Comparametric HDR (High Dynamic Range) Imaging for Digital Eye Glass, Wearable Cameras, and Sousveillance

Mir Adnan Ali, Tao Ai, Akshay Gill, Jose Emilio, Kalin Ovtcharov, and Steve Mann
Department of Electrical and Computer Engineering
University of Toronto
10 King's College Rd, Toronto, ON, M5S3G4

***Abstract*—Wearable computing can be used to both extend the range of human perception, and to share sensory experiences with others. For this objective to be made practical, engineering considerations such as form factor, computational power, and power consumption are critical concerns. In this work, we consider the design of a low-power visual seeing aid, and how to implement computationally-intensive computational photography algorithms in a small form factor with low power consumption.**

We present realtime an FPGA-based HDR (High Dynamic Range) video processing and filtering by integrating tonal and spatial information obtained from multiple different exposures of the same subject matter. In this embodiment the system captures, in rapid succession, sets of three exposures, "dark", "medium", and "light", over and over again, e.g. "dark", "medium", "light", "dark", "medium", "light", and so on, at 60 frames per second. These exposures are used to determine an estimate of the photoquantity every 1/60th of a second (each time a frame comes in, an estimate goes out).

This allows us to build a seeing aid that helps people see better in high contrast scenes, for example, while welding, or in outdoor scenes, or scenes where a bright light is shining directly into the eyes of the wearer. Our system is suitable for being built into eyeglasses or small camera-based, lifeglogging, or gesture-sensing pendants, and other miniature wearable devices, with low-power and compact circuits that can be easily mounted on the body.

I. Introduction

There are two broad categories of veillance, **sur**veillance, and **sous**veillance. The primary dictionary definition, given in [1], of the word "surveillance" is

> surveillance, *n.* : a watch kept over a person, group, etc., especially over a suspect, prisoner, or the like: *The suspects were under police surveillance*.

The etymology of this word is from the French word "surveiller" which means "to watch over". Specifically, the word "surveillance" is formed from two parts: (1) the French prefix "sur" which means "over" or "from above", and (2) the French verb "veiller" which means "to watch".

A more recently coined word is the word "sousveillance", which is an etymologically correct opposite formed by replacing the prefix "sur", in "surveillance", with its opposite, "sous". Table I summarizes the veillances (surveillance and sousveillances) and the etymologies of these words. Sousveillance often refers to cameras borne by people, e.g. hand-held cameras or wearable cameras [2], [3], [4], [5], [6], [7].

TABLE I: The Veillances: Surveillance and Sousveillance

English	**French**
to see	voir
to look (at)	regarder
to watch	**veiller**
watching (monitoring)	**veillance**
watching over (oversight)	**surveillance**
to oversee (to watch from above)	**surveiller**
over (from above)	**sur**
under (from below)	**sous**
"undersight" (to watch from below)	**sousveillance**

A. Motivation

In traditional photography or surveillance, a great deal of effort is often expended to arrange the camera and lighting for best exposure. Photographers will often position themselves for best view, i.e. with the sun behind them, so that they are not shooting directly into the light.

In our application, however, we wish to create a computer vision system (seeing aid) that can work in any lighting situation as might occur in day-to-day life. The Digital Eye Glass, for example, observes the real world in a similar fashion as the user observes (or would have observed, in the case of a blind individual). The sousveillance user's total lack of control over lighting, combined with the wide variety of lighting situations encountered in the course of a day (e.g. from direct sunlight to candlelight) necessitate that any sousveillance apparatus be equipped to produce a useful signal, without requiring large battery packs or a large-aperture camera.

B. Related Work

Cameras can only take in photographs with limited dynamic range. One method to overcome this is to combine differently exposed images of the same subject matter, producing a High Dynamic Range result [8], [9], [10]. HDR digital photography started almost 20 years ago. "The first report of digitally combining multiple pictures of the same scene to improve dynamic range appears to be Mann" [11]. Now, it is possible to produce HDR photography [12], [13] and video [14], [15], [16], [17] in realtime, on both high-power CPU/GPU systems, as well as low-power FPGA boards

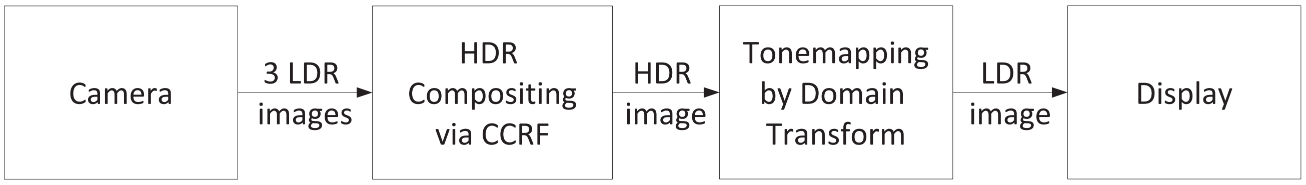


Fig. 1: System diagram. Input low-dynamic range (LDR) images are composited into an high-dynamic range (HDR) image, which is then tonemapped (spatio-tonal mapping) to allow the end user to see details in the highlights and lowlights of the scene. In this work we examine resource-efficient implementation of HDR Compositing and Tonemapping suitable for wearable computers. In particular, we target a low-power FPGA architecture with severe memory constraints.

