Computational Circuit for use in Real Time Color Amplification

by

Andrew Yokubaitis
Vinayak Raju

Senior Project
Electrical Engineering Department
California Polytechnic University
San Luis Obispo
December 2016

Table of Contents

Abstract

The Eulerian Video Amplification algorithm developed at MIT allows for amplification of very small changes in seemingly static video scenes. The algorithm helps visualize these small changes by amplifying them to pull out signals of interest, such as the human pulse and the motion of hot air. This algorithm is computationally intensive and, as is, is difficult to run in real-time as a serially executed program. Our project is to design a parallel processing circuit that allows the algorithm to be calculated in hardware in order to magnify these small changes in real time. This will be done by using a highly parallel circuit for computing a Gaussian Pyramid Reduction calculation. The main advantage of a hardware solution is that we can exploit parallelism in order to calculate multiple parts of the algorithm at once, rather than having to perform thousands of calculations in sequence. Unlike previously attempts to develop software methods make to run the algorithm run in real-time, a highly parallel hardware solution allows us to not only run faster, but also save all the intermediate calculations thereby allowing better detection and amplification. The circuit is built to intake a frame of 32 x 32 pixels, and undergo 5 reductions before producing a single reduced pixel that is used in the amplification calculation. Due to multiple issues we had using Cadence we were able to simulate, but not synthesize our Verilog blocks. Unfortunately we could not obtain any timing diagrams or test bench results but, assuming a clock speed of 16MHz and the worst case scenario where only 1 calculation is performed per cycle, the total time from inputting a 32 x 32 frame to receiving an amplified frame out is about .00477 seconds. Since most cameras record at a rate of 60Hz, which equates to .0167 seconds per frame, we are well within the margins needed to be considered “real time” because we can calculate the amplified frame about 3 times faster than the camera can record. Unfortunately, doubling the size of a side of each frame increases computation time nearly 4 fold. This means that, assuming a lock speed of 16MHz and worst case scenario computation time, our circuit could not longer perform in real time if the frame size was increased to 64 x 64 pixels.

Chapter 1: Introduction

1-1: Background and General Introduction

The Eulerian Video Amplification algorithm developed at MIT takes small changes in color or motion in a video and amplifies those changes to make them visible to the human eye. The algorithm helps visualize these small changes by amplifying them to pull out signals of interest, such as the human pulse and the motion of hot air. This algorithm is computationally intensive and, as is, is impossible to run in real-time when computed in serial. The parallel processing circuit we designed allows the algorithm to magnify these small changes in real time. This will be done by using a highly parallel circuit for multiplying and averaging RGB color values in hardware. The primary application we are looking into is for heart rate monitoring. Each time your heart beats, the capillaries in your skin dilate, and your skin turns redder. These subtle changes are not visible to the naked eye, but when ran through the Eulerian Video Amplification algorithm these changes become much more apparent. Figure 1.1 below shows a few frames of an unmodified input video as seen by the naked eye.

Figure 1.1: Unmodified Input Video [1]

At first, there seems to be no change between any of the frames, but after running the file through the video amplification algorithm, the frames visibly change color and the man's heart beat can be detected as seen in Figure 1.2 below.

Figure 1.2: Eulerian Color Amplification Video [1]

We can only show individual frames in this report, but when observing the preprocessed video, it is easy to see the man's face fading from the redder version in frames 1 and 3 to the paler versions in 2 and 4. In effect, this is

visualizing his heart beat since the redder complexion is caused by capillary dilation due increased blood flow from the beating heart. This change can be tracked over the course of the video and his heart rate can be calculated purely from observing his face.

While the example here is to capture heart rate, color amplification can also be used for such things as biometrics, chemistry, outdoor gas leak detection, and many other potential applications. Multiple sensors can also be used in tandem for tracking propagation of fluids through a medium such as observing the blood flow through your arm, or outdoor gas leak detection. This introduces a much wider range of uses which are limited only by the ingenuity of the engineers designing systems with this technology.

One problem that currently exists in hospitals is every patient requires an individual cardiac monitor. Our product will allow multiple patients' heart rates to be monitored simultaneously by detecting multiple people in the same video frame. Another problem that currently exists is doctors have a long time diagnosing injuries and diseases. Our product could allow doctors to quickly observe how fast blood is flowing in certain parts of the body and could potentially enable them to better diagnose injuries, such as burns, malnutrition, and diseases, like skin cancer and other skin conditions.

1-2: Advantages of a Highly Parallel Hardware Solution

Our project allows two main improvements over a software solution. First, allowing real time color amplification through the use of a highly-parallel hardware algorithm instead of software allows for faster, more robust calculation of the algorithm and has the ability to keep as much detail as the original algorithm without slowing the calculation down. Secondly, because of much smaller memory requirements and the absence of need to save video, a hardware solution increases the versatility of the algorithm, enabling it to be used in a wider range of monitoring systems. In essence, our circuit will be mainly composed of two sub-circuits: the pixel reduction circuit that computes the Gaussian Pyramid Reduction, and the Color Amplification Circuit that actually amplifies the desired color in the final frame. Each circuit will be described in more detail in following sections.

Chapter 2: Theory and Explanation

2-1: Theory

Of the various forms of image processing used in computer vision, one basic technique is known as pyramid representation. Pyramid representation is done by taking an image or signal and subjecting it to repeated blurring or subsampling, which allows the user to extract specific information depending on the type of pyramid representation used. There are two main types of pyramids: lowpass and bandpass. A lowpass pyramid is made by blurring the image with an appropriate blurring filter and then subsampling the blurred image, usually by a factor of 2 along each coordinate direction. Each resulting image is then subjected to the same procedure, and the cycle is repeated multiple times. Each cycle of this process results in a smaller image with increased blurring, but with decreased image resolution. If illustrated graphically, the entire multi-scale representation will look like a pyramid, with the original image on the bottom and each cycle's resulting smaller image stacked one atop the other, much like Figure 2.2 below. On the other hand, a bandpass pyramid is made by calculating the *difference* between images at adjacent levels in the pyramid and performing some kind of comparison between adjacent levels of resolution, to amplify pixelwise differences. There are also two different pyramid generation kernels used to perform the image blurring calculation: Gaussian Pyramids, and Laplacian Pyramids. In a Gaussian pyramid, subsequent images are weighted down using a Gaussian average and are then scaled down. Each pixel contains a local average that corresponds to a pixel neighborhood on a lower level of the pyramid. This is the kernel utilized in our circuit because it lends itself well to color amplification. A Laplacian pyramid is very similar to a Gaussian pyramid but saves the difference image of the blurred versions between successive levels. Only the smallest level is not a difference image to enable reconstruction of the high resolution image using the difference images on higher levels. This technique is utilized in MIT's motion amplification algorithm which is not discussed in this document.

