A Smoothed Local Histogram Percent Filter

Stevo Bailey
EECS Department
UC Berkeley
Berkeley, California
stevo.bailey@eecs.berkeley.edu

Garen Der-Khachadourian
EECS Department
UC Berkeley
Berkeley, California
gdd9@berkeley.edu

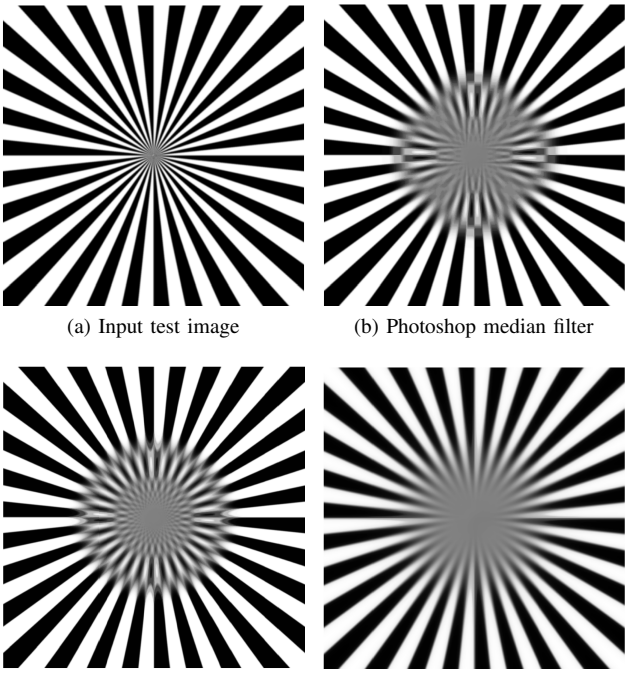
***Abstract*—A local histogram filter uses the information about the neighborhood around individual pixels to process an image. By constructing a smooth, isotropic, centrally-weighted histogram for each pixel, these filters have an advantage over pre-existing histogram filters. In addition to better-looking percent filters (e.g. the median filter, erosion, dilation, etc.), such histogram filters allow new mode-based filters as well. This project focused on smoothed local percent filters. This design's throughput was independent of neighborhood size, and a frame rate of 30 fps was easily reached with a 320x200 pixel image. The SRAMs used to buffer intermediate calculations grossly dominated the design's metrics, but further design exploration could reduce or eliminate this hinderance.**

***Index Terms*—histogram filter; IIR filter; percent filter, image processor; median filter; 2D spatial convolution**

I. Introduction

The local histogram of a pixel is defined as the set of data containing the pixel value and its nearest neighbors' values. These neighbor values may be weighted by their spatial distance from the pixel of interest, decreasing their significance the further they are from the current pixel. To smooth the local histogram, it may be convolved with a desired smoothing function, for example a Gaussian. Given a local histogram, a variety of filters may be easily computed. The popular median filter represents the 50% point of the histogram. Erosion and dilation morphological image processing can be expressed as the 0% and 100% points, respectively. By smoothing the histogram, modal filters like the mean-shift may also be constructed. These filters require three variations of the smoothed local histogram: the histogram itself, the histogram's derivative, and the histogram's integral.

The goal of this project were to implement a version of the algorithm presented by Kass and Solomon [1] in Chisel, a Berkeley object-oriented HDL built on Scala. The benefits of their smoothed, isotropic, centrally weighted histogram filtering are reduced spatial artifacts and new modal filters. Figure 1 shows a comparison of median filter results produced by Photoshop, by an isotropic equally-weighted neighborhood, and by their method. To verify the accuracy of the design, the algorithm was first implemented in MATLAB. Multiple designs were pushed through the Synopsys ASIC tool flow, producing results for a design space. These results were used to iteratively improve the design metrics. The metrics include total area, energy per cycle, energy per image, and throughput (frames per second, fps). The goal throughput was 30 fps for a VGA or CGA image (640x480 or 320x200).



(a) Input test image (b) Photoshop median filter

(c) Isotropic equally-weighted median filter (d) Kass [1] algorithm

Fig. 1. Comparison of median filter techniques

II. Algorithm

For our project, we restricted our design to percent filters, which require only the integral of the smoothed local histogram. The algorithm presented originates in [1], with 2D Gaussian spatial convolution as an IIR filter taken from [2]. A smoothed local histogram's integral may be expressed as a combination of two kernels: a neighborhood weighting kernel and a histogram smoothing kernel. To reduce artifacts and have pixel weights drop off with distance, an isotropic, center-weighted Gaussian weighting kernel was selected. For the smoothing kernel, an error function was used, in line with [1]. The algorithm may be summarized as

$$R_p(s) = erf(I_p - s) * W \tag{1}$$

where $R_p(s)$ is the smoothed local histogram's integral at a position s, erf represents the error function, I_p is the intensity of a pixel at point p, and W symbolizes 2D spatial convolution with a Gaussian function. Note that both kernels must sum to one for the histogram's integral to range from 0 to 1.