[18]. However, other FPGA implementations have relied upon methods that are less accurate than current CPU and GPU-based methods. Namely, the earlier FPGA approach used weighted sums for image compositing, and a simple inverse power function to provide compression of the dynamic range.

In the present work, we demonstrate that it is possible to implement an high-quality HDR system on FPGA. First we present a novel method for implementing 2D lookup tables (LUT) on an FPGA architecture, bounding the error and space of a quadtree representation of the lookup table by the expected use of the table, so that the LUT is compressed by roughly $60\times$, this fitting within the block RAM of a typical FPGA. Second, we present an FPGA implementation of the domain transform, used for tonemapping, which is optimized for wearable application since the FPGA platform uses far less power than a traditional CPU or GPU implementation.

C. Description of Implementation

The overall system is as shown in Fig. 1, where a wearable camera provides multiple exposures of the subject matter, which are then composited into a single HDR image. This image is then reduced to an LDR image for display using the domain transform for tonemapping. This pipeline is implemented completely in FPGA.

II. CCRF Compression

In [19], a novel method for HDR compositing is presented. This method makes extensive use of a 2D lookup table, called the CCRF (comparametric camera response function). The CCRF is computed such that for two input pixel values as returned from a camera, it returns a refined estimate of the scene's photoquantity.

The CCRF is used for HDR compositing by giving a pairwise estimate of photoquantity $\hat{q}$ from two pixel values f_1 and f_2 with exposure difference ΔEV. On systems without significant memory constraints, the CCRF can be stored in a 2D lookup table. The lookup table contains $N \times N$ elements of pre-computed CCRF results on pairs of discretized f_1 and f_2 values. Every CCRF has the domain of a comparagram [20] and the range of a camera response function, as depicted in Fig 2.

A. Quadtree Representation

The CCRF can also be represented in a tree structure, rather than as a look-up table (LUT). For $N \times N$ elements, we can generate a quadtree, where all nodes are square and each parent node in such a tree contains four children nodes, to fully represent the CCRF. If the quadtree is a complete tree then it would have with $\log_4 N^2$ or $\log_2 N$ levels. However, to compress the tree, we use larger leaf nodes in smooth areas, and in areas that are less-used, as shown in Fig. 2 (rightmost).

One method of generating such a tree is to recursively divide a unit square into four quadrants (four smaller but equal size squares). We can visualize the center of divided unit square as the parent node of the four quadrants. The center of each quadrant is considered a child node. Such process is performed recursively in each quadrant until the root unit square is divided into $N \times N$ equally sized squares. The bottom nodes of the quadtree are the leaves of the tree, each of which stores the CCRF lookup value of the corresponding pixel pair $(f_1,\ f_2)$.

B. Reducing the Quadtree

We observe that the most frequently accessed CCRF values lie along the comparagram. Therefore, higher precision on interpolation may not be necessary for CCRF lookup points that are distant from the comparagram. This suggests that the error constraint for the pair $(f_1,\ f_2)$ should vary depending on its likelihood of occurrence. To further compress the CCRF lookup table, we can scale e by the number of observed occurrences of the pair $(f_1,\ f_2)$.

To reduce the number of elements needed for storage, we proceed to interpolate the CCRF value f_{quadtree} of a specific pair $(f_1,\ f_2)$ based on the corner values of the leaf node that the point falls in. The value f_{quadtree} interpolated based on the corner values of the corresponding leaf node is compared against the actual CCRF lookup value f_{CCRF}, giving the error e per table entry:

$$e = |f_{\text{quadtree}} - f_{\text{CCRF}}|. \qquad (1)$$

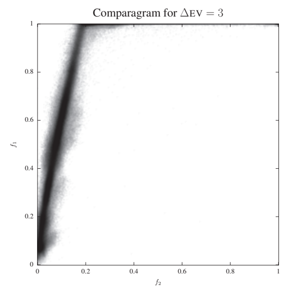
We accept the approximated result if e is within the error threshold e_{thresh}:

$$e_{\text{thresh}} = e_{\max}/C_{\text{normal}} \qquad (2)$$

where $C_{\text{normal}} = (C_{ij}/C_{\max} + 1)$ is the normalized mass of the comparagram at the point being estimated, and $e_{\max}$ is the maximum allowed error.

The purpose of the division is to obtain a more accurate reconstruction at that point, at the cost of making the quadtree larger. The closer the corner values are to the point of interpolation yield a lower e, as CCRF lookup table generally varies smoothly over a continuous and wide range of f_1 and f_2 values. Therefore, we expect that the density of the divisions corresponds to the local gradient of the CCRF lookup table: the higher the local gradient the more recursive divisions are required to bring corners closer to the point; whereas the points formed of a large and smooth region of CCRF with low local gradient share the same corners.

