

# Real-time hardware acceleration of the trace transform

Suhaib A. Fahmy · Christos-Savvas Bouganis ·  
Peter Y. K. Cheung · Wayne Luk

Received: 10 May 2007 / Accepted: 12 November 2007 / Published online: 24 November 2007  
© Springer-Verlag 2007

**Abstract** The trace transform is a novel algorithm that has been shown to be effective in a number of image recognition tasks. It is a generalisation of the Radon transform that has been widely used in image processing for decades. Its generality—allowing multiple functions to be used in the mapping—leads to an algorithm that can be tailored to specific applications. However, its computation complexity has been a barrier to its widespread adoption. By harnessing the heterogeneous resources on a modern FPGA, the algorithm is significantly accelerated. Here, a flexible system is developed that allows for a wide array of functionals to be computed without re-implementing the design. The system is fully scalable, such that the number and complexity of functionals does not affect the speed of the circuit. The heterogeneous resources of the FPGA platform are then used to develop a set of flexible functional blocks that can each implement a number of different mathematical functionals. The combined result of this design is a system that can compute the trace transform on a  $256 \times 256$  pixel image at 26 fps, enabling real-time processing of captured video frames.

**Keywords** Trace transform · Image processing · Field programmable gate arrays · Reconfigurable systems

---

This work was partially funded by the UK Research Council under the Basic Technology Research Programme (GR/R87642/02) and by the EPSRC Research Grant (EP/C549481/1).

---

S. A. Fahmy (✉) · C.-S. Bouganis · P. Y. K. Cheung · W. Luk  
Circuits and Systems Group, EEE Department,  
Imperial College London, Exhibition Road,  
London SW7 2BT, UK  
e-mail: s.fahmy99@imperial.ac.uk

## 1 Introduction

The extraction of information from images is a complex process. An image, in its pixels and intensities, serves no purpose until useful information can be deduced about it. This information may be the segmentation or even identification of objects, or in a temporally changing stream of images, some deduction of the movement or higher level behaviour of an object. One of the tools that is of great benefit in these tasks is the ability to transform an image from the spatial domain to an alternate domain, in which information is more easily extracted.

The Radon transform is a classical piece of image processing theory, introduced in 1917 and developed throughout the last century. Its primary utility is in identifying information on the internal properties of an object from a set of projections. These projections are obtained by summing pixel values along all lines crossing the image. Its main application has been in the field of computed tomography (CT), a primarily medical application in which an internal “image” of a patient can be obtained by computation from a set of X-ray profiles [4].

In 1998, Kadyrov and Petrou introduced the trace transform [8], a generalisation of the Radon transform, whereby any functional can be used, in place of the sum, to reduce a vector of pixel intensities along lines crossing the image to single values in the transform domain. Indeed, a typical implementation would include a number of functionals being used separately on the lines, yielding a “Trace” of the image for each functional. These Traces can be used for image analysis tasks, or through further steps, a set of features can be extracted from the image. Selection of appropriate functionals can give features that are invariant or measurably sensitive to standard transformations (e.g., translation, rotation and scaling) in the image

domain. The algorithm has been shown to perform very well in tasks such as image database search [9], token registration, activity monitoring and face authentication [16, 17] among others.

One of the difficulties in working with the algorithm has been its high computational complexity. The application of many functionals (some of which are intrinsically complex) to the multitude of lines crossing the image requires significant processing power. By harnessing the power of modern field programmable gate arrays (FPGAs), it is possible to design a hardware system that can implement the algorithm at high frame rates. This opens new possibilities in applying the trace transform to video as well as still images.

By exploiting the heterogeneous resources on modern FPGAs it is possible to create a flexible platform that can implement a wide array of different functionals. Furthermore, a hardware-accelerated platform as part of a combined software/hardware system allows for further experimentation with the algorithm. By allowing a host PC to take advantage of the flexibility in terms of functional selection, it is possible to use the system to search for functionals that are effective for a given task. Flexibility coupled with speed makes this platform an ideal springboard for further investigation of the Trace transform.

## 2 Introducing the trace transform

The trace transform of an image is a mapping from the spatial domain with Cartesian coordinates  $(x,y)$  to a domain with coordinates  $(\phi,p)$  where the value at each  $(\phi,p)$  point is equal to the result of applying a functional  $T$  to the image intensity function along a line that crosses the image tangential to angle  $\phi$  and at a distance  $p$  from the centre. Figure 1 shows this mapping. The flexibility of the transform comes from the freedom to use any functional to reduce the line to a single value in the  $(\phi,p)$  domain. Typically, a number of functionals are applied and thus a number of Trace images are produced.

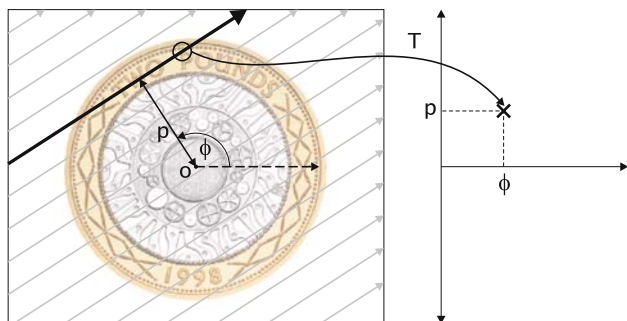


Fig. 1 Mapping of an image to the new space

While the transform is defined in the continuous domain, its discrete implementation is a trivial development. Parameters  $\phi$  and  $p$  are sampled discretely. Considering  $\phi$ , it is clear that lines crossing the image can be at any angle from  $0^\circ$  through  $360^\circ$ . It is, in fact, possible to only use angles up to  $180^\circ$ , since a line with angle  $\theta$  and distance  $x$  could be represented by the parameters  $180 - \theta$  and  $-x$ . However, it is important to note that some functionals take into account the direction of the line, and hence this simplification cannot be applied generally. For functionals that do not use the line direction, the Trace is effectively odd-symmetrical, due to the aforementioned property. A line in the discrete domain is just an array of values that represent the image intensity at each of the sampled points.

The functionals used to reduce these arrays to single values can be arbitrarily defined. One might choose some variation of sum of differences, mode, median, maximum difference, root mean squared, etc. For different applications, different functionals may exhibit strengths that are not apparent in others. Furthermore, functionals may vary significantly in their computational complexity.

A further extension of the transform involves the extraction of “Triple features”. This is done by applying a further functional called the diametrical functional,  $P(p)$ , to the each of the  $p$  vectors in the resultant trace. Finally a “circus functional”,  $\Phi(\phi)$ , is applied to the resultant vector, yielding a single triple feature. This process is shown in Fig. 2. Applying a number of functionals at each stage gives a number of triple features. If  $n_T$  functionals are used for the trace functional ( $T$ ),  $n_P$  functionals are used for the diametrical functional ( $P$ ) and  $n_\Phi$  functionals are used for the circus functional ( $\Phi$ ), then a total of  $n_T n_P n_\Phi$  triple features can be extracted. Appropriate selection of functionals at each stage can result in these features being invariant [10, 13] or measurably sensitive [11] to spatial transformations.