To elucidate the algorithm, we will walk through an example step-by-step. Keep in mind that, for each pixel, the goal is to step through the histogram's integral, find the desired percentage value, and report that value as the new pixel intensity.

Thinking ahead, we know that convolution on a certain domain will extend the domain, so convolving a histogram with a function will actually extend the domain of the histogram. However, our histogram's domain is restricted to integers in [0, 255] for an 8-bit pixel intensity value, so we must throw away values outside this domain. In essence, this means that the integral's range over the histogram's domain may not be [0, 1]. To properly compute the desired percentage value, we must find a "target" integral value for each pixel. For example, as seen in Figure 2, if a certain pixel's integral ranges from [0.2, 1], the median should be the histogram domain value at 0.6, half of this range. To get the integral's range and thus the target value, we first compute the integral value at the boundaries of the histogram's domain. Once a target value is calculated, we must step through each value along the histogram's domain, denoted by the s variable, until we reach this target and note the result.

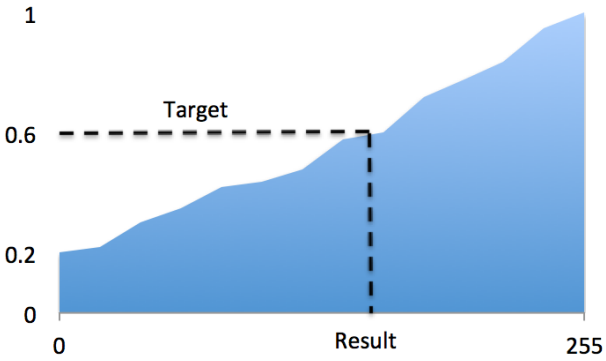


Fig. 2. Example smoothed histogram integral with target median value noted

At a specific value of s for a pixel, computing the integral value is a two-step process. The first step is to compute $erf(I_p - s)$, where I_p is the pixel's intensity, s is the histogram domain value, and erf is the error function (the integral of a Gaussian). To do this, we implement a look-up table (LUT). Since I_p is restricted to integers in [0, 255], and s is restricted to values in [0, 255], we precompute the error function over the set of inputs [-255, 255] and store these values in the LUT. If we stopped here, we would have the smoothed histogram integral for each pixel, but this histogram would not contain information from neighboring pixels.

To include a neighborhood, 2D spatial convolution of the integrals is performed. Conveniently, 2D spatial convolution can be broken into two discrete, sequential 1D convolutions, one in the x direction and one in the y direction. Deriche [2] presents an effective and stable method for 1D Gaussian convolution using two IIR filters. Because the Gaussian is non-causal, a forward IIR filter and a reverse IIR filter are independently computed and added. If y_k^+ gives the forward convolution, and y_k^- gives the reverse convolution, then y_k in (4) gives the 1D convolution of a non-causal function as an IIR filter of order 2. Our design used a fourth-order IIR filter.

$$y_k^+ = n_0^+ x_k + n_1^+ x_{k-1} - d_1^+ y_{k-1}^+ - d_2^+ y_{k-2}^+ \quad \text{with } k = 1, ..., N \tag{2}$$

$$y_k^- = n_1^- x_{k+1} + n_2^- x_{k+2} - d_1^- y_{k+1}^- - d_2^- y_{k+2}^- \quad \text{with } k = N, ..., 1 \tag{3}$$

$$y_k = y_k^+ + y_k^- \text{ with } k = 1, ..., N \tag{4}$$

The coefficients n_i^+, d_i^+, n_i^-, and d_i^- are given in [2] for a fourth-order filter (i from 0 to 4).

As an example, the first row of the image is passed through an IIR filter in the normal, forward direction. At the same time, the row is also passed backwards through another IIR filter with different coefficients, taken from [2]. These two convolutions are summed, completing the 1D convolution in one row of the x direction. After all the rows are finished, these results are pushed through two more IIR filters, one with the column data in the forward direction and one with the column data in reverse. When these results are summed, the smoothed local histogram integral has been found.

The final step is to find the target integral value, and note the domain value (s value) associated with it as the new pixel intensity value. Because this is all done in hardware, the histogram domain must be discretized. The error function is a low-pass filter, so we could ideally quantize s with large step sizes and interpolate between them. However, to save on having a divider in hardware, we instead opted for a smaller quantization step and no interpolation.