The error of the interpolation against the original lookup value at the input pair $(f_1,\ f_2)$ should be within e_{thresh}. This error is affected by the interpolation method. Empirically,



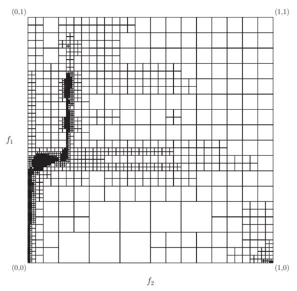


Fig. 2: *Leftmost:* A comparagram is a joint histogram of pixel values from images of the same scene taken with different exposures. For any given sensor, the comparagram is directly related to the camera response function. Areas that are dark in this plot correspond to joint values that are more likely to occur. *Rightmost:* Quadtree based representation of the CCRF based on weighting from the comparagram. Areas of rapid change or high use are more finely subdivided for greater accuracy. Inside each square the CCRF value is approximated using bilinear interpolation based on the corner values. Note the self-similar nature of the representation, with a granularity that matches "regions of interest", defined by the error in reconstruction and expected frequency of use.

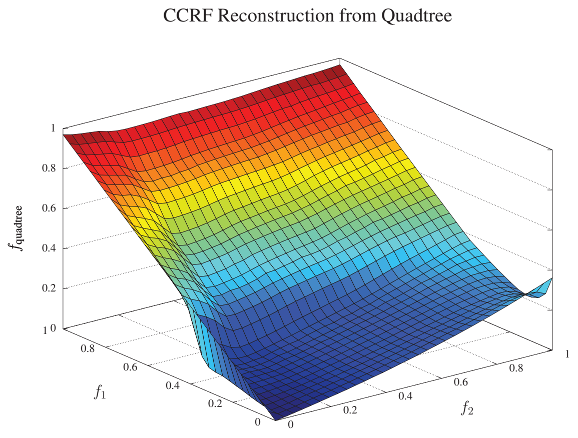


Fig. 3: Reconstruction of CCRF lookup table based on compressed quadtree with error constraint of within one pixel value (α set to 1.0). The error bound is weighted by the expected usage, so that values used more often have a smaller error bound.

we find that bilinear interpolation works better than quartic interpolation in terms of minimizing the number of lookup points while satisfying the error constraint.

The CCRF lookup table we use has 1024 columns for f_1 and 1024 rows for f_2. Therefore, there is no point to construct a tree that has more depth than $\log_2 1024 = 10$. We may constrain the depth of the tree to fewer levels than 10 as long as the error constraint is met. This affects the resulting number of entries of the CCRF quadtree as well as the number of iterations required for the search of a leaf node.

C. Corner Value Access

Each node of the quadtree is the center point of the square that contains it. To access the corner values of a leaf node, we can perform a recursive comparison of pair (f_1, f_2) to (f_x, f_y) of the non-leaf nodes in the tree until a leaf node has been reached, seen in Algorithm 1. The leaf nodes contain memory addresses of corresponding corner values, which are also stored in memory.

Algorithm 1 Recursive Quadtree Search

```
procedure GET CORNERS($f_1$, $f_2$, Node)
    if Node is not a leaf then
        if $f_1$ < Node→ $f_x$ then
            if $f_2$ < Node→ $f_y$ then
                get corners($f_1$, $f_2$, Node→Child(left, down))
            else
                get corners($f_1$, $f_2$, Node→Child(left, up))
        else
            if $f_2$ < Node→ $f_y$ then
                get corners($f_1$, $f_2$, Node→Child(right, down))
            else
                get corners($f_1$, $f_2$, Node→Child(right, up))
    else
        retrieve corner values
end procedure
```

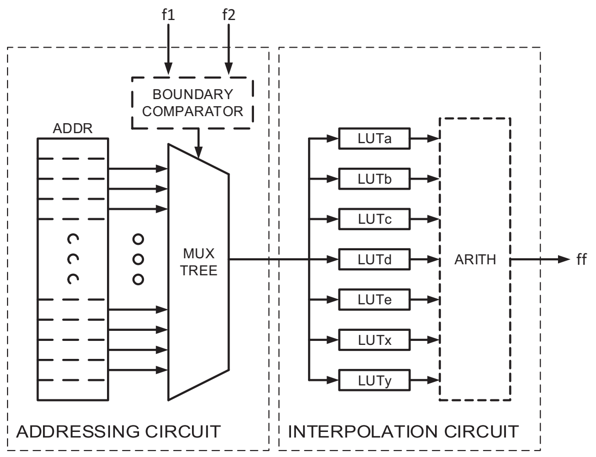


Fig. 4: The top-level system consists of two main part: the addressing circuit and the interpolation circuit. Input pixel values are normalized first to 16-bit fixed point representation (f_1, and f_2) before entering the Boundary Block. According to the boundary conditions of the two values, the circuit generates controlling signals to the multiplexer tree, which selects the correct address that correspond to the original quadtree node. The address is then used in the interpolation circuit to retrieve the stored values that are necessary for bilinear interpolation. After arithmetic circuit operation, the output f_{quadtree} can be further combined with f_{quadtree} from another instance of the same circuit.

III. Quadtree Implementation

The algorithm can be implemented efficiently on an medium-sized FPGA. Given a finalized quadtree data structure, a system can be generated using software. The 4 corner values are stored in ROMs implemented with on-chip Block RAM (BRAM), and then selected using multiplexer chains based on the input f_1 and f_2. An arithmetic circuit that follows after can then calculate the result based on bilinear interpolation. Thus as shown in Fig 4, the system consists of two major parts: an addressing circuit and an interpolation circuit.

Since the size of quadtree can grow to ten levels, a C program is written for generating the implementation of the addressing circuit in Verilog HDL. Given a compressed quadtree data structure, this program generates four ROM initialization files and a circuit that retrieves corner values stored in the ROMs based on the inputs.