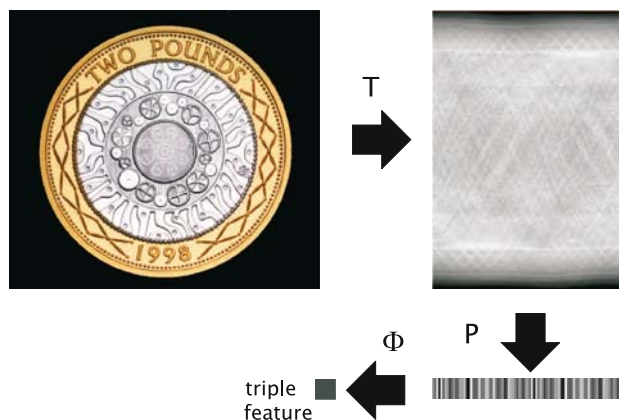


Fig. 2 An image, its trace, and the subsequent steps of feature extraction

### 2.1 Computational complexity

The main issue with the transform thus far has been that of computational complexity. To investigate this, it is first necessary to understand the parameters that can be controlled in an implementation. We will concern ourselves here with only the first step of the Trace transform, the mapping from the spatial domain to the Trace domain, since that is the most computationally intensive step.

Firstly, a fixed sampling density of angles can be considered,  $n_\phi$ ; one could consider angles down to a  $1^\circ$  accuracy or lower, or alternatively choose a coarser sampling. The amount of information carried over to the Trace domain clearly depends on this parameter. Secondly, it is possible to sample an arbitrary number of lines,  $n_p$ , per angle, again this has a bearing on the accuracy of the transformation. For an  $N \times N$  image, we might require an inter-line distance of a single pixel, so  $n_p$  would have a maximum value of  $\sqrt{2}N$  (for a diagonal line). Since each line maps to a single point in the  $(\phi, p)$  domain, an image transformed with the above parameters will yield a trace of size  $n_\phi \times n_p$ . Another parameter that can be varied is the sampling granularity along each line, or more intuitively, the number of samples taken into account for each line,  $n_t$ . If the requirement is to read every pixel along each line, then the maximum value of  $n_t$  is also  $\sqrt{2}N$ . This will not affect the density of data in the parameter domain but will affect the accuracy of the results of applying the functional to the line. Other parameters used in this section are shown in Table 1.

There are two main steps in computing the trace transform. The first is to extract the necessary pixels from the source image, given values  $(\phi, p)$  and the second is the computation of the Trace results. For each of  $n_\phi$  angles and  $n_p$  lines per angle, let  $C_\phi$  represent the number of operations required to compute the pixel addresses for a line. To compute each of  $N_T$  functional results,  $n_t$  points on each of the  $n_p$  lines for each of the  $n_\phi$  angles must be processed. If

$C_T$  denotes the number of operations required on average per pixel per functional, then a total of  $n_\phi n_p n_t N_T C_T$  operations are required to compute the traces for an image, while  $n_\phi n_p C_\phi$  operations are required for the line extraction.

For an  $N \times N$  image, these parameters may take values as follows:  $n_\phi = 180$ ,  $n_p \simeq N$  and  $n_t \simeq N$ . This would give a computation complexity of  $180N C_\phi + 180N^2 n_t C_T$ . The number of functionals,  $n_T$ , may be 8–10 in an implementation, and so the high computational cost is apparent. The amount of time taken for  $C_\phi$  and  $C_T$  would depend on the implementation; in hardware, it might be possible to do these operations more efficiently than in software. Furthermore, by parallelising in the number of angles ( $n_\phi$ ), the number of lines ( $n_p$ ) and the number of functionals ( $n_T$ ), the total run time can be reduced. The key to accelerating any algorithm in hardware is to identify the inherent parallelism and then exploit it in the design. By harnessing this parallelism, the algorithm could well be accelerated to run at the speeds required for real-time video.

### 3 Related work

The work presented in this paper and in [5] is the first to deal with hardware implementation of the trace transform. The most closely related work is that which deals with the Radon transform and so that is what will be considered here.

In [15], a system based on four parallel DSP processors for computation of the Radon and Inverse Radon Transform is presented. The parallelism of angles is exploited to increase performance. Different interpolation techniques are compared, and while Linear Interpolation is shown to be slower than nearest neighbour, it is chosen due to the increase in quality.

In [1], the authors use progressively larger line segments to approximate the line sums, thus significantly reducing processing time. The authors of [7] further develop this algorithm, presenting a hardware implementation that can process  $21\,512 \times 512$  pixel frames per second.

In [14], the presented implementation first maps an image to the Polar coordinate system, then uses this to transform it to the Radon domain. The system only deals with binary images. The authors also suggest parallelisation in the angles.

In [12], the authors make use of the Radon transform’s relationship with the Fourier transform. Using efficient implementations of the FFT and IFFT, they are able to accelerate computation of the Radon and inverse Radon transforms.

Finally, in [3], the authors present two architectures for the acceleration of the Finite Radon Transform. They

**Table 1** Trace transform computational parameters

Parameter	Explanation
$n_\phi$	The number of angles to consider
$n_p$	The number of distances (inverse of the interline distance)
$n_t$	The number of points to consider along each trace line
$N_T$	The number of Trace functionals
$N_P$	The number of diametrical functionals
$N_\Phi$	The number of circus functionals
$C_\phi$	The operations required for trace line address generation
$C_T$	The average operations per sample for each $T$ functional
$C_P$	The average operations per sample for each $P$ functional
$C_\Phi$	The average operations per sample for each $\Phi$ functional

mention the clear distinction between the Finite Radon Transform and the Discrete Radon Transform. The theory is thus distinct.

There are a few important notes to be mentioned, that preclude much of this previous work from being useful in regard to the trace transform. Firstly, Radon transform implementations assume the function to be applied to each line is a sum. For the trace transform, this is clearly not the case, so ideas such as partial results and the summing of line segments (as in [7]) cannot be applied. The work in [14] actually links closely to the idea of using rotations instead of line extractions that will be shown in the next section. Furthermore, the trace transform does not retain the mathematical properties of the Radon transform nor its relationship to other transforms, hence work such as that in [12] cannot be adapted for a trace transform architecture.

The presented architecture builds from the ground up, introducing parallelism wherever possible in order to allow speed-up. Further simplifications are made, but with care not to make any assumptions about the types of functionals to be considered. This paper builds on the architecture developed in [5] with the introduction of flexible functional blocks which enable this architecture to be used in functional exploration for trace transform applications.

## 4 Hardware implementation of the trace transform

### 4.1 Target platform

The target platform for this implementation is the relatively mature Celoxica RC300 Development board [2]. This board hosts a Xilinx Virtex II XC2V6000 FPGA, alongside a vast array of peripherals. The only other components that concern us are the on-board pipelined ZBT SRAMs—there are four 8 MB banks—and the USB connection to a host computer. The RAMs can be accessed in pipelined mode, accepting a single read or write instruction per cycle and are 36 bits wide. The FPGA has a large logic capacity as well as providing 144 hard-wired multipliers and 144 18 Kb BlockRAMs. These are small on-chip RAMs that can be used as buffers and to optimise data flow in the design. The system was designed and implemented using Celoxica Handel-C, a high-level C-syntax-based hardware description language. It is important to note here, that while it is possible to write mostly standard C and compile it to an FPGA design, this gives relatively little performance improvement over software. The key to exploiting the full power of the FPGA is to write the Handel-C code in a manner that suits the hardware implementation.

### 4.2 System framework

Before discussing the details of the hardware implementation, it is necessary to look at the overall system in its constituent parts. A host PC captures image data by way of a standard USB camera, at a rate of 25 frames per second. The camera outputs frames of size  $640 \times 480$  pixels, the central  $256 \times 256$  pixels are then extracted in software. The target is to process these frames in real-time, so the processing rate must meet this minimum requirement. The image data is pre-processed, including resizing and conversion to greyscale, before being sent to the development board via a USB interface. The image data is stored in one of the on-board SRAMs before any processing occurs. As each frame becomes available in the SRAMs, the FPGA reads the frame and computes the results. These results are stored in another SRAM, from which the result is transmitted back to the PC, again via the USB interface. On the PC, the data is reorganised and used in the subsequent processing steps of a higher level system. Hence the crux of the hardware implementation deals with the data between the input and result RAMs. (The FPGA is also used to control the communication with the PC as well as the data reading and storage.)