III. MATLAB Implementation

The proposed algorithm was first implemented in MATLAB both to understand its subtleties and to provide a benchmark for the hardware code. After writing a floating-point representation of the algorithm, a sample image from [1] was applied. Figure 3a shows the original, noisy image. After applying a median filter, or 50% filter, the reference paper obtained the output image shown in Figure 3b, while our MATLAB implementation produced Figure 3c. Note that to get these results, only the V channel of the HSV input image was filtered, in accordance with [1]. As seen, our algorithm correctly produces the desired output. The remaining few spots of noise in the actual output are attributed to a lack of interpolation before filtering. The reference paper produced Figure 3b by bilinearly interpolating before filtering, while our program did not. The floating-point MATLAB implementation had a VGA throughput of 0.015 fps, while the CGA throughput was 0.064 fps.

(a) Noisy test image (b) Expected output (c) Actual output

Fig. 3. Floating-point MATLAB implementation result

Since floating-point operations are more complex to implement than fixed-point operations, we opted for an entirely fixed-point design. To test the necessary fixed-point precision, a variation of the initial MATLAB algorithm was written. However, large images required substantial computation time, so fixed-point analysis was conducted only using the native 16-bit and 32-bit numerical types in MATLAB. Third-party fixed-point functions took too long. Figure 6 shows the results of the fixed-point analysis. Again only the V channel was filtered. Visual inspection probably identifies no differences between any of the filtered results.

IV. Chisel Implementation

After testing the algorithm in MATLAB, we mapped the design to hardware by constructing block diagrams. Storage elements were needed to hold pixel values, integral values, convolution coefficients and results. Computational blocks performed comparisons in the percent filter and 2D spatial convolution.

Figure 4 shows the top level block diagram of the design. The signals `frame_sync_in` and `frame_sync_out` indicate when the first 8-bit pixel intensity value of an image enters and exits the module. Look-up table entries, IIR convolution coefficients, percent values, and an s quantization step size (`s_step`) are loaded into the module during a loading phase. When the `ready` signal goes high, the user may either load a new configuration or load a new image to process.

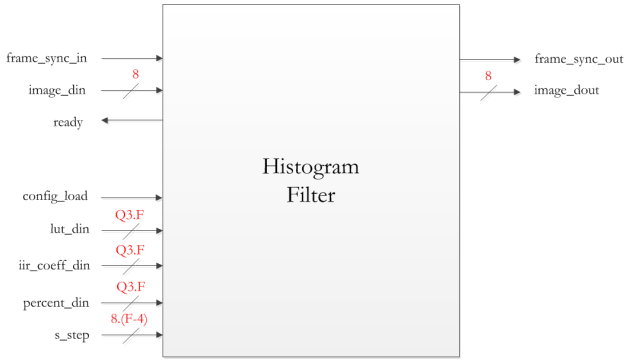


Fig. 4. Top level block diagram

The contents of the top level system are shown in Figure 5. Since the filter must access the entire input frame many times, an image buffer stores an entire frame. The image buffer was implemented with an SRAM to minimize area. The look-up table buffer (LUT buffer) holds error function look-up table entries. These entries, as well as IIR coefficients and the desired percentage, are calculated externally (ideally from a CPU) and loaded before filtering the image. The control unit tells the image buffer and datapath when to start a new slice and which slice to process, where a "slice" corresponds to a particular s value.

As stated in Section II, each histogram slice must be mapped through the error function look-up table, spatially convolved in both the x and y directions, and finally compared with the target integral value. The datapath, shown in Figure 7, is a 2-stage pipelined implementation of this operation on a single slice. Specialized subsystems were designed to perform these individual tasks.

A look-up table mapping module was built to map a pixel intensity value to an integral value stored in the look-up table buffer. Each row is then passed through convolution modules

