Journal of Real-Time Image Processing (2021) 18:2347–2360
https://doi.org/10.1007/s11554-021-01119-6

ORIGINAL RESEARCH PAPER

Real-time implementation of fast discriminative scale space tracking algorithm

Walid Walid[1] · Muhammad Awais[2] · Ashfaq Ahmed[2,3] · Guido Masera[1] · Maurizio Martina[1]

Received: 16 October 2020 / Accepted: 24 April 2021 / Published online: 22 May 2021

Abstract
Real-time object tracking is an important step of many modern image processing applications. The efficient hardware design of real-time object tracker must achieve the desired accuracy while satisfying the frame rate requirements for a variety of image sizes. The existing methods of visual tracking employ sophisticated algorithms and challenge the capabilities of most embedded architectures. Discriminative scale space tracking is one algorithm that is capable of demonstrating good performance with affordable complexity. It has a high degree of parallelism which can be exploited for efficient implementation of reconfigurable hardware architectures. This paper proposes a real-time implementation of the discriminative scale-space tracker on FPGA for the major blocks. A careful design exploration of core mathematical operations of the tracking algorithm is performed to improve their hardware utilization and timing performance. Among the core functional units optimized in this work, the discrete Fourier transform achieves a computational time improvement of 92% relative to existing works, QR factorization achieves a 2.3× reduction in resource utilization, and singular value decomposition yields a 3.8× improvement in processing time. The proposed data path architecture is designed using Vivado HLS tool set and implemented for Zync Zed Board (xc7z020clg484-1). For an input image size of 320 × 240, the proposed architecture achieves a mean 25.38 fps.

Keywords Discriminative correlation filter · DSST · Real time · FPGA · QR · SVD · DFT

1 Introduction

Visual object tracking has got significant research interest in recent years. The purpose of visual tracking is to identify the updated location of the target object in the incoming video sequence, given an initial target location in one frame. It finds its application in several exciting scenarios, including, but not limited to, computer vision, smart video surveillance, robotics, automation. Real-time object tracking is a challenging task whose performance is influenced by various factors, including camera motion, background variations of the scene, and complex motion of the object. To deal with these challenges, sophisticated algorithms with an optimal set of parameters are required to achieve a good degree of accuracy. Moreover, the use of high-resolution cameras further increases the computations required for successful tracking.

At present, visual tracking is mostly performed using software-based platforms, including PCs and embedded processors. However, the frame rate performance of software systems is mostly not enough to support several real-time, mission-critical applications such as tracking for accident prevention, security and defence, etc. Moreover, scale variations and requirements of multi-target tracking limit the use of the serial approach for data-centric applications. Therefore, significant improvement is required at algorithmic and implementation levels to build a real-time, stand-alone object tracking devices for most mission-critical systems. Field programmable gate array (FPGA) is a kind of hardware inherently suited for such applications, thanks to its parallel processing structure, large data throughput interfaces, and integration capability. Existing works on object tracking

✉ Walid Walid
walid.walid@polito.it

[1] Department of Electronics and Telecommunications, Politecnico di Torino, Corso Duca degli Abruzzi, 24, 10129 Turin, Italy

[2] Department of Electrical and Computer Engineering, COMSATS University Islamabad, Wah Campus, Wah Cantt , Pakistan

[3] Department of Electrical Engineering and Computer Science, Khalifa University, Abu Dhabi 127788, United Arab Emirates

are either based on discriminative [1–3] or generative [4, 5] approaches. The discriminative approaches use machine learning methods to learn the target location employing a filter, which is later used to estimate the target location. The generative approaches deal with creating the statistical model of the target. Studies have shown that discriminative approaches show better performance and require less computation [6–8]. An emerging approach is a multi-aspect detection, which considers both the size and location of the target. In this context, two techniques are proposed. The first approach is named as joint scale space tracking and utilizes a 3D correlation filter. The second approach, named multi-resolution tracking, utilizes a 2D filter at multiple resolutions, creating a 3D pyramid for detection. Both approaches are computationally intensive and not suitable for efficient hardware implementation. Recently, in [9], the authors proposed a technique named as discriminative scale space tracking (DSST), which demonstrates a good performance with reasonable complexity. The DSST algorithm achieves better performance using separate filters for translation and scale estimation [9, 10]. At first, the change in the target location is estimated using a translation filter. Next, the updated location is fed to the scale filter to estimate the target size. Finally, both filters are updated for the image frame. The method continues iteratively for the video sequence. The high degree of parallelism of the DSST algorithm makes it a suitable candidate for hardware implementation. The major mathematical operations involved in DSST are singular value decomposition (SVD), QR factorization, two-dimensional discrete Fourier transform (DFT2), and histogram of gradients (HOG). The filter is applied to an image by performing pointwise multiplication with all the pixels. This process is called windowing, which is the most critical minor operation in terms of resources.

The majority of existing works are based on software implementation of these mathematical operators, mainly focusing on the performance rather than hardware resources for higher dimensions. Therefore, there is a substantial requirement for hardware implementation of these operations targeting a complete visual tracking system on a standalone device. A survey on hardware implementations on visual object trackers is also provided in [11]. This work deals with the FPGA implementation of major blocks of a real-time DSST algorithm. We propose suitable implementation strategies for the core operations of the DSST algorithm. The implementation of the DSST is carried out using Xilinx Vivado HLS 2016.1 tool with Zedboard with Xilinx Zynq xc7z020clg484-1 System on Chip as target device. The proposed architecture is able to operate at 100 MHz clock frequency for an image of dimensions 320 × 240 pixels. It is able to achieve an estimated mean frame rate of 25.38 fps. The proposed system is fully scalable for higher image dimensions.

The rest of this paper is organized as follows. Section 2 deals with the description of the algorithm and state of the art regarding mathematical operations. Section 3 describes the proposed architecture of these operations. Section 4 deals with the implementation strategies and results. Finally, Sect. 5 concludes the paper.

2 Introduction to discriminative scale space tracking (DSST) algorithm

This section deals with the details of the DSST algorithm [9] and discusses the state-of-the-art implementations for the mathematical operations involved. Algorithm 1 demonstrates the main computation steps of fast DSST (FDSST). The algorithm receives an image and the initial target location as input. An image patch *f* centered around an initial target location *I* is extracted using a HOG extractor. These image features are utilized for learning the target translation Discriminative Correlation Filter (DCF). As the domain dimension of *f* is arbitrary, this can also be used for the scale translation DCF. H^l refers to a filter under which the correlation error between the extracted image patch and the desired output is minimum. The detailed derivation of (1) is provided in [9]. The filter equations are given as

$$H^l = \frac{\overline{G}F^l}{\sum_{k=1}^{d} \overline{F^k}F^k + \lambda}, \quad l = 1, \dots, d \tag{1}$$

$$Y_t = \frac{\sum_{l=1}^{\tilde{d}} \overline{\tilde{A}^l_{t-1}} \circ \tilde{Z}^l_t}{\tilde{B}_{t-1} + \lambda}, \quad \forall t \tag{2}$$

$$\tilde{A}^l_t = \overline{G} \circ \tilde{U}^l_t, \quad l = 1, \dots, \tilde{d} \tag{3}$$

$$\tilde{B}_t = (1 - \eta)\tilde{B}_{t-1} + \eta \sum_{k=1}^{\tilde{d}} \overline{\tilde{F}^k_t} \circ \tilde{F}^k_t, \quad \forall t. \tag{4}$$

The capital letters denote the Fourier transform of the quantities. All quantities are described in Table 1. Equation (2) is the final equation of the correlation filter. As the approach is iterative, for each new frame the filter is updated. The numerator $\tilde{A}^l_t$ in (2) is updated according to (3) while the denominator $\tilde{B}_t$ in (2) is updated according to (4). The dimensions are compressed using standard principal component analysis (PCA) to reduce the size of DFTs considering the compression suggested in [9] to realize a fast DSST. Mathematically, it is performed by exploiting a template $u_t = (1 - \eta)u_t + \eta f_t$. The *tilde* terms are obtained by windowing the quantities with *P*, which is a low-dimension subspace of the features. The compressed dimensions are obtained using the eigenvalue decomposition of the autocorrelation