The system is composed of a number of blocks, shown in Fig. 3. A Top-Level Control block oversees the communication between separate blocks and ensures synchronisation. The Rotation Block reads an input image from the on-board RAM and produces a rotated copy at its output. Each Functional Block takes the pixels in this rotated image and uses them to compute the relevant results for each line crossing the image at that angle. This is the first example of parallelism in the design; these functional blocks work in parallel thus producing all their results in the time it would normally take to produce a single functional result. Further parallelism will be examined in subsequent sections. Finally, an Aggregator Block reads the results from each functional in turn and outputs them serially to the result RAM.

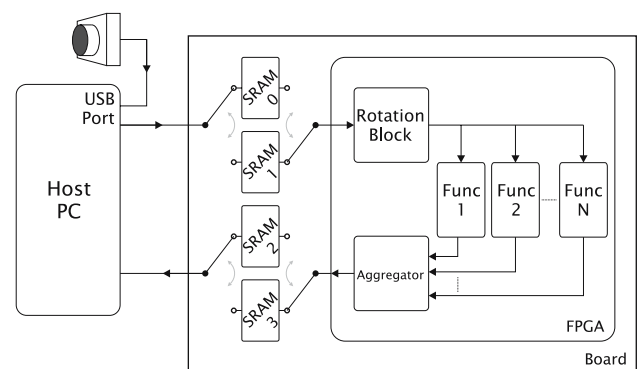


Fig. 3 Architecture overview

The host PC sends the image data, frame by frame to the image RAM on the board. The FPGA reads from the USB port and selects a RAM to write the image to. The RAMs are double-buffered to increase performance. This means that while an image is being loaded into one RAM, the other RAM in the pair is being used for calculation. When one calculation cycle is completed, the roles of the RAMs are swapped. Since loading the input memory and reading the result memory takes less time than the processing, the system is constantly busy with computation, with no need to wait for data loading.

The rotation block produces rotated versions of the original image with angles increasing by  $2^\circ$  per iteration. This increment can be modified as required for an implementation. The reason for selecting this value, was to attempt to retain a similar amount of information in the trace domain representation as in the image domain. This results in a mapping of a  $256t \times 256$  pixel image to  $256 \times 180$  points in the transform domain.

The functional blocks read the resultant stream, keeping track of the row beginnings and endings, and passes the results for each row to the Aggregator. Each row corresponds to a single line across the original image. Once the calculation of all lines for all rotations is complete the host PC reads from the result RAM, while the system writes the next set of results to the other output RAM. The results can then be extracted and organised on the host PC for further processing.

### 4.3 Top level control

The Top-Level Control block oversees all the other blocks. It initiates the rotations and ensures that each new rotation is synchronised with the completion of results calculation for the previous rotation. It also manages the double-buffering of the external RAMs.

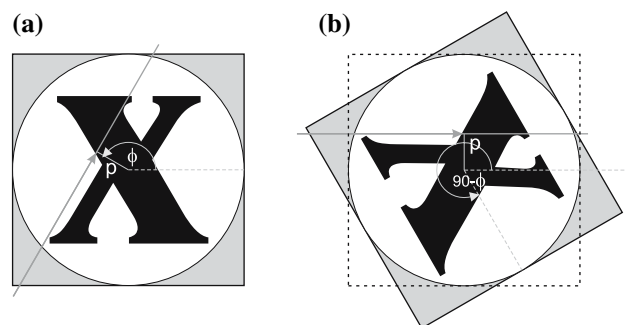
### 4.4 Rotation block

Conceptually, the first step in the algorithm is a line tracer: a block that takes a  $(\phi, p)$  value and produces the coordinates of the relevant pixels in the source image. These coordinates are then used to extract pixel intensity values from the source image, stored in off-chip RAM. It is worth noting, however, that to compute a trace, all values of  $(\phi, p)$  must be used, hence, the whole image is traced for all angles and at all distances from the centre. Bearing this in mind, a simplification can be made that makes for a more efficient implementation. Rather than iterate over values of  $\phi$  and  $p$  then mapping them to the Cartesian domain, it is possible to rotate the whole image by an angle  $\theta$  and

sample pixel values along the horizontal rows in that rotated version. This would be equivalent to computing all the trace results for a fixed value of  $\phi = 90 - \theta$ . This equivalence is illustrated in Fig. 4.

It is necessary here to mention an important caveat. When rotating an image, some parts of the image fall outside the original size limits of the source image. The rotated version must either be larger than the source image, to accommodate this extra information, or the information must be lost. An equivalent amount of empty canvas is also introduced from areas “underneath” the image that become exposed. If a square image of size  $N \times N$  is rotated through all angles from zero through  $360^\circ$ , then only the portions of the image that fall within a concentric circle with diameter  $N$  would be present in all possible rotations. This is also shown in Fig. 4. Here, it is worth mentioning that the trace transform has only been shown to perform well with masked images in the presence of noise [9]; since the lines that trace the image may include part of the background too, a lack of masking would allow the background to contribute significantly to the functional results. Due to this fact, it is necessary for the object of interest to be masked; that is, that a binary overlay be present that determines whether or not the corresponding pixel in the image is used in calculations. It is thus a fair assumption for this system to require any object to fall within the aforementioned area and to be masked appropriately.

To understand why this modification simplifies the system, consider first the initial approach. A block would be required that takes a  $(\phi, p)$  value and outputs a vector of addresses. To do this, each line must have a starting point (which must be computed), that may well fall outside the image coordinates. A counter must then be incremented for both image axes and coordinates that fall outside the image must be discarded. The lines will vary in length, and so, some way of tracking the position of the perpendicular to the centre is needed. Furthermore, some way of tracking



**Fig. 4** Rotating an image then reading across its rows can replace the address generation required for extracting line pixels. The *line* shown in **a** is equivalent to the row of the rotated image in **b**. The *shaded* area shows the part of the image that will fall out of the image frame for some rotations, and so must not contain an area of interest

the correct  $(\phi, p)$  values for each line is required, since each line is extracted independently. Due to the variation in line lengths, there is the added problem of reading from the image RAM inefficiently. Alignment of vectors to take account for the gaps in between readings would also be necessary.

Now consider the approach where the whole image is rotated. Each rotation produces a set of all  $p$  values for the given rotation angle (note the offset mentioned above). Since the resultant rotated image is produced in raster scan format, there are no gaps, and the vectors are all aligned. This means that no further logic is required before reading from the source RAM. Dealing with a fixed line length of  $N$  also simplifies tracking of the  $p$  values in the functional blocks, since this is simply the row number in the rotated image (again there is a fixed offset). Once a rotation has been completed, the source image is again rotated by a new angle, and this produces the vectors for another value of  $\phi$  and so on. This technique simplifies the system significantly and allows for full pipelining of the architecture.

The Rotation Block thus takes an angle as its input and produces the raster scan of the source image rotated by that angle at its output. The source image is read out of order, and since the output is in order, there is no need for image buffers or other signals, since all addressing is inherent in the data.

So far, this modification has dealt with data handling. To fully harness the power of hardware implementation, it is also worthwhile to look at exploiting parallelism in the algorithm. Since results for one angle,  $\phi$ , are in no way computationally related to other values of  $\phi$ , the algorithm could be said to be independent in  $\phi$ . Hence a number of parallel calculations could be computed, equal to the number of angles considered. It is, however, necessary to consider data-path limitations. For each rotation that occurs in parallel, separate accesses would be required to the source image, due to the out-of-order access imposed by the design specified above. Hence we would require multiple copies of the image in separate RAMs, since the board RAMs on our development board only provide single-port access to only four banks, and this is unfeasible.