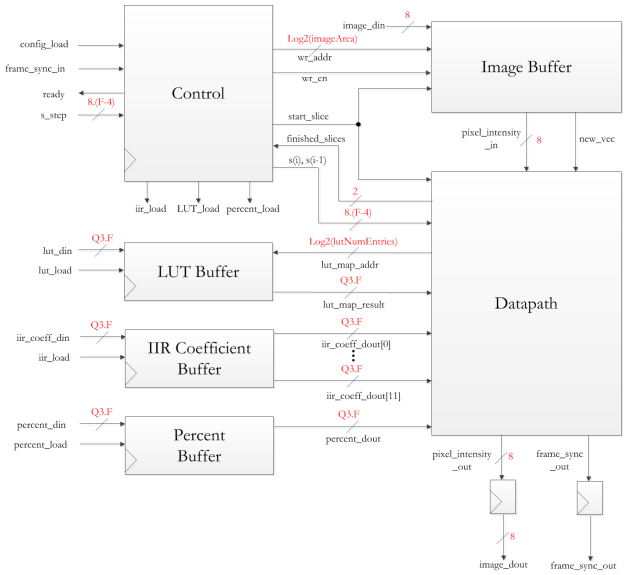


Fig. 5. Histogram block diagram

(a) Input test image

(b) Floating-point output

(c) 32-bit fixed-point output

(d) 16-bit fixed-point output

Fig. 6. Fixed-point MATLAB analysis results

which traverse in both the forward and reverse directions. Stack-like memory structures flip the row orientations to ensure that the correct convolution results are added together. The row convolution results are stored into an SRAM, which outputs data as columns to the column convolution modules. After the two-dimensional convolution is completed for a particular pixel, the result arrives at the percent filter, which compares the current integral value to the target based on the specified percent value.

The target values are stored in the percent filter by passing the last slice through the datapath first and then comparing the integral values with those of the first slice. The datapath then processes each slice in increments of the specified step size. Once an integral value for a particular pixel meets the target value, the 8-bit value that corresponds to the current slice replaces the target value in the target SRAM. In this way, the target SRAM serves as both a storage unit for target values and final results. After each slice has been processed, the filter outputs each pixel intensity value in the SRAM in sequential order and asserts the `frame_sync_out` signal.

Instead of designing a fairly complex control unit that handled all the control logic for the entire design, we incorporated some control logic in several of the more complex modules. For example, the percent filter in the datapath had a local state variable to indicate when it was storing target values, comparing integral values to the stored targets, and passing the final pixel intensity values to the output. This approach made the subsystems more autonomous and easier to test in isolation. The complexity of the control unit was also drastically reduced as the control unit was not responsible of keeping track of the states of the individual submodules.

A handshaking protocol was used between the control unit and the datapath to indicate when to start and stop processing a new frame. The main bottlenecks of the datapath are the row SRAM and the percent filter. These subsystems send control signals to the control unit to indicate when they have finished processing an image slice. After all the necessary configuration data have been loaded, the control unit asserts a signal called `start_slice` to inform the image buffer and the modules in the datapath that a new slice will be processed.

A signal called `new_vec` is asserted by the image buffer unit to inform the datapath that a new row will be processed. Each row is injected into the datapath from right to left, allowing the stack-like memory cells to invert the row such that the pixels arrive at the row SRAM from left to right. The `new_vec` signal is appropriately buffered along the datapath so that it arrives at the row SRAM when the pixel corresponding to the start of a row arrives at the input port.

After the forward and reverse convolution operations have been completed on each row and every pixel has been stored, the row SRAM asserts the `new_vec` signal and begins sending its entries in column order to the second half of the datapath. The data is then processed at the percent filter. When the row SRAM and percent filter have completed working on a slice, they assert `finished_slice` signals to indicate that they are ready to process a new one.

