HotEE:

A Hardware Accelerated Thermal Imager

6.375 Final Project
Spring 2013
Sumit Dutta, Namir Jawdat, and John Donnal

Introduction

Thermal imaging is a powerful tool and can be used in fields as diverse as detecting “hot spot” electrical faults, measuring the efficiency of building insulation, and even non-intrusively monitor patient vital signs. Unfortunately the high cost of IR Imagers has prevented their wide spread adoption. Our project aims to address the need for low cost thermal imagers by using new IR acquisition technology coupled with custom hardware to provide a full thermal imaging solution at under $100. Additionally, we incorporate a visual filter bank that can manipulate images before applying the thermal overlay. To demonstrate the power of this filter architecture we have developed a proof of concept EVM process that works on still images. EVM filtering amplifies small motions making it possible to visually detect subtle movement such like the rise and fall of the chest during breathing. Coupling temperature data with motion amplification provides a powerful new diagnostic tool making it easy to focus on the items of interest in an image.

Pipeline Overview

The architecture consists of two parallel video paths and an alpha masking module. The steps in each path are flexible and arbitrary filters can be applied to both. The block diagram is shown below:

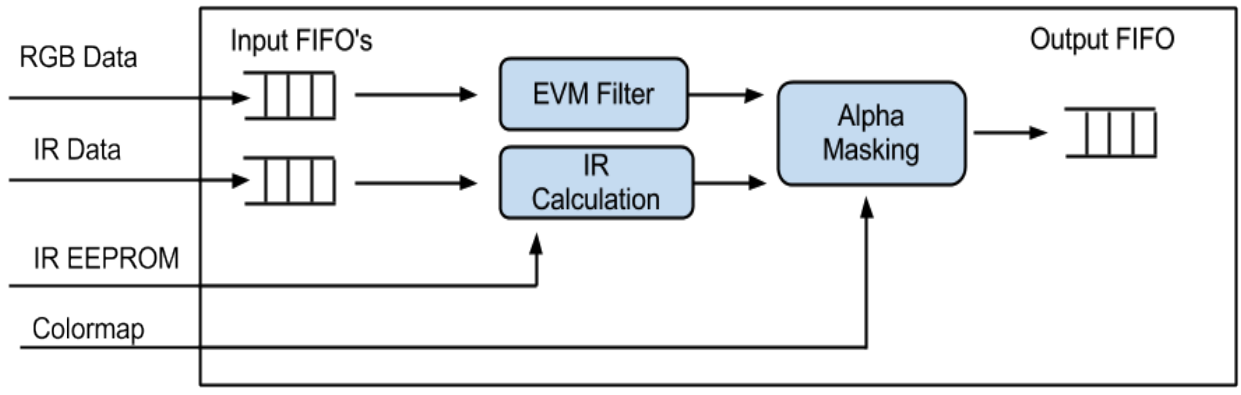


Figure 1: System block diagram

Supporting Hardware/Software

The FPGA accesses the video feeds via SceMi. Our custom hardware communicates to the host PC via two USB connections, one for IR and one for regular video. We use the OpenCV library to retrieve the raw pixel data from the webcam feed and the LUFA USB library to access the IR feed as a serial device. Both of these feeds can be passed to simulator code which verifies our algorithms, or to SCEMI which then sends it to the FPGA. The frames are reconstructed into video by OpenCV and displayed to the user. Using an FPGA this can all be done in real time. This setup is shown below:

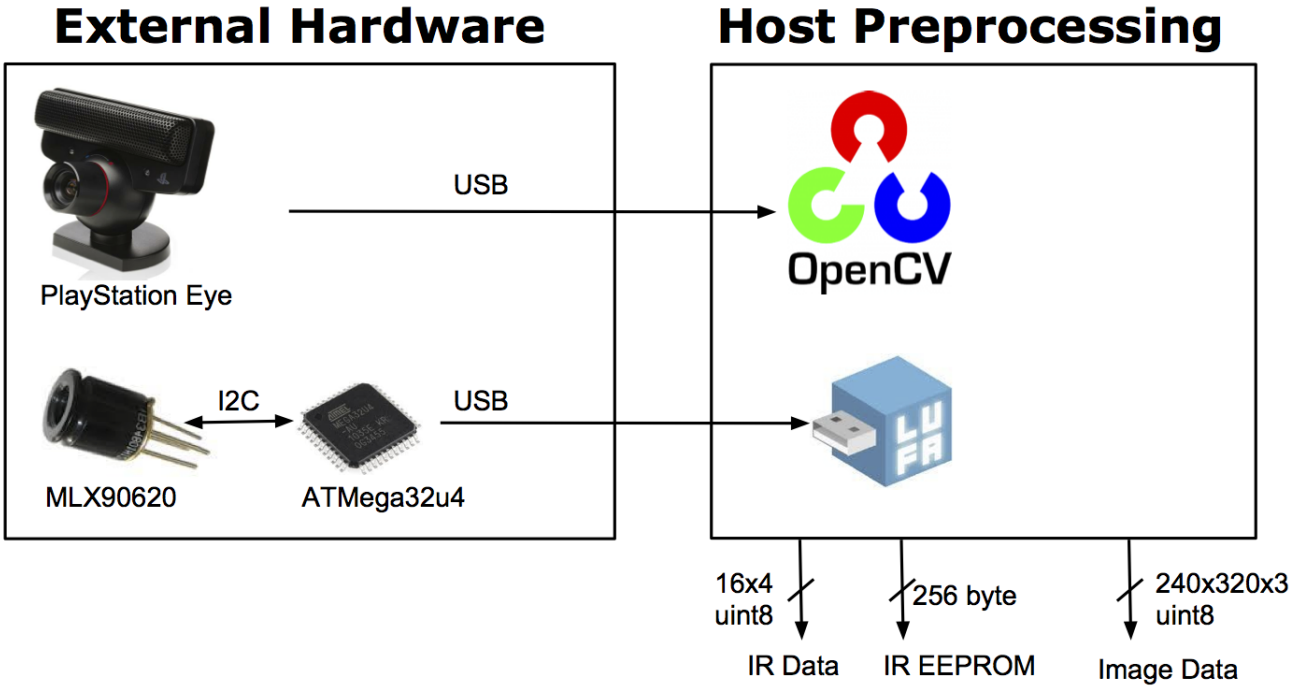


Figure 2: External hardware and libraries

FPGA Processing Details

Eulerian Video Magnification

Professor Freeman's group in MIT CSAIL developed a method called Eulerian Video Magnification (EVM) to amplify subtle changes in standard videos, revealing variations that generally cannot be perceived by

the naked eye. Today, the video analysis is performed on a computer, but we believe that the analysis can be performed faster and more efficiently on hardware specifically designed for the analysis. The MIT CSAIL team released the Matlab code used to process example videos, providing us a clear framework to build our hardware upon. Analysis is done in the spatial domain and in the time domain. The paper by the MIT CSAIL team [1] and corresponding Matlab code show several methods for the spatial processing and temporal processing needed for EVM.

The spatial and temporal processing must be done within the limits of our FPGA, requiring us to evaluate how suitable each method is for implementation on our FPGA. Spatial processing is done either with Laplacian pyramidal decomposition or Gaussian blurring. Laplacian pyramidal decomposition involves working with the original image, its once-downsampled image, its twice-downsampled image, and so on, whereas Gaussian blurring is a one-time process run on each image. Both of those spatial processes could have been implemented on the FPGA. However, the temporal processing required for each type of spatial processing is different and that helped us choose what to implement on the FPGA. Among the methods for temporal processing are bandpass filtering or IIR filtering. Bandpass filtering uses FFT and IFFT blocks that we already understand and it can be accomplished with a linear arithmetic pipeline. IIR filtering is similar except that it has a feedback loop, which makes the process more complex for a hardware implementation. This means bandpass filtering, either in the frequency domain or with FIR filters, is desirable for our FPGA implementation because it is simpler and we have building blocks for it.