MIT came up with an algorithm to amplify small instances of color change on a pre-recorded film. The basic approach was to consider the varying color values at any location (pixel) and amplify the variation based on how adjacent pixels are changing. MIT used a Gaussian pyramid type calculation to extract and visually reveal the corresponding changes in color. They considered varying color at any location to the value of a pixel in a frequency band and applied a

bandpass filter to extract the frequency bands of interest. The results were generated using non-optimized MATLAB code and the computation time per video was on the order of a few minutes. The main use of the software was essentially serving as a microscope for temporal variations in previously recorded video. This analysis allowed them to amplify and visualize the pulse signal at each location on the face.

The project has been done by Cal Poly students, but their solution required them to eliminate the intermediate calculations when calculating the reconstruction of the pyramid. This disallowed the solution to be used for mid-size objects, as the intermediate steps of the pyramid were left out. Our solution saved the intermediate calculations and allowed full reconstruction of the pixel frame. This allowed us amplify the color change of all sized objects.

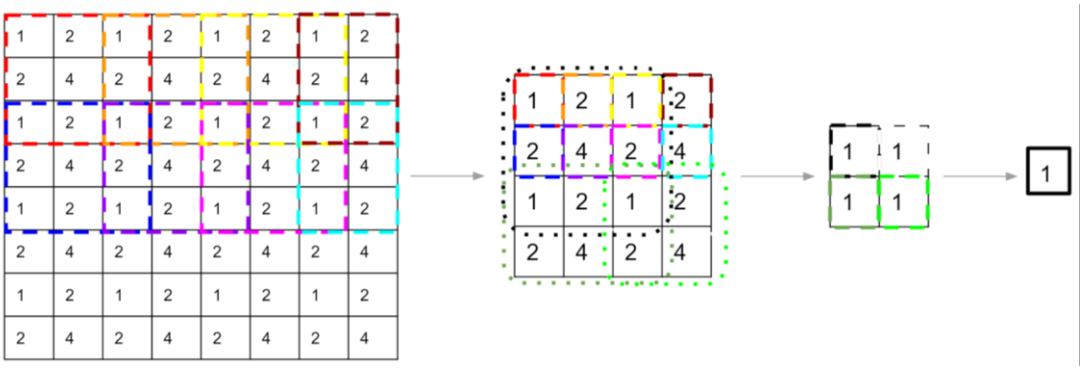

Figure 2.1: Pixel Frame Reduction [2]

Figure 2-1 above shows a visual representation of how the mathematics behind the Gaussian pyramid calculation are performed. Each individual box represents a pixel, and the number in the box is the “weight” of that pixel used during the calculation. To perform the calculation, first each frame of pixels is broken up into overlapping boxes of 9 pixels, such as those enclosed by the colored boxes in the far left frame of figure 2.1. The RGB color value of each pixel is then multiplied by its weight, then all the pixels in the box of 9 are summed up, effectively reducing the block of 9 pixels to a single pixel ready to be used in the next stage of the pyramid. This reduction is illustrated in figure 2.1 and can be seen by following boxes of similar colors as the reduction calculation is performed. For example, the red group of pixels in the top left corner of the 8x8 frame reduces to the single red pixel in the top left of the 4x4 frame. The exact same operation is performed on all the colored groups in the 8x8 frame and their resulting pixel can be seen in the 4x4 frame, and the same calculation is then performed on the 4x4 frame to reduce it to the 2x2 frame, then eventually a

single pixel. Once reduction to a single pixel has been performed, the pixel's RGB value is broken up into its constituent red, green and blue values. The value of the desired color to be amplified is then extracted and is compared to its value in the following reduced frame to give a change over time for the given color. This change is generally very small and is invisible to the human eye (such as a heartbeat), but the change is then amplified and extrapolated back to the original frame size to perform the actual color amplification.

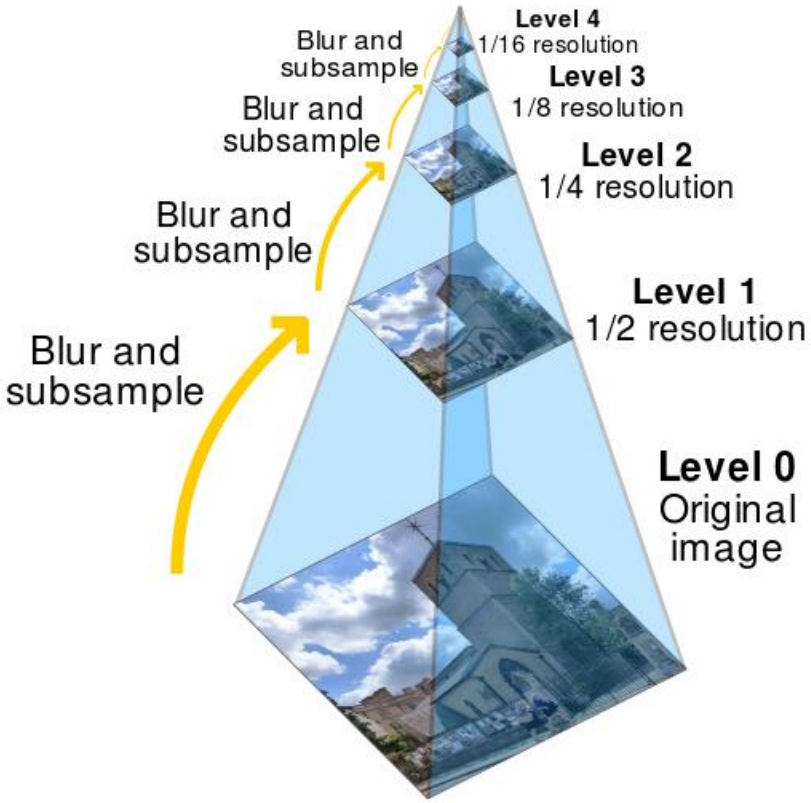

Figure 2.2: How the Gaussian pyramid works [3]

Figure 2.2 above provides a better visual of how the Gaussian pyramid reduction is performed. In a Gaussian pyramid, subsequent images are weighted using a Gaussian average, and scaled down. Each pixel contains a local average that corresponds to a pixel neighborhood (block of 9 pixels) on a lower level of the pyramid. Essentially, the algorithm repeatedly blurs and resamples each successive frame, reducing its resolution after every operation, until the original frame is reduced to a single pixel.