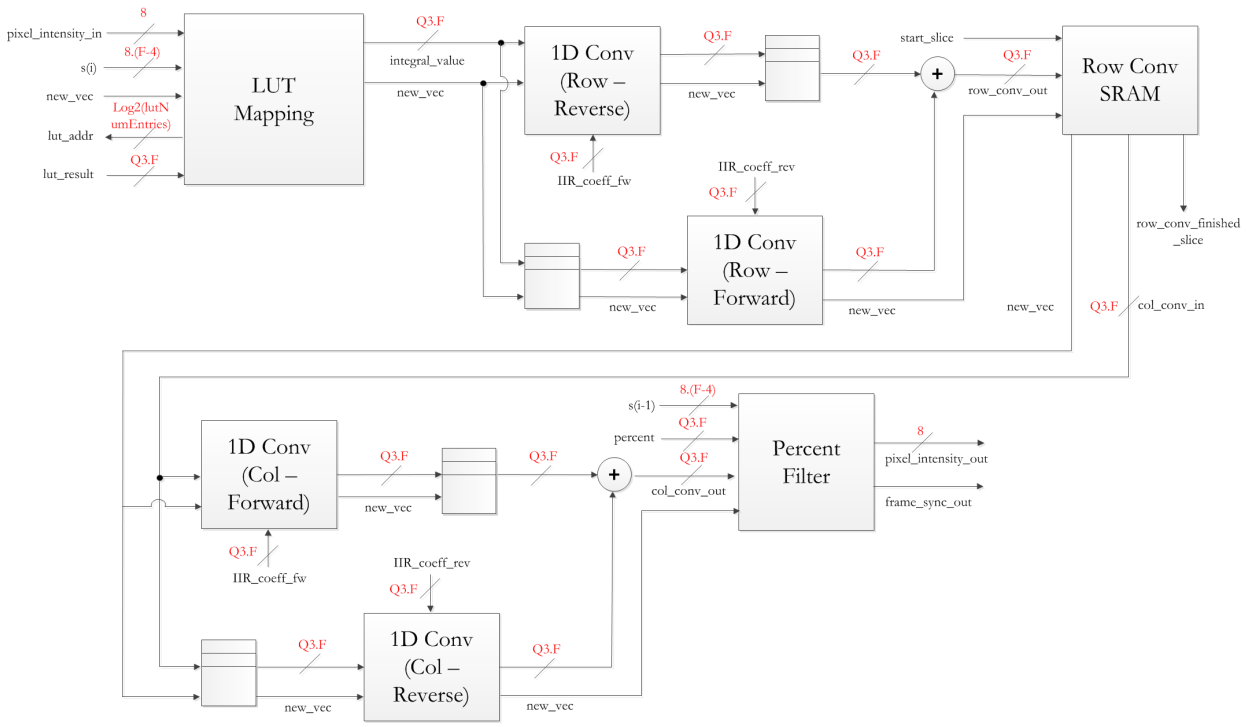


Fig. 7. Datapath block diagram

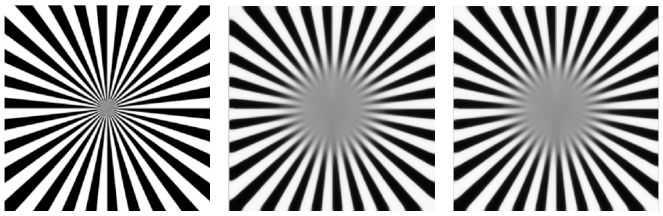
V. Testing Strategy

An incremental design and testing methodology was used for this project. The various subsystems were verified in isolation using unit tests. The approach provided a means of debugging issues earlier on in the design process. A top-level Chisel test was developed to evaluate the functionality of the entire design. A similar testbench was written in Verilog to assess the RTL in addition to the results produced by the synthesis and place and route steps in the Synopsis tool flow.

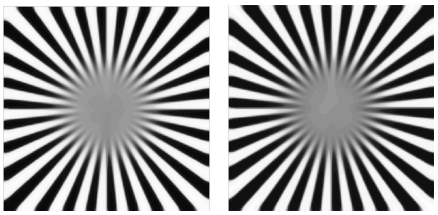
The design was originally tested for very small image sizes, in which case the SRAMs were replaced with register arrays. This methodology facilitated the debugging process because it led to shorter simulation times and generated results more quickly. After both the Chisel and Verilog tests produced results similar to the ones generated by the MATLAB implementation, the design was tested with larger image sizes. The SRAMs were eventually included in the tests, thus verifying the the overall functionality of the the entire design.

VI. Results

Because our algorithm does not interpolate when calculating an output value, the number of loop iterations our datapath performs can be quite large. If we calculate an integral value for every integer value of s, then the datapath iterates over 255 slices. Pipelining improves the throughput, but the total throughput could be dramatically increased by having larger quantization steps in s, and hence fewer datapath loop iterations. With a larger quantization, however, output image quality decreases. Figure 8 shows the artifacts introduced by increasing the quantization step size of the histogram domain.



(a) Input test image (b) step size of 1 (c) step size of 3



(d) step size of 5 (e) step size of 7

Fig. 8. Histogram quantization effects

A. Area

The design was pushed through the Synopsys ASIC tool flow with varying image sizes, bit widths, clock periods, and voltage settings. Although each build produced layouts with different areas, image size and bit width affected the area the most. Typical area numbers for 16-bit builds with CGA image sizes (320x200) resulted in total area sizes of 1.9 mm^2. Builds with the same bit-width but VGA-sized images (640x480) had total area values of around $8mm^2$. Doubling the bit-width led to areas of around 3.7 mm^2 and 15.3 mm^2 for CGA- and

VGA-sized images, respectively. The SRAMs constituted 90-98% of the total chip area in each build. Looking at the area without the SRAMs provides a more telling story of the impact of image size, bit width, clock period, and supply voltage on area.

Figure 9 below shows a breakdown of the area of the different subsystems with the SRAMs excluded. The percentages did not vary significantly across the different builds. The convolution modules were the majority of the area, which was expected since they contain many multiplication and addition blocks. The interconnect proved to be another major percentage of the area, which was expected since the SRAMs were fairly large and would require longer interconnect. The area of the row SRAM and percent filter modules without the large SRAMs were relatively small in comparison.