Table 1 Symbols in DSST [9]

Symbol	Meaning
$\bar{}$	Complex conjugate
$\sim$	Compressed dimensions
λ	Regularization parameter
$\eta, \circ$	Learning rate, element-wise multiplication
F	Input image extracted features
l, d	Feature channel and length dimensions
G	Gaussian function for CF output
Y_t	Correlation scores
A^l, B_t	Numerator and denominator of the CF
Z^l	Image features extracted from new location
P	Projection matrix using PCA
U^l	Iterative compression of features f

Algorithm 1 FDSST algorithm [9]

```
Inputs: Image $I_t$, Prior target position $p_{t-1}$ and scale $s_{t-1}$,
    Translation model $A_{t-1,trans}$, $B_{t-1,trans}$,
    Scale model $A_{t-1,scale}$, $B_{t-1,scale}$
Outputs: Estimated target position $p_t$ and scale $s_t$,
    Updated translation model $A_{t,trans}$, $B_{t,trans}$,
    Updated scale model $A_{t,scale}$, $B_{t,scale}$
for all $frames\ t$ do
    if $t \neq 1$ then
        Translation estimation:
        Extract $z_{t,trans} \leftarrow I_t$ at $\{p_{t-1}, s_{t-1}\}$ using HOG
        $Z_{t,trans} \leftarrow z_{t,trans}$ using DFT2 and compute correlation scores $y_{t,trans}$ using (2)
        Set $p_t = \max\{y_{t,trans}\}$
        Scale estimation:
        Extract $z_{t,scale} \leftarrow I_t$ at $\{p_t, s_{t-1}\}$ using HOG
        $Z_{t,scale} \leftarrow z_{t,scale}$ using DFT2 and compute correlation scores $y_{t,scale}$ using (2)
        Set $s_t = \max\{y_{t,scale}\}$
    end if
    Model update:
    Extract $f_{t,trans} \leftarrow I_t$ at $p_{t-1}$ , $f_{t,scale} \leftarrow p_t$ at $s_{t-1}$ using HOG extractor
    Using SVD compute $P_{t,trans}$, calculate DFT2 of $u_{t,trans}$ and $f_{t,trans}$ then update the translation model $A_{t,trans}$, $B_{t,trans}$ using (3) and (4)
    Using QR compute $P_{t,scale}$, calculate DFT of $u_{t,scale}$ and $f_{t,scale}$ then update scale model $A_{t,scale}$, $B_{t,scale}$ using (3) and (4)
end for
```

matrix of u_t. As shown in Algorithm 1, the compression step is achieved with SVD and QR decomposition. This step is the training or learning step. The filter application or detection part is implemented by (2), which calculates the correlation scores. z_t is a new extracted sample from the estimated 2D target location using HOG extractor. By taking the inverse DFT of Y_t and maximizing the correlation scores, the new target estimation is obtained. This step completes the translation estimation. The scale estimation is obtained by repeating the above steps for scale one-dimensional filter, using the updated target location from translation estimation. This way, searching the scale in the updated location saves a lot of computations. The algorithm performs the scaling, translation, filter estimation and update process, as described in (1), (2), (3) and (4). They involve the scale and translation filter estimation and update equations. As mentioned earlier, the core mathematical operations of DSST algorithm involve DFT2, QR, SVD and HOGs. These operations are discussed as follows.

2.1 DFT and DFT2

The synthesis and analysis equations of discrete Fourier transform (DFT) are given, respectively, as

$$X_k = \sum\nolimits_{n=0}^{N-1} x_n e^{-\frac{j2\pi}{N}kn} x_k = \frac{1}{N}\sum\nolimits_{n=0}^{N-1} X_n e^{\frac{j2\pi}{N}kn}, \tag{5}$$

respectively, where $W_N^{nk} = e^{-\frac{j2\pi}{N}kn}$ is the twiddle factor, X_n and x_n are complex numbers. The twiddle factor can also be expressed as $W_N^{nk} = \cos(\frac{2\pi n}{N}) - j \cdot \sin(\frac{2\pi n}{N})$ using Euler's identity.

$$X_k = \sum\nolimits_{n=0}^{N-1} x_n \left[\cos\left(\frac{2\pi n}{N}\right) - j \cdot \sin\left(\frac{2\pi n}{N}\right)\right] \tag{6}$$

A straightforward implementation of N-Point DFT has a complexity of O(N^2) operations. Fast Fourier transform (FFT) is a hardware friendly algorithm which reduces the DFT computations to O($N\log_2 N$), using well-known decimation in time (DIT) and decimation in frequency (DIF) techniques [12]. A number of hardware implementations of FFT exist including parallel [13] and serial approaches [14]. FFT requires $N = 2^n$ which is not fixed in this case. So, instead DFT is implemented. An implementation for DFT is provided in [15], which uses the Coordinate Rotation Digital Computer (CORDIC) algorithm to calculate the twiddle factor. Our approach uses precalculated twiddle factors and thus improves performance.

The DFT2 is the two-dimensional Fourier transform applied to a matrix. To compute DFT2, first DFT is applied along the rows of the matrix, and the result is transposed. Then DFT is applied along with the columns. Finally, results are transposed and assigned to the output. In most of the published works, the DFT2 is approximated by two dimensional fast Fourier transform (FFT2). Instead, this work adopts the approach of [16] based on modified row–column decomposition.

2.2 QR factorization

QR factorization is the decomposition of a matrix A of dimensions $m \times n$ into two matrices, i.e. Q matrix of dimensions $m \times m$ and R matrix of dimensions $m \times n$. Q is the orthogonal matrix, while R is the upper triangular matrix. The literature proposes three main approaches for QR factorization, namely, Gram-Schmidt approach [17], Householder transformation [18] and approach based on Givens rotation [19, 20]. This work adopts the third approach, i.e. Givens rotation. The idea is to apply Givens rotations to elements of the lower triangle of A and turn them to zero. When all the lower triangle elements are zeroed, matrix R is obtained. Applying the same transformation to an identity matrix in parallel gives the matrix Q^{T}. The Givens rotation matrix is of the form, $G = \begin{bmatrix} c & s \\ -s & c \end{bmatrix}$, where $c = \frac{a}{\sqrt{a^2+a^b}}$, $s = \frac{b}{\sqrt{a^2+a^b}}$, a is the first element of the row pair and b is the element which has to be turned to zero below a. A systolic array-based implementation is proposed in [20]. Our approach is similar, but we focus on resource optimization rather than performance. We also employ row-parallel approach, which improves performance by offering more parallelism.

2.3 SVD

SVD is the eigenvalue decomposition of a matrix A of dimensions $m \times n$ into three matrices U, S and V of dimensions $m \times m$, $m \times n$ and $n \times n$, respectively. U and V contain the left and right eigenvectors of A, while S is a diagonal matrix containing real eigenvalues. SVD in hardware is mostly implemented using two-sided Jacobi method [21]. The idea is to divide the matrix into 2×2 small matrices. Jacobi rotations to elements of the matrix A are applied from left and right, hence the name two-sided Jacobi. This multiplication turns non-diagonal elements to zero giving the matrix S. Similar transformations to the identity matrix gives matrix U and V. The Jacobi rotation matrix is of the form $\begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} c & s \\ -s & c \end{bmatrix} = \begin{bmatrix} a1 & 0 \\ 0 & a2 \end{bmatrix}$, where $c = \cos(\theta)$, $s = \sin(\theta)$ and θ is given by $\theta = \frac{1}{2} \arctan \frac{c+b}{d-a}$. Fixed point based implementation are given in [21, 22]. Our approach is similar but we focus on time optimization because this unit has small dimensions in the algorithm, so it is operated in parallel.