2-2: Explanation

Calculating Gaussian pyramids is an extremely repetitive process that repeatedly multiplies and adds values in a very specific order. Due to the repetitive nature of the Gaussian pyramid calculation, we believed a highly parallel circuit could be created to calculate different parts of the algorithm simultaneously in order to decrease computation time, and increase the robustness of the data gleaned from performing the calculation. The project is composed of two different circuits: the pixel reduction circuit, and the color amplification circuit. The Pixel Reduction Circuit performs the actual Gaussian pyramid calculation to reduce the frame size, and obtain the amplification factor value. The Color Amplification circuit uses the amplification factor and a frame from the Pixel Reduction circuit and performs the actual color amplification on each frame. Each circuit was designed to perform their job quickly, and have minimal inputs and outputs in order to make wrapping the two circuits together as easy as possible. The two different circuits will be discussed more deeply in the following section.

Chapter 3: Circuit Design

3-1: The Problem

MIT's software that is currently used to perform the Gaussian pyramid calculation is slow, unoptimized, requires a full blown PC to perform the calculations, and cannot save intermediate steps. This drastically limits the use of the software because it can really only be used for analysis of an event after that event has taken place, instead of being utilized as a detection mechanism in real time systems.

3-2: Overview of the Solution

Our hardware is a parallel IC that intakes 24-bit RGB pixel values (8 bits for red, 8 for green and 8 for blue) for each pixel in the frame, and runs the calculations through matrix multiplication and addition based on pixel values at the input. This is different from previous attempts to make the algorithm run real-time in software, because the software version did not save the intermediate calculations, which foregoes the ability to run the algorithm over small sections of the frame, and the software takes much longer to complete the computation.

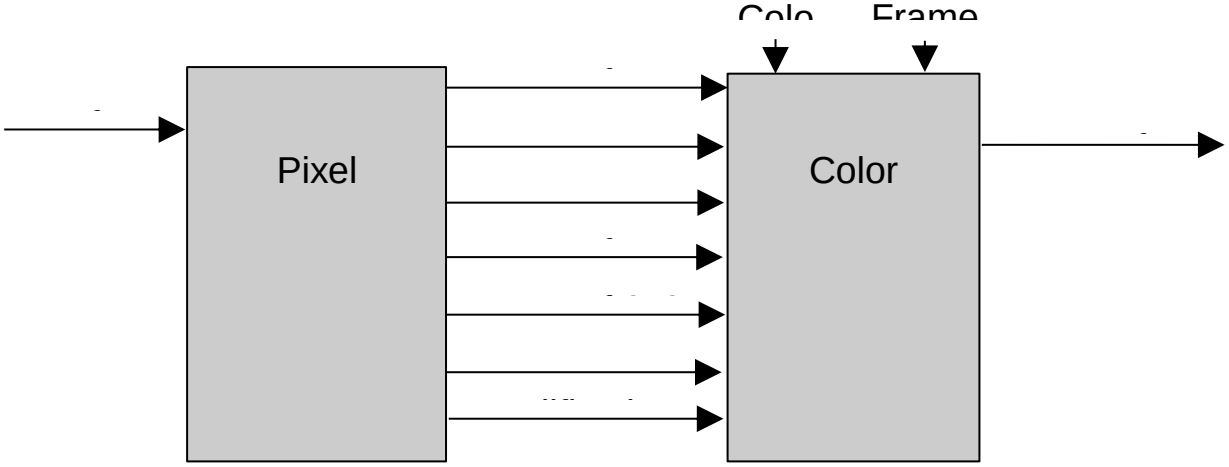

Figure 3.1: Overall Block Diagram

The circuit itself will be designed in Verilog using the Cadence software and verification tools. Each component will be built, simulated and attempt to be

tested in the Cadence environment, including construction and operation of the final circuit.

Figure 3.1 above illustrates the high level block diagrams and the inputs and outputs of the whole Eulerian Color Amplification Circuit. At a high level, the circuit works by intaking a video frame, or portion of a frame, (our circuit starts with a frame of 32x32 pixels) and passes each pixel from the frame into the pixel reduction circuit. This circuit utilizes Gaussian pyramid reduction to reduce frame size and calculate the modified color values for each pixel until a single frame consisting of a single pixel remains. This is effectively performing a Gaussian blur over the whole frame. After a full reduction calculation has been completed on a frame, the reduced frame is immediately passed to memory to be used by the color amplification circuit later on. A schematic and a more detailed description of this circuit can be found later in this chapter.

The color amplification part of the circuit takes in a specific frame of desired size, the desired color to be amplified, the frames fully blurred result from the Pixel Reduction Circuit, and the previous result from the Pixel Reduction Circuit. It then compares the R G and B values in the previous single blurred pixel to the newly blurred pixel by breaking them into their 8 bit chunks representing its Red, Green and Blue values and subtracting the current values from the previous in order to find the difference in each color between the two reduced frames. The actual amplification is performed by selecting the color to be amplified, then multiplying all of it’s values in the full sized frame by the difference in its value between the two single blurred pixels. Multiplying the whole frame by the delta between the single blurred pixels it effectively amplifies the change between two subsequent frames. This amplification factor can be increased or decreased simply by changing its value accordingly. We chose not to simply double the desired color, because we would need an extra bit if the color being amplified was over half of its max value. A schematic and a more detailed description of the circuit can be found below.