A. Addressing Circuit

Each of the quadtree leaves needs a unique address. This address is then used to retrieve the corresponding corner values from ROM. As shown in Fig. 5, the circuit outputs the address

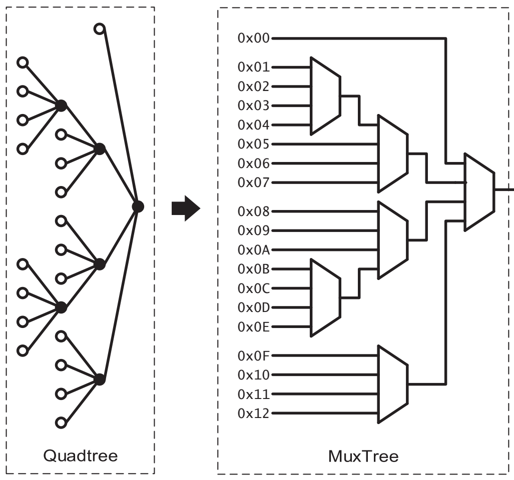


Fig. 5: The relation between the original quadtree and its multiplexer implementation makes it very easy to generate the Verilog using the same quadtree date structure in software. Efficiently using 4-to-1 multiplexers in the 6-LUT FPGA architecture can significantly reduce the resource usage (i.e. each multiplexer is mapped to one logic slice, instead of three if 2-to-1 multiplexer is used) and code generation algorithm.

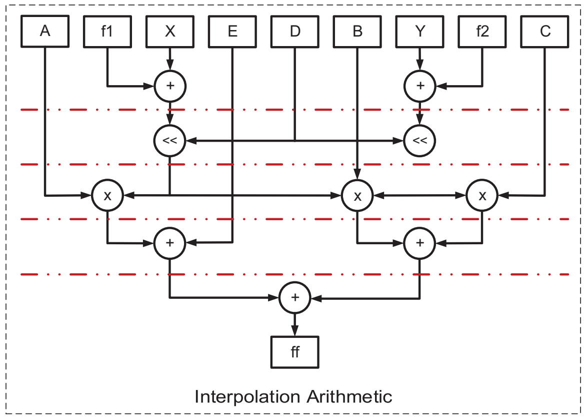


Fig. 6: This circuit performs pipelined arithmetic that is necessary for bilinear interpolation. The inputs to this circuit are loaded from Block RAM, which is initialized according to the compressed quadtree structure. The red dotted lines indicates the stage after which the data can be pipelined in order to have higher throughput.

by comparing f_1 and f_2 with constant boundary values, in the same way as we traverse the quadtree. The main function of the Boundary Comparator is to send the controlling signal to traverse through the multiplexer tree, based on the given input pair. At each level, it compares the input values to the pre-stored center coordinate values, and determine which branch (if exits) it should take next. Otherwise, the current node is a quadtree leaf and a valid address is selected.

The algorithm generates the circuit by traversing through the quadtree. Since a new unique address is needed for every leaf being visited, a global counter is used to determine the addresses. The width of the circuit datapath is then determined using the last address generated (i.e. the maximum address).

B. Interpolation Circuit

The circuit takes the address provided by the addresing circuit as and input and uses it to look up values that are pre-stored in the Block RAM. These values can be used to perform arithmetic operation (as shown in Fig. 6) for bilinear interpolation. In order to maintain high throughput, the intermediate stages are pipelined using registers.

	Size of Compressed CCRF			
	α=1	α=4	α=16	α=64
Number of entries	4087	4102	4234	4954
Compression factor	64.1	63.9	61.9	52.9
Mean error	0.00099	0.00099	0.00099	0.00098
Expected error	0.00091	0.00090	0.00088	0.00064
Mean depth	5.5	5.5	5.5	5.5
Expected depth	6.6	6.6	6.6	6.9
Actual slice usage	869	894	932	1129

TABLE II: The number of entries depending on the choice of α. The compression factor is calculated based on the number of CCRF lookup entries required divide by the number of the entries after compression. The CCRF without compression contains 1024×1024 floating-point entries.

C. Compression Performance

We wrote the compression in C programming language to output the compressed CCRF lookup table. From Table II, we list the minimum compression factor, expected and mean error over the entire CCRF lookup, with four selections of α value.

The minimum compression factor summarizes the amount of compression achieved by taking the ratio of the total number of entries in the CCRF lookup table over the compressed one:

$$\text{minimum compression factor} = \frac{N^2}{\text{\# leaf nodes} \cdot 4} \qquad (3)$$

The constant 4 is the upper bound of the maximum redundancy of the corner value storage that overlap with adjacent CCRF lookup points.

For α = 1, the effective maximum error bound is to one pixel value. As α increases, the tighter error bound demands higher precision from the interpolated results. However, the number of entries after compression scales sublinearly with respect to increase in α. This shows our compression algorithm is capable of achieving high compression rate with tight error bound.

For resource estimation without considering optimization effort performed by the CAD tool, we have:

$$\begin{aligned} \text{\# Multiplexers} &= \text{\# None-leaf Nodes} \\ \text{\# Comparators} &= \text{\# None-leaf Nodes} * 2 \\ \text{\# Addresses} &= \text{\# Leaves} \end{aligned}$$

The resource usaged reported by the vendor CAD tool are also summarized in Table II.