2.4 Histogram of gradients (HOG)

HOG is a classifier used for target recognition. It is constructed with the help of image gradients and extracts the features contained in image pixels. HOG is a computationally intensive operation, and its implementation is proposed by a number of approaches [23–25]. The authors of [23] provide a comparative study of different implementations as well. The implementation in [24] utilizes the least amount of resources but operates below 100 MHz. In [25], an approach is proposed which achieves a frame rate of 60 and requires less amount of hardware resources. It avoids expensive angle calculation using integer multiplication and inequality comparisons.

As a first step, the magnitude and orientation of an image gradient are computed. The magnitude of the gradient is assigned to suitable bins among the nine available. This assignment is done based on the gradient orientation (0–180) and for (0–360). The window for detection is composed of 8×8 pixels of non-overlapping cells. Afterwards, an aggregate module is utilized for summing the 64 pixels of each bin to create the histogram. Finally, they are passed to a normalization phase. The details of the algorithm are given in [25].

3 Proposed architectures

This section deals with the details of the architecture of the mathematical operations described in the previous section. The flowchart is depicted in Fig. 1. The proposed DSST algorithm has four major steps. The translation search, i.e. the 2D position of target and translation filter update steps involve HOG extraction and DFT2 of 3D matrices. The scale search and scale filter update involve HOG extraction and DFT of a 2D matrix. SVD and QR are involved in translation filter update and scale filter update steps respectively. The DFT unit is the most critical block in terms of performance because of operation on 3D 320×320 matrix. Vivado HLS is used as the base tool for synthesizing and simulation.

3.1 Discrete Fourier transform and DFT2

The vector DFT acts as a building block for matrix DFT, i.e. DFT2. The architectures for both are discussed as follows.

3.1.1 DFT

Figure 2 demonstrates the proposed architecture to compute one dimensional DFT. In the first clock cycle i (input loop counter), the sample $X\_real\,[i]$ and $X\_imag\,[i]$ are received along with twiddle factors $\cos\,[i]$ and $\sin\,[i]$. Four multipliers and two adders/subs are required to complete the complex multiplication which is the basic DFT block. This goes to the

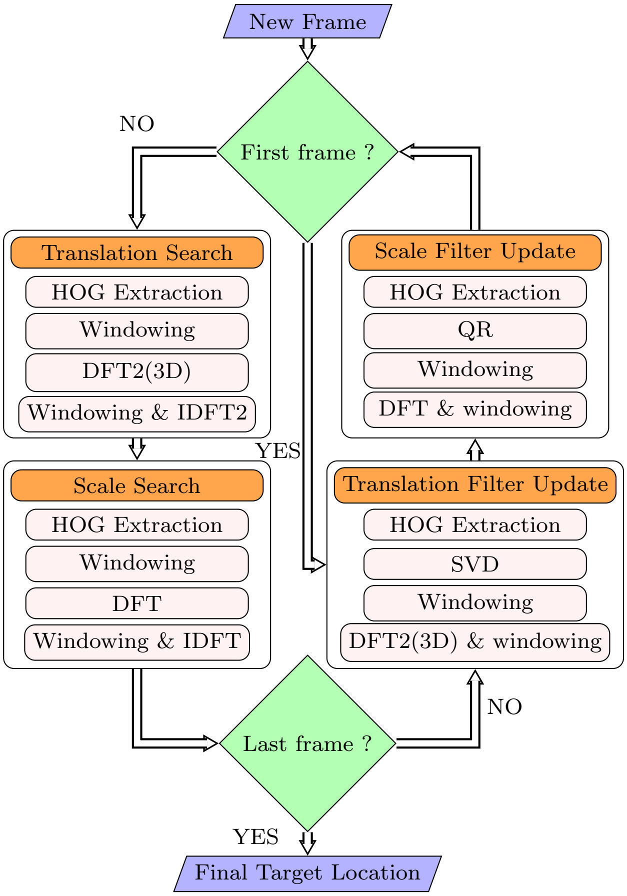

Fig. 1 Flowchart of the DSST algorithm [9]

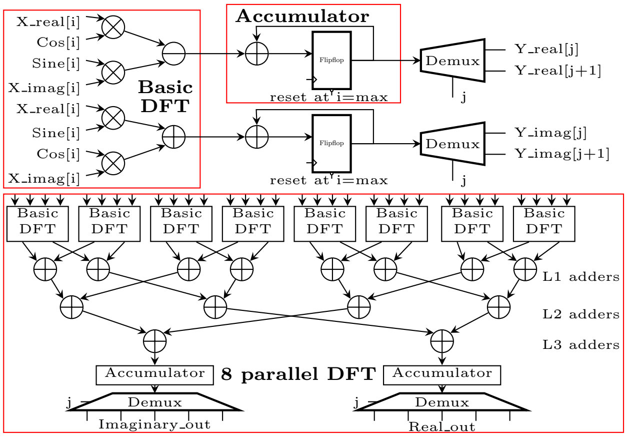

Fig. 2 RTL diagram of DFT 1D unit

accumulator (at first, the other input to the accumulator is 0 because the register is at reset). This produces real part of complex multiplication, i.e. $X\_real\,[i]\cdot(\cos[i]) - X\_imag\,[i]\cdot(\sin[i])$. Similarly, the other output of accumulator produces $X\_real\,[i]\cdot(\sin[i]) + X\_imag\,[i]\cdot(\cos[i])$. Now, in the next clock cycles, the basic DFT receives the inputs $X\_[i+1]$ along with the twiddle factors. After it performs the complex multiplication, the results are accumulated. When the last input is processed, j (the output loop counter which is 0 at first) assigns the first output $Y\_real[i]$, $Y\_imag[i]$ via demultiplexers. In the end, j is incremented and the register is reset. This whole process is repeated until all the N outputs are produced. This approach is serial as it takes N cycles to produce one output and consumes 1 input per cycle.

Using the basic blocks discussed above, now the architecture is parallelized to support fast DFT for higher image dimensions. This parallelization is shown in the lower part of Fig. 2. By operating eight elements in parallel, Equation (6) can be modified as

$$X_k = \sum_{n=0}^{N/8-1} x_n\left[\cos\left(\frac{2\pi n}{N}\right) - j\cdot\sin\left(\frac{2\pi n}{N}\right)\right] + \cdots + x_{n+7}\left[\cos\left(\frac{2\pi(n+7)}{N}\right) - j\cdot\sin\left(\frac{2\pi(n+7)}{N}\right)\right] \tag{7}$$

x_n is complex. Each complex multiplication i.e. $C_n = x_n\cdot\left[\cos\left(\frac{2\pi n}{N}\right) - j\cdot\sin\left(\frac{2\pi n}{N}\right)\right]$ is performed by a basic DFT block. The L1 adders add the outputs of basic DFT in pairs of $\{C_n, C_{n+1}\}, \{C_{n+2}, C_{n+3}\}, \{C_{n+4}, C_{n+5}\}, \{C_{n+6}, C_{n+7}\}$. Let $M_n = C_n + C_{n+1}$ then L2 adders add the following pairs: $\{\{M_n, M_{n+1}\}, \{M_{n+2}, M_{n+3}\}\}$. Finally, the L3 adders add the M_n terms to produce 8 point DFT. Accumulator sums the DFTs until N inputs are processed. Similarly to the serial approach when the last input is processed demultiplexers assign the output, j is incremented and accumulators are reset. Thus, with an adder tree between the basic unit and accumulator, 8-parallel DFT is performed. All the DFT coefficients are pre-computed and stored in memory. The HLS compiler mostly handles the intermediate computation results. However, in some cases, we instantiate and partition the BRAMs for intermediate values so, the elements can be processed in parallel. For this unit, the input and output arrays are partitioned into 8 BRAMS. So, eight elements can be accessed in parallel. The general DFT has a delay of $O(N^2)$. This architecture improves it to delay $= O(\frac{N^2}{8}) +$ pipeline stages $\approx O(\frac{N^2}{8})$. The architecture is implemented for maximum $N = 320$. If $N < 320$ then a comparator (not shown in figure for simplicity) limits the number of traverses through the block. For improving performance, task-level parallelism is used with the help of Vivado HLS Dataflow pragma.