3-3: Pixel Reduction Circuit

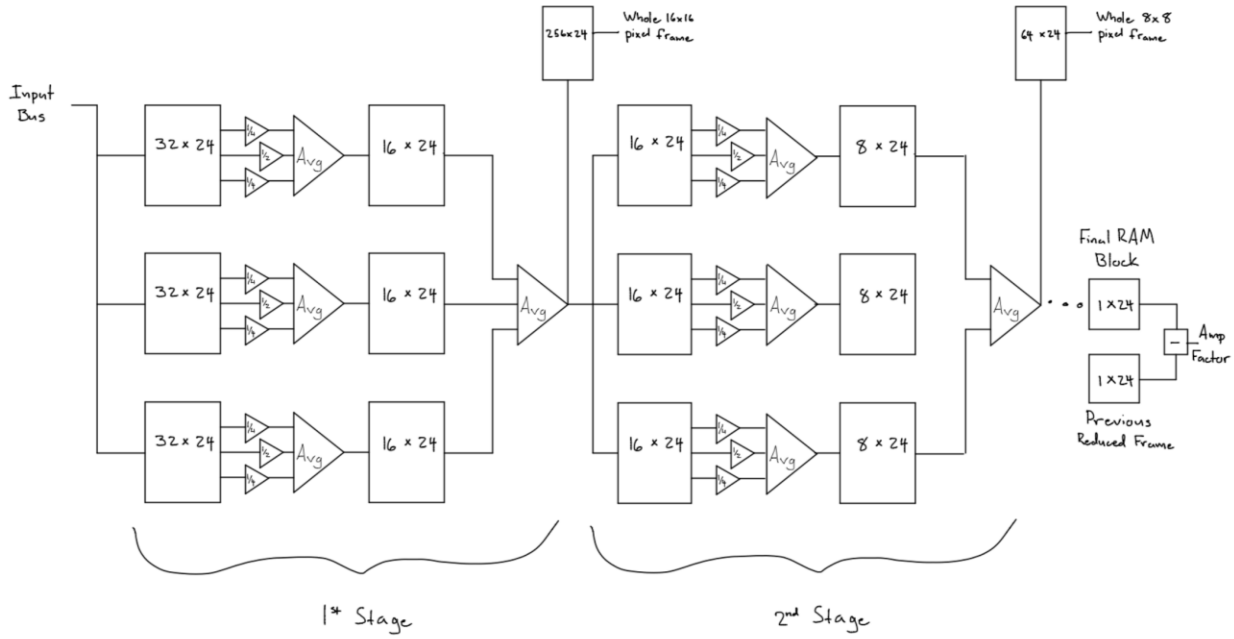

Figure 3.2: Low level block diagram of the Pixel Reduction Circuit

Figure 3.2 above is low level block diagram of the pixel reduction circuit. As the picture shows, the input is entered into three RAM blocks of size 32x24. The 32 represents how many pixels are on one side of the frame, and there are 24 bits in each pixel. There are 3 RAM blocks, because 3 rows of pixels are needed to start the calculation, and each block stores one row. Each pixel is weighted differently depending on its position in the frame. To begin the blurring calculation, the first 3 pixels from each RAM block are extracted and combined to create our first Gaussian kernel of 9 pixels. Each pixel is then broken up into its individual Red Green and Blue values. Once all the pixels have been split into their constituent colors, each color value is weighted according to its location in the kernel, and an average value is calculated for each color over the whole kernel creating a single weighted average for each color, then the individual color values are concatenated to produce the final, blurred, pixel. To perform the weighting calculation, values are either left alone, are halved, or are divided by four (we originally tried multiplying the values to perform the weighting function). Details about why we decided to go with division instead of multiplication will come later. Each blurred pixel from the first stage is then passed into the next stage where another row of RAM blocks, size 16x24, are used to store the reduced frame. This process basically just repeats itself, with smaller and smaller RAM blocks. The final output is a one cell RAM block which represents the 1x1 pixel frame, which is then compared to the previous single blurred pixel

to find the delta between the weighted averages. This delta will be referred to as the amplification value for the rest of this report. The Pixel Reduction Circuit consists of very similar, repetitive stages used to reduce the original frame. The arithmetic logic in each stage remains the same, and the only difference is the size of RAM blocks used to store the reduced frames. Each frame size is stored individually in its own RAM block. After the full calculation has been performed, data from all intermediate stages, as well as the amplification value, are passed onto the Color Amplification Circuit where the desired color is selected and amplified.

3-4: Reasons division is advantageous to multiplication:

Dividing the color values rather than multiplying them guarantees that the weighted average will always be under the maximum value for a given color. This prevents us from having to add additional bits to hold color values past their maximum 8 bit values and eases the difficulty of the calculation.To see how this division trick works, lets assume all colors in a kernel are the maximum value 0xFF. To simplify this example, let the maximum value of 0xFF be represented by 1.00 and the weights are applied like in Figure 3.3 below, so the calculation for finding the weighted average over the kernel with division is:

(.25 + .5 + .25) + (.5 + 1 + .5) + (.25 + .5 + .25) / 9 = .444

.25	.5	.25
.5	1	.5
.25	.5	.25

Figure 3.3: Gaussian kernel weights with division

Because the maximum weighted value is .444 of the absolute maximum, this shows, that no matter what colors we throw at the circuit, it will always produce a weighted average that is less than the maximum value. On the other hand, if we multiplied the heavily weighted blocks according the the weights in Figure 3.4 rather than dividing the lighter weighted ones, the calculation becomes:

(1 + 2 + 1) + (2 + 4 + 2) + (1 + 2 + 1) / 9 = 1.77

1	2	1

2	4	2
1	2	1

Figure 3.4: Gaussian kernel weights with multiplication

This clearly shows that performing the weighting function by multiplication could nearly double our output value. Since the values only increase with every subsequent calculation (because there is never a division or subtraction operation) this value could easily climb to multiple times the original maximum value after multiple weighted average calculations have been performed. This would require much larger RAM blocks to store the data, and would greatly increase the difficulty of our calculations