There is, however, another way of multiplying performance without sacrificing much area. Consider that any rotation by a multiple of  $90^\circ$  is simply a rearrangement of data (or alternatively a reassigning of axes values); this is easily implemented in software. It is also clear that a rotation by any (positive) angle,  $\theta$ , is equivalent to a rotation by some multiple of  $90^\circ$  plus the remaining angle. Formally:

$$\theta = \lfloor \theta/90 \rfloor \times 90 + \theta \bmod 90. \quad (1)$$

As an example, rotation by  $212^\circ$  is equivalent to rotation by  $180^\circ$  followed by a further rotation by  $32^\circ$ .

This fact can be exploited in order to parallelise rotations as follows. The source image is an 8-bit greyscale image. The external board RAMs are 36-bits wide, and hence, storing a single image is a waste of the word widths. Instead, what can be done is to store the four orthogonal base rotations ( $0^\circ$ ,  $90^\circ$ ,  $180^\circ$  and  $270^\circ$ ) concatenated in a single RAM word. Since the host computer can easily construct the other three rotations from a standard image, there is no real computational cost to be considered. Table 2 shows that any orthogonal rotation can be obtained with no more than simple calculations that can be performed extremely quickly on a host PC. Now when a rotation by angle  $\theta$  is carried out, the RAM word that is output can be spliced to give the relevant pixel for the rotations by  $\theta$ ,  $\theta + 90^\circ$ ,  $\theta + 180^\circ$  and  $\theta + 270^\circ$ . This effectively quadruples performance with only minimal impact on area. (The area impact is only as a result of increasing the size of the registers.)

The construction of the orthogonal rotations occurs on the host PC, and only once per frame. While this takes  $4 \times N^2$  reads on the PC (ignoring the effects of caching), the resultant concatenated image is rotated 45 times instead of the 180 that would be needed for a standard image, in order to construct a full trace. This saves over 9 million cycles per trace, at a cost of  $3 \times N^2$  extra cycles on the PC (for which cycle times are much shorter). Hence the overhead is minimal.

Since all images are also masked, the four mask rotations are also stored in the RAM. With four 8-bit image words and four 1-bit masks, the total wordlength is 36-bits which matches the RAM perfectly. The makeup of a single RAM word is shown in Fig. 5. Loading an image onto the board over USB requires the data to be sent in single bytes, this means a  $256 \times 256$  pixel image takes  $256 \times 256 \times 5 = 327,860$  cycles to be transferred from the PC to the board RAM.

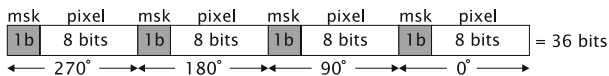
In order to compute the rotations, the system proceeds as follows. The input angle is used to address sine and cosine lookup tables (stored in BlockRAMs). The resultant values are then used to compute the standard Cartesian rotation equations:

$$x' = x \cos \theta - y \sin \theta \quad (2)$$

$$y' = x \sin \theta + y \cos \theta. \quad (3)$$

**Table 2** Base orthogonal rotation coordinates

Rotation	$x$	$y$
$0^\circ$	$x$	$y$
$90^\circ$	$y$	$N - x$
$180^\circ$	$N - x$	$N - y$
$270^\circ$	$N - y$	$x$



**Fig. 5** Structure of a single word in external RAM

The  $x$  and  $y$  coordinates obtained from this computation are used to read specific pixels from the source image in off-chip RAM. Recall that these source pixels are in fact a set of four concatenated orthogonal base rotations. When one of these concatenated “pixels” is read, it is spliced into its four separate parts and each of those is used to build a separate rotated image. This means that in  $N^2$  cycles, four separate rotations are completed.

The calculations are carried out by using an 8-bit fixed-point calculation. Nearest neighbour approximation is used whereby each sample point takes the value of its nearest pixel; this avoids the more complex circuitry and scheduling required for bilinear or bicubic approximation. The result is that a complete rotation of an  $N \times N$  image is complete in  $N^2$  clock cycles.

The resultant rotated images stream through the system in raster scan format. This is where each row is transmitted, one pixel at a time, followed by the next row and so on. This makes the subsequent blocks simpler since there is no complex buffering or ordering to be considered.

Most Radon transform implementations use sub-pixel interpolation; this is done primarily to preserve accuracy in order to allow the original image to be obtained through the inverse transform. In the case of the trace transform, there is no defined inverse, rather the transform is used to map an image to an alternate domain. While the point in the transform domain to which a pixel is mapped will depend on the type of interpolation used, consistency between different images is of greatest importance. Furthermore, subsequent stages in trace transform applications thus far throw away much of the transformed data, instead using it to construct a simpler representation. As an example, consider the face authentication application in [17]; the trace image is thresholded and the outline converted to a shape. In such a situation, fine numerical accuracy is of little bearing. Furthermore, when used for a recognition task, it is effectively a closed system, where both reference and candidate images will be processed in an identical fashion. Further accuracy discussions are presented in Sect. 5.5.

#### 4.5 Functional blocks

The computation of results for the trace transform occurs in the functional blocks. Each block must process the output of the rotation block, computing the results of applying the appropriate calculation to each row of the input, and then

pass this result on to be stored in the result RAM. A few aspects of the implementation should be noted here. Firstly, the input to the functional blocks is the raster scan of the rotated image. This means that a row of  $N$  pixels, clocked at one pixel per cycle, is followed immediately by the  $N$  pixels of the subsequent row. To assist in keeping the data aligned, a “new row” signal is also passed by the rotation block. Each rotation is processed independently, so there is a short gap in processing between subsequent rotations. This is needed to allow for functional blocks with different latencies to process the correct data at the same time.

A second point of importance is to recall that the output of the rotation block is four rotations rather than one. Each functional block therefore splices these data and computes results for each of the four rotations independently. This means that all the computation datapaths and registers are duplicated four times. The control circuitry is kept combined for compactness and synchronisation purposes.

Finally, it is important that each functional block makes use of the mask associated with each pixel to decide whether it is used in the calculation. Recall that masked-out pixels are ignored in the Trace transform.

When the results for each row are ready, they are stored in an output buffer and a “result ready” signal is passed to the Aggregator. Since the board RAMs have a 36-bit wordlength, the widths of the data paths are tailored to ensure that the final result fits within this limit.

In order to take advantage of the flexibility afforded by programmable logic, the functionals should be designed such that they are reprogrammable in some way. Since many functionals that might be used share a common basic structure, it is possible to create a single functional that can, with some configuration, compute multiple possible functional results. In order to afford this flexibility, on-chip BlockRAMs are used as lookup tables for the basic arithmetic functions within each block. This allows for alternative functions to be defined. Furthermore, a configuration register is optionally available to select from multiple possible datapaths within each block. This is discussed in detail in Sect. 5.

#### 4.6 Aggregator

The Aggregator polls the functionals in a round-robin fashion awaiting a “new result” signal. It is preferable that the polling is ordered such that the functional that finishes first is the first polled. This would provide the most efficient use of the storage time window. When received, it proceeds to store the four results from the current functional in a serial manner. This is done to avoid having a large data bus between each of the functionals and the aggregator. Since there is only a new result every  $N$  cycles

(256 in this implementation), there is sufficient time to read each result from each functional in series. The results are stored in an on-board RAM addressed using a concatenation of functional number, rotation and row number. The contents of this RAM can then be read by the host PC over USB and the results used in further stages of processing.