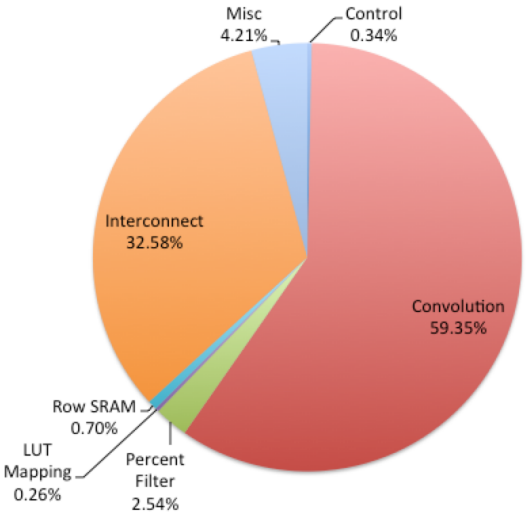


Fig. 9. Area breakdown

A layout for a 32-bit build with a CGA image size is shown in Figure 10. The large black boxes are the SRAMs: the image buffer in the top left corner, the row SRAM in the bottom right corner, and the two remaining SRAMs corresponding to the ones in the percent filter. The convolution blocks, shown in red, make up a large percentage of the remaining area. The percent filter, shown in green, is another noticeable chunk of the area. The control module (blue), the look-up table mapping module (purple), and the row SRAM logic (yellow) are relatively small in comparison. This diagram highlights the area percentages shown in Figure 9.

Pushing the design through the tool flow with a 4x larger image resulted in an area increase of about 25% when the other parameters were held constant and the SRAMs were excluded. The SRAMs were significantly larger for the VGA-sized images than for the CGA-sized images since more pixel data would have to be stored. The rest of the hardware should not have been significantly affected by larger image sizes. The primary reason for the area increase is due to the additional interconnect required because of the larger SRAMs.

Changing the bit-width from 16-bit to 32-bit more than

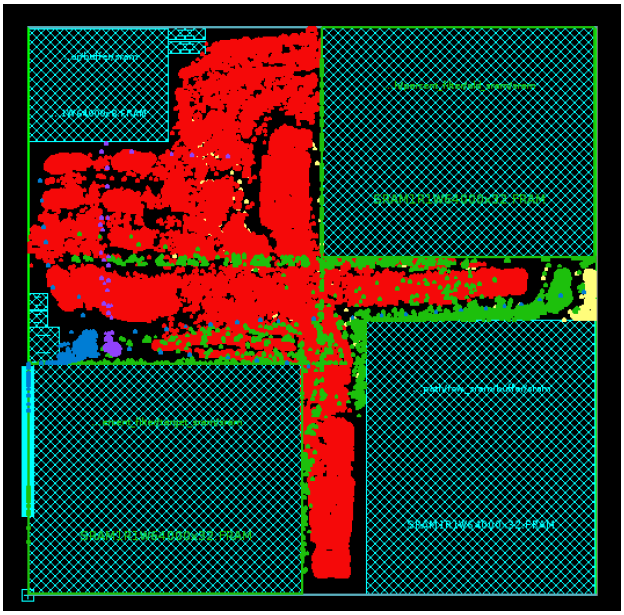


Fig. 10. Chip layout

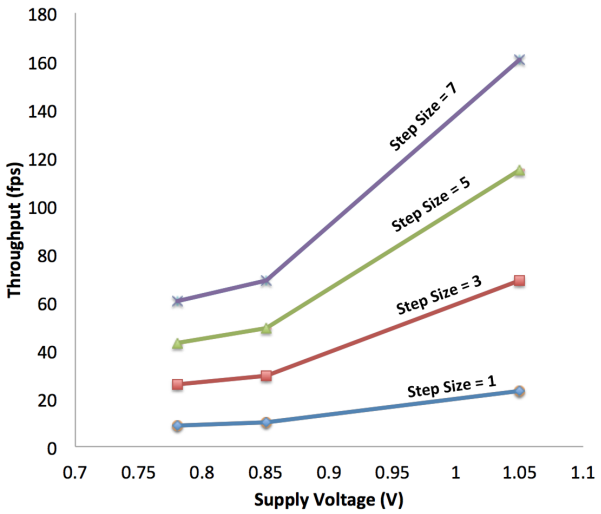
doubled the area of the filter. The wiring for many of the paths doubled in area as did the sizes of the computational blocks. Even when the SRAMs are included, the area increases by a factor of two.

When the designs were pushed through the flow with longer clock periods, slightly reduced area numbers were observed. The tools are able to make more optimizations when handling a design with a longer clock period. When a design is running at a clock frequency that barely satisfies the timing constraints, the tools make less effective optimizations which can lead to larger area sizes.

