Politecnico di Torino

Department of Electronic Engineering

Real Time Implementation of Discriminative Scale Space Tracker (DSST) Algorithm

THESIS

Master degree in Electronic Engineering

Supervisors: Prof. Maurizio Martina and Prof. Guido Masera

Author:

WALID WALID(252665)

October 13, 2019

Abstract

Real-time on field scale estimation of a target with accuracy is a research problem in visual tracking. It is important due to its applications in domains like surveillance, robotics and automation. In general, only the initial position of the object is known, and the trajectory of the object motion is desired to be traced. It becomes further difficult due to problems like fast motion of the objects, motion blur, size and scale variations. Existing algorithms estimate the size of target based on exhaustive scale search which faces problems when scale varies a lot and involves expensive computation. They learn size or appearance model of target by using generative or discriminative approach. This model is then utilized to estimate the state target in a new frame. Typically, the model is evaluated at multiple resolutions by an exhaustive scale search, hence computationally expensive.

To tackle these problems, we used an algorithm which relies on online explicit filtering based target sampling at different scales. This proposes an alternative, discriminative approach for scale adaptive visual tracking. Separate scale filters are used for scale estimation and translation by Discriminative Scale Space Tracker (DSST). To reduce the computational cost Fast Fourier Transform (FFT) and pointwise operations are used, hence the fast Discriminative Scale Space Tracker (fDSST) is developed.

This thesis aims at FPGA or hardware based implementation of fast discriminative scale space tracking algorithm. FPGAs are used for many DSP applications. They are fast prototyping devices for real-time applications. The hardware created on FPGA will be fast and use resources to a minimum to decrease the cost and will be optimized based on different matrices like performance-cost, power-performance and cost-power etc.

For this purpose, the major mathematical blocks in this algorithm are studied and their best implementation approach for this algorithm is described. The best implementation approach is given in terms of the complexity and dimensions of inputs involve, less area and fast operation. The whole algorithm is also depicted with the step by step operations involved, to better understand the decisions. The implementation strategies and the implemented blocks are implemented in VIVADO HLS tool, which is a tool for high level synthesis. It is suggested to keep the area of target image at maximum, half of the frame size for good performance. Also, the synthesized version of Discrete Fourier Transform hardware is depicted with simulating on real image data.

Dedication

This work is dedicated to my parents. Especially I would dedicate this to my mother, for it was her foremost desire to see me succeed and achieve higher level of education. I wish them a long and healthy life.

Acknowledgements

In the beginning I would like to thank Almighty Allah (God) for all the things He blessed me with and strengthen me to achieve what I am achieving. I am very thankful to my supervisors Prof. Maurizio Martina and Prof. Guido Masera for their support and professional guidance during the course of this thesis work. It has been really very inspiring work under their supervision. Especially I am thankful to them for their input at my work and deciding the direction of my work. Alongside, I would like to thank the Department of Electronics of Politecnico di Torino for providing good facilities and online support of material for my thesis work. I wish to give my gratitude and thanks to my family and friends for their support.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

Image processing is used in vast varieties of applications. It extends from simple image enhancement algorithms to face detection algorithms. From object detection or identification to more complex algorithms. Based on this it has a vast domain of research areas. One active research area of Image processing is in the domain of visual object tracking. In a lot of applications like surveillance, robotics, automation and other security purposes it is important to have sophisticated algorithms for safety and reliability. As these fields progress it is necessary to have hardware support for these algorithms to be implemented in real world. The more sophisticated and complex an algorithm becomes, it requires more computations. Which in turn mean huge burden on hardware. Therefore there is need of efficient and smart hardware implementation of these algorithms.

The research in Visual object tracking algorithms have mostly been done on two major categories. Namely generative or discriminative approaches. Before looking into these approaches let's first get an insight of what Visual object tracking means. Usually, in most of the cases Visual object tracking means given an initial position/location of target object in a frame, one now has to trace the next location of target in a sequences of other frames. As the locations of target changes it can move in the horizontal or vertical directions along with motion along the camera axis. There are some difficulties in connection with it. Some of these are variations in target appearance, occlusions[33] shown in figure 2.2, fast motion of target, target motion blur and scale variations of the target. They make the detection process more complicated.