#### 4.7 Initialisation

Before the system can commence operation, all signals need to be initialised and parameters, such as the number of angles and lines to consider, set. The major step in the initialisation step is to define the configuration of each of the functional blocks. As mentioned in the previous section, the functional blocks are designed to be flexible and thus use a configuration register to select a specific datapath and BlockRAMs to define arithmetic operations. A typical functional block may contain two or three lookup RAMs; these are used to implement functions such as  $x$ ,  $x^2$ ,  $\cos(x)$  and so on. For each Functional Block, the lookup RAMs must be pre-loaded with the relevant values to enable the correct functional to be computed. The configuration register must also be set.

Since there may be 8–10 functionals on the chip at one time, a large number of RAMs must be initialised and doing so from one central location would thus be inefficient since signals would have to be routed to more than 80 locations (since there are effectively four functionals in each block) from the initialisation block. Rather, the system reads the initialisation values from the PC via USB and stores them in one of the board RAMs. From there, the data is streamed by a distributor onto an initialisation bus. This bus is accessed by a dedicated unit beside each Functional Block that only acts upon the instructions relevant to its own functional.

The initialisation process only needs to occur at the beginning of a processing run. The time taken depends on the number of Block RAMs that need to be initialised with each requiring 514 cycles to complete this step. At 80 MHz, this translates to just over 640 ns per Block RAM.

### 5 Flexible functional blocks

An initial system, complete with three basic fixed functional blocks was implemented as a proof of concept and detailed in [5]. It computed functionals 1, 2 and 4 from Table 3, processing captured video in real-time at 26 fps for  $256 \times 256$  pixel frames. However, the scope of a hardware implementation is more significant than just a fixed implementation. Using Programmable Logic for the

implementation of the trace transform carries with it a further benefit aside from hardware acceleration: flexibility. By exploiting some of the heterogeneous resources provided on today's FPGAs, an architecture can be designed. In this section, we present a much more flexible set of basic functional blocks that leverage the FPGA's flexibility in order to allow for a wide range of different functionals to be implemented.

Modern FPGAs afford the designer a plethora of heterogeneous resources that can be employed in a hardware design. Hard-wired multipliers that can compute a wide multiplication in a single cycle are ideal for many Signal Processing applications; specialised DSP blocks are completely tailored to this domain. There are also small on-chip memories which serve as ideal resources for use in buffers, and for managing data flow and complex computations. On the Xilinx Virtex II chip used here, there are  $18 \times 18$ -bit multipliers and 18Kb Block-RAMs. These RAMs are dual-ported and so can be read twice in a single cycle and can be set-up in multiple width and depth arrangements. Some of their common uses include hashing, FIFOs, buffers, lookup tables and more.

It is worth briefly mentioning the sorts of functionals that have been used in trace transform implementations to date. In [16, 17] the authors design a face authentication system, based on the trace transform. They use the trace transform for authentication in two ways. Firstly, the traces constructed are directly compared to each other using the weighted trace transform (WTT); this, however, shows mediocre performance. The second method is the shape trace transform (STT), where the resultant traces are threshold and the resultant shapes compared rather than the traces themselves. This method proved much more accurate. The functionals used for both are shown in Table 3. These were selected due to pre-existent work that showed their performance in texture classification. Of the 22 functionals listed, numbers 1, 2, 7, 9, 11, 12, 13, 14, 20, 21 and 22 were found to be useful for the STT.

By using the heterogeneous resources available on a modern FPGA, it is possible to design a small number of flexible functional blocks that can compute the majority of functionals in Table 3, in which similar ones have been grouped together. By designing a functional block that can, after reconfiguration, compute any member of a group, or further possibilities, flexibility can be afforded to the system. Thus, a few generalised blocks can allow the implementation of a whole range of functionals, saving on area, and providing a wider array of functional possibilities.

The resources that most assist in this endeavour are the on-chip BlockRAMs, used as lookup tables. A lookup RAM contains pre-computed values for some function stored at each location. For example, for a lookup RAM to



**Table 3** The trace functionals  $T$  [17]

No.	Functional	Details
1	$T(f(t)) = \int_0^\infty f(t)dt$	Radon transform
2	$T(f(t)) = \left[ \int_0^\infty  f(t) ^{\frac{1}{2}} dt \right]^2$	
3	$T(f(t)) = \left[ \int_0^\infty  f(t) ^4 dt \right]^{\frac{1}{4}}$	
4	$T(f(t)) = \int_0^\infty  f'(t)  dt$	$f'(t) = (t_2 - t_1), (t_3 - t_2), \dots, (t_n - t_{n-1})$
5	$T(f(t)) = \text{median}_l\{f(t),  f(t) \}$	Weighted median
6	$T(f(t)) = \text{median}_l\{f(t),  f'(t) \}$	
7	$T(f(t)) = \left[ \int_0^{n/2}  \mathcal{F}\{f(t)\}(t) ^4 dt \right]^{\frac{1}{4}}$	$\mathcal{F}$ means taking the Discrete Fourier Transform
8	$T(f(t)) = \int_0^\infty \left  \frac{d}{dt} \mathcal{M}\{f(t)\} \right  dt$	$\mathcal{M}$ means taking the Median over a length 3 window, and $\frac{d}{dt}$ means taking the difference of successive samples
9	$T(f(t)) = \int_0^\infty rf(t)dt$	$r = ll - cl; l = 1, 2, \dots, n; c = \text{median}_l\{l, f(t)\}$
10	$T(f(t)) = \text{median}_{t^*} \left\{ \sqrt{rf(t)},  f'(t) ^{\frac{1}{2}} \right\}$	
11	$T(f(t)) = \int_0^\infty r^2 f(t) dt$	
12	$T(f(t)) = \int_{c^*}^\infty \sqrt{rf(t)} dt$	$c^*$ signifies the nearest integer to $c$
13	$T(f(t)) = \int_{c^*}^\infty rf(t) dt$	
14	$T(f(t)) = \int_{c^*}^\infty r^2 f(t) dt$	
15	$T(f(t)) = \text{median}_{t^*} \left\{ f(t^*),  f(t^*) ^{\frac{1}{2}} \right\}$	$f(t^*) = \{f(t_{c^*}), f(t_{c^*+1}), \dots, f(t_n)\}$
16	$T(f(t)) = \text{median}_{t^*} \left\{ rf(t^*),  f(t^*) ^{\frac{1}{2}} \right\}$	$l = c^*, c^* + 1, \dots, n; c = \text{median}_l\{l,  f(t) ^{\frac{1}{2}}\}$
17	$T(f(t)) = \left  \int_{c^*}^\infty e^{i4 \log(r)} \sqrt{rf(t)} dt \right $	
18	$T(f(t)) = \left  \int_{c^*}^\infty e^{i3 \log(r)} f(t) dt \right $	
19	$T(f(t)) = \left  \int_{c^*}^\infty e^{i5 \log(r)} rf(t) dt \right $	
20	$T(f(t)) = \int_c^\infty \sqrt{rf(t)} dt$	$r = ll - cl; l = 1, 2, \dots, n;$
21	$T(f(t)) = \int_c^\infty rf(t) dt$	$c = \frac{1}{S} \int_0^\infty l f(t)  dt; S = \int_0^\infty  f(t)  dt$
22	$T(f(t)) = \int_c^\infty r^2 f(t) dt$	

be used in computing  $\cos(x)$ , the contents of each memory location would have to contain the value of  $\cos(addr)$ , where  $addr$  is the address of the RAM word. Thus when value  $a$  is applied to the address input of the lookup,  $\cos(a)$  emerges at the data output. Note that the RAMs are in fact used as ROMs in this situation, however, they are still referred to as RAMs because they retain their write capability. This write capability is what allows us to exploit them for adding flexibility. Since these RAMs can be configured with data at runtime, there is no need to re-synthesise the design in order to change the RAM contents, and thus the function computed by each lookup.