The proposed CCRF compression algorithm simply utilizes the logic slices as an alternative memory available on the hardware. However, the slice usage is sublinear with respsect to the increase of the error bound, indicating an efficient usage of logic slices.

This result is amenable for implementation on highly memory-constrained low-power FPGA platforms. Construction of the compositing function is performed by non-linear optimization of a Bayesian formulation of the compositing problem, where we select our prior algorithm to create an accurate estimator that is smooth for robustness and enhanced compression. We solve the estimator over a regular grid in the unit square, forming a two-dimensional lookup table.

Implementation of this solution on FPGA-based I Eye Glass then relies on the compression of this looku into a quadtree form that allows for random access, an bilinear interpolation to approximate values for interm points. This form allows for selective control over bounds, depending on the expected use of the table, is easily obtained for a particular image sensor.

IV. Domain Transform Filtering Implementa

Spatial tonal mapping and compression needs performed on the HDR video since ordinary monito not display the produced HDR video stream. The d transform method proposed by Gastal et al. [21] reduc computational intensity of many existing multi-dimen high-quality filter kernels [22], [23].

The domain transform method reduces the computa intensity of many existing multi-dimensional filter kernel The key idea is to construct a 1-dimensional filter kern preserves the original distance in higher dimensions between adjacent pixels, thus maintaining the edge-preserving property of filter kernels in lower dimensions [23].

Assuming that the number of independent channels c in the input frame is three. Let $I(x, y)$ represent the two dimensional input frame.

$$I'_{cx} = \frac{dI_c}{dx} = I(x+h) - I(x)$$
$$I'_{cy} = \frac{dI_c}{dy} = I(y+h) - I(y)$$

where, h = 1.

Partial finite differences are taken to compute the domain transform. The domain transforms, ct'_x and ct'_y are calculated as:

$$ct'_x(u) = 1 + \frac{\sigma_s}{\sigma_r} \sum_{i=1}^{c} |I'_{cx}|$$
$$ct'_y(u) = 1 + \frac{\sigma_s}{\sigma_r} \sum_{i=1}^{c} |I'_{cy}|$$

σ_s and σ_r are the spatial standard deviation and range standard deviation of the filter kernel respectively [21]. The domain transforms $ct'_x(u)$, $ct'_y(u)$, describe the difference in pixel values between adjacent pixels in 1-dimensional space. It is important to process all channels of the frame at once for edge-preserving filtering. Processing each channel independently can introduce artifacts near the edges [21].

The input image is then 1-dimensionally convolved with the filter kernels $V_x(u)$ and $V_y(u)$. The horizontal convolution is performed first on the input image with $V_x(u)$. The result, $F_c x$, is then vertically convolved with $V_y(u)$ to obtain the output image.

$$V_x(u) = \exp\left[(-\sqrt{2}/\sigma_s)(ct'_x(u))\right]$$
$$V_y(u) = \exp\left[(-\sqrt{2}/\sigma_s)(ct'_y(u))\right]$$

The effect of the filter is scaled inversely proportional to the domain transform, $ct'(u)$. Namely, if pixel u has a large $ct'(u)$ value, an edge occurs at u. The filtering has little effect on pixel u when it is large, hence preserving the edges.

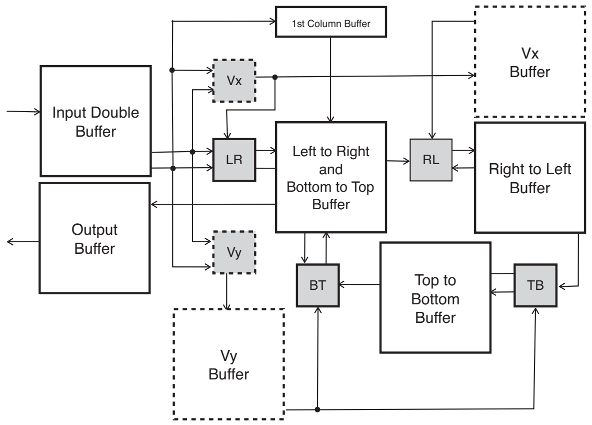


Fig. 7: The top-level block diagram of the filter datapath. Vx and Vy are calculated just once from the original frame. The output of each filter is the input of the next. Therefore the output of each of the three inner filters needs to be fully buffered in an intermediate block RAM.

$$F_{cx}(u) = \int_{-\infty}^{\infty} I_c(x) V_x(u-x)dx$$
$$F_c(u) = \int_{-\infty}^{\infty} F_{cx}(y) V_y(u-y)dy$$

This algorithm applies a filter on the input pixel stream, as shown in Fig 7. The filter performs horizontal filtering and then vertical filtering using kernels Vx and Vy respectively, which are computed from the input image. In total this takes four passes on the image: left-to-right, right-to-left, downwards and upwards.

When performing the horizontal filtering, the filter first processes the frame left to right as shown in Fig 8. The filtering result of a given pixel relies on the result of computations performed to all pixels along the same line up to the current pixel. When computing the horizontal filtering, since there is no vertical dependency between the filtering, we pipelined columns of data instead of proceeding line by line. This greatly improved the performance of the algorithm by reducing the latency and producing continuous throughput. In a similar way, when implementing the vertical filtering, we took advantage of the horizontal independency and computed whole lines of data on each pass over the image.