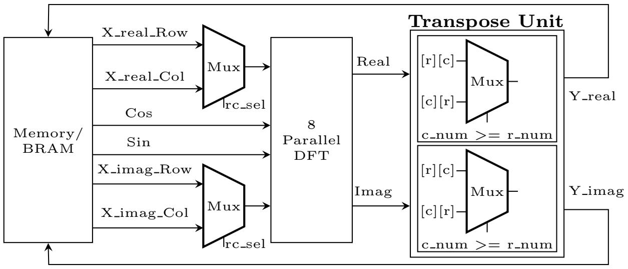

Fig. 3 RTL diagram of DFT2 unit

3.1.2 DFT2

The proposed DFT2 architecture is shown in Fig. 3 and uses the row–column decomposition approach. The real and imaginary parts are kept separate so that operations run in parallel. The 2D matrix is taken as input. Rows are selected and passed to the DFT calculation blocks first, while the coefficients are precomputed and saved in BRAM blocks. 8-parallel unit is used as a basic block for this unit. The output goes to a transpose unit and afterwards to the memory. In the next steps, relevant outputs are fed along with the weights to 8-parallel DFT block, to compute the column-wise DFT.

The transpose unit has a delay proportional to N. It is given by $O\left(N^2 - \sum_{i=1}^{N-1} i\right)$. It is parallelized to $O\left(\frac{N^2}{8} - \sum_{i=1}^{\frac{N}{8}-1} i\right)$. The total delay is $2M \times (T_{\mathrm{DFT}} + T_{\mathrm{Trans}})$), where M is the number of rows of the matrix. This is nearly equal to $2M \times (T_{\mathrm{DFT}})$. As this unit lies in the critical path, at the cost of twice the hardware, the maximum delay can be reduced to half that is $M \times (T_{\mathrm{DFT}})$. It was synthesized for a maximum size of 320 × 320. To save the BRAM resources, the same input matrix is used for saving the outputs. For DFT2 (3D), DFT2 is used as a base unit. The maximum delay will be $P \times (T_{\mathrm{DFT2}})$, where P is the third dimension, i.e. the number of 2D matrices. The maximum value of P is 18 for fDSST. The same DFT2 unit is used with a divisor in the accumulator before the delay element to divide by N.

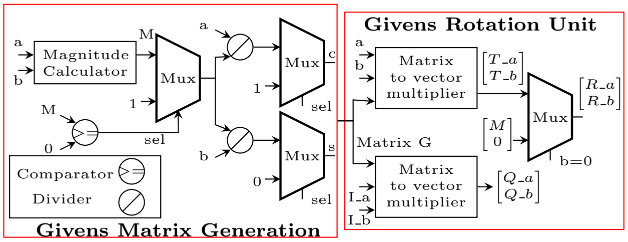

Fig. 4 RTL diagram of QR factorization

3.2 QR factorization

The proposed DSST architecture performs the QR factorization using the Givens Rotation Method [20]. The Vivado HLS QR factorization library [26] is used as a reference unit and modified for our architecture. This implementation is for real numbers and is shown in Fig. 4. Algorithm 2 highlights the main computational steps of QR factorization. The QR factorization unit consists of Givens matrix generation and rotation units. The Givens matrix generation takes as input, two elements from two rows of matrix A and provides the Givens matrix as output. The input 2D matrix A and the Givens matrix G are given by

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{14} \\ \overbrace{a_{31}} & a_{32} & a_{33} & a_{34} \\ \underbrace{a_{41}} & a_{42} & a_{43} & a_{44} \end{bmatrix} \quad \text{and} \quad G = \begin{bmatrix} c & s \\ -s & c \end{bmatrix}, \tag{8}$$

where $c = \frac{a}{M}$ and $s = \frac{b}{M}$. Magnitude M is calculated as

$$M = \sqrt{a^2 + b^2} = x \times \sqrt{1 + y \times y}, \tag{9}$$

where $x = \max(a, b)$ and $y = \frac{\min(a,b)}{x}$.

Lets take $a = a_{31}$ and $b = a_{41}$. With the help of (9) matrix G is calculated. Equation (9) is implemented by magnitude calculation unit shown in Fig. 5. In this unit, the maximum of the two inputs is determined using a comparator and two multiplexers. Afterwards, the divider calculates y. In the end, magnitude M is obtained by a combination of adder, multiplier and square root unit. To obtain matrix G, c and s are determined. As the magnitude appears in the denominator, divide by zero is checked using a comparator and a multiplexer. Matrix R is obtained by turning the lower triangular elements of matrix A to zero. For this purpose, the Givens rotation is applied in the following manner:

$$G \cdot \left[a_{31} a_{41}\right]^{\mathrm{T}} = \left[a_{31} {*} 0\right]^{\mathrm{T}}. \tag{10}$$

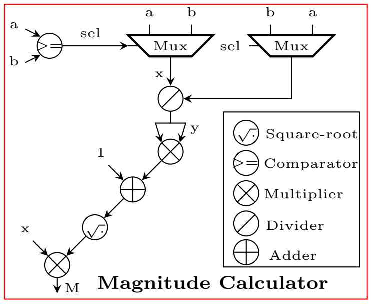

Fig. 5 RTL diagram of magnitude calculator for QR

The Givens rotation block is shown on the left of Fig. 5. It is simply a 2×2 matrix to vector multiplication. If the second element b is already zero, then Givens rotation is simply $a = M$ and $b = 0$. This assignment is implemented with multiplexers. Givens rotation unit is accompanied by a comparator to avoid assigning wrong values to zeroed positioned elements shown in the top left of Fig. 5. In that case, the first element is equal to the magnitude. For complete matrix R now select $a = a_{21}$ and $b = a_{31}$ $*$ and repeat until all the lower triangular elements are turned to zero. The matrix Q is obtained by performing the same generated rotations to an identity matrix. A small address generation unit is used for row selection. Row pairs are selected to be operated in parallel. The rotations are applied to all the columns of the selected rows.

A single matrix is used for input A and output R to reduce the number of resources. The critical path is in the Givens generation block because of the magnitude unit. This block has a division and square root operator. The generated Verilog code can be modified at RTL level to pipeline the architecture. Also, the number of parallel rotations impact resources and performance. Resource optimization is employed since this unit is not on the critical path. The generation and rotation blocks are parallelized by a factor of two and pipelined with an initiation interval of 4.

3.3 SVD

This work uses the two-sided Jacobi method [21] to perform SVD. The Vivado HLS library [26] is adopted as a reference unit and optimized for the target application. This implementation is for real numbers. The implementation is shown in Fig. 6. Algorithm 3 demonstrates the execution of SVD. The SVD unit consists of the diagonal processor (DP) and non-diagonal processor. The DP takes as input the 2D matrix A, divided into $N/2$ 2×2 submatrices. Matrix A is the same as Eq. (8) and Jacobi left and right matrices are given by $u = \begin{bmatrix} c1 & s1 \\ -s1 & c1 \end{bmatrix}$ and $v = \begin{bmatrix} c2 & s2 \\ -s2 & c2 \end{bmatrix}$, where $c = \cos(\theta)$ and $s = \sin(\theta)$. For θ, we have $\theta = \frac{1}{2} \arctan \frac{b+c}{d-a}$ to avoid calculation of arctan, consider

$$\tan(2\theta) = \frac{2b}{(d-a)}. \tag{11}$$

Algorithm 2 QR calculation algorithm.

```
Inputs: Matrix: A
Outputs: Orthogonal matrix: Q , Triangular matrix: R
    Givens rotation Generation:
for all $r, c \in A$ do
    if $r > c$ and $A[r][c] \neq 0$ then
        for all non overlapping $[ra, rb]$ pairs do
            Compute magnitude M using (9)
            Generate the matrix G using $c = \frac{a}{M}$, $s = \frac{b}{M}$
        end for
    end if
end for
    Givens rotation Application:
for all $r, c \in A$ do
    if $r > c$ and $A[r][c] \neq 0$ then
        for all non overlapping $[ra, rb]$ piars do
            for all $c \in [ca, c_{\max}]$ do
                Triangular matrix R computation:
                Obtain R by applying (10) to [ra,rb] pairs
                Orthogonal matrix Q computation:
                Generate matrix Q by applying (10) to $I$
            end for
        end for
    end if
end for
```