3-5: Color Amplification Circuit

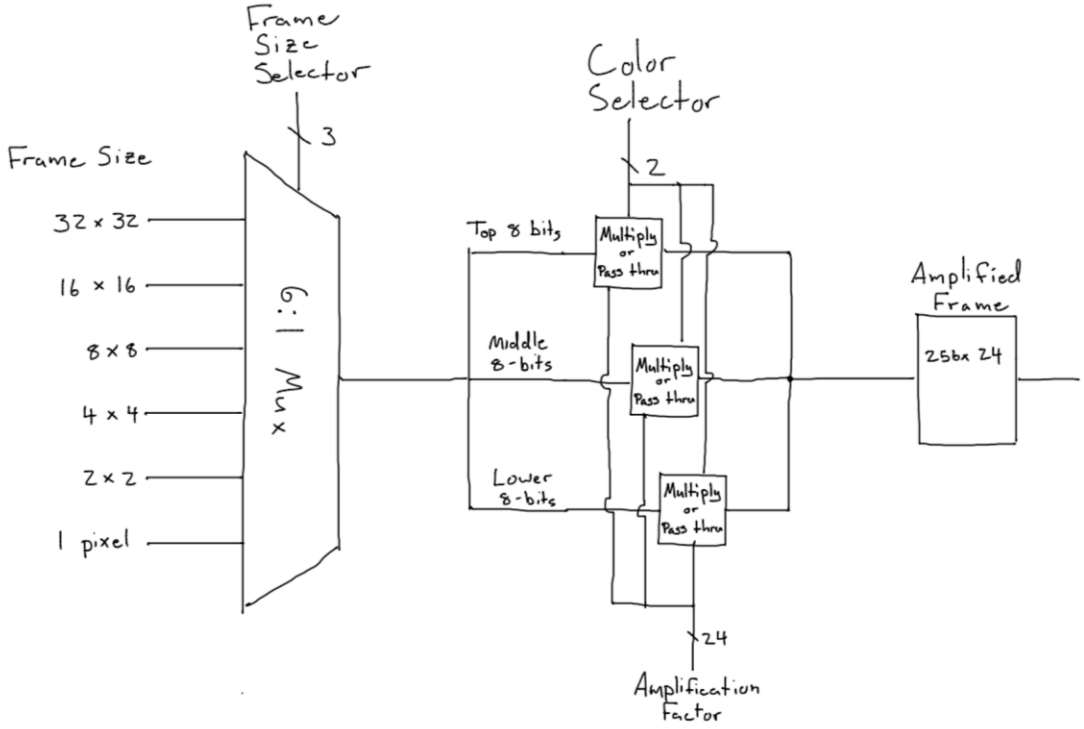

Figure 3.5: Low level block diagram of the Color Amplification Circuit

The color amplification part of the circuit performs the actual amplification of the desired color for the final frame. As inputs, it takes in a specific frame size from the Pixel Reduction Circuit via a MUX , the desired color to be amplified via a 2-bit signal, and the amplification value from the Pixel Reduction Circuit. Then the circuit breaks each pixel into its 8-bit R, G and B values and multiplies the desired color in every pixel by the amplification value + 0xFF. This ensures that the amplified value is larger than its initial value, but less than twice the maximum. Once the operation has been performed on all the pixels in the frame, the frame has finally been amplified and is ready to display on a screen.

Chapter 4: Testing and Verification

Testing and verification was by far the most difficult and frustrating part of the project for our team. While we were able to simulate the two separate blocks in Cadence Virtuoso and verify their correct construction, we were unable to synthesize the code due to the absence of the necessary .tcl files. We were unable to create the correct .tcl files as the required modifications were vague and unknown, however our main challenge was navigating through Cadence and its patched-together software.

The structure and workflow of the Cadence tools is rather aggravating because designing, simulating, synthesizing, verifying, and compiling the circuit are all done in different programs of Cadence's design suite. This means that files have to be created, saved, and accessed all in very specific locations, from multiple different programs, at different times in order to produce a working solution. Due to our limited linux experience and ignorance of most aspects of the Cadence software, we inadvertently produced many errors while trying to load or simulate files, each of which took very long to fix, or could not be remedied at all. Also, due to the proprietary nature of Cadence's software, there is a dearth of documentation and tutorials produced by Cadence about how to use their tools. Most of the tutorials and information we found online were made by professors at other universities for very specific classes, and were not of much use to us. This lack of documentation made it very hard to progress the project since it was very difficult to find solutions to many of the problems we encountered.

Below is the simulation verification screenshots showing that the current structure of the circuits does not produce any errors or warnings.

4-1: Simulation verification for the Pixel Reduction Circuit

```
[gen5xx@CadenceProd2 ~]$ sim-ncg.sh PixelReductionCkt.v
SOURCELIST:  PixelReductionCkt.v
Running sim-ncg.sh - Verilog stand-alone simulator
Calling setup-scr7.sh
Using the setup-cadence7 script from F2015

Basic Cadence script finished
You are now set up to run Cadence.
Working directory is /homeF15/gen5xx
ncverilog: 08.20-s029: (c) Copyright 1995-2011 Cadence Design Systems, Inc.
file: PixelReductionCkt.v
        module worklib.PixelReductionCircuit:v
                errors: 0, warnings: 0
                Caching library 'worklib' ....... Done
        Elaborating the design hierarchy:
        Building instance overlay tables: .................... Done
        Generating native compiled code:
                worklib.PixelReductionCircuit:v <0x3973391f>
                        streams:   4, words:  7339
        Loading native compiled code:     .................... Done
        Building instance specific data structures.
        Design hierarchy summary:
                          Instances  Unique
                Modules:          1       1
                Registers:       44      44
                Scalar wires:     1       -
                Vectored wires:   2       -
                Always blocks:    1       1
                Initial blocks:   2       2
        Writing initial simulation snapshot: worklib.PixelReductionCircuit:v
```