Waiting for the previous output would not be acceptable, since this would introduce significant pipleine latency. Utilizing these pipeline stages to carry out additional computations would allow us make better use of the highly pipelined functional units and achieve a high throughput.

Since there is no vertical depedency between pixels during the horizontal filtering process, the hardware is piplelined, making use of the vertical independece by performing the horizontal filtering on two columns at a time.

Once the left to right pass is completed, the filter acts right to left on the computed results. At present, this is achieved by buffering the results in on-chip block RAM.

As a proof of concept, this paper presents the FPGA implementation of this recursive filtering process.

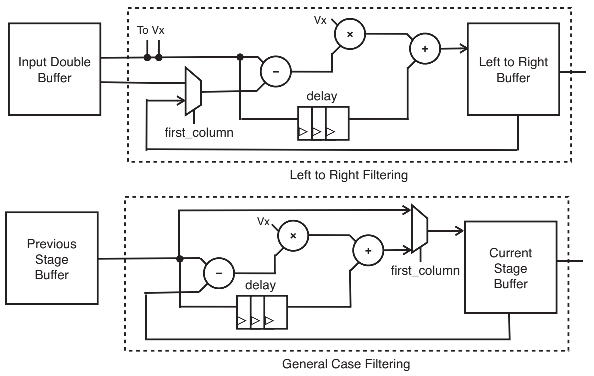


Fig. 8: Filter operations are described here in detail. The figure above shows the datapath for the left to right filter, which is the only filter that does not need to read Vx or Vy from the buffer. Its output needs to be buffered since the right to left filter cannot start until the image has been completely processed. All the filters follow the pattern shown here.

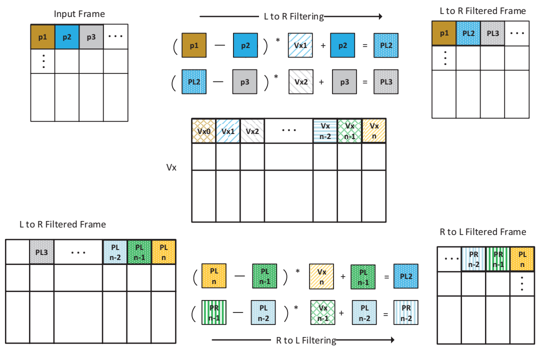


Fig. 9: Visual representation of the horizontal filtering operation in the recursive filter operations. The left to right filtering is shown on the top half of the figure, and the right to left filtering is shown on the bottom half of the figure. Pixels and their relative positions in the matrices involved are uniquely identified by colors and patterns. Note the recursive nature of the algorithm and the horizontal data dependency.

A. *Input Block RAM*

To take advantage of the data independency described earlier, this block RAM stores transposed versions of the input frames. Incoming data is stored and replicated in two buffers in order to provide an additional read port. This allows for two adjacent column pixels to be accessed simulatenously at every clock cycle allowing for maximum throughput.

Two buffers are used to store the incoming integer pixel data of all three 8-bit colour channels. The RAM blocks store a window of the incoming image or video frame.

B. *Floating Point Conversion and Normalization*

The data is read column wise in fixed point format from the RAM modules and is then converted to floating point representation. The data is then normalized to the range[0,1]. Floating point computation provides more accuracy as compared to its fixed point counterpart. Since this algorithm is computationally intensive, highly pipelined functional units that run above 200 MHz provide the necessary computational bandwidth.

C. *Double Buffering and Addressing*

Two columns must be read simultaneously, which requires the original frame to be buffered twice. The pixels are accessed by assigning each value a line address and a column address. This way of uniquely addressing the pixels with two coordinates allows better control of the reading and writing of values that access the buffers. Another advantage of this type of addressing is that a frame can effectively be transposed by swapping line and column addresses.

D. *Kernel Computation*

Before performing recursive filtering, two kernel matrices Vx and Vy must be computed from all three RGB channels.

1) Horizontal Difference: This module ("compute Vx" in Fig 7) calculates the difference between adjacent pixels in the original image by subtracting two adjacent columns. This contributes towards the detection of vertical edges in the image. The result of the difference is returned in the absolute form.

2) Vertical Difference: In a similar way ("compute Vy" in Fig 7) to the horizontal difference, the vertical difference of neighbouring pixels in rows is computed in parallel. This provides the horizontal edges of the image.

3) Channel Merging and Zero Padding: The differences of the RGB channels are computed at the same time. At this stage of the pipeline, they are added together. In the case of the horizontal differences, a column of zeroes is added before the first column of the image, and in the case of the vertical differences, a row of zeroes is padded at the top. This is to match the dimensions of these matrices to that of the original image. At this point the domain transform of the image has been calculated.

4) Derivative of the Domain Transform: Every element in both the horizontal and vertical differences matrices is multiplied by a factor, σ_s/σ_r, and then added to 1. This results in two new matrices, dHdx and dVdy. The mathematical constants σ_s and σ_r are precomputed.

5) Computing Vx: To implement this block in hardware, a multiplier module is used together with a natural exponential module. The following mathematical relation was used to compute the result.

$$a^b = \exp\left[b \cdot \ln(a)\right]$$

The output of this block gives us Vx and Vy, which completes the first stage of the pipeline. The computed values of Vx are now used to perform the recursive horizontal filtering.