The processing methods also must run on fixed-point numbers rather than floating-point numbers in order for our implementation to fit on an FPGA. The reference implementation in Matlab takes at least several minutes to process each video, so we first worked with a cropped 88x88 video and later worked with a single frame of a larger image that can still be processed by EVM but in less time. Two EVM methods in the reference implementation were converted from working with floating-point numbers to working with fixed-point numbers with a user-specified number of bits. These methods are: (1) Laplacian pyramidal decomposition for spatial processing and IIR filtering for temporal processing, and (2) Gaussian blurring for spatial processing and bandpass filtering for temporal processing. Initially, an essential 2D convolution step could not be done in fixed-point arithmetic, but later that 2D convolution function was explicitly written and successfully done in fixed-point arithmetic. Further experimentation with fixed-point bit sizes and algorithm coefficients in our modified reference implementation aided us in designing the respective components of our hardware design.

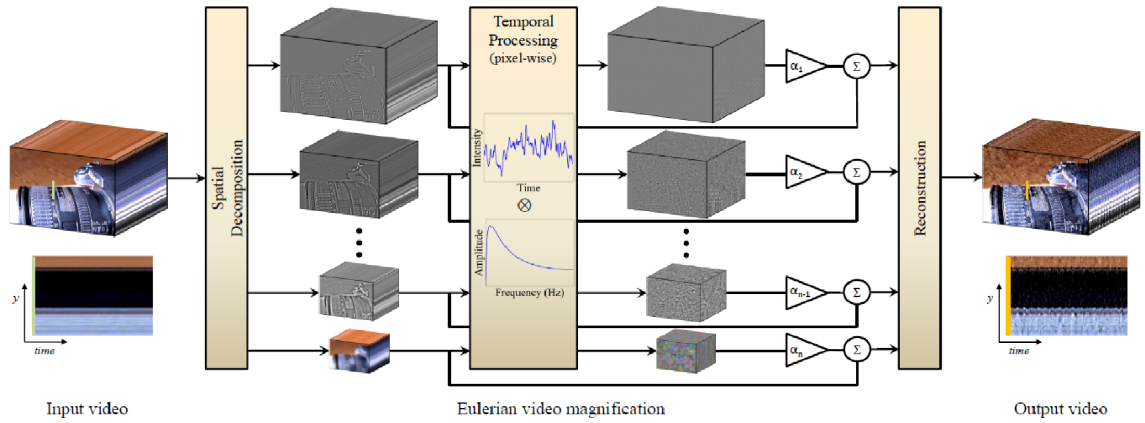


Figure 3: Block diagram of Eulerian Video Magnification from [1].

To operate on real-time video input, our FPGA implementation should be designed to process video at a resolution up to 320x240 at around 30 Hz. We realized that this goal is too ambitious and would have reduced speed due to FPGA limitations discussed later. Nevertheless, we decided on how we could efficiently implement EVM.

At first, we ruled out Laplacian pyramidal decomposition in favor of Gaussian blurring and downsampling for spatial processing. Then we looked closer at the two methods and noticed that the underlying process of blurring with 2D convolution is common to both methods and that the Laplacian pyramids method is more scalable. Rather than implementing Laplacian pyramids or Gaussian blurring as done in the EVM reference code, we designed a simple Laplacian pyramids flow that is essentially takes a video frame, does convolution with downsampling, does upsampling, and performs necessary sums for reconstruction. This process requires a large number of multiplications, so we were careful in the hardware implementation we picked for the blurring. There are 10 convolutions performed for the Gaussian blurring of an 88x88 grayscale image with a filter of size 5x1. The actual hardware implementation chosen for convolution is a linear pipeline using FIR filters. The process we chose to implement can be scaled up or down as needed, by choosing a different square kernel size. More information on the 2D convolution is discussed later in this paper.

The Laplacian pyramids flow that we designed comes essentially from a section about Laplacian pyramidal decomposition in an image processing textbook [2]. This reference shows that the spatial decomposition can be done with a one-level pyramid and the reconstruction can be done with one-level reconstruction. Additional levels of decomposition can be added if desired using multiple one-level pyramid and one-level reconstruction blocks. Block diagrams of the Laplacian pyramids flow are shown later.

When we initially tried Gaussian blurring in Matlab, we got the results on a black and white image that are shown in Figure 4. In the blurred version the stray pixel groups are removed and only the main features remain. We continued working on a grayscale image. We wrote and understood Matlab code of our own to do the EVM process with our designed Laplacian pyramids flow. It works well. We implemented the Laplacian pyramidal decomposition and reconstruction first. Then we planned to work

on the temporal process. According to our calculations, we have enough BRAM to do the temporal process. Currently we use only 2 BRAM modules, one for the input image frame and one for the output image frame. Since we have room left, we considered using additional BRAM modules to store a history of sums in the FIR Series. The Bluespec code was completed for all of the blocks in the Laplacian pyramids process. Then we started testing it and packaging them into the decomposition module and reconstruction module. Then we planned to do the bandpass filter of the temporal process.

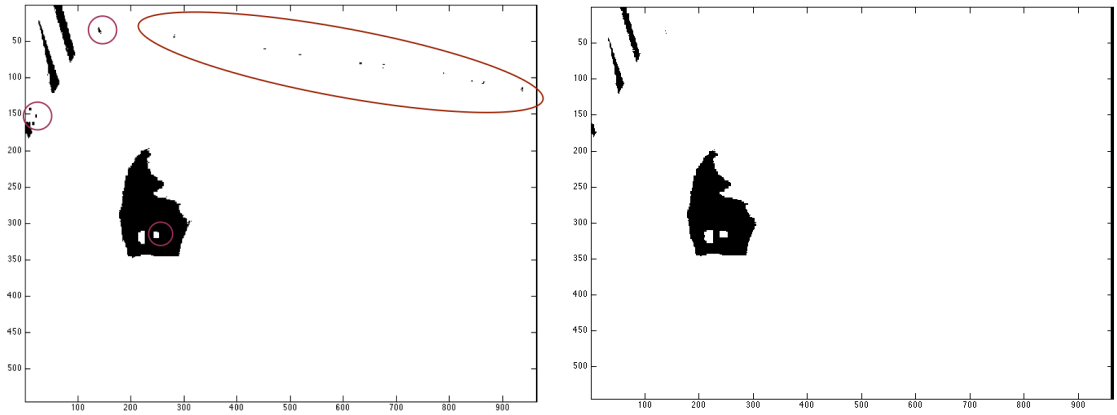


Figure 4: Blurring of a test image with a custom-made reference code

The full subsystem block diagram for the “EVM Filter” block is shown in Figure 5. The block framework has been created in Bluespec. As seen in the figure, a single image frame can be stored in memory after it comes from the video camera, and the resultant single image frame is stored in memory after the image data is processed. These two memories are called the Input Image BRAM and Output Image BRAM, respectively. The image data is stored as 8-bit unsigned integers, taking data from one of the red, blue, or green channels in their raw values between 0 and 255. The spatial processing of individual pixels involve multiplications with numbers between 0 and 1 as described later, and thus image data is converted to fixed point for processing. After the core spatial processing, data is converted back into unsigned integers to be stored in the Output Image BRAM.