Now consider that a wide range of functions can be implemented in this fashion, and it becomes clear that a functional block with lookup RAMs incorporated can implement a wide array of different functionals. Some of the arithmetic functions that can be computed in this manner include  $x, x^2, \sqrt{x}, \ln(x), \sin(x)$ , to name but a few. The only limitation is that the input value must be bound since the depth of the RAM must be predetermined. A small increase in the range of an input can impact the resources used dramatically. The on-chip BlockRAMs on a Xilinx Virtex II are 18 Kb in size, and can be configured in a number of ways, between  $512 \times 36$ -bits to  $16 \text{ K} \times 1$ -bit

[18]. The specific configuration must be set in advance. Fortunately, for the purposes of this system, the input pixels are adequately represented in 8-bits. As a result, and to simplify the datapaths, the BlockRAMs are set up to store 16-bit wide lookup values. This provides sufficient range for some of the larger numbers but also good fixed-point accuracy for the smaller numbers.

In the following sections, three different generalised functional blocks are presented. Each of them can implement a number of functionals listed in Table 3 as well as some others obtained by changing the lookup functions. Block diagrams are shown for each type, though for simplification, only a single datapath is shown whereas, as mentioned previously, each functional block actually processes four rotations in parallel. The signal wordlengths are shown on the connections. Each lookup table is implemented on a single Block RAM. In reality, since four rotations are needed, four Block RAMs would be required for each lookup. However, given that the BlockRAMs on the target FPGA can be set up to be dual-ported, only two, are in fact needed per lookup table. This is why the hardware resource usage results, shown in Table 4, show four times the number of multipliers and double the number of Block RAMs, when compared to the diagrams.

**Table 4** Synthesis results

Unit	Slices	Multipliers	BlockRAMs
Framework	1,300	4	2
Type A functional	800	4	6
Type B functional	17,500	4	22
Type C functional	1,300	8	6
Total available	33,792	144	144

Note that in the figures, each circuit element takes a single cycle to run and that the dashed parts are optional, determined by the configuration register. “D” is a single cycle delay register. Intermediate registers have been omitted for clarity.

### 5.1 Type A functional block

This functional block is able to compute functionals 1, 2 and 4 from the original list of 22. A block diagram is shown in Fig. 6.

The block takes an input pixel then applies a function,  $l_1$  to it. Optionally, function  $l_2$  is applied to a one cycle delayed version of the input pixel and the absolute difference is taken; the actual datapath is decided by the configuration register. Each of the resultant values is then summed and at the end of the row, the final result is optionally squared.

### 5.2 Type B functional block

This functional block is more complex than Type A; it implements functionals 9, 11, 12, 13 and 14 from Table 3, which depend on the weighted median calculation. The weighted median is implemented using an efficient hardware block that can compute medians for very large windows using a cumulative histogram [6]. Since the median block can only return a result after a whole row has been processed, the current implementation uses the result from the previous row in each calculation. Lookup  $l_1$  offers

flexibility in modifying the skew of the median calculation with the default being  $\sqrt{}$ . It is clear from the implementation results that this functional block uses significant resources. The reason for this is that each of the four angles must have its own median calculation circuit, and each of these is quite resource intensive.

The functional computes the intermediate values  $c$  and  $r$  as described in row nine of Table 3. Note that the value  $c$  is only updated once per row (shown shaded in the figure). Function  $l_2$  is then applied to  $r$  and  $l_3$  to  $f(t)$ . These results are then multiplied before being summed in an accumulator (Fig. 7).

### 5.3 Type C functional block

This functional block implements functionals 20, 21 and 22 of the initial list. The system follows similar design to the Type B block, except that  $c$  is computed as described in row 21 of Table 3. Note that the values  $c$  and  $S$  are only updated once per row (Fig. 8).

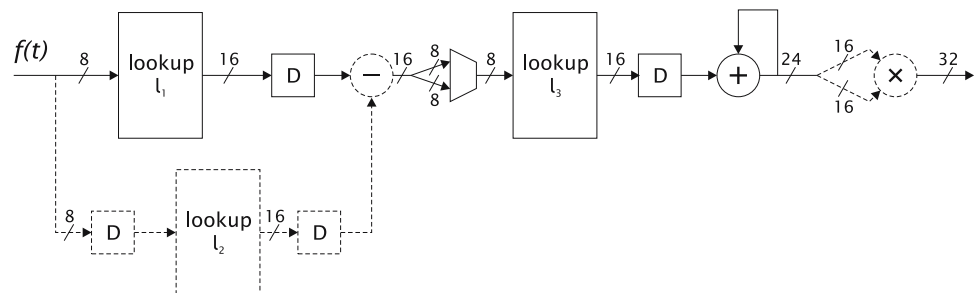
### 5.4 Functional coverage

These three functional types cover 10 of the 11 functionals required by the STT for face authentication [17]. Furthermore, by using alternative look-up functions in the BlockRAMs, it is possible to compute additional functionals.

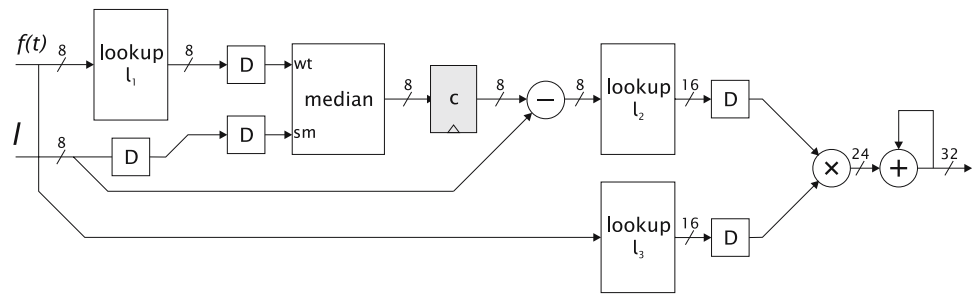
### 5.5 Accuracy considerations

Before discussing the accuracy of the architecture, it is worth reiterating what was mentioned in Sect. 4.4. The trace transform is used within a closed system, primarily for recognition tasks. Furthermore, the transform is not used as an encoding tool, nor is its inversion defined, hence, its accuracy need only be considered as much as is required by subsequent blocks in the specific application. Therefore, fine numerical accuracy is of a lesser concern than with Radon transform implementations.

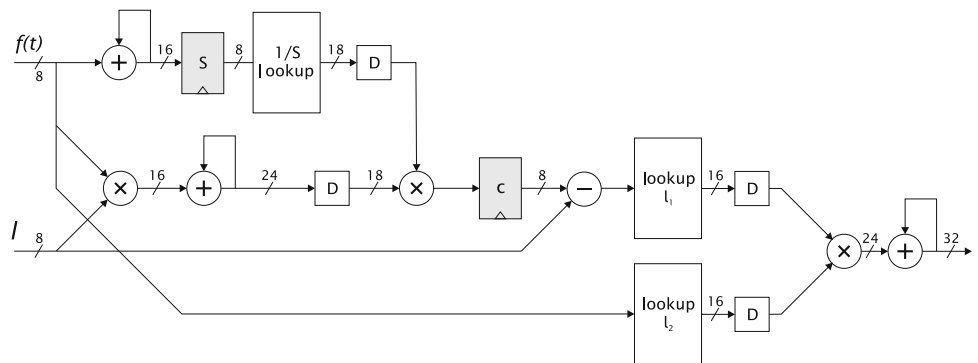
**Fig. 6** Type A functional block



**Fig. 7** Type B functional block



**Fig. 8** Type C functional block



The accuracy of this implementation is affected by two factors. The first is the use of nearest-neighbour approximation in the rotation block. The second is the use of constrained wordlengths in the functional blocks. These two factors are combined in the discussion of accuracy in this section. The comparison is made to a Matlab implementation that uses an identical rotation engine (with Nearest Neighbour approximation) but computes functionals using floating point values.