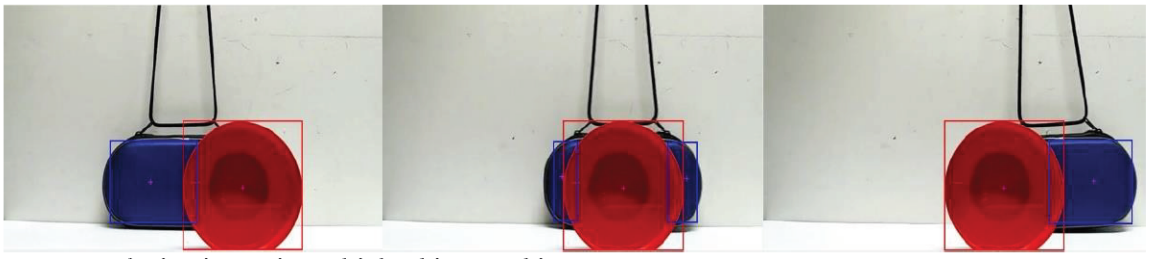


Figure 1.1: Occlusions

Keeping these complications in mind, tracking the object requires an appearance model of the target. This can be learned by discriminative approach[15],[23],[24] or generative approach[30],[13],[3]. By using either generative or discriminative approach the appearance model of target is developed, which is than put into work for estimating state of the target in the new frame. Usually the state only involves the vertical and horizontal position of the target object in the frame i.e, its initial position. Along with this position, the target size is also important in many of the applications. In Surveillance

cameras and robotics it is important to compute and estimate the size of target along with its position. The target size changes due to the motion of target along the axis of camera or the changes that occur in appearance of the target. To take this into account, the scale has to be estimated. Correct and precise estimation of the changes in scale is a complicated problem which is further affected by other factors like fast motion, occlusions and variations in illumination.

A simple and direct method for implementing scale estimation of an object tracking scenario is to compute the appearance model of target at different resolutions, by computing an extensive scale search approach. But, this brute-force strategy, for scale search, is computation hungry and in case hardware implementation poses limits on speed. In applications involving real-time, efficiency in terms of computation is a vital factor to be considered. Thus, an ideal strategy for tracking better be robust for handling variations in scale while operating in real-time. The paper[18] on which this work is based on, investigates the issue of precise scale adaptive object tracking with focus on the real-time performance.

Before going into the implementation used by[18], lets first look at the two approaches mentioned earlier that is discriminative and generative.

The Generative models are based on describing the appearance of target by utilising statistical models or templates. Thus, it does require constructing a template appearance model and than using it to search the next frame. Hence it has a high computational cost for implementing in hardware. On the other hand the discriminative approach uses methods of machine learning to distinguish between the target object appearance and the surrounding background. Thus, thw Discriminative approach tries work by only riling on the given data while learning the classification from the observed statistics. This makes it less computation hungry with respect to generative approach. Examples containing learning approaches are boosting techniques and Support Vector Machines (SVM).

In the recent past, visual trackers that are based on discriminative correlation filter (DCF) [18], [15], [3] have depicted excellent result in terms of performance. Moreover, these trackers are efficient in terms of computation which is a very good advantage, thereby making them a good approach for many real-time applications. Hence really good for hardware implementation. Their success can be seen from the Visual Object Tracking (VOT) Challenge (2014) [17], in which the first 3 visual trackers were based on correlation filters. Also methods involving DCF [15], [10] have depicted good results on OTB data set [32]. This is also true while working at speed of over 100 frames per second(fps). First they learn an optimal correlation filter that is then utilized to find the target in a new frame sequence. The real advantage in speed(fps) is achieved by using the fast Fourier transform (FFT). This is applied both at the detection and learning stages. The approaches that use DCF for visual object tracking has emphasis on translation estimation. A paper[18] utilizes DCF based approach for visual tracking while incorporating real time scale adaptation. Hence suitable for such hardware.