The Bluespec implementation of the full subsystem in Figure 5 is completed, with the correct bits flowing from end to end. This is subsystem is called the EVM Filter block. This includes the block that accesses the BRAMs, the Chunker, the Splitter, and the skeleton of the block containing the FIR Series. The EVM Filter module was verified to pass data from one end to the other. Additionally, the module includes the 1-Level Pyramid and 1-Level Reconstruction blocks. These two blocks pass data correctly as desired. Initially, the FIR Series block was a simple pass-through skeleton block because its actual functionality was being written at the same time. Later, we got the completed FIR Series block to fit into the same spot and the full EVM Filter block still worked with the correct output.

The 1-Level Pyramid and 1-Level Reconstruction blocks were also tested with a test bench for each block, and the sub-blocks of those two blocks were also tested with their own test bench. The Upsampler, Downsampler, Adder, and Subtractor blocks work correctly. The 1-Level Pyramid and 1-Level

Reconstruction blocks appear to work correctly, and they were later tried in the larger EVM Filter subsystem block for a complete verification in simulation.

We have begun testing the blocks in the FIR Series and have a working two-dimensional convolution, though there are some odd results which we are looked into to fix any potentially remaining bugs.

In addition to simulation we have also successfully synthesized the EVM Filter submodule with its correct sub-blocks with the exception of the FIR Series which is only a pass-through skeleton block for synthesis. Currently, the critical path in the synthesis of the "EVM Filter" block is in the Input Image BRAM, but this will change once we integrate the working FIR Series block into the system. That is, although our critical path was 8.16 ns at first, this number increased when the arithmetic-heavy FIR Series was fully included in the synthesis. Nevertheless, our processing time is expected to be much shorter than the time needed for video frame input/output, so our subsystem was fine as long as it is synthesized successfully again when our working FIR Series was completely integrated with the subsystem.

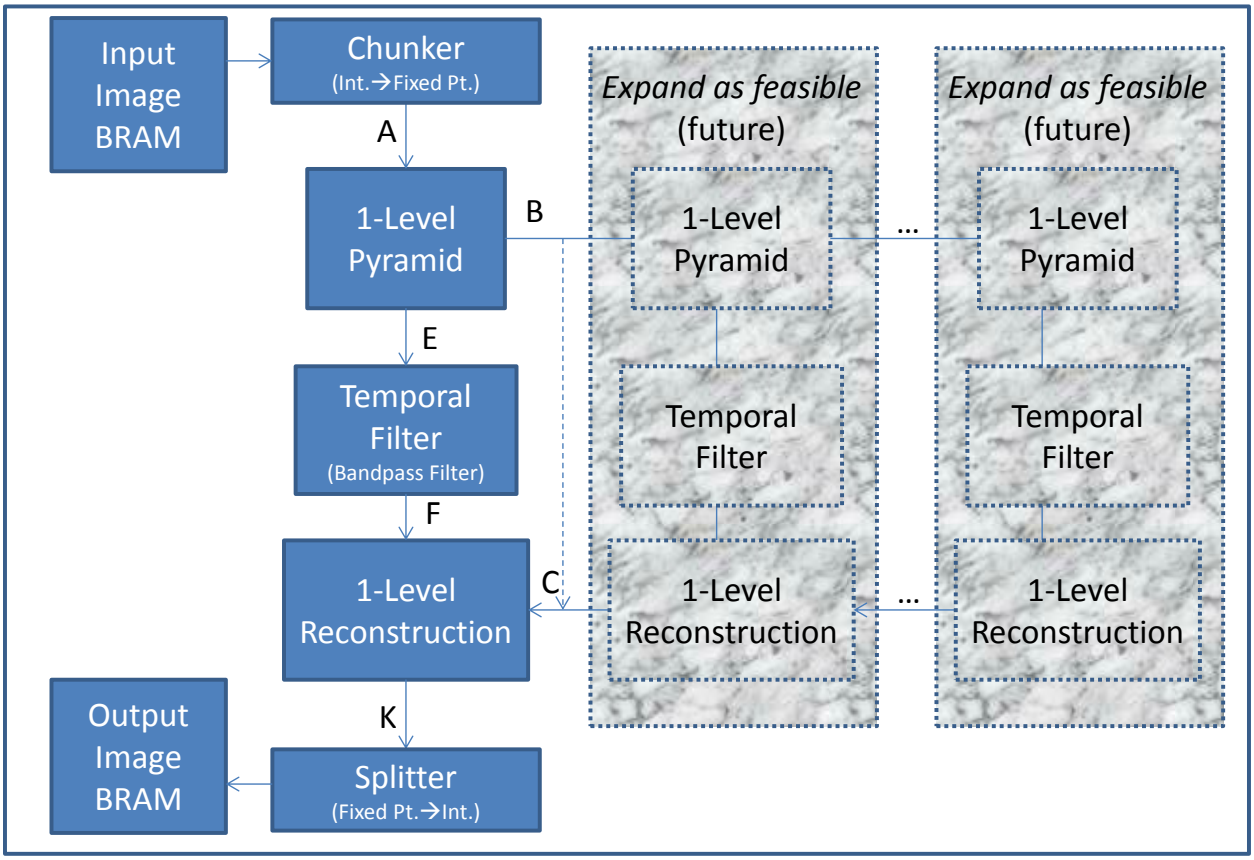


Figure 5: Subsystem block diagram of the "EVM Filter"

The 1-Level Pyramid and 1-Level Reconstruction blocks are shown in Figures 6 and 7, respectively. The 1-Level Pyramid does the effective blurring of an input frame, doing convolution using FIR filters in the FIR Series. After decomposing the frame into a pyramid layer, we reconstruct the output frame with the 1-

Level Reconstruction block. In order to start simple, we first pursued the simplest possible spatial processing without temporal processing, leaving room to add more if time permits. The pipeline above has been built in Matlab.

First-in-first-out (FIFO) queues are used at the input and output of many blocks. FIFO queues were in some intermediate steps before, but we realized that FIFO queues are necessary mainly at the block module boundaries. We also use FIFO queues for shift registers. Note that certain paths in the pipeline shown in Figures 6 and 7 make use of shift registers, such that the appropriate data is available at the right cycle, such as for certain summation operations. The Shift Register block is simply a FIFO queue that has enough elements to support keeping some data for the desired delay in cycles, and it does not count cycles because that would be contrary to Bluespec principles.

The FIR Series idea comes from a previous 6.375 final project [3], where a two-dimensional convolution of an image frame with a one-dimensional kernel is performed using FIR Filter blocks, shown in Figure 9, in a larger system, shown in Figure 10. Figure 10 is the FIR Series block. The FIR Series block implements convolution using the method shown in Figure 8. Note that the FIR filter coefficients are the kernel to convolve the frame with. The kernel used in the EVM study [1] is called “binom5” and we use the same numbers of the kernel in our design. However, the EVM with the one-dimensional “binom5” kernel requires multiple image reads, whereas we want the frame data to flow through only once. Because of this, we found that it is better to use a different two-dimensional kernel instead that we can use with the convolution shown in Figure 10 to process a frame bit-by-bit only once.