With the Type A functional block, the accuracy for the three defined functionals was found to be 100% for the functionals defined above when compared to the software implementation.

For the Type B functional block, the use of the median result from the previous row in the calculation introduced some inaccuracy. The mean relative error was 2% with 52% of samples having zero relative error and 97% having less than 10% relative error.

For the Type C functional block, the use of values of the  $c$  variable from the previous row caused error, but with a different profile to that of the Type B functional block. In this case, the error had a more even spread, and there were a number of positions where the relative error was very large. About 28% of pixels had zero error, while 86% had less than 10% relative error. About 0.66% of positions had greater than 100% relative error, however.

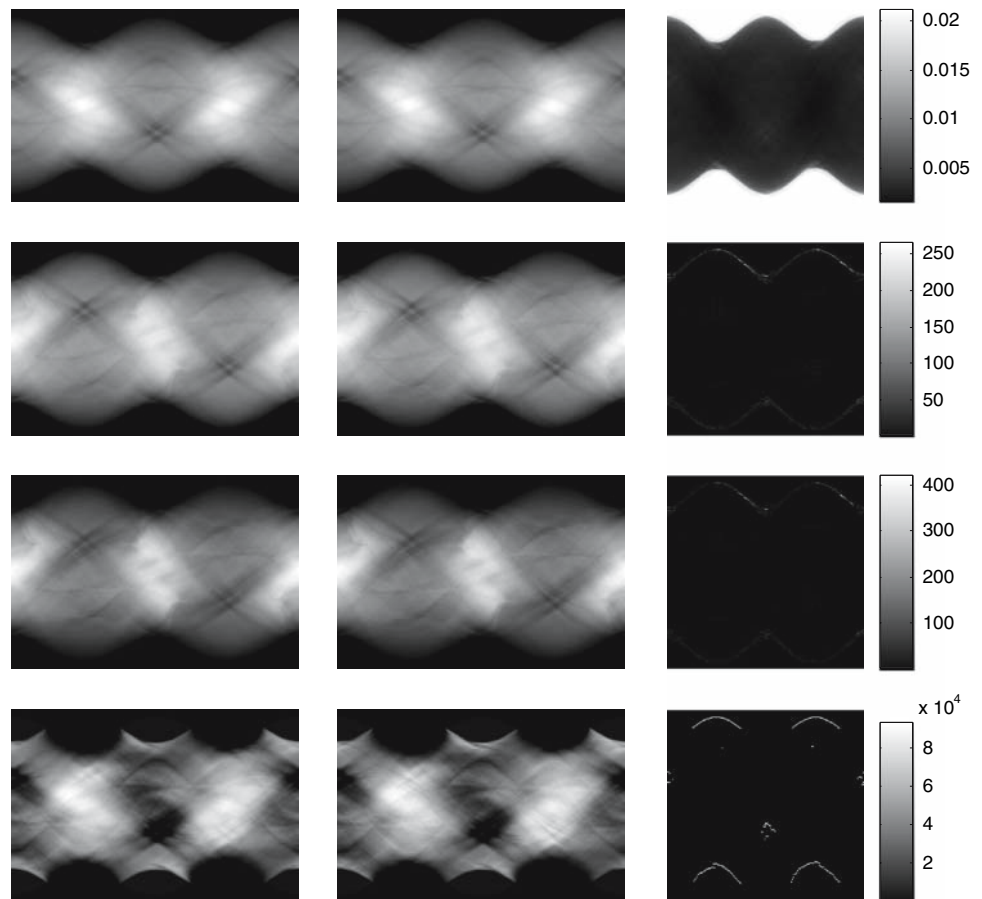
It is important to note that these errors have little effect on the trace images when treated as images. The outliers are very few in overall terms and are spread sparsely at certain points where there is already high

contrast. It depends largely on the next step in processing as to what constitutes acceptable error. Figure 9 shows traces computed using floating point arithmetic on the left and the equivalent computed in hardware on the right.

Note that the errors for the type A and B functional blocks, as a result of using one-row delayed intermediate can be overcome by adding a single full line latency to the whole system. While 256 cycles may, at first, seem significant, recall that a complete trace takes over 3 million cycles to compute, and so it is a relatively low cost. There is no other way of overcoming this issue, since these functionals produce results that depend on some property of the whole row being computed in advance. In this case, the errors were found to be tolerable for the types of processing expected to follow, and so the extra latency was not added.

Replacing the nearest neighbour approximation in software with bicubic interpolation resulted in very small discrepancies. For all functional types, the mean relative error in the resultant trace image was below 1%. Over 75% of pixels in the output trace images for all functionals had zero error. Those pixels with more significant error were at the periphery of the trace image, and in pixels with lower values, which would not contribute to subsequent steps in the recognition process, due to thresholding (see Sect. 4.4). Finally, the errors due to nearest neighbour approximation fell in the same areas as the errors shown in Fig. 9, which represent the lines that intersect with the borders of the masked object of interest.

**Fig. 9** Trace images obtained using floating point arithmetic (*left*), the equivalents using the hardware architecture (*centre*), and error images with percentage range (*right*)



## 6 Performance and area results

Before discussing the results of implementing the trace transform in hardware, an overview of the operating parameters is needed. Recall that the source image is stored in an off-chip SRAM; pipelining the reads means that a pixel can be extracted every cycle. A single  $256 \times 256$  image contains 65,536 pixels, and so it would take this number of cycles to read. As such, a new rotation is complete every 65,536 cycles plus a few cycles used to fill and flush the pipeline. With a rotation angle step size of  $2^\circ$ , 180 rotations are needed for a full set of results. Since 4 rotations are computed in parallel, the actual number of rotation operations is 45. Hence, assuming a rotation latency of 65,560 cycles (to include the margins mentioned above) a full set of rotations is complete in just under 3 million clock cycles. Each of the functionals instantiated runs in parallel, and computes results as the pixels stream through, so they do not add to the cycle length or throughput. The latency of the system is only as long as required by the functionals with greatest latency.

The tool used to synthesise the design, Xilinx ISE 8.1, gives final area and speed results that quantify the efficiency of the design. The area of a design is typically

measured in terms of logic usage. On a Xilinx FPGA, the most basic unit is a Slice which includes two four-input Look-Up Tables (LUTs) and some other logic. The FPGA fabric also includes Block RAMs and embedded  $18 \times 18$  bit multipliers. These are the three common types of available resources in a modern FPGA, and usage figures for each give a measure for area.

Table 4 shows the area results for each of the units in the design. Multiple combinations of functionals were implemented and results for single functional blocks calculated from those combined implementations. Due to the modular, fully pipelined nature of the design, there was no performance penalty associated with instantiating multiple functionals. An example implementation with three functionals was synthesised and run on the RC300 board, and further combinations tested for timing.

All units and combinations were successfully synthesised to run at 79 MHz. This limitation is enforced by the board libraries that are used to access resources on the board. Since each full trace takes 3 million cycles, this means the full system can process  $256 \times 256$  pixel images, producing traces at 26 frames per second; this satisfies the real-time requirement of the proposed system. Note that multiple traces are produced in parallel per image.