The implementation of angle calculation is shown in Fig. 7. A divisor depicted in the figure generates $\tan(2\theta)$. By the use of trigonometric identities, cos and sine are derived. They are as under

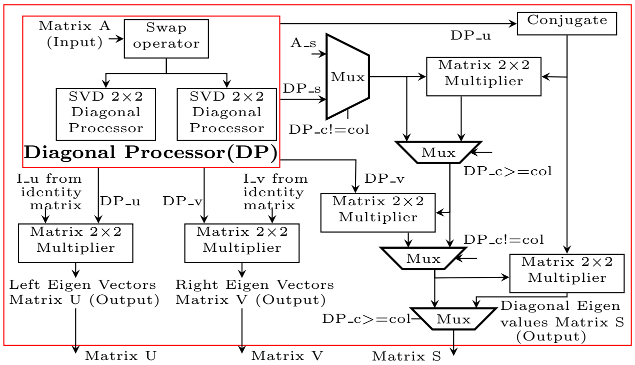

Fig. 6 RTL diagram of SVD

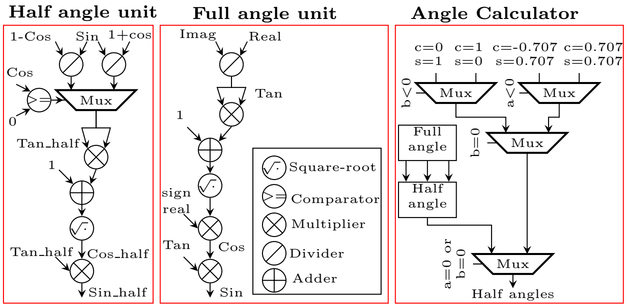

Fig. 7 RTL diagram of angle calculator

$$\cos(\theta) = \frac{1}{\sqrt{1 + \left(\tan^2(\theta)\right)}},$$
$$\sin(\theta) = \cos(\theta).\tan(\theta), \tag{12}$$
$$\tan\left(\frac{\theta}{2}\right) = \frac{(1 - \cos(\theta))}{\sin(\theta)}.$$

The computation of half-angle identities is shown in the left of Fig. 7. Angle calculator is shown in the right of Fig. 7, which consists of a tree of multiplexers to select the correct angle based on whether the number is real, imaginary or complex.

The Jacobi matrix generation is shown in Fig. 8. The numerator and denominator of (11) are computed via adder and subtractor and passed to the angle calculator unit. The Jacobi matrices are generated using $c = \cos(\alpha \mp \beta)$ and $s = \cos(\alpha \pm \beta)$ identities, where α and β are half angles calculated previously. The identities are implemented with simple vector multiplier. The Jacobi rotations are given by

$$\begin{bmatrix} c1 & -s1 \\ s1 & c1 \end{bmatrix} \begin{bmatrix} a11 & a12 \\ a21 & a22 \end{bmatrix} \begin{bmatrix} c2 & s2 \\ -s2 & c2 \end{bmatrix} = \begin{bmatrix} a11 & 0 \\ 0 & a22 \end{bmatrix}. \tag{13}$$

Thus Jacobi rotations are simple matrix multiplications and are implemented by vector multipliers in Fig. 8.

As all the building blocks are described, next is the demonstration of how the three U, S and V matrices are generated. As shown in Fig. 6, SVD consists of DP and non-DP. The quantities used are defined as follows: *A_s* denotes diagonal submatrix of A, terms with DP mean newly updated submatrix from DP unit and terms with I indicate they are from identity matrix. DP_*c* and col represent 2×2 submatrix the current iteration and currently selected column pairs, respectively. The algorithm is repeated for a minimum number of iterations to achieve convergence. Literature suggests that 6–10 iterations are enough for it. In our case, for a dimension of 32, the iteration factor is 6. The iteration factor is determined from the table given in [21].

The diagonal processor receives a 2×2 main diagonal submatrix *A_s* of matrix A. It has a swap operator implemented with a mux. The swap operator swaps the columns to arrange the diagonal elements in ascending order as SVD requires it. After the swap, DP generates the Jacobi matrices and applies the rotations. *A_s* is rotated by DP_*v* first and then it is post multiplied by DP_*u*. Now the non-diagonal elements of *A_s* are turned to zero. DP now outputs the new matrices DP_*s*, DP_*u* and DP_*v* to the non-DP. Non-DP in Fig. 6 receives 2 identity matrices U, V and matrix A. It also receives DP_*u*, DP_*s* and DP_*v* from the DP. Now there are two subcases, current 2×2 submatrix DP_*c* is less than current column indices col and DP_*c* is greater than col. In the first case, *A_s* is pre-rotated by Hermitian transposed DP_*u* then post-rotated by DP_*v*. In the second case, *A_s* is pre-rotated by DP_*v* then post-multiplied by Hermitian transposed DP_*u*. If the submatrix overlaps with DP_*s*, then DP_*s* values from DP are used otherwise from matrix A. This process is repeated until the matrix A is diagonalized. This diagonalized matrix is the matrix S containing the eigenvalues. Matrix U (left eigenvectors) and V (right eigenvectors) are obtained by applying the same DP_*u* and DP_*v* rotations to identity matrices *I_u* and *I_v*, respectively.

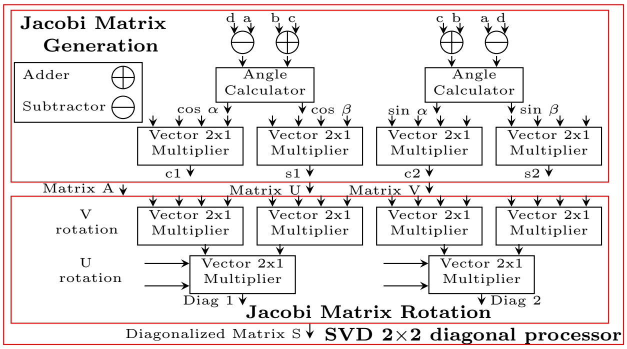

Fig. 8 RTL diagram of SVD 2 × 2 unit

Algorithm 3 SVD algorithm.

```
Inputs: Matrix: A
Outputs: Diagonalized eigen values matrix: S
    Left and right eigen vectors matrices: U and V
repeat
    for all $c \in A$ do
        Diagonal Processor:
        for all $2 \times 2$ $DP\_c \in [0, columns/2]$ do
            if $c_{diag1} < c_{diag2}$ then
                Swap the columns.
            end if
            Jacobi matrices generation:
            Compute half angles $\cos\alpha, \cos\beta$ and $\sin\alpha, \sin\beta$ using (11) and (12)
            Generate the matrix DP_u and DP_v by using c $= \cos\alpha \mp \beta$ and s $= \sin\alpha \pm \beta$ identities
            Jacobi two sided rotation:
            Matrix S_diagonal is computed by applying the Jacobi rotations on A using (13)
        end for
        Non Diagonal Processor:
        for all $2 \times 2$ $DP\_c \in [0, columns/2]$ do
            Jacobi rotations for matrix S:
            if $DP_c < col$ then
                Pre-rotate A by Hermitian DP_u and post-rotate by DP_u
            end if
            if $DP_c > col$ then
                Pre-rotate A by DP_v and post-rotate by Hermitian DP_u
            end if
            Jacobi rotations for matrices U and V:
            For matrix U rotate identity matrix I_u by DP_u
            For matrix V rotate identity matrix I_u by DP_v
        end for
    end for
until minimum number of iterations to converge
```

A single matrix is used for input A and output U to reduce the number of resources. The critical path is in the DP block because of the SVD 2×2 unit, which has the angle calculator. The generated Verilog code can be modified at the RTL level to pipeline the architecture as the critical path is too long. At the cost of resources, frequency and latency optimization is employed. The 2 generations and rotation blocks are used in parallel and pipelined with the initiation interval of 8.