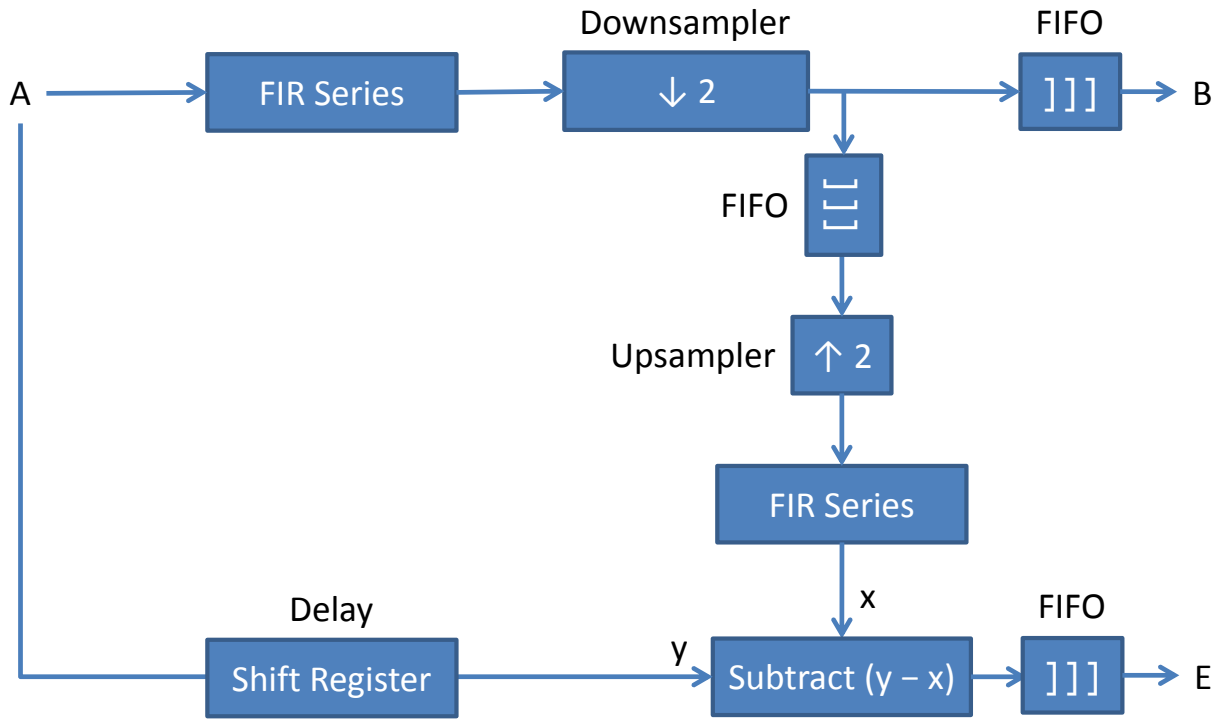


Figure 6: 1-Level Pyramid block diagram

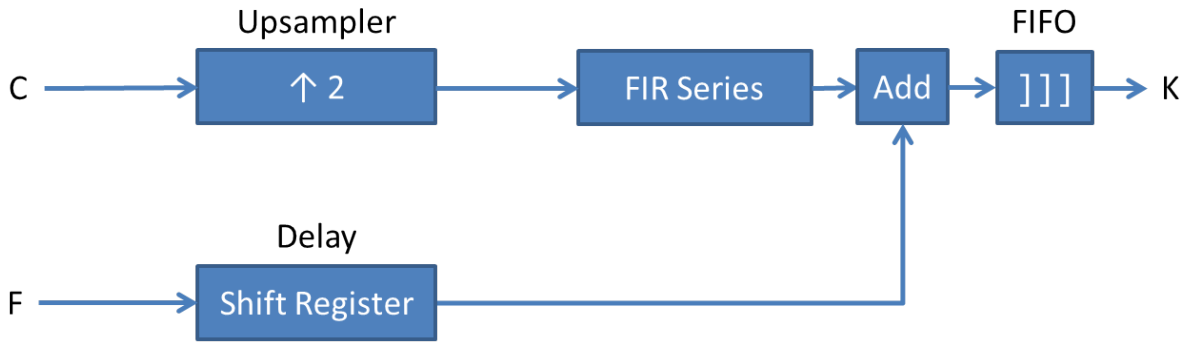
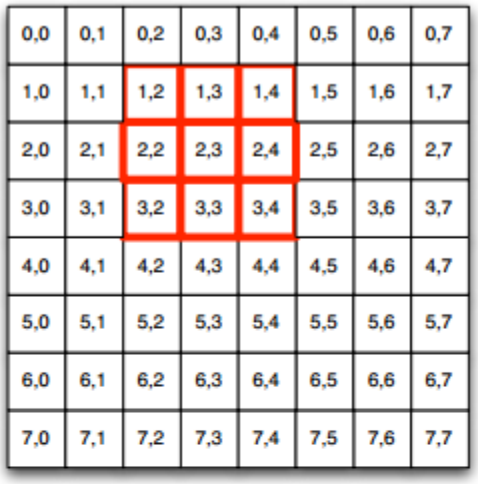