4-2: Simulation verification for the Color Amplification Circuit

```
[gen5xx@CadenceProd2 ~]$ sim-ncg.sh ColorAmpCkt.v
SOURCELIST:  ColorAmpCkt.v
Running sim-ncg.sh - Verilog stand-alone simulator
Calling setup-scr7.sh
Using the setup-cadence7 script from F2015

Basic Cadence script finished
You are now set up to run Cadence.
Working directory is /homeF15/gen5xx
ncverilog: 08.20-s029: (c) Copyright 1995-2011 Cadence Design Systems, Inc.
file: ColorAmpCkt.v
        module worklib.ColorAmpCircuit:v
                errors: 0, warnings: 0
                Caching library 'worklib' ....... Done
        Elaborating the design hierarchy:
        Building instance overlay tables: .................... Done
        Generating native compiled code:
                worklib.ColorAmpCircuit:v <0x182d3f2e>
                        streams:   2, words:  4444
        Loading native compiled code:     .................... Done
        Building instance specific data structures.
        Design hierarchy summary:
                          Instances  Unique
                Modules:          1       1
                Registers:       11      11
                Scalar wires:     1       -
                Vectored wires:   5       -
                Always blocks:    1       1
                Initial blocks:   1       1
        Writing initial simulation snapshot: worklib.ColorAmpCircuit:v
```

Chapter 5: Conclusion

The goal of this project was to successfully simulate MIT's Eulerian Color Amplification algorithm using a Gaussian pyramid created in Verilog to calculate the algorithm in hardware. Computing the algorithm in hardware allows to achieve two primary goals: to greatly reduces computation time, and increase the versatility of the algorithm in general. This increased versatility could aid the development of many new technologies such as giving hospital wards the ability to monitor numerous patients at once, allow doctors to diagnose injuries quicker, enabling factory workers to view gas leaks in an effective manner, aiding in measurements of precise chemical reactions, or many other unforeseen applications.

Our hardware solution was composed of two main circuits. The first portion of the solution was the Pixel Reduction Circuit. This circuit is used to perform the Gaussian pyramid/weighted average calculation, and its goal was to find the change of individual colors between two frames after reducing them to a single pixel. The second portion of the circuit is the Color Amplification Circuit. This circuit is used to actually amplify the color of choice. This is done by taking the amplification factor of the desired color from the Pixel Reduction Circuit, and multiplying that color's values in the original frame by the amplification factor. This effectively amplifies the change between the two frames and produces the desired result.

As for what we accomplished, we successfully designed the two circuits that would reduce the pixel and amplify the color. We also successfully simulated the circuit in Cadence Virtuoso.

In our previous education at Cal Poly, we took two classes dedicated to development of VHDL systems implemented on an FPGA through the use of Xilinx hardware and its accompanying software. Over the course of those classes we became pretty familiar with the Xilinx tools but due to some inconsistencies between Xilinx, industry standards, and their own hardware, we were advised not to use Xilinx products despite being much more familiar with them. This led us to developing our designs in the Cadence Design Suite using Verilog.

While we were able to simulate the two separate blocks in Cadence Virtuoso and verify their correct construction, we were unable to synthesize the code due to the inability to create the necessary .tcl files, and difficulty navigating through Cadence and its software.

As discussed previously, the structure and workflow of the Cadence tools is aggravating and full circuit design requires moving files between multiple different programs in Cadence's design suite. Our inexperience with Cadence and Linux in general combined with lacking documentation on the Cadence Design Suite led us to produce multiple errors while trying to load or simulate files, each of which took very long to fix, or could not be remedied at all. This lack of documentation made it very hard to progress the project since it was very difficult to find solutions to many of the problems we encountered.

Another challenge was learning a completely new language. Again, in our previous education we were taught VHDL. This project required us to learn Verilog, which has many of its own nuances despite being somewhat of a combination of VHDL and C. This took hours of research and note taking, but we were eventually able to understand how the language worked, and construct our prototypes.

The greatest benefit of this project is that it allowed us to learn many new things. As previously discussed, we learned a new language that will definitely help us in our careers and beyond. Verilog is a quite useful language that is used in a variety of hardware solutions. Also, we learned a lot about the Gaussian pyramid blurring of pictures, computer vision, and image manipulation.

As for the future steps of the project, it would start with creating the test benches to actually simulate the circuits. Then, we would need to synthesize the two circuits. Since we were unable to modify the .tcl files, we were not able to synthesize. With proper guidance and documentation, the .tcl files could be modified to synthesize with our code and actually simulate within Virtuoso. After synthesizing, we would need to add registers for memory storage. The registers would serve the purpose of saving the intermediate calculations, as well as…… After those preliminary steps were done, we would wrap the entire circuit together and synthesize again. Once the synthesis of the entire wrapped circuit is completed, we would proceed to load it into an FPGA. The FPGA would have to have enough RAM for the entire circuit to load. We would then proceed to test the circuit on the FPGA.

References

[1] Wu, Hao-Yu, Michael Rubinstein, Eugene Shih, John Guttag, and William T. Freeman. "Video Magnification." *Video Magnification*. MIT Computer Science and Artificial Intelligence Laboratory, n.d. Web. 2 Mar. 2016.

-Source of color amp pics at beginning of document

[2] Raju, Vinayak, and Andrew Yokubaitis. *Pixel Frame Reduction*

[3] Cmglee. Visual representation of an image pyramid with 5 levels. Digital image.*Pyramid (image Processing)*. Wikipedia, 21 Aug. 2015. Web. 20 Oct. 2016.