**Table 5** Running times and speedup

Functional type	S/W (ms)	H/W (ms)
Type A functional	1,400	38.5
Type B functional	3,900	38.5
Type C functional	2,600	38.5

Table 5 shows a comparison between hardware and software in computing a single functional. As a software reference an optimised MATLAB implementation is used that is running on a Pentium 4 at 2.2 GHz with 1 GB of memory. It was coded making full use of MATLAB's vector operations and avoiding the use of loops. Using the Matlab compiler yielded very minimal gains in performance, so it was not incorporated in these figures. While the absolute performance gain may not be representative of a highly optimised software implementation, the important observation is that the performance of the hardware functional blocks is not affected by the type of functional, whereas the software will reflect the complexity of the functional in its runtime. It is also important to note that these numbers are for a single functional. In the hardware implementation, additional functionals are computed in parallel, resulting in an even greater performance boost. Consider a system with three functionals—one of each type. The software implementation would have a runtime equivalent to the sum of the functionals' runtimes, whereas the hardware system would still take an amount of time equal to one functional to complete. Increasing the number of functionals further, would magnify this difference.

The number of functionals that can be implemented in hardware is limited by two factors. Firstly, the resource requirements of a functional block and the resources available on the target device must be taken into account. Clearly, the number of functionals possible in an implementation depends entirely on the combination of functional block types required.

The second factor is timing-related. A full trace computation takes just under 3 million cycles to complete, and since the input and output memories are double-buffered, it is necessary for the data transfer to and from the board to be completed in this time. The loading of input data from the PC takes 327,860 clock cycles as detailed in Sect. 4.4. Each functional produces a trace image that is  $256 \times 256$  pixels in size, with each pixel being 32 bits wide. Hence reading a trace image in bytes over USB takes  $256 \times 256 \times 4 = 262,144$  cycles. This means that the maximum number of functionals that can be accommodated in this implementation is  $\lfloor (3M - 327,680)/262,144 \rfloor = 10$ . This limitation is only due to the use of the USB port to transfer data. An FPGA board with a wider and/or faster interface to the host PC would alleviate this problem.

## 7 Conclusion

This work has shown how exploiting the heterogeneous resources on modern FPGAs enables the acceleration of complex algorithms such as the Trace transform. By analysing the algorithm in detail and making some computational simplifications it is possible to tailor the implementation to hardware. At the same time, the on-chip BlockRAMs are exploited to provide a degree of flexibility, allowing an arbitrary set of trace transform functionals to be computed using generalised functional blocks. The speedup over software becomes more pronounced, when multiple functional blocks are implemented, since in software this increases execution time; in this parallel architecture, multiple functionals are computed in the same time as a single functional, so performance is not affected. Maintaining a point-to-point approach, by taking a source image from one board RAM and depositing the results in another makes the system flexible in that the input and output data can be either processed by other blocks within the hardware or using a host PC.

This work can be extended in a number of ways. Firstly, further generalised functional blocks, defined outside the functionals proposed thus far can be implemented. By following the same methods in constructing flexible functionals, a wide array of additional functionals can be computed. Since the functionals that are most useful depend entirely on the application domain, it is clear that the ability to identify effective functionals from a large pool is a great strength. Since the hardware system functions with a host PC, it is possible to make the host PC control the functional exploration work. These results can then be used to identify functionals that match to the criteria required by the specific application. Finally, extending the system to take advantage of the runtime partial reconfiguration that is available on modern devices would represent a logical extension to the flexibility presented thus far.

## References

1. Brady, M.L., Yong, W.: Fast parallel discrete approximation algorithms for the Radon transform. In: Proceedings of ACM Symposium on Parallel Algorithms and Architectures, pp. 91–99 (1992)
2. Celoxica ltd (2004) URL <http://www.celoxica.com/>
3. Chandrasekaran, S., Amira, A.: High speed/low power architectures for the finite Radon transform. In: Proceedings of International Conference on Field Programmable Logic and Applications (FPL), pp. 450–455 (2005)
4. Deans, S.: The Radon Transform and Some of its Applications. Wiley, London (1983)
5. Fahmy, S., Bouganis, C.S., Cheung, P., Luk, W.: Efficient real-time FPGA implementation of the trace transform. In:

- Proceedings of Field Programmable Logic and Its Applications (2006)
6. Fahmy, S., Cheung, P., Luk, W.: Novel FPGA-based implementation of median and weighted median filters for image processing. In: Proceedings of Field Programmable Logic and Its Applications (2005)
  7. Frederick, M.T., VanderHorn, N.A., Somani, A.K.: Real-time H/W implementation of the approximate discrete Radon transform. In: Proceedings of IEEE International Conference on Application-Specific Systems, Architectures and Processors, pp. 399–404 (2005)
  8. Kadyrov, A., Petrou, M.: The trace transform as a tool to invariant feature construction. In: Proceedings of 14th International Conference on Pattern Recognition, vol. 2, pp. 1037–1039 (1998)
  9. Kadyrov, A., Petrou, M.: The trace transform and its applications. *IEEE Trans. Pattern Anal. Mach. Intell.* **23**(8), 811–828 (2001)
  10. Kadyrov, A., Petrou, M.: Object signatures invariant to Affine distortions derived from the trace transform. *Image Vis. Comput.* **21**, 1135–1143 (2003)
  11. Kadyrov, A., Petrou, M.: Affine parameter estimation from the trace transform. *IEEE Trans. Pattern Anal. Mach. Intell.* **28**(10), 1631–1645 (2006)
  12. Mitra, A., Banerjee, S.: A regular algorithm for real time Radon and inverse radon transform. In: Proceedings of International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 105–108 (2004)
  13. Petrou, M., Kadyrov, A.: Affine invariant features from the trace transform. *IEEE Trans. Pattern Anal. Mach. Intell.* **26**(1), 30–44 (2004)
  14. Shapiro, V.A., Ivanov, V.H.: Real-time Hough/Radon transform: algorithm and architectures. In: Proceedings of IEEE International Conference on Image Processing (ICIP), vol. 3, pp. 630–634 (1994)
  15. Shieh, E., Current, K.W., Hurst, P.J., Agi, I.: High-speed computation of the Radon transform and backprojection using an expandable multiprocessor architecture. *IEEE Trans. Circuits Syst. Video Technol.* **2**(4), 347–360 (1992)
  16. Srisuk, S., Petrou, M., Kurutach, W., Kadyrov, A.: Face authentication using the trace transform. In: Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2003, vol. 1, pp. 305–312 (2003)
  17. Srisuk, S., Petrou, M., Kurutach, W., Kadyrov, A.: A face authentication system using the trace transform. *Pattern Anal. Appl.* **8**(1–2), 50–61 (2005)
  18. Xilinx, Inc.: Virtex-II Platform FPGA Handbook (2000)

#### Author Biographies

**Suhaib A. Fahmy** Graduated with and MEng in Information Systems Engineering in 2003 from Imperial College London. Since then has been researching acceleration of image- and video-processing algorithms on FPGA platforms. Submitted a PhD thesis on Hardware Acceleration of Scene Understanding at Imperial College London. Now working as a Research Fellow at the Centre for Telecommunications Value-Chain Research (CTVR) in Trinity College, Dublin.

**Christos-Savvas Bouganis** Received first degree in Computer Engineering and Informatics from University of Patras, Greece in 1998. Completed an MSc course in Signal Processing and Communications at Imperial College London, UK in 1999. Received Ph.D. degree from Imperial College London in computer vision in 2003. Currently Lecturer in the Department of EEE at Imperial College London.

**Peter Y.K. Cheung** Professor in Digital Systems and Head of the Circuits and Systems Research Group, EEE Department, Imperial College London.

**Wayne Luk** Professor and Head of the Custom Computing Research Group in the Department of Computing, Imperial College London.