Figure 7: 1-Level Reconstruction block diagram



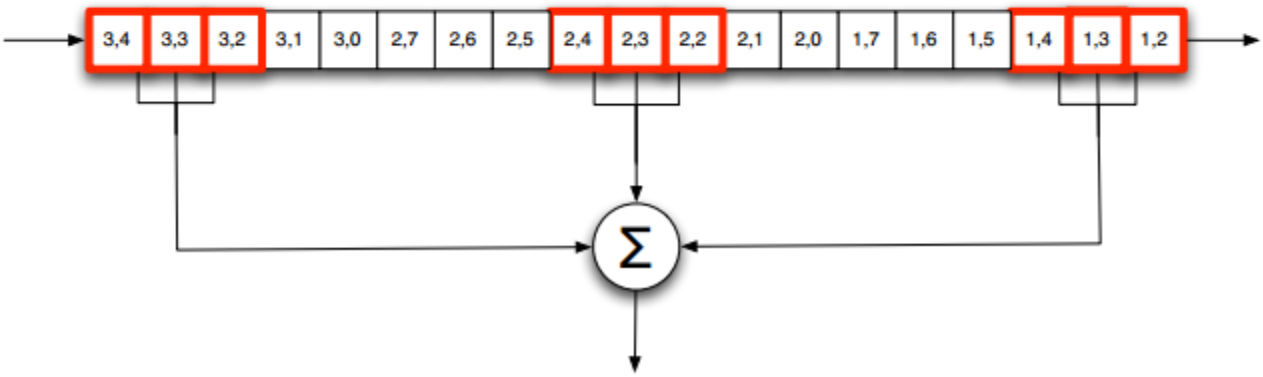


Figure 8: Convolution algorithm from [3] used in the FIR Series

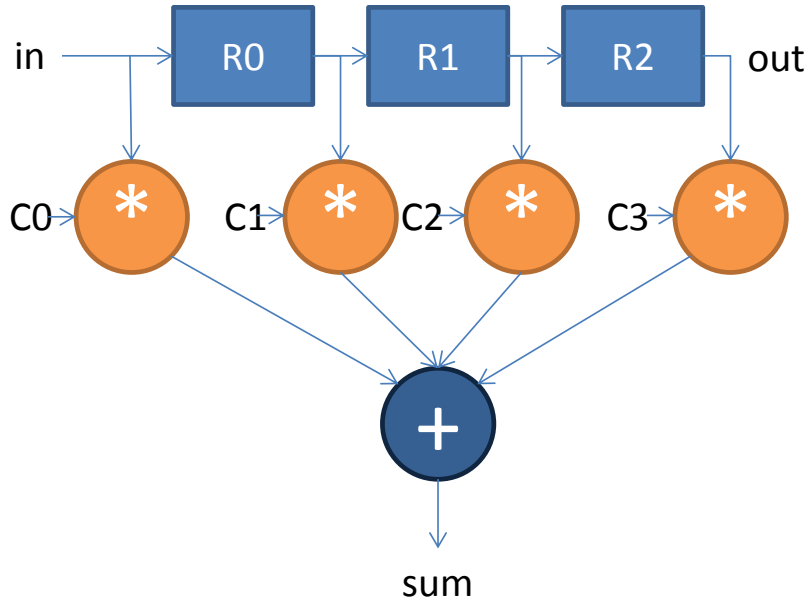


Figure 9: FIR Filter block diagram

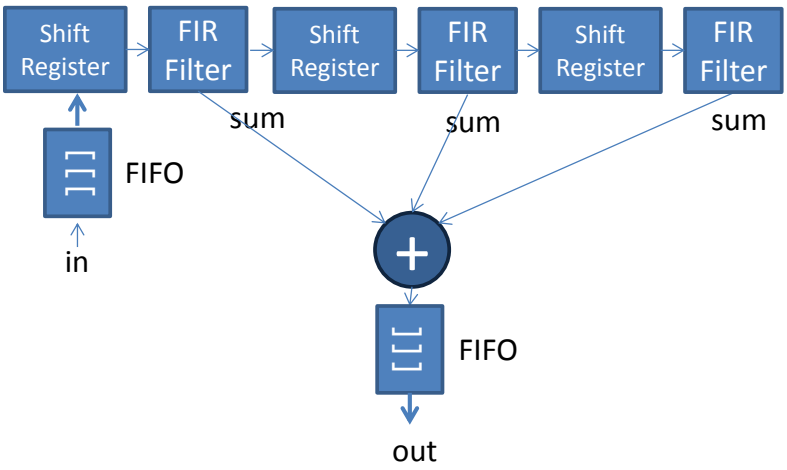


Figure 10: FIR Series block diagram, where each FIR filter corresponds to one set of three adjacent red boxes in Figure 8

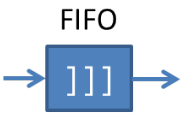


Figure 11: The Shift Register block is simply a FIFO that has enough elements to accommodate the desired delay

We decided to add temporal processing if time permitted, which it did. The temporal process would enable us to get results from video and not just from a single image frame. This would help us potentially identify features that can help us see if the warm object is human. The temporal process block diagram was implemented as the figure below. There are two lowpass filters that together they

work as a bandpass filter. However, we still had a bandwidth limitation with SceMi preventing us from sending in multiple video frames to the FPGA, so our final FPGA design only worked with a single frame.

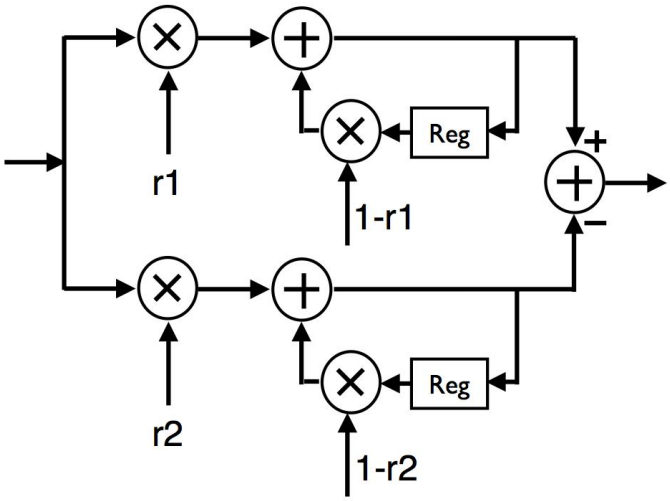


Figure 12: Temporal filter block diagram

To test the FIR Series Bluespec module, sample image data was piped into the block in a test bench and the piped-out results were plotted as an image. The expected result is that of a convolution of the input image with the kernel, essentially a blurred image. Initial results from the FIR Series block are shown in Figure 13 where the input image is the US flag. The original color image was converted to grayscale before it was put into the FIR Series block. The result is a blurred flag in grayscale, where the stars appear duller and the edges of the stripes are no longer sharp boundaries. Likewise, another example is shown in Figure 14 where an input image in grayscale is blurred by the FIR Series block. The result in this case does not appear blurred exactly as one would expect because there are some black regions. We are currently debugging any potential flaws in the FIR Series design and are making sure that our image displaying with Matlab is working correctly.

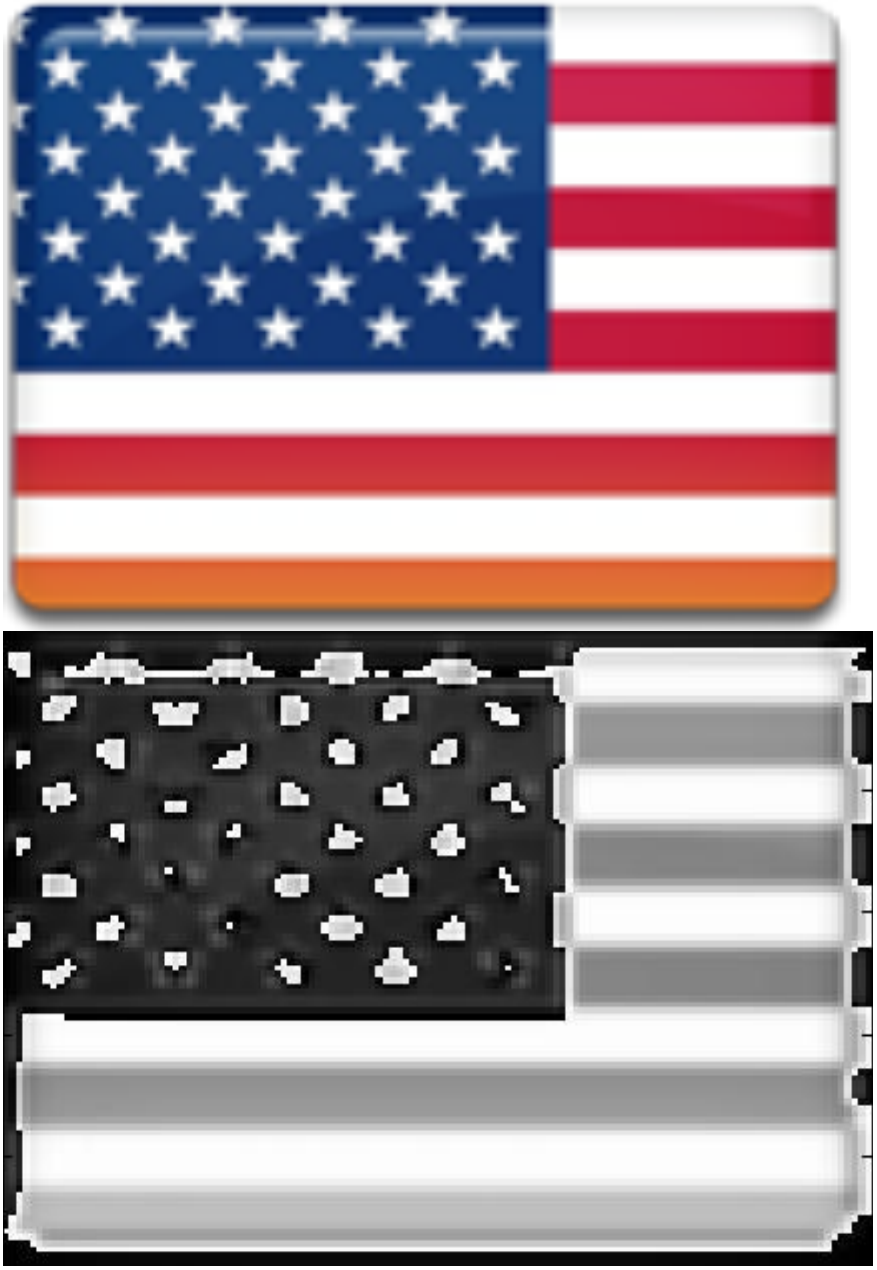


Figure 13: Example image that is blurred by the FIR Series, which effectively does a convolution between a kernel and the grayscale image; the original image is on top, and the result after it is converted to grayscale and blurred by the FIR Series module in Bluespec is on the bottom