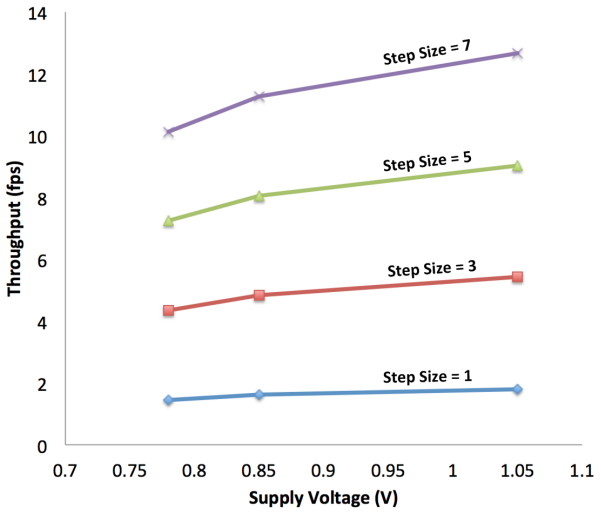
An increase in area was observed when the voltage was reduced and the other variables were held constant. Lowering the voltage makes it more difficult for the design to meet the timing constraints for a fixed clock period. For that reason, the tools cannot optimize the area as effectively.

B. Performance

We measured performance in frames per second (fps). The goal was 30 fps. The charts in Figure 11 show that this goal was met only for CGA images at specific voltage levels and quantization levels. The number of quantization levels is given by $255/(step\ size)$. As expected, the VGA design had a much lower throughput because of its larger area. Lowering the supply voltage also lowered throughput because the clock frequency decreased to meet timing. Additionally, increasing the histogram quantization step by a factor of two decreased the performance by a factor of two. Fewer quantization steps mean fewer datapath loop iterations. Bigger bit widths decreased throughput due to increased area and delay in the SRAMs and computation blocks. Because of the poor throughput of 32-bit designs, their data are not included here.

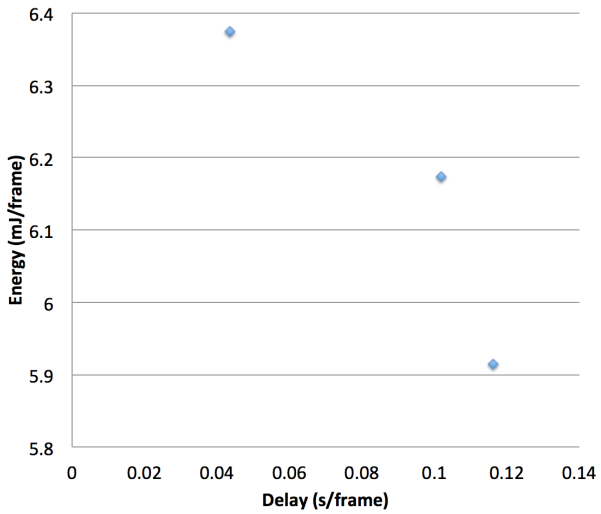


(a) 16-bit CGA performance

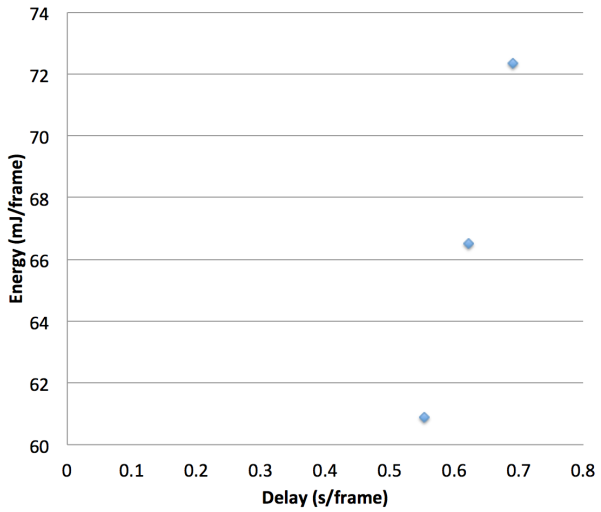


(b) 16-bit VGA performance

Fig. 11. 16-bit performance with CGA (320x200) and VGA (640x480) sizes



(a) 16-bit CGA energy



(b) 16-bit VGA energy

Fig. 12. 16-bit energy with CGA (320x200) and VGA (640x480) sizes

The critical path in the design appeared to be through the convolution blocks. Specifically, after optimizations were made, the critical path was often through one multiplier and two adders within an IIR filter computation block. Further buffering or pipelining this critical path was not possible because of the required single-cycle delay inherent in IIR filters.

C. Energy

Simulation of larger images required too much computation time, so a small, 8x8 image was pushed through the Synopsys flow and simulated. The extracted activity factors were used to obtain a more accurate dynamic power for our design. Since the loading phase uses only 5% of the cycles of an 8x8 image, and the power during loading is probably less than 5% of the total because the other modules are not switching during loading, the loading phase power was assumed small. Extrapolating the results from the 8x8 image to larger images produced the energy-delay curves seen in Figure 12.

