced extremely small numbers that required at least 20 bits for a fractional portion. Due to this, a larger than desired architecture will be needed for hardware implementation. The fixed-point conversion was completed with a final architecture of 20 fractional bits and 11 integer bits,

$$(W, F) = (31, 20).$$

Note:
All fixed-point software can be found at:

`https://github.com/bbrown1867/ObjectTracking/tree/master/sw`

All floating-point software can be found at:

`https://github.com/bbrown1867/ObjectTracking/tree/floating_`
`point_stable/sw`

6 Future Work

The most immediate future work that needs to be completed is finishing fixed-point software implementation by finding the bug in `applyKalman.m` described in the previous section. Once the fixed-point software is complete, hardware description must be completed by translating the MATLAB code to VHDL. It is expected that this will not be very time consuming given that we have written fixed-point software with no dependencies. Once this is completed, or perhaps while it is being completed, interfacing the FPGA with a camera and display and handling the incoming video data will most likely be the biggest challenge in Phase 2 of the project. Figures 2 and 3 show two proposed system architectures for Phase 2 of the project. Decisions will need to made about whether or not the system should be entirely self-sustaining (i.e. completely embedded) like in Figure 2 as opposed to a system which utlizes camera and display resouces from a laptop of desktop CPU like in Figure 3.

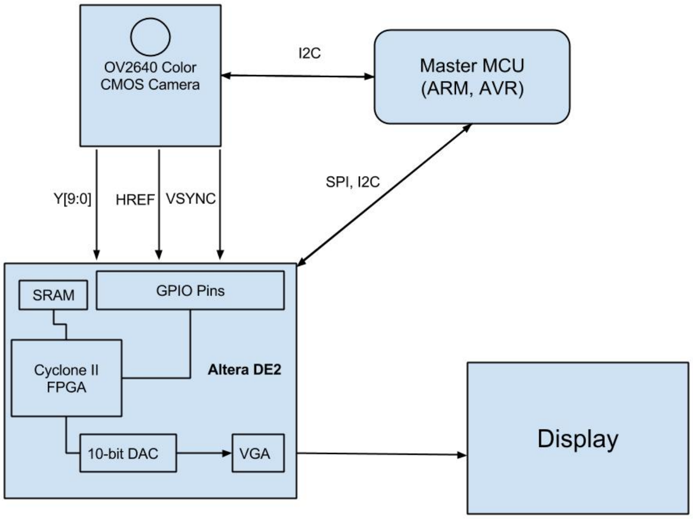

Figure 2: Proposed Hardware Implementation #1

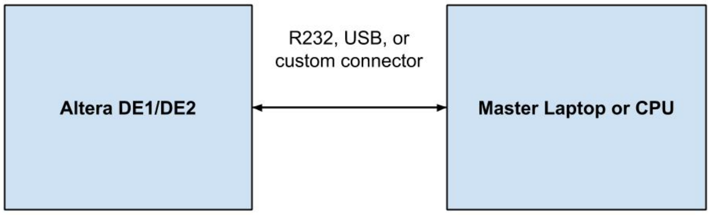

Figure 3: Proposed Hardware Implementation #2

7 Impact on Society

Our dedicated hardware object tracking system will have some impacts in society, but no profound negative impacts. The most notable impact will be its automating effect in the applications it is integrated with. Its introduction to another system will fulfill a task that could have been previously accomplished by a human or even by software previously embedded within the system. The consumer would have to purchase our hardware to implement our system. Our choice of hardware, the Altera DE2 Board has an upfront cost of around $500. This can be a substantial price for the consumer but the performance benefits outweigh that of embedded software, and in the case of surveillance for example, the cost of running our system will be less than paying a worker to watch a video feed for objects. The introduction of new technology that makes a manned job obsolete is often viewed as a negative impact on society, but this sort of progress is inevitable.

In the design process so far, we have not made any sort of environmental impact as we have just been developing software. Our system is going to be a combination of previously manufactured parts which were presumably made with the environment in mind. There are so many electronics already in use in society today, the addition of our system's environmental impact can not be quantified. Our product does not produce waste nor does it need constant physical additions. It requires a small constant supply of electricity. When it is no longer needed it can be properly recycled in the same way that computers and other electronics are. Our system does not pose any health or safety risks.

As mentioned earlier, the value of our device comes from the additional performance gained by a secondary system using it. It is possible that the secondary system has some sort of malicious intent but our product cannot be responsible for any negative consequences the secondary system produces.

8 Allocation of Work

Work allocation was quite simple as a team of two members, and there were never any issues about work disparity or in communication. Using Google Drive for document management and Github for software development made callaboration on notes and code very simple. The algorithm and platform research was easy to divide in two, as both of us researched independently and took notes in shared documents on the Google Drive. We would then meet to discuss the conclusions drawn from the research and what design decisions to make from them. Floating-point software development was split 50/50 and fixed-point software development was split about 60/40, with Ben doing more work towards the end of the semester.

The only real challenge we faced was finding collaborative ways to develop software together, which we both feel is more natural to do alone. This was remedied by individually writing functions, and then meeting to debug code together when issues arose. It is difficult for us to identify all individual contributions due to the dynamic of such a small group, but major individual contributions would be Ben researching and developing the Kalman filter and Taylor researching and developing DFG and extracting measurements from it.