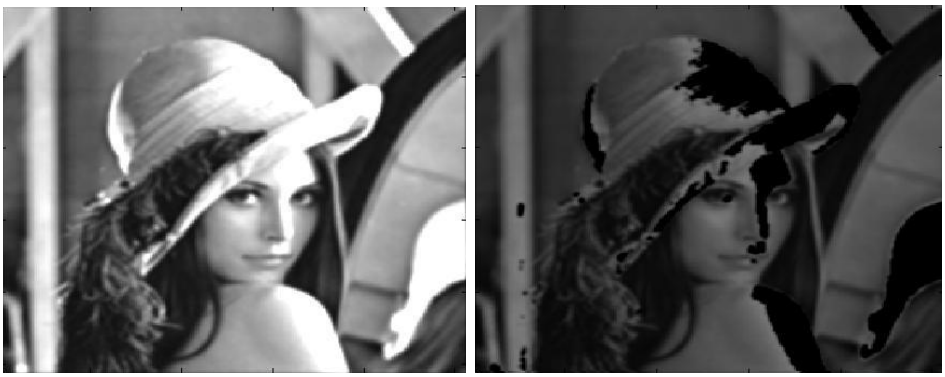


Figure 14: Another example of a grayscale image (left) blurred with the FIR Series block (right)

Figure 15: Another example showing a grayscale image after going through various stages of the one-level pyramid Bluespec module: original (left), after the downsampler but skipping the FIR Series (middle), and after the downsampler and its following upsampler but skipping the FIR Series (right)

Figure 15 shows the results from intermediate steps in the one-level pyramid. The downsampled image in the middle is indeed half the length and half the width of the original picture on the left. The upsampled version of the downsampled picture is indeed the same size as the original picture and it has a grid of black going across the rows and columns as expected.

The entire one-level pyramid to one-level reconstruction flow was verified to work in simulation and on the Xilinx XUPV5 FPGA. However, since the output result of this flow is the same image as the input, we decided to simply take the result from one of the intermediate steps in Figure 15 that look interesting. The result is shown in Figure 16, where we take the upsampled downsampled FIR filtered original image and display that to the user. Hence the exact functionality of the EVM Filter block in the whole system is to FIR filter, downsample, and upsample the input image. The resulting image is sent to the output, and all remaining reconstruction is not done.

Nevertheless, the infrastructure exists to use the working pyramids and reconstruction flow to process multiple pyramid levels on video, but we avoided integrating that because we knew we could not get any more than one frame in at a time due to slow SceMi bandwidth speed. The temporal filter was fully constructed and tested, but we were still SceMi bandwidth-limited to run the whole temporal process.

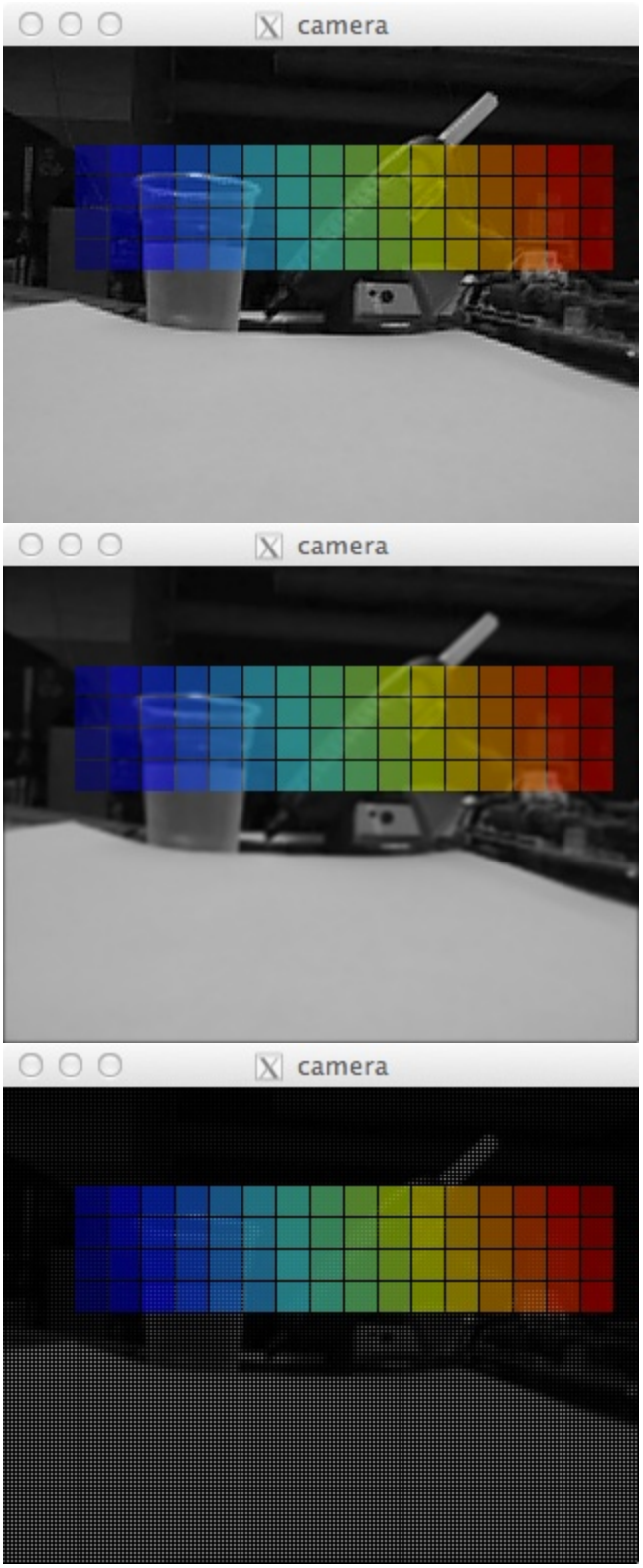


Figure 16: Complete system output: original image bypassing the EVM filter (top), original image after going through only the FIR series (middle), and complete system final result, where the input goes through the FIR series, downsampler, and then upsampler (bottom)

Complete synthesis, of the system depicted in Figure 16, appears to work for the Xilinx ML605 (Virtex 6) because it has enough resources to fit the design.