Appendix A. Verilog Code

Pixel Reduction Circuit

```
module PixelReductionCircuit (pixel, clk, outPixel, ampFactor);
    reg [8:0] block [23:0];
    input clk;
    input [23:0] pixel;

    //Create register for each pixel
    reg [23:0] pixel1;
    reg [23:0] pixel2;
    reg [23:0] pixel3;
    reg [23:0] pixel4;
    reg [23:0] pixel5;
    reg [23:0] pixel6;
    reg [23:0] pixel7;
    reg [23:0] pixel8;
    reg [23:0] pixel9;

    //Create registers for each color of each pixel
    reg [7:0] red1;
    reg [7:0] red2;
    reg [7:0] red3;
    reg [7:0] red4;
    reg [7:0] red5;
    reg [7:0] red6;
    reg [7:0] red7;
    reg [7:0] red8;
    reg [7:0] red9;

    reg [7:0] blue1;
    reg [7:0] blue2;
    reg [7:0] blue3;
    reg [7:0] blue4;
    reg [7:0] blue5;
    reg [7:0] blue6;
    reg [7:0] blue7;
    reg [7:0] blue8;
    reg [7:0] blue9;

    reg [7:0] grn1;
    reg [7:0] grn2;
    reg [7:0] grn3;
    reg [7:0] grn4;
    reg [7:0] grn5;
    reg [7:0] grn6;
    reg [7:0] grn7;
    reg [7:0] grn8;
```

```
reg [7:0] grn9;


//Create registers to hold final weighted average color values
reg [7:0] red;
reg [7:0] blue;
reg [7:0] grn;
reg [23:0] hold;

integer temp = 0;
integer pixelCount = 0;

//Holds values of the previously solved kernel
reg [7:0] oldRed;
reg [7:0] oldBlue;
reg [7:0] oldGrn;

output reg [23:0] outPixel;
output reg [23:0] ampFactor;

always @ (posedge clk)
begin
    if (block[8] == 0)
            begin
            block[pixelCount] = pixel;
            pixelCount = pixelCount + 1;
            end
    else
            begin
            pixel1 = block[temp];
            temp <= temp + 1;
            pixel2 = block[temp];
            temp <= temp + 1;
            pixel3 = block[temp];
            temp <= temp + 1;
            pixel4 = block[temp];
            temp <= temp + 1;
            pixel5 = block[temp];
            temp <= temp + 1;
            pixel6 = block[temp];
            temp <= temp + 1;
            pixel7 = block[temp];
            temp <= temp + 1;
            pixel8 = block[temp];
            temp <= temp + 1;
            pixel9 = block[temp];
            temp <= temp + 1;

            red1 = pixel1[23:16];
            grn1 = pixel1[15:8];
            blue1 = pixel1[7:0];
            red2 = pixel2[23:16];
```

```
                grn2 = pixel2[15:8];
                blue2 = pixel2[7:0];
                red3 = pixel3[23:16];
                grn3 = pixel3[15:8];
                blue3 = pixel3[7:0];

                red4 = pixel4[23:16];
                grn4 = pixel4[15:8];
                blue4 = pixel4[7:0];
                red5 = pixel5[23:16];
                grn5 = pixel5[15:8];
                blue5 = pixel5[7:0];
                red6 = pixel6[23:16];
                grn6 = pixel6[15:8];
                blue6 = pixel6[7:0];

                red7 = pixel7[23:16];
                grn7 = pixel7[15:8];
                blue7 = pixel7[7:0];
                red8 = pixel8[23:16];
                grn8 = pixel8[15:8];
                blue8 = pixel8[7:0];
                red9 = pixel9[23:16];
                grn9 = pixel9[15:8];
                blue9 = pixel9[7:0];

                red = (red1 / 4) + (red2 / 2) + (red3 / 4) + (red4 / 2) + red5 + (red6 / 2) +
(red7 / 4) + (red8 / 2) + (red9 / 4);
               blue = (blue1 / 4) + (blue2 / 2) + (blue3 / 4) + (blue4 / 2) + blue5 + (blue6 /
2) + (blue7 / 4) + (blue8 / 2) + (blue9 / 4);
                grn = (grn1 / 4) + (grn2 / 2) + (grn3 / 4) + (grn4 / 2) + grn5 + (grn6 / 2) +
(grn7 / 4) + (grn8 / 2) + (grn9 / 4);


                //create register to send to output, then sends it
                hold [23:16] = red;
                hold [15:8]  = grn;
                hold [7:0]   = blue;

                outPixel = hold;

                //create register to send to output for amp factor, then sends it
                hold [23:16] = red - oldRed;
                hold [15:8]  = grn - oldGrn;
                hold [7:0]   = blue - oldBlue;

                //actualy sends the output
                ampFactor = hold;

                //saves the output for the next calc
                oldRed = red;
```

```
                oldGrn = grn;
                oldBlue = blue;

                temp = 0;
                pixelCount = 0;

                block[0] = 0;
                block[1] = 0;
                block[2] = 0;
                block[3] = 0;
                block[4] = 0;
                block[5] = 0;
                block[6] = 0;
                block[7] = 0;
                block[8] = 0;

                end

    end
endmodule
```

Color Amplification Circuit

```
module ColorAmpCircuit (inPixel, frameSel, colorSel, ampFactor, clk, ampFrame);
    input clk;
    input [2:0] frameSel;
    input [1:0] colorSel;
    input [23:0] ampFactor;
    input [23:0] inPixel;

    reg [7:0] red;
    reg [7:0] blue;
    reg [7:0] grn;
    reg [7:0] ampColor;
    reg [23:0] myPixel;
    reg [7:0] myColor;
    reg [255:0] regFrame;
    reg [255:0] ampCalcFrame;
    output reg [255:0] ampFrame;


    integer i, j = 0;

    always @ (posedge clk && frameSel)
    begin
        //checks to see if a whole frame has been stored, if not, then keep storing pixels
        if((frameSel == 1 && regFrame[0] == 0) || (frameSel == 2 && regFrame[3] == 0) ||
(frameSel == 3 && regFrame[15] == 0) || (frameSel == 4 && regFrame[63] == 0) || (frameSel == 5
&& regFrame[255] == 0))
```

```
	begin
		regFrame [j] = inPixel;
		j = j + 1;
	end


	//extract the correct colors and store the amplified frame in ampCalcFrame
	else
	begin
		if(colorSel == 1) //amplify red
		begin
			j = 0;
			for(i = 0; i < 256; i = i + 1)
			begin
				//get a pixel
				myPixel = regFrame[i];
				//get desired color from pixel
				red = myPixel [23:16];
				//get amp value for specific color
				ampColor = ampFactor [23:16];

				//get amplified color
				myColor = red * ampColor;
				//replace unamplified color with the amplified color
				myPixel [23:16] = myColor;
				//place the amplified pixel in a register to send to output
later
				ampCalcFrame [i] = myPixel;
			end

		end

		else if(colorSel == 2) //amplify green
		begin
			j = 0;
			for(i = 0; i < 256; i = i + 1)
			begin
				myPixel = regFrame[i];
				grn = myPixel [15:8];
				ampColor = ampFactor [15:8];

				myColor = grn * ampColor;
				myPixel [15:8] = myColor;
				ampCalcFrame [i] = myPixel;
			end

		end

		else if(colorSel == 3) //amplify green
		begin
			j = 0;
```

```
                        for(i = 0; i < 256; i = i + 1)
                        begin
                                myPixel = regFrame[i];
                                blue = myPixel [7:0];
                                ampColor = ampFactor [7:0];

                                myColor = grn * ampColor;
                                myPixel [7:0] = myColor;
                                ampCalcFrame [i] = myPixel;
                        end
                end
        end

    //output the amplified frame
    ampFrame = ampCalcFrame;
    regFrame = 0;

    end
Endmodule
```