E. *Recursive Filtering*

The recursive filtering process consists of two major operations. The first operation performs horizontal filtering and it consists of left to right followed by right to left filtering, as shown in Fig 9.

In the left to right operation, the algorithm starts operating on the second column, subtracting the previous column pixel to the current one, and multiplying the result with the value of Vx in the position of the current pixel.

The result is added back to the current pixel of the input frame to give the final result of the left to right filtering process. This result is used recursively to calculate the pixel values for the next column. Once the left to right filtered frame is computed, in a similar way the right to left frame is also recursively computed.

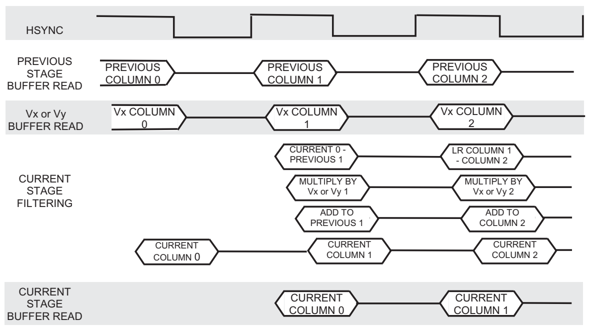


Fig. 10: Timing diagram that summarizes the data flow of the recursive filtering process. The hsync (horizontal synchronization) signal is used for timing synchronization of operations on columns or rows and vsync (vertical synchronization) signal to synchronize the different stages of the pipeline, namely left to right, right to left, top to bottom and bottom to top.

Resource				
ALUT	Registers	ALM	DSP18	BRAM(M9K)
29009	40891	29007	263	1094

TABLE III: Resource usage of domain transform implementation.

The second process is a vertical filtering operation which performs a process similar to horizontal filtering. In hardware implementation, the result of the horizontal filtering process is transposed and fed through the same horizontal filtering module, with the difference that the computations now use Vy instead of Vx.

F. Sequencing and Timing

Three control signals will be referred to in the following section; $hsync$ which is the horizontal synchronization signal, $vsync$, which is the vertical synchronization sistageand de, which is the input data enable signal.

The $hsync$ signal is used for timing synchronization of operations on columns or rows, and $vsync$ to synchronize the different stages of the pipeline. The left to right filtering process begins after a delay of 71 clock cycles from the rising edge of the de signal. This is the time that it takes to produce the first Vx result.

Taking the example of the left to right filter module, the subtraction is performed between the two input image columns $I1$ and $I0$. The result is produced in 7 clock cycles. The subtraction result is multiplied by the Vx values corresponding to the $I1$ column. This multiplication takes 5 clock cycles to complete. The result of the multiplication is added to the input image $I1$ to produce the result of the left to right filtering process. This result is stored in the buffer and made available to the computation of the next column at the next $hsync$, as well as to the next module, which is the right to left filter during the next $vsync$ signal. A generalized representation of the timing is shown in Fig. 10.

G. Domain Transform Performance

The latency observed in the Domain Transform implementation on an FPGA was 0.1143ms for a 128×128 pixel color image as compared to a 7ms latency reported by Gastal et. al for a 1 megapixel color image implemented on a GPU [21]. The resource ultilization of the implementation is listed in Table III.

V. Results

The compressed CCRF lookup table requires storage of all corner values per pair (f_1, f_2) for interpolation. Without any compression, this method would require as many as four times of the storage space due to redundancies of identical CCRF values shared between adjacent lookups. This is achieved by implementing a quadtree structure for leaf node searches that holds the address to the corners stored in the actual memory units. However, the compression is able to reduce the number of lookup entries by a factor greater than four times. This compression factor depends on the selection of α.

For the HDR compositing using the quadtree approach, the implementation features a $60\times$ compression of the memory requirements needed for the Comparametric Camera Response Function (CCRF) implementation, which itself achieves an approximately $5000\times$ speedup over other non-linear optimization based HDR imaging implementations [19].

The domain transform filtering implementation we present has been adapted for use on an low-power FPGA platform. In comparison to implementations on GPU, the FPGA implementation produces competitive results with added advantage of low power consumption and portability. This is ideal for implementations on wearable computers.

This is the first implementation of the domain transform on an FPGA platform, to our knowledge. The implementation presented in this paper is power efficient and can be used in portable embedded applications such as wearable computers. The domain transform can be used for [21] detail manipulation, tone mapping, stylization, filtering, colorization and recoloring of images or video frames in real time, depending on the selection of relevant parameters.

VI. Future Work

There are two areas in which work is presently on-going. One is in correctly pipelining the entire quadtree lookup on FPGA, so that many lookups may be “in-flight” at once, but that the circuit returns one value per clock cycle. This would enable higher throughput, or equivalently allow the circuit to be turned off for a longer period, per output frame. The other area is in adapting the domain transform to use external (DDR) memory, which would enable a substantially higher resolution to be processed.

VII. Conclusion

We have presented a highy efficient implementation of HDR (High Dynamic Range) imaging system suitable for implementation in low-power FPGA (Field Programmable Gate Array) chips that can be built into a seeing aid or visual memory aid such as the Digital Eye Glass, lifeglogging devices, or other sousveillance (inverse surveillance) technologies.