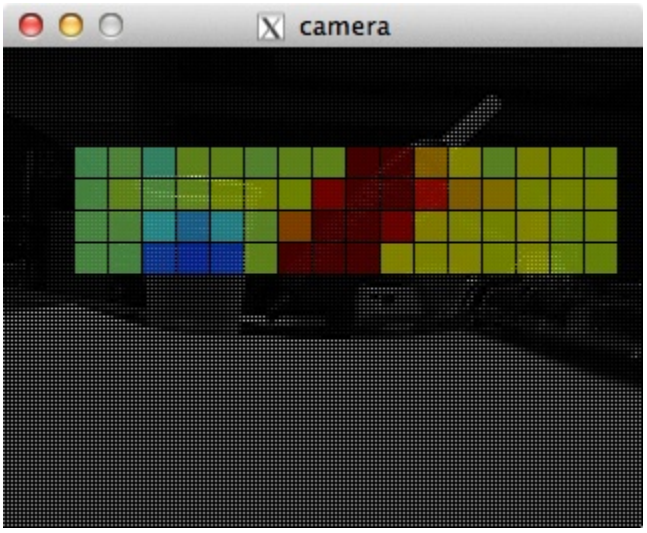


Figure 17: FPGA output

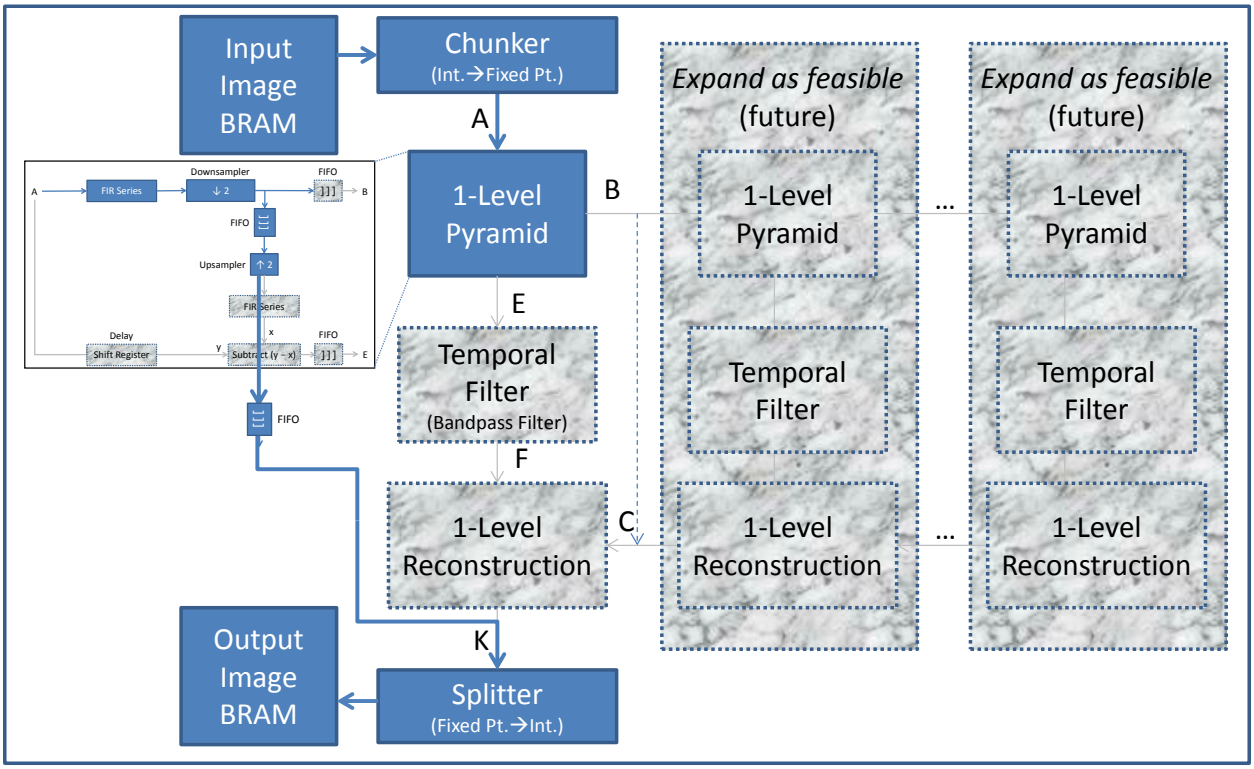


Figure 18: Block diagram of the system as synthesized for and run on the ML605 FPGA, where gray-filled blocks are commented out in the Bluespec code while preserving remaining functionality

Thermal Video

Theory

Thermal video records infrared energy and produces heat maps of the imaged objects. Traditional thermal cameras are prohibitively expensive but a new compact device manufactured by Melexis, the MLX90620, provides a resolution of 16x4 in a compact package that can be connected to an FPGA or traditional microcontroller with I^2C. This device is a combination EEPROM and IR Array in a single TO-39 package.

The MLX90620 Thermal Image Camera

Some complex arithmetic must be performed on the data to calculate the temperature at each pixel. First the temperature of the die must be calculated:

$$T_A = \frac{-K_{T1} + \sqrt{K_{T1}^{\ 2} - 4K_{T2}\left[V_{TH}(25) - PTAT\_data\right]}}{2K_{T2}} + 25,[°C]$$

The K constants are stored in EEPROM. Then the temperature at each pixel must be calculated:

$$T_{O(i,j)} = \sqrt[4]{\frac{V_{IR(i,j)\_COMPENSATED}}{\alpha_{(i,j)}} + (T_a + 273.15)^4} - 273.15,[°C]$$

The alpha can be computed directly using values from the EEPROM:

$$\alpha_{(i,j)} = \frac{(256.\alpha_{0\_H} + \alpha_{0\_L})}{2^{\alpha_{0\_SCALERE}}} + \frac{\Delta\alpha_{(i,j)}}{2^{\Delta\alpha_{SCALERE}}}$$

V_{IR} is more complex to calculate. In general there are three compensation steps. First the non-linearity of the sensor must be compensated so each pixel has a slope and offset coefficient:

$$V_{IR(i,j)\_OFF\_COMP} = V_{IR(i,j)} - \left(A_{i(i,j)} + \frac{B_{i(i,j)}}{2^{B_{i\_scale}}} \left(T_a - T_{a\_0} \right) \right)$$

Secondly the thermal gradient across the die must be compensated by reading the value of the compensation pixel and applying the following formula (where V_{IR} of the compensation pixel is determined from the formula above:

$$V_{IR(i,j)\_TGC\_COMP} = V_{IR(i,j)\_OFF\_COMP} - \frac{TGC}{32} . V_{IR\_CP\_OFF\_COMP}$$

Finally, if the emissivity of the object under inspection is known, this can be compensated using:

$$V_{IR(i,j)\_COMPENSATED} = \frac{V_{IR(i,j)\_TGC\_COMP}}{\varepsilon}$$

This fully compensated V_{IR} value is used in the T_O calculation. This calculation sequence must be carried out for each pixel. The full frame is then transmitted to the alpha masking module which combines it with the video.

Implementation

All of the IR calculation code is in the TempCalc module. This module has an input vector with 65 16-bit entries representing the entire RAM of the MLX90620. The first entry is the PTAT sensor and the next 64 are the voltage readings from each pixel. The module has an output vector with 64 8-bit entries that are temperatures in Fahrenheit for each pixel in the camera. The formulas described above have been simplified to minimize the number of operations and all constants have absorbed appropriate signs so there is no subtraction. The calculation for ambient temperature is simplified to:

$$T_A = (A + sqrt(B+C*ptat)/D$$

Where A, B, C, and D are defined in TempConstants.bsv which is generated by a python script using the camera's EEPROM.

The calculations for each pixel require ambient temperature as a parameter. The first step is to compensate the voltage reading at the pixel:

$$V_{IR\_comp} = V_{IR} + (A+B*(Ta))$$

Again, A, and B are defined in TempConstants.bsv. This compensation equation balances these pixels which each have a separate offset and slope. Therefore this equation must be computed 64 times. There are equations to compensate for the thermal gradient across the sensor die but the scaling coefficient for this correction factor is 0 in the EEPROM so this has no effect on the calculated temperature. Future versions of this sensor might have a nonzero value for this coefficient but currently Melexis does provide

it for the MLX90620. Finally we assume an emissivity of unity since we do not know beforehand the material we will be viewing. Therefore the linearized V_{IR_comp} value is used directly to compute the temperature read by the pixel using this equation:

$$T_O = \text{sqrt}(\text{sqrt}(V_{IR\_comp} /A + (T_A +B)^4)) + C$$

As before the constants are defined statically. This gives the temperature in Celsius. The final step is to convert to Fahrenheit. While not strictly necessary, the Fahrenheit scale is desirable because of familiarity and because it has larger variation in the temperature range of interest. This conversion is straightforward:

$$T_{O\_Fahrenheit} = T_O *A+B$$

Here, A=1.8 and B=32. These are defined in TempConstants.bsv along with all the other constants.