Appendix B. Senior Project Analysis

Project Title: Computational Circuit for use in Real Time Color Amplification

Student(s): Vinayak Raju and Andrew Yokubaitis

Advisor: Tina Smilkstein

1. **Summary of Functional Requirements**
    a. Describe the overall capabilities or functions of your project or design.
        i. The Color Amplification circuit is an integrated circuit that will allow small changes of color to be amplified. Our hardware is a parallel IC that intakes 24-bit RGB pixel values for a 32x32 pixel frame, reduces the frame size through division and addition logic, then amplifies the desired color to change the pixel values of the original frame.
        ii. The system will be implemented in Cadence using Verilog and will consist of numerous RAM blocks, MUXes, multipliers, adders, and splitters.

2. **Primary Constraints**
    a. Describe significant challenges or difficulties associated with your project.
        i. One major difficulty in the project was grasping the concept of color amplification. This concept was rather difficult to understand at first, but once we had a grasp of the ideas constructing a block diagram was rather easy.
        ii. After understanding of the concept, the next challenge was trying to arrange the different functional blocks we needed to implement the device. This was done by numerous meetings with Tina Smilkstein to verify we understood the problem conceptually, and to verify the circuits we drew up would function correctly.

3. **Economic**
    a. What economic impacts result?

i. Human Capital: Engineers will be needed to construct, monitor, upgrade, and optimize the circuit. This device could result in medical companies creating new jobs for optimization of the circuit.

ii. Financial Capital: Hospitals will be able to save money by the monitoring of numerous patients with only one heart rate monitor.

iii. Natural Capital: The production and disposal of electronic devices create waste that takes does not decay, and leaves rare, expensive metals in landfills where they are extremely hard, if not impossible to recover on a large scale.

iv. Manufactured or Real Capital: This device will result in manufacturing of the IC and will require manpower to manufacture the circuit.

v. Costs: The sale price will be comparable to other heart rate monitors and will be determined by the price of all the components used to build the heart rate monitor, like cameras, power supplies, the IC itself and any other components needed. The profit would come from the markup on top of the production cost, which should be rather high since actual production costs should be low compared to the prices listed for other heart rate monitoring devices. Actual cost per device will depend on the total number of ICs made, since making only a few costs a lot more per device than mass producing them.

4. **If manufactured on a commercial basis:**
    a. This is largely dependent on whether the circuit is completed and optimized by the end of the year.
5. **Environmental**
    a. The environmental impact of this project comes from the fabrication of the integrated circuit. Factories are needed to manufacture this IC, which require energy and materials. Per all electronics, proper disposal of old devices is needed. If not appropriately disposed, harmful chemicals can leak into the environment, affecting the Earth’s atmosphere and animal life.
6. **Manufacturability**
    a. The manufacturing of the integrated circuit will be done by the normal fabrication process of all integrated circuits. The ingot will be cut into wafers. The wafers will be doped, then the wafers will be

sliced into small dies. Proper care must be taken when developing the ingots because they must be extracted at a proper temperature and must be sliced very carefully into wafers.

7. **Sustainability**
    a. A major issue with maintaining the completed device will be Electrostatic Discharge (ESD). ESD is the sudden flow of electricity between two electrically charged objects. ESD is a major complication in industry, as almost every object in the universe has a charge on it. Proper care must be taken when handling the IC because contact with our hand to the IC can cause the device to short out. Also, ESD approved bags and boxes will be needed to transport the device across locations.
    b. To maintain the product, access to Automated Testing Equipment (ATE) will be needed to test and optimize the device. An IC is unique in the fact that it can only be tested using specific testing equipment.
    c. Upgrades to our device include faster RAM modules within the device, faster adders to perform the matrix multiplication quicker, and layout changes to change signal path lengths.
    d. A major challenge associated with the upgrades to our device is Scan Chain. Scan chain is a test which sends a signal through the logic of a device, which includes the registers, and observing the output. By observing the output and points within the logic, a faulty register can be found. Faster RAM modules require quicker registers, which gives the scan chain less time to correctly output the necessary input.
8. **Ethical**
    a. Positive ethical implications that this product provides for is for doctors to provide good healthcare services, diagnose injuries quicker, and monitor patients non-invasively.
    b. Negative ethical implications include the use of monitoring one's heart rate when they are unaware. This could be used to determine when they are nervous, excited, lying etc.
9. **Health and Safety**
    a. There are not any specific health or safety concerns with the use of this device, unless the device stops working properly. In that case it will be a reliability issue which will require extensive testing to minimize before the device is released.

10. Social and Political

a. The project impacts many people. First, it impacts doctors by allowing them to monitor several patients with one heart rate monitor. It also could allow doctors to diagnose injuries quicker, by looking through the camera lense and observing parts of the body, and checking to see if they are getting proper blood flow. Second, it helps patients in hospitals by allowing them to be monitored by one heart rate monitor throughout the room without needing to be hooked up to a machine. Lastly, it affects anyone that utilizes fitness trackers and heart rate monitors. This product will optimize the sensors in heart rate monitors and allow for more accurate monitoring.
b. This project can affect stakeholders by encouraging them to invest more money into companies. Once they recognize the true potential of the device, they will want to invest more of their time and money into the device and into the company.

11. Development

a. This project will allow us to learn and utilize Verilog. Before this, we had no experience with Verilog. It is also a great exercise in hardware/software co-design, since we are designing hardware to work with very specific software.