For Phase 2 of the project, we plan to keep the work allocation similar to Phase 1, and feel that hardware implementation does not present any situations that cannot be approached in a similar way.

9 Conclusion

Phase 1 of this project has provided a framework both theoretically and practically for Phase 2 of the project, where the ultimate goal of hardware implementation for a complex video processing algorithm will be done.

In section 5.1 it was made clear that an Altera FPGA would be the best hardware platform for implementing the system, and ideally a breakout board like the DE1 or DE2 would provide easier prototype development. Section 5.2 showed that the best algorithm for motion based object tracking would be using Delta Frame Generation to provide object measurements to a predictive Kalman filter, which would take this measurements in conjunction with a theoretical model and produce improved coordinates. Finally, section 5.3 described the process of implementing the chosen algorithm in software, and proved from the extremely slow execution time that practical applications of this algorithm require dedicated hardware. In conclusion, Phase 1 of the project was a complete success aside to the difficulties encountered when converting the Kalman filter from floating to fixed-point, that will hopefully be resolved in the near future. This phase of the project taught us a lot about processing and interpreting video data for practical applications, and exposed us to the challenges that come with implementing a motion based object tracking system in software. Hopefully, others can learn from these experiences.

References

[1] F. Roth, “Using low cost FPGAs for realtime video processing”, M.S. thesis, Faculty of Informatics, Masaryk University, 2011.

[2] A. Saeed et al., “FPGA based Real-time Target Tracking on a Mobile Platform,” in 2010 International Conference on Computational Intelligence and Communication Networks, 2010, pp. 560-564.

[3] ”Video and Image Processing Design Using FPGAs.” Altera. 2007. January 2014. http://www.altera.com/literature/wp/wp-video0306.pdf

[4] E. Trucco and A. Verri, “Chapter 8 - Motion,” in Introductory Techniques for 3D Computer, pp. 177-219.

[5] S.A. El-Azim et al., “An Efficient Object Tracking Technique Using Block-Matching Algorithm”, in Nineteenth National Radio Science Conference, Alexandria, 2002, pp. 427 - 433.

[6] Caner et al., “An Adaptive Filtering Framework For Image Registration”, IEEE Trans. Acoustics, Speech, and Signal Processing, vol. 2, no. 2, 885-888. March, 2005.

[7] Yin et al. *Performance Evaluation of Object Tracking Algorithm* [Online]. Available: http://dircweb.kingston.ac.uk/

[8] G. Shrikanth, K. Subramanian, “Implementation of FPGA-Based object tracking algorithm,” Electronics and Communication Engineering Sri Venkateswara College of Engineering, 2008.

[9] E. Pizzini, D. Thomas, “FPGA Based Kalman Filter,” Worcester Polytechnic Institute, 2012.

[10] M. Shabany. (2011, December 27). *Floating-point to Fixed-point Conversion* [Online]. Available: http://ee.sharif.edu/ digitalvlsi/Docs/Fixed-Point.pdf

[11] N. Devillard. (1998, July). *Fast median search: an ANSI C implementation* [Online]. Available: http://ndevilla.free.fr/median/median.pdf

[12] D. Kohanbash. (2014, January 30). *Kalman Filtering - A Practical Implementation Guide (with code!)* [Online]. Available: http://robotsforroboticists.com/kalman-filtering/

Appendices

A Additional Figures

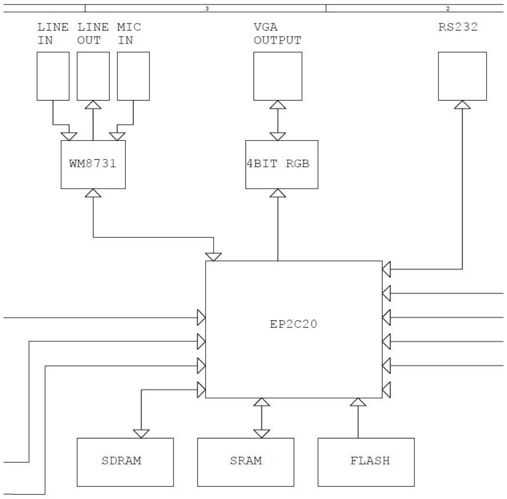

Figure 4: Altera DE1 Block Diagram (Partial)

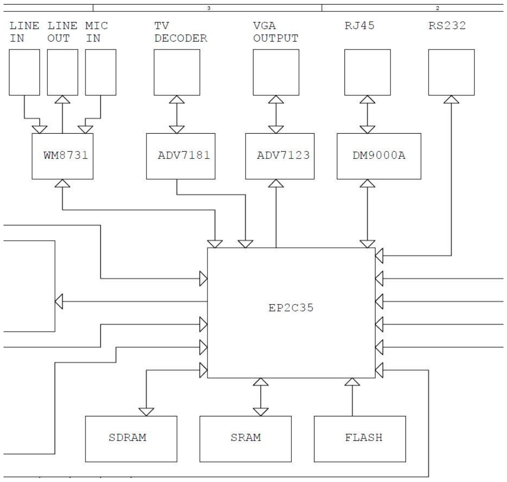

Figure 5: Altera DE2 Block Diagram (Partial)