Two kind of search strategies are considered for taking into account the target scale estimation in discriminative correlation filter(DCF) based visual object tracking. First method does the joint estimation of target scale and translation by utilising a 3 dimensional correlation filter. While the other strategy uses a 2 dimensional correlation filter at different resolution. Thus both being highly computational demanding are not a good option for the real-time object tracking. This work utilises another approach using discriminative filter for scale adaptive object tracking based on [18] called fast Discriminative Scale Space Tracker(fDSST).

The paper[18] firstly, suggests the discriminative scale space tracker (DSST), which learns different discriminative correlation filters (DCF) for target scale estimation and target translation. For learning scale filter, samples of target at different scales are considered. But before scale estimation, translation estimation is preformed. In new frame the visual target translation is predicted by using a standard filter for translation. Than for estimating the accurate target size, the learned scale filter is applied at the new target location. Thus discriminative scale space tracker[18] in comparison with other exhaustive scale search strategies reduces the search space and learns the change in appearance of

target, by changes induced due to variations in object size. Secondly, authors [18] suggest the fast discriminative scale space tracker (fDSST) by improving the computational cost, increasing the search space and hence making it more robust and accurate. Thus making it more suitable for real-time applications. The implementation of current algorithm is considered in this work for targeting a FPGA.

The paper[18] validates its approach by, a comprehensive evaluation done on Online Tracking Benchmark (OTB) data set [32], having fifty videos. The results are evaluated on the VOT challenge 2014 [17]. Qualitative and quantitative experiments were executed. This approach refines the Discriminative Correlation Filter methods for scale search, both for speed and accuracy. This DCF tracker overcomes nineteen approaches on OTB data set. It is further shown to achieve high rank on VOT 2014 data set by leaving behind 37 trackers.

This work mainly focuses on hardware implementation of [18].It discusses the major blocks which are needed for the implementation of the algorithm. For the implementation of these block Vivado HLS 2016.1 is used. The major mathematical blocks in this algorithm are studied and their best implementation approach for this algorithm is described. The best implementation approach is given in terms of the complexity and Dimensions of inputs involve, less area and fast operation. The whole algorithm is also depicted with the step by step operations involved to better understand the decisions. It is suggested to keep the area of target image at max half of the frame size for good performance. Also, the synthesized version of Discrete Fourier Transform hardware is depicted.

Major blocks involve Discrete Time Fourier Transform(DFT), Singular value decomposition(SVD) to find eigenvalues and eigenvectors, QR factorization, image resizing, Hann windowing, image integral, fhog features. This work suggests the suitable methodology for their implementation. With main focus on less area to be able to fit in an FPGA meanwhile achieving latency for operating in real time.

The remaining of this thesis is organized as follows. Chapter 2 gives an overview of the literature review of work most related to this work. In Chapter 3 we introduce our methodology. Chapter 4 to 8 discusses the best approaches for implementation of the major blocks of the algorithm, i.e, DFT, SVD, QR, Resizing image, Hann windowing, integral image. Chapter 9 gives the best approaches for implementation of the minor blocks of the algorithm. Finally the conclusion is given in Chapter 10.

CHAPTER 2

Literature Review

2.1 Object Tracking

2.1.1 Visual object tracking

In computer vision Visual object tracking is a basic research problem. In most of the cases, Visual object tracking means given an initial position of target in a frame, one has to trace the location of the target in a sequences of frames. In general visual tracking, the target to be tracked can be any item/object and is only defined by its initial position. Therefore the only information available to the tracker is its position. Since it has this generic nature many applications exist for it. Mostly it includes surveillance cameras [11] robotics [14], and road view understanding [1].

Given the initial frame than the tracking methods involve creating an appearance model for the target from the initial frame information. This task can be achieved by either of the two methods.

- Generative Methods
- Discriminative Methods

Generative Methods

The Generative appearance models are based on describing the appearance of target by utilising statistical models or templates. This does require constructing a template appearance model and than using it to search the next frame. Thus it has a high computational cost for implementing in hardware.

Discriminative Methods

Discriminative approach use machine learning methods to distinguish between the target appearance and the surrounding back-ground. Discriminative approach work by only riling on the given data while learning the classification from the observed statistics. This is less hungry for computations with respect to generative approach. Examples of involved learning approaches are Support Vector Machines (SVM) and boosting techniques.