In summary, initially, Jacobi's left and right rotations are generated. They are applied to original matrix A to obtain S. Left rotations on identity matrix produce Matrix U while the right rotations produce matrix V^T. But this approach can only be applied to symmetric matrices. If the matrix is not symmetric, then a further step is needed to symmetrize the matrix. The symmetrization can be done with the help of Givens rotations.

3.4 Histogram of gradients (HOG)

The architecture suggested relies on the implementation provided by the authors of [25]. The 2D grayscale image is taken as input. Gradients p_x and p_y are calculated for each pixel in both x and y directions, respectively, using two subtractors. The gradient magnitude and direction are given by

$$M = \sqrt{p_x^2 + p_y^2},$$
$$\tan(\theta) = \frac{p_y}{p_x}.$$

The angle (0–180) is used to assign a gradient magnitude to 9 different bins [24]. The histogram is built from these bins. To save resources angle calculation, i.e. the orientation of the gradients is computed using integer multiplications instead of division. This improvement is achieved by performing angle calculation and bin assignments together. If the angle lies between any one of the nine available slots, then that gradient magnitude is assigned to the corresponding bin. As demonstrated by authors of [25], the gradient magnitude is assigned to bin one if the following inequality holds:

$$0 \leq p_y/p_x \leq \tan(20) \rightarrow 0 \leq p_y \leq p_x \tan(20).$$

This step simply involves a multiplication of integer constant to satisfy the inequality. This step helps save resources because it avoids a calculation of $\tan(\theta)$, which involves divisions. Based on the orientation, the gradients are assigned to 9 different bins. They are the contrast insensitive bins. Afterwards, they are aggregated for smoothening among all bins. At last, they are normalized using L1-norm. As compared to the L2-norm, L1-norm avoids squaring. Also, instead of dividing the reciprocal is multiplied. This improvement further saves hardware resources without sacrificing accuracy too much. To obtain Felzenszwalb's HOG, two extra steps need to be performed. First bin assignment step is performed again. This time the gradients are assigned to 18 different bins based on orientation (0–360). They are the contrast-sensitive bins. These 18 bins from each block are averaged together. Also, the previously calculated 9 bins are averaged for each block. For 4 blocks, the 9 normalized bin elements are also averaged. These 18 directional, 9 non-directional and 4 normalization bins form the 31 third dimensional features of fHOG. After this step, the output is assigned. The implementation diagram is demonstrated in [25].

4 Implementation results and discussion

The data path of the proposed DSST system is specified using the Vivado HLS tool. The prototyping is performed on Zedboard with Xilinx Zynq xc7z020clg484-1

Table 2 Timing results for the FPGA-based DFT in us

N	Cycles [15]	Cycles S_HLS	Cycles S_HLS	Time [15]	Time P_HLS	Time P_HLS
10	24,179	260	247	484	2.2	2.4
12	28,999	358	342	580	3.3	3.32
20	96,509	910	883	1930	7.7	8.6

Table 3 Area results for the DFT implementation in FPGA

N	LUT [15]	LUT S_HLS	LUT S_HLS	MUL [15]	DSP48E P_HLS	DSP48E P_HLS
10	616	395	3702	4	8	32
12	648	421	3710	4	8	32
20	776	460	3823	4	8	32

Table 4 AREA and TIME results of DFT and DFT2

N	DFT type	Time (ms)	LUT	DSP
320	DFT	0.475	1722	32
320 × 320	DFT2	273	11,352	32
320 × 320	DFT2 (2 parallel)	170	14,934	64

System-on-Chip. The block-wise implementation results are discussed as follows.

4.1 Discrete Fourier transform

DFT 1D has maximum size $N = 320$. For DFT, the results are compared with [15]. Vivado HLS post-synthesis timing and resources results, referred to as HLS, are shown in Tables 2, 3 and 4. HLS timing results in Table 2 outperform the one in [15] by factor of 92%. This improvement is because of using precalculated twiddle factors. Also, in contrast to the implementation in [15], our implementation is extensively unrolled. Two different implementations are provided here, serial S_HLS and parallel P_HLS. Resources in Table 3 indicate that our serial DFT uses less LUTs than the one in [15] but uses twice the DSP48E units. The higher number of resources is because of separate hardware for imaginary and real parts. Also, for 8 parallel DFT, our resource consumption is much higher than [15]. The higher resource consumption is justifiable because optimization is done for performance in the case of higher dimensions. The maximum operating frequency for 8 parallel DFT is 112 MHz while [15] operates at 50 MHz. The frequency can be further improved by pipelining the Verilog code generated at the RTL level. It cannot be improved in Vivado HLS as the pipeline commands unroll the architecture in the scope available. So only inner loops are pipelined.

The simulation results were compared against MATLAB-generated golden matrices. The relative percentage error is computed by $E = \frac{(R_{\mathrm{MATLAB}} - R_{\mathrm{HLS}})}{R_{\mathrm{MATLAB}}} \cdot 100$, where E is the error. The golden matrices were generated using intermediate values of the algorithm at the input of each block. This is then applied to HLS-based units. Afterward, the outputs of both are compared. The average error values, in this case, were 6.52 and 6.9 for real and imaginary matrices, respectively. This is because MATLAB uses double-precision values. Our results are still approximate enough because double precision in hardware will consume four times more resources. Also, the latency will be much slower. DFT2 is implemented for maximum dimensions of 320 × 320. The results for DFT and DFT2 are reported in Table 4. It has a maximum frequency of 112 MHz.

Table 5 TIME and area for FPGA-based 4 × 4 QR

Architecture	F (MHz)	Cycles	DSP	FF	LUT
Mult A [20]	117.1	116	48	10,844	11,337
Mult B [20]	377.6	140	48	11,520	11,225
HLS	115.9	467	21	6054	8824

Table 6 TIME results for SVD implementation in FPGA

N	Time [22] (μs)	Time [21] (μs)	Time HLS (μs)
4 × 4	–	12.1	35
10 × 10	3570	1001	207
20 × 20	4280	12,100	1135
30 × 30	12,550	–	3445
40 × 40	26,860	–	7799
50 × 50	–	69,500	14,872

Table 7 AREA results for SVD implementation in FPGA

Architecture 4 × 4	LUT	BRAM	DSP48E
Implementation [22]	5283	8	12
Implementation [21]	1504	3	16
HLS	10,110	14	30

4.2 QR factorization

This unit has the maximum dimensions of 800 × 17. For QR, the results are compared with [20] where implementation is for a 4 × 4 matrix. QR [20] is fully unrolled and uses fixed-point iterations while the approach is based on floating-point. The comparison can be made from the 32-bit fixed-point version. A comparison of the timing results from Table 5 indicates that HLS-based approach takes 4 times more clock cycles than the one in [20]. But HLS based area results are significantly better. This approach takes 2.3 times less DSP48E resources as this is not the critical block, so resource optimization was our target. The operating frequency is almost similar to the non-pipelined version. Our approach is generic while the one in [20] is a fixed size. The worst-case delay for an input of size 800 × 17 is 337 μs for QR economy. At the same time, it consumes 26 DSP48Es resources. The maximum frequency is 116 MHz. The error is calculated using the same procedure as in Sect. 4.1. The average error value is 0.034 for matrix Q as only matrix Q is needed for the DSST algorithm. Our results are approximate enough because double precision used by MATLAB in hardware will consume four times more resources.