All computation is serialized to preserve resources for the image processing module. All of these calculations are implemented in Bluespec and work on the FPGA.

Floating Point

As suggested above the operations to calculate temperature are best done with floating point. The presence of powers of 4 and square roots make fixed point implementations very difficult and error prone. Bluespec does not support floating point natively so we are using a set of Verilog libraries provided by Xilinx. The AWB/Leap computing group in CSAIL has a set of wrappers to expose the floating point modules to Bluespec. While not exactly straightforward I have successfully integrated their work into our project as a set of modules. One drawback of using raw Verilog is that designs can no longer be run in Bluesim. This makes working with floating point much more difficult because everything must be synthesized and run directly on the FPGA to check if it works. In order to preserve our ability to use Bluesim for the rest of the project I have two versions of the TempCalc module. The first uses floating point modules to calculate the temperature, and the second fills the temperature vector with a set of dummy values and does no other math. For simulations we use the dummy module and for synthesis we substitute in the real module.

The Xilinx libraries are exposed as modules with a server/client interface. Modules receive requests with either one or two operands, and produce responses containing the result as well as the floating point status flags. We are using the following floating point modules:

1. Addition
2. Multiplication
3. Division
4. Squareroot
5. Integer to Double precision conversion

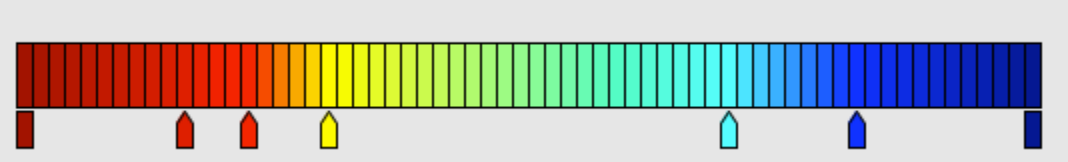
6 Double precision to Integer conversion

Constants are stored as Reals in TempConstants.bsv and converted to floats statically using the helper function $realtobits().

One additional complication with the Xilinx libraries is that the Verilog files are just for reference and cannot be synthesized. The actual functional code is in a set of *.ngc files which is done to protect the Xilinx IP. Unfortunately the standard build utility for Bluespec does not allow the inclusion of additional *.ngc files (to our knowledge) so a custom build script had to be designed. Our build script adds the necessary flags to the Xilinx tool chain to incorporate the libraries. This is wrapped up nicely in custom_build.sh that is included in our repository. This can be called just like the normal build command.

Alpha Masking

The result from both pipelines must be combined to create a single image. The traditional video pipeline has 3 channels for R, G, and B but the thermal pipeline only has the temperature values. To convert the temperatures to colors a color map lookup table is used. We created a custom color map based off the common Jet profile using Matlab's color map editor. This provides an intuitive interface to assign colors to values. We place the reds around human body temperature and the blues around the freezing temperature of water since this should cover the majority of the images we expect to encounter. The sensor provides readings from -50 to 300 C though. Shown below is our color map:

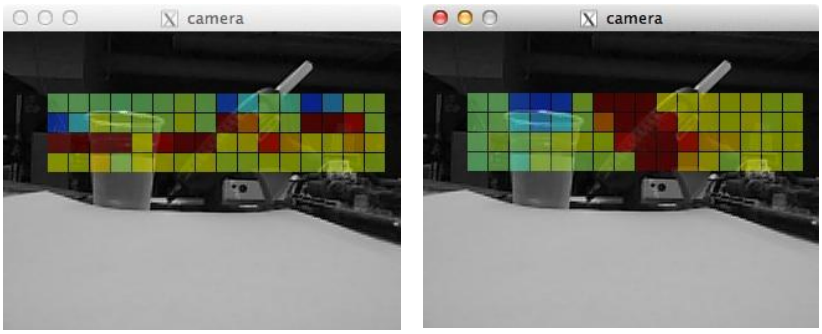


The map has 70 entries and the temperatures in Fahrenheit are indexed into this table. Any temperature outside [30,100) is fixed the respective max or min. The next step is to map each thermal pixel to the region of pixels it matches on the visual image. This is a set of static offsets and scale factors that are specified at compile time since the cameras are physically attached so the aspect does not change with time.

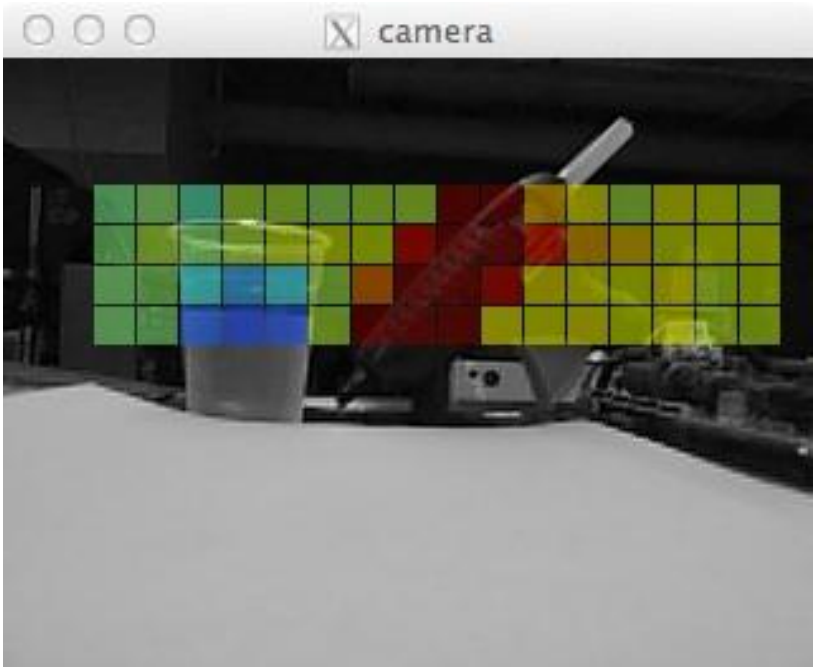
Finally the thermal image (now three-channel RGB) must be masked with the visual image. We use the traditional alpha compositing technique which is simplified from its general form with the assumption that the background image is opaque (3):

$$\begin{cases} out_A = 1 \\ out_{RGB} = src_{RGB} src_A + dst_{RGB}(1 - src_A) \end{cases}$$

We send the final combined image back to the user via a SCEMI port. This is fully implemented in Bluespec and works on the FPGA.



(Working to correctly align IR pixels with the image is a hard task)



Success!

References

1. H. Wu, M. Rubenstein, E. Shih, J. Guttag, F. Durand, and W. Freeman, “Eulerian Video Magnification for Revealing Subtle Changes in the World,” ACM SIGGRAPH Conference 2012, New York, NY.
2. R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, Third Edition, Addison-Wesley, 2008.
3. J. Porter, M. Thomson, and A. Wahab, “Lucas-Kanade Optical Flow Accelerator,” 6.375 Final Project, Spring 2011.
4. MLX90620 datasheet.
5. Alpha Compositing. Wikipedia. http://en.wikipedia.org/wiki/Alpha_compositing