Based on this Discriminative approach seems more feasible for hardware implementation than generative approaches.

2.1.2 Discriminative Correlation Filters

The discriminative correlation Filter is an algorithm. It trains a template to distinguish between an image and its translation. It is feasible for object tracking because of its formulation in Fourier transform domain provides a fast solution, allowing the tracker to be re-trained one time per frame.

In Recent years, Discriminative Correlation Filters(DCFs) have successfully been employed for visual object tracking [15], [3], [10], [16]. In real-time operation, these approaches have shown to give excellent performance on benchmark tracking data sets [17], [32]. These trackers learn a discriminative correlation filter for target estimation in a new frame.

Author of [3] trained the filter on a set of sample gray-scale patches by minimizing the total squared error between the desired correlation and the actual output. The authors showed that with the help of circular correlation, the filter can be computed using just FFTs and point wise operations thus making it efficient. Author [9] further showed that the formulated Discriminative Correlation Filter can equally be casted as learning a least squares regression or ridge regression on the set of all cyclic shifts of the involved sample patches. This formulation than led to the introduction of fast kernelized correlation filters.

2.1.3 Multi-channel Discriminative Correlation Filters

In recent, several works have utilized generalizations of the Discriminative Correlation Filter approach [3] for multidimensional features [7], [28], [8]. Given the set of training samples they learn an exact multi-channel filter.

The authors of [18] show generic approach for Multi-channel Discriminative Correlation Filters. From target appearance a single sample f is considered for learning a multi-channel correlation filter. In the general scenario f is an image patch that is centered around the target object. This is used for learning a 2 dimensional correlation filter for computing the target translation. Normally, domain of f has arbitrary dimension, which is very useful. This means that the same technique can be used for one - dimensional, two- dimensional or even three- dimensional filters. One-dimensional filter is for scale estimation, two-dimensional filter is for translation estimation and three-dimensional is for combined scale and translation estimation. To achieve this, for each case, the feature extraction step needs to be adapted. The authors of [18] further explore that the sample of object target f, at each location n in a rectangular domain, comprises of a d-dimensional feature vector f(n) $\in R^{\mathrm{d}}$.

For the translation case the gray scale intensity value/ RGB value at every pixel position of the patch is considered. In general usually, any grid-based feature representation can be considered. The feature channel l $\in \{1, 2, ..., d\}$of image patch f is denoted by f^{l}.

The goal then is to learn a correlation filter h for each feature channel comprising of one filter h^{l} . This is accomplished by minimizing the L^2 of correlation response error compared to the required correlation output g,

$$\epsilon = ||g - \sum_{l=1}^{d} h^l * f^l||^2 + \lambda \sum_{l=1}^{d} ||h^l||^2 \tag{2.1}$$

where * means circular correlation, g is the required correlation output which usually is a Gaussian function having its standard deviation parametrized [3]. Lambda is weighting parameter of the second regularization term in Eq (2.1). Also it is important to see that the all terms h^{l}, f^{l}, and g poses equal sizes and dimensions. The authors[18] further suggested that Eq (2.1) is a linear least square problem. Parseval's formula can be used to solve it efficiently by transforming Eq (2.1) to the Fourier analysis domain. Than, the filter that minimizes Eq (2.1) is given by,

$$H^l = \frac{\overline{G}F^l}{\sum_{k=1}^{d} \overline{F^k}F^k + \lambda} \tag{2.2}$$

where,

$$l \in \{1, 2, ..., d\}$$

and the Upper case letters means the discrete Fourier transform (DFT) of the terms. The symbol bar means complex conjugation of corresponding quantities. The divisions and multiplications in Eq(2.2) are not matrix divisions and multiplications but are point wise.