Working within the resource constraints of wearable platforms, we find that computationally- and memory-intensive algorithms can be implemented efficiently, typically by taking advantage of the recursive and self-similar nature of computer-vision algorithms. Thus, using current technology, we have successfully demonstrated how to implement algorithms that substantially enhance the human visual system, in a form that is practical for wearable applications.

References

[1] "Online etymology dictionary, douglas harper," 2010. [Online]. Available: http://dictionary.reference.com/browse/surveillance

[2] S. Mann, J. Nolan, and B. Wellman, "Sousveillance: Inventing and using wearable computing devices for data collection in surveillance environments." *Surveillance & Society*, vol. 1, no. 3, pp. 331–355, 2002.

[3] K. Michael and M. Michael, "Sousveillance and point of view technologies in law enforcement: An overview," 2012.

[4] J. Bradwell and K. Michael, "Security workshop brings' sousveillance'under the microscope," 2012.

[5] G. Fletcher, M. Griffiths, and M. Kutar, "A day in the digital life: a preliminary sousveillance study," *SSRN, http://papers.ssrn.com/sol3/papers.cfm?abstract_id=1923629*, September 7, 2011.

[6] C. Reynolds, "Negative sousveillance," *First International Conference of the International Association for Computing and Philosophy (IACAP11)*, pp. 306 – 309, July 4 - 6, 2011, Aarhus, Denmark.

[7] V. Bakir, *Sousveillance, media and strategic political communication: Iraq, USA, UK*. Continuum, 2010.

[8] F. M. Candocia, "A least squares approach for the joint domain and range registration of images," *IEEE ICASSP*, vol. IV, pp. 3237–3240, May 13-17 2002, avail. at http://iul.eng.fiu.edu/candocia/Publications/Publications.htm.

[9] ——, "Synthesizing a panoramic scene with a common exposure via the simultaneous registration of images," *FCRAR*, May 23-24 2002, avail. at http://iul.eng.fiu.edu/candocia/Publications/Publications.htm.

[10] A. Barros and F. M. Candocia, "Image registration in range using a constrained piecewise linear model," *IEEE ICASSP*, vol. IV, pp. 3345–3348, May 13-17 2002, avail. at http://iul.eng.fiu.edu/candocia/Publications/Publications.htm.

[11] M. A. Robertson, S. Borman, and R. L. Stevenson, "Estimation-theoretic approach to dynamic range enhancement using multiple exposures," *Journal of Electronic Imaging*, vol. 12, no. 2, pp. 219–228, 2003.

[12] C. Wyckoff, "An experimental extended response film," *SPIE Newslett*, pp. 16–20, 1962.

[13] S. Mann, "Compositing multiple pictures of the same scene," in *Proceedings of the 46th Annual IS&T Conference*, vol. 2, 1993.

[14] R. Lo, S. Mann, J. Huang, V. Rampersad, and T. Ai, "High dynamic range (hdr) video image processing for digital glass," in *Proceedings of the 20th ACM international conference on Multimedia*. ACM, 2012, pp. 1477–1480.

[15] M. D. Tocci, C. Kiser, N. Tocci, and P. Sen, "A Versatile HDR Video Production System," *ACM Transactions on Graphics (TOG) (Proceedings of SIGGRAPH 2011)*, vol. 30, no. 4, 2011.

[16] S. Kang, M. Uyttendaele, S. Winder, and R. Szeliski, "High dynamic range video," *ACM Transactions on Graphics*, vol. 22, no. 3, pp. 319–325, 2003.

[17] S. Mann, R. Lo, J. Huang, V. Rampersad, and R. Janzen, "Hdrchitecture: real-time stereoscopic hdr imaging for extreme dynamic range," in *ACM SIGGRAPH 2012 Emerging Technologies*. ACM, 2012, p. 11.

[18] S. Mann, R. Lo, K. Ovtcharov, S. Gu, D. Dai, C. Ngan, and T. Ai, "Realtime hdr (high dynamic range) video for eyetap wearable computers, fpga-based seeing aids, and electric eyeglasses," *image*, vol. 1, no. 2, pp. 3–4, 2012.

[19] M. A. Ali and S. Mann, "Comparametric image compositing: Computationally efficient high dynamic range imaging," in *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*. IEEE, 2012, pp. 913–916.

[20] S. Mann, *Intelligent Image Processing*. John Wiley and Sons, November 2 2001, iSBN: 0-471-40637-6.

[21] E. S. Gastal and M. M. Oliveira, "Domain transform for edge-aware image and video processing," *ACM Transactions on Graphics (TOG)*, vol. 30, no. 4, p. 69, 2011.

[22] J. Chen, S. Paris, and F. Durand, "Real-time edge-aware image processing with the bilateral grid," *ACM Trans. Graph.*, vol. 26, no. 3, Jul. 2007. [Online]. Available: http://doi.acm.org/10.1145/1276377.1276506

[23] Z. Farbman, R. Fattal, D. Lischinski, and R. Szeliski, "Edge-preserving decompositions for multi-scale tone and detail manipulation," in *ACM Transactions on Graphics (TOG)*, vol. 27, no. 3. ACM, 2008, p. 67.