4.3 SVD

This unit is implemented for maximum dimensions of 32 × 32. For SVD, the results are compared with [21, 22]. The results are compared for low values of N with [21] and for higher values with [22]. Authors of [21, 22] use fixed-point iterations while we have a floating-point implementation. Comparison of the timing results from Table 6 indicates HLS-based approach performs better than both implementations for all dimensions except for $N = 4$ with [21]. Timing is improved by nearly a factor of 3.7 and 4.8 as compared to [22] and [21], respectively. But HLS based area consumption is 2 times higher, as shown in Table 7. The reason being our angle calculation unit uses division and the square root of floating-point numbers, while [21, 22] uses the CORDIC algorithm for it. The worst-case delay for an input of size 32 × 32 is 4 ms. At the same time, it consumes 30 DSP48Es resources. The maximum frequency is 108 MHz. The error is calculated similarly as in Sect. 4.1. The average error value is 7.156 for matrix U. As only matrix U is needed for the DSST algorithm. Our results are approximate enough because double precision used by MATLAB in hardware will consume more resources. Also, the latency will be higher.

4.4 Histogram of gradients (HOG)

The maximum dimensions of this unit are 320 × 240. The results of [25] are reported here. The maximum operating frequency is 270 MHz. 60 fps for an image size of 1920 × 1080. It consumes only 12 DSP blocks.

4.5 Overall resources and speed

Table 8 shows the maximum resources used by the units in terms of DSP48E, BRAM, FF and LUTs. Also, the maximum power for each unit is displayed. Power is reported using power reports of post place-and-route from VIVADO HLS. The values are for the maximum dimensions of each unit. The speed/frames per second (fps) is calculated for each unit separately. For each block, an average fps is considered using a range of sizes as input. The fps is calculated as

$\text{fps} = \frac{F_{\max}}{\text{Cycles}}$, where cycles is the number of clock cycles required to process one frame. Table 8 shows the mean fps of each unit. As for the whole architecture, there are four stages, namely, Scale Search (SS), Scale Filter (SF), Translation Search (TS), and Translation Filter (TF). Table 9 shows the mean fps of each stage.Total fps for a stage is computed by adding reciprocal fps of units involved in the

Table 8 Area, fps and power results of proposed architecture for DSST algorithm

Unit	Bram	DSP	FF	LUT	FPS	P (mW)
DFT	0	32	4419	6305	6153	258
DFT2	192	64	11,922	18,343	173	662
QR	33	26	6662	5837	2948	238
SVD	16	30	8294	10,808	248	358
HOG[a]	7	12	3642	3924	60	244
Misc	0	8	591	1545	4392	156
Used	248	172	35,530	46,762	–	1916
(%)	(88.6)	(78.2)	(33.4)	(87.9)	–	–
Total	280	220	106,400	53,200	–	–

[a] The resources and fps reported are from [25]. The power is calculated using the implementation in https://github.com/nikkatsa7/HOG_Zedboard.git

Table 9 FPS results for proposed DSST architecture for 240 × 320 image

Unit	SS	SF	TS	TF	TS2	TF2
HOG	60	60	60	60	60	60
DFT	6153	6153	–	–	–	–
IDFT2	–	–	29.9	–	59.8	–
DFT3	–	–	86.52	43.26	173	86.52
SVD	–	–	–	248	–	248
QR	–	1474	–	–	–	–
Misc	4392	4392	4392	4392	4392	4392
Total	58.63	56.38	16.16	22.71	25.38	30.78

Table 10 Comparison against other implementations

Algorithm	Platform	Power	FPS
DSST [9]	Intel Xeon 2 core CPU	–	25.4
fDSST [9]	2.66 GHz 16GB RAM	–	54.3
RPCF [27]	Xilinx Soc Zynq dual core Cortex A9 + Artix7 FPGA	–	39.3
RADSST [28]	Intel i7 6700k CPU 64 GB	–	20
MOSSE [29]	Zynq US+ MPSoc (4k)	8.84 W	60
VDSP [30]	Vision DSP 4GB RAM	–	165
HLS	Zync Zedboard	1.92 W	25.4

corresponding stage. The critical stages are the TS and TF stages, as they involve DFT2 units. Using 2-parallel architecture for DFT2, the fps is improved. This is shown as TS2 and TF2 in Table 9. The most critical stage (stage with the minimum fps) dictates the mean fps of the whole architecture. Thus, for an image size of 320 × 240, it can fit in Zync Zed board with a frame rate of 25.38 fps. The maximum frequency is 108 MHz. Increasing the image size in the same FPGA would decrease fps. If frame size doubles, the fps is lowered by a factor of 1.5. With a different FPGA, the same fps can be achieved at the cost of higher resources. For the input and output images, (hls::Mat) from HLS is used. They are transformed into an 8-bit integer matrix for processing.

Table 10 shows the comparison of our whole HLS architecture against other tracker implementations. Our frame rate is equal to the original DSST [9], but it is less than the fDSST tracker. In terms of power HLS-based solution is much better as the fDSST runs on Xeon CPU and uses more resources. Another correlation filter-based tracker [27] reported is for IoT applications and is implemented for edge devices. It relies on the server for computations, which helps it speed up then our HLS based approach. Again in terms of power our solution is much better. The authors of Rotation Aware DSST tracker [28] use DSST but also integrate rotation awareness for accurate scale estimation. Our solution dominates both in terms of power and speed the RADSST [28]. MOSSE [29] is not DSST-based tracker but relies on programmable logic. It has a higher speed at 4K resolution, but it is reported to compare power. Again for power HLS solution is better. The best fps is of Moving Target Tracker MTT [30] which uses SIMD based DSP platform. It also uses 4GB of RAM. So in terms of power, resources and portability our solution is better. In comparison to these trackers, HLS based solution consumes less power. The speed is comparable to some of the trackers, but the fewer resources make it feasible to operate on the field.

5 Conclusions

In this paper, the RTL level implementation of the major blocks of Discriminative Scale Space Tracker (DSST) [9] is presented. The implementation is given in terms of the major mathematical operations involved including SVD, QR, DFT2 and HOG extractor. DFT is implemented by 8-parallel architecture; this is the base for DFT2 unit. This approach improves the timing by 92% with increased resources. For QR, the resource consumption is improved by a factor of 2.3 compared to [20]. For SVD, the timing is improved by a factor of nearly 3.8 compared to [22]. For an image size of 320 × 240 it is able to fit in Zync Zed board with a mean frame rate of 25.38 fps, thus can be operated as a stand-alone unit. Future research can be performed at further optimizing the operations involved. The effort can also be made to integrate the whole algorithm implementation and interface with a real camera to test it on the field. Finally, the overall accuracy can be compared to the databases available.

Funding Open access funding provided by Politecnico di Torino within the CRUI-CARE Agreement.

Declarations

Conflict of interest The authors have no conflicts of interest to declare and no funding was received for conducting this study.

References

1. Hare, S., Golodetz, S., Saffari, A., Vineet, V., Cheng, M.M., Hicks, S.L., Torr, P.H.S.: Struck: structured output tracking with kernels. IEEE Trans. Pattern Anal. Mach. Intell. **38**(10), 2096–2109 (2015)
2. Danelljan, M., Khan, F.S., Felsberg, M., Van de Weijer, J.: Adaptive color attributes for real-time visual tracking. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 1090–1097 (2014)