Given a sample f for training of target Eq (2.2) gives the optimal filter h. In practice robust correlation filter h is learned by taking many samples $(f_j)^t{}_1$ at different instances of time, which can be accomplished by taking the mean of the the correlation error in Eq (2.1) of all training samples f_1,...,f_t. As depicted by author in [7], by using Discrete Time Fourier Transform the least squares problem which is linear can be block diagonalized. As a result N number of d * d linear systems can be solved to find the final H, where N being the elements in the filter h^l. But, for the real-time online learning this will result in a bottleneck computationally. Hence, the authors[18] consider the exact solution of Eq (2.2) for one training sample to get a robust approximation. Authors[18] use the update rule of [3] for single feature i.e (d = 1) to update numerator and denominator of the filter for a sample f_t. Let the current numerator $A_t{}^l$ and the current denominator be $B_t{}^l$, than the update is given as,

$$A_t^l = (1 - \eta)A_{t-1}^l + \eta\overline{G}F_t^l \tag{2.3}$$

$$B_t = (1 - \eta)B_{t-1} + \eta \sum_{k=1}^{d} \overline{F_t^k} F_t^k \tag{2.4}$$

where, the scalar

$$\eta$$

is a learning rate parameter. To apply the filter in the next frame t, a sample z_t is taken from the region of transformation considered. Where in the standard case, z_t is an image patch centered around the estimated target position. The sample z_t is taken similarly to the other training samples f_t, considering the same representation of features. The Discrete Time Fourier Transform(DFT) of the correlation scores y_t is computed in the Fourier domain

$$Y_t = \frac{\sum_{l=1}^{d} \overline{A_{t-1}^l} Z_t^l}{B_{t-1} + \lambda} \tag{2.5}$$

To compute correlation scores at the positions reflected in z_t the inverse Discrete Time Fourier Transform(IDFT) is taken as

$$y_t = F^{-1}\{Y_t\}$$

The prediction of the current state of target is obtained by taking the maximum of the correlation scores.

2.1.4 Correlation filter for translation estimation

For tracking involving only translation first learn a 2-dimensional multi channel Discriminative Correlation Filter(DCF). For this in a frame number t, firstly the target location is given. Then extract a sample training patch f_t centered around the object target. The translation filter numerator and denominator are then updated using Eq (2.3) and Eq (2.4). Then to compute the target position in a new frame t, a sample patch z_t is taken from the previously computed location. The correlation scores are taken from Eq (2.5).

2.1.5 Correlation filter for scale estimation

In visual object detection an approach for detection of an object at variety of scales is to apply the filter at multiple resolutions [21]. This approach is used for the standard DCF-based tracker [31]. Same strategy as previous section is utilized. In the detection step, many patches are sampled at different resolutions centered around the last target position. The translation filter is then used for each sample patch independently by Eq (2.5). Both the scale and translation of the desired target is obtained by taking the resolution or scale and position with the highest correlation score considering all the patches.

2.1.6 Joint Scale Space Filter

A direct approach for taking into account scale estimation is to create a 3-dimensional scale space filter. This filter can estimate jointly the scale and translation of the target. It is done by estimating the correlation scores of a box-shaped(rectangular cuboid) area of a scale pyramid representation. Both the scale and translation estimates are computed by maximizing the correlation score. The update of this joint scale space filter is done as follows.

- First create a feature pyramid in a rectangular region centered around the desired target location such that the object size at current resolution belongs to the spatial filter dimension i.e M * N.
- f_t, the sample for training, is set to the box or rectangular cuboid of dimensions M * N * S around the object scale and position. S being the size of filter in the dimension of scale.
- This filter is than updated for numerator and denominator using Eq (2.3) and Eq (2.4). In this case Gaussian function for the desired correlation output is taken to be three-dimensional.
- After updating the filter, in the detection stage, create a feature pyramid centered around the previously computed target scale and position.
- This rectangular cuboid of dimension M * N * S around the location is used to extract z_t.
- Finally using Eq (2.5) the correlation scores for this scale space region are computed.

However, these approaches are not directly applicable to the real-time object tracking case due to a significant amount of the computational cost. On the other hand, approximate formulations for multi-channel filters learning are investigated [15], [10]. These methods are robust and they scale linearly with the number of feature channels. Also, to reduce the cost author of [15] introduced a technique for adaptive feature dimensionality reduction while preserving performance of the tracker. The DCF based methods have shown the capability of precise target localization in many challenging situations.

The authors[18] aim directly at learning the change in appearance induced by scale variations. This allows to track at a higher frame-rate while achieving accurate scale adaptive tracking. This is described in the next section of Discriminative Scale Space Tracker(DSST). In section 2.1.9 DSST algorithm is analyzed for techniques to reduce its computational cost. This allows not only to increase the robustness of tracker by increasing the target search area but also it doesn't sacrificing real-time performance hence very good solution to be implemented in hardware. These improvements result in higher tracking performance with twice the speedup.

2.1.7 Discriminative Scale Space Tracker(DSST)

The authors[18] propose the Discriminative Scale Space Tracker. The idea is that to use separate filters for scale and translation. It learns a separate one-dimensional correlation filter for scale. This

scale filter is applied at a target location in an image to compute correlation scores in scale dimension. These correlation scores are then utilized to predict the scale of target. To create $f_{t,scale}$, the training sample, features are extracted using different patch sizes centered around the target location. Assume, P * R denotes the size of target in the present frame and let S be the scale filter size. For each

$$n \in \left\{ \left[-\frac{S-1}{2} \right], ..., \left[-\frac{S+1}{2} \right] \right\}$$

an image patch I_n is extracted with a size of a^n P X a^n R around the target position. where, a represents the scale factor between the feature layers. The sample $f_{t,scale}$ for each scale level n has the value $f_{t,scale}(n)$ which is set to the d-dimensional feature descriptor of I_n. In the end of, Eq (2.3) and Eq (2.4) are used for updating the scale filter $h_{t,scale}$ with the new sample $f_{t,scale}$. As expected a one-dimensional Gaussian is used for the desired correlation output g.

For estimating the target translation, the standard translation filter is used. Usually, the scale target difference between two consecutive frames is less compared to the difference in target translation. Hence translation filter $h_{t,scale}$ is applied first for a new frame. Then the scale filter $h_{t,trans}$ is applied at the new estimated target location. $z_{t,trans}$. Same procedure is used for extracting the scale estimation of test sample $z_{t,scale}$. Finally, by maximizing the scale correlation scores using Eq (2.5), the relative difference in scale from previous is obtained. The algorithm is depicted in figure 2.1 of the DSST [18].

2.1.8 Fast Discriminative Scale Space Tracker(FDSST)

The authors[18] suggest strategies for reduction of the computational cost of the DSST method. Two methods for reducing the number of computations in the learning and detection steps of the multi-channel DCF are described.

- Sub grid based interpolation of correlation scores
- Feature dimensionality reduction using Principal Component Analysis (PCA)

Sub grid based interpolation of correlation scores

Sub-grid interpolation is used for creating coarser feature grids for the samples of detection and training. This decreases the computational cost by the reduction in the size of the performed DFTs required to compute Eq (2.3),(2.4) and (2.5) for detection and training respectively. The interpolation is done with trigonometric polynomials[25]. This is feasible since the coefficients of DFT for the correlation score, required for implementing the interpolation, are available from in Eq (2.5). The interpolated scores y_t" are obtained by zero-padding the high frequencies of Y_t in Eq(2.5) keeping its size equal to that of interpolation the grid. The interpolated scores y_t" are then obtained by taking the inverse DFT of the padded Y_t.

Feature dimensionality reduction

Most of the computational cost of the DSST is due to the DFT. Since equation (2.3) (2.4 and (2.5) requires one DFT per feature dimension, as a result the number of DFT computations scale linearly with the feature dimension d. Hence a dimensionality reduction technique need to used to decrease the number of DFT computations. The dimensionality method is based on the standard Principle Component Analysis (PCA).

For reducing the number of DFT computations, the update is to be done on a target template instead given by,

$$u_t = (1 - \eta)u_{t-1} + \eta f_t$$

Input:

Image I_t.
Previous target position $\boldsymbol{p}_{t-1}$ and scale s_{t-1}.
Translation model $A_{t-1,\text{trans}}$, $B_{t-1,\text{trans}}$.
Scale model $A_{t-1,\text{scale}}$, $B_{t-1,\text{scale}}$.

Output:

Estimated target position $\boldsymbol{p}_t$ and scale s_t.
Updated translation model $A_