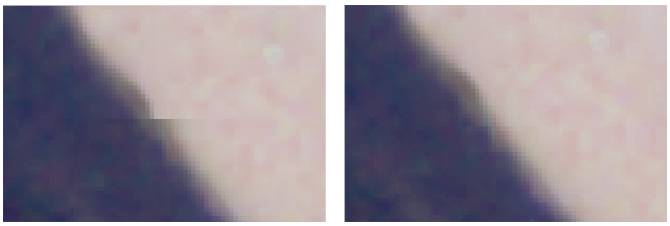
Figure 12 shows how the energy and delay of processing a frame vary with changing supply voltage. Lower supply voltages increased the delay, and, for the CGA design, decreased the energy. However, because the VGA design is so large, the energy gets worse as supply voltage scales down. The savings in switching energy are offset by the increased delay.

VII. Extensions

The design's area severely limits its realization in a co-processor. The large SRAMs needed take up 90-98% of the total design area. Design alterations that reduce the area may reduce performance, but some current designs already have well over 30 fps throughput. Further exploration should be done to reduce area while keeping throughput at reasonable levels.

A naive way to reduce SRAM area would be to recalculate necessary values instead of storing them in a memory unit. However, because calculating a specific integral value for one pixel requires both look-up table mapping and 2D spatial convolution for the entire image, recalculating values to avoid storing them in memory would reduce throughput to undesired levels.

One possible design alteration would be processing the image in pieces. Dividing a large image into four subimages would require the same size image buffer, but the datapath buffers could be about one-fourth as big. Issues arise on the edges, where pixels outside the subimage must be included to obtain the correct neighborhood. The number of outside pixels to include depends on the sigma value of the weighting kernel. Figure 13 shows the boundary artifacts produced without properly expanding the subimages, as well as diminished boundary artifacts by properly expanding. While this design drastically reduces the area, it also limits the user's choice of weighting kernel size. The control unit would change to accommodate this new algorithm, but the throughput would remain close to original algorithm's throughput.



(a) No extra boundary pixels included (b) 2σ extra boundary pixels included

Fig. 13. Subdividing an image produces boundary artifacts if the subimages are not extended properly

Another design choice would be using an FIR filter in place of the IIR filter for 2D spatial convolution. Like subdividing the image, an FIR filter limits the possible neighborhood size. Increasing the available neighborhood size would decrease the throughput dramatically, so an easy trade-off exists between FIR filter size and throughput.

In a coprocessor chip, the image would ideally be stored in the CPU cache or main memory, eliminating the need for the image buffer. Also, if the histogram filter had access to main memory, it could store intermediate data values there instead of in large, local SRAM blocks.

VIII. Conclusion

This project verified the algorithm presented by Kass and Solomon [1] and implemented it in Chisel. To explore the design space of this algorithm, certain parameters were varied. These parameters include the image size, the histogram quantization, the supply voltage, and the clock frequency. While smaller image sizes and fewer quantization steps produced better metrics, they limit the processor's capabilities. In general, supply voltage and clock frequency provide knobs which adjust the energy and throughput of the design. The target throughput of 30 frames per second was met with a CGA image size, a standard supply voltage, and a moderate number of quantization levels. However, output image quality varies somewhat with histogram domain quantization, so care must be taken when selecting a desired number of levels.

Our project merely scratched the surface of what is possible with smooth local histogram filters. A percent filter is one application, but there are many other interesting algorithms that could be implemented with a similar approach. Percent filters use the integral values of the histogram to determine the output pixels. Other examples that use local histogram data are dominant-mode and closest-mode filters, which utilize the raw histogram as well as its derivative and integral. Designing these filters on top of our percent filter would require more look-up tables for the additional histogram information as well as different computation blocks at the tail end of the datapath.

IX. Course Reflections

This course was an incredibly valuable learning experience. This project taught the importance of following an incremental design approach and provided the opportunity to take a algorithm through the entire design cycle. Learning the algorithm proved to be a major obstacle in the beginning due to its complexity. Ideally, we would have learned the algorithm and implemented a design in MATLAB more quickly so that we would have had more time to develop and refine the RTL. An important facet of the class is learning about the flow tools and making design decisions based on the results generated by pushing RTL through synthesis and place and route. If we were able to push the design earlier, it would have allowed us to revise our design and make more informed decisions.

References

[1] M. Kass and J. Solomon, "Smoothed local histogram filters," SIGGRAPH 2010.

[2] R. Deriche, "Recursively implementing the Gaussian and its derivatives," Tech. Prep. 1893, INRIA, Unit de Recherche Sophia-Antipolis, 1993.