3. Varfolomieiev, A.: Channel-independent spatially regularized discriminative correlation filter for visual object tracking. J. Real-time Image Process. **18**, 1–11 (2020)
4. Jia, X., Lu, H., Yang, M.H.: Visual tracking via adaptive structural local sparse appearance model. In: 12th IEEE Conference on Computer Vision and Pattern Recognition, Providence, RI, USA, pp. 1822–1829 (2012)
5. Sevilla-Lara, L., Learned-Miller, E.: Distribution fields for tracking. In: 12th IEEE Conference on Computer Vision and Pattern Recognition, Providence, RI, USA, pp. 1910–1917 (2012)
6. Bolme, D.S., Beveridge, J.R., Draper, B.A., Lui, Y.M.: Visual object tracking using adaptive correlation filters. In: 2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, San Francisco, CA, USA, pp. 2544–2550 (2010)
7. Henriques, J.F., Caseiro, R., Martins, P., Batista, J.: Exploiting the circulant structure of tracking-by-detection with kernels. In: European Conference on Computer Vision, (Springer Berlin Heidelberg), pp. 702–715 (2012)
8. Böettger, T., Steger, C.: Accurate and robust tracking of rigid objects in real time. J. Real-time Image Proces. (2020). https://doi.org/10.1007/s11554-020-00978-9
9. Danelljan, M., Häger, G., Khan, F.S., Felsberg, M.: Discriminative scale space tracking. IEEE Trans. Pattern Anal. Mach. Intell. **39**(8), 1561–1575 (2016)
10. Li, C., Liu, X., Su, X., Zhang, B.: Robust kernelized correlation filter with scale adaption for real-time single object tracking. J. Real-time Image Process. **15**(3), 583–596 (2018)
11. El-Shafie, A.H.A., Habib, S.E.: Survey on hardware implementations of visual object trackers. IET Image Proc. **13**(6), 863–876 (2019)
12. Cooley, J.W., Tukey, J.W.: An algorithm for the machine calculation of complex Fourier series. Math. Comput. **19**(90), 297–307 (1965)
13. Shirbhate, R., Panse, T., Ralekar, C.: Design of parallel FFT architecture using Cooley Tukey algorithm. In: IEEE International Conference on Communications and Signal Processing (ICCSP), Melmaruvathur, pp. 574–578 (2015)
14. Awais, M., Vacca, M., Graziano, M., Masera, G.: FFT implementation using QCA. In: 19th IEEE International Conference on Electronics, Circuits, and Systems (ICECS), Seville, Spain, pp. 741–744 (2012)
15. De, D., Gumar, G.K., Ghosh, A., Saha, A.: FPGA implementation of discrete Fourier transform using CORDIC algorithm. Adv. Model. Anal. B **60**(2), 332–337 (2017)
16. Yu, C.L., Irick, K., Chakrabarti, C., Narayanan, V.: Multidimensional DFT IP generator for FPGA platforms. IEEE Trans. Circuits Syst. I Regul. Pap. **58**(4), 755–764 (2010)
17. Wu, J., Fang, S., Li, L., Yang, Y.: QR decomposition and gram Schmidt orthogonalization based low-complexity multi-user MIMO precoding. In: 10th International Conference on Wireless Communications, Networking and Mobile Computing (WiCOM), Beijing, China, IET, pp. 61–66 (2014)
18. Merchant, F., Vatwani, T., Chattopadhyay, A., Raha, S., Nandy, S., Narayan, R.: Achieving efficient QR factorization by algorithm-architecture co-design of householder transformation. In: 29th International Conference on VLSI Design (VLSID), pp. 98–103 (2016)
19. Fan, W., Alimohammad, A.: Givens rotation-based QR decomposition for MIMO systems. IET Commun. **11**(12), 1838–1845 (2017)
20. Muñoz, S.D., Hormigo, J.: High-throughput FPGA implementation of QR decomposition. IEEE Trans. Circuits Syst. II Express Br. II **62**(9), 861–865 (2015)
21. Shiri, A. and Khosroshahi, G.K.: An FPGA implementation of singular value decomposition. In: 27th Iranian Conference on Electrical Engineering (ICEE), Yazd, Iran, pp. 416–422 (2019)
22. Mohanty, R., Anirudh, G., Pradhan, T., Kabi, B., Routray, A.: Design and performance analysis of fixed-point Jacobi SVD algorithm on reconfigurable system. IERI Procedia. **7**, 21–27 (2014)
23. Ngo, V., Castells-Rufas, D., Casadevall, A., Codina, M., Carrabina, J.: Low-power pedestrian detection system on FPGA. In: Multidisciplinary Digital Publishing Institute Proceedings, vol. 31, p. 35 (2019)
24. Bourrasset, C., Maggiani, L., Sérot, J., Berry, F.: Dataflow object detection system for FPGA-based smart camera. IET Circuits Devices Syst. **10**(4), 280–291 (2016)
25. Hahnle, M., Saxen, F., Hisung, M., Brunsmann, U., Doll, K.: FPGA-based real-time pedestrian detection on high-resolution images. In: IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPR), pp. 629–635 (2013)
26. Xilinx, VH.: Vivado design suite user guide-high-level synthesis. UG902 (v2020.1) (2020)
27. Zhang, H., Zhang, Z., Zhang, L., Yang, Y., Kang, Q., Sun, D.: Object tracking for a smart city using IoT and edge computing. Sensors **19**(9), 1987 (2019)
28. Marvasti, Z., Seyed, M., Ghanei-Yakhdan, H., Kasaei, S.: Rotation-aware discriminative scale space tracking. In: Iranian Conference on Electrical Engineering (ICEE), pp. 1272–1276 (2019)
29. Kowalczyk, M., Przewlocka, D., Kryjak, T.: Real-time implementation of adaptive correlation filter tracking for 4K video stream in Zynq UltraScale+ MPSoC. In: Conference on Design and Architectures for Signal and Image Processing (DASIP), pp. 53–58 (2019)
30. Gong, X., Le, Z., Wang, H., Wu, Y.: Study on the moving target tracking based on vision DSP. Sensors **20**(22), 6494 (2020)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Walid Walid received his B.Sc. (with distinction) and M.Sc. (summa cum laude) degrees in Electronic Engineering from University of Engineering and Technology Taxila, Pakistan (2015) and Politecnico di Torino, Italy, (2019) respectively. He is currently a Ph.D. student at VLSI lab, Department of Electronics and Telecommunications, Politecnico di Torino, Italy. His research interests include VLSI architectures for multi-core processor design. He is currently working on high performance real-time design using ASIC/FPGAs in the domain of object tracking and computer vision.

Muhammad Awais completed his MS and PhD degrees in 2010 and 2014 respectively from VLSI Lab, Department of Electronics and Telecommunications, Politecnico di Torino Italy. Since then, he is serving at Department of Electrical and Computer Engineering, COMSATS University Islamabad, Wah Campus, Pakistan. His research interests include high performance digital design using ASIC/FPGAs of various applications from the domain of communications, signal processing and computer vision, and applied optimization for resource allocation in next generation wireless communications.

Ashfaq Ahmed received the M.Sc. and Ph.D. Degree from the Department of Electronics and Telecommunications, Politecnico di Torino, Torino, Italy, in 2010 and 2014, respectively. He is currently affiliated with the center for cyber-physical systems, Department of Electrical Engineering and Computer Science, Khalifa University, Abu Dhabi, UAE, and simultaneously working as an Assistant Professor at the Department of Electrical and Computer Engineering, COMSATS University Islamabad, Wah campus, Pakistan. His current research interests include the topics related to applied optimization, targeting telecommunication and smart system applications, ASICs and FPGA design and implementation of digital signal processing architectures, especially the areas closely related to digital communications, and video and image coding.

Guido Masera received the Dr.-Ing. (summa cum laude) and Ph.D. degrees in Electronic Engineering from Politecnico di Torino, Italy. He is a Professor with the Electronic Department, Politecnico di Torino, since 1992. His research interests include several aspects in the design of digital integrated circuits and systems, with a special emphasis on high-performance architectures for communications, forward error correction, image and video coding, cryptography and hardware accelerators for machine learning. He has more than 200 publications, two patents and was a designer of several ASIC components. Dr. Masera is an Associate Editor of MDPI Electronics and a former Associate Editor of the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS I, IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS II and the IET Circuits, Devices and Systems.

Maurizio Martina received the M.S. and Ph.D. in electronic engineering from Politecnico di Torino, Italy, in 2000 and 2004, respectively. He is currently an Associate Professor of the VLSI-Lab group, Politecnico di Torino. His research interests include VLSI design and implementation of computer architectures and specialized architectures for digital signal processing, video coding, communications, artificial intelligence, machine learning and event-based processing. He has more than 100 scientific publications, 3 book chapters, 1 book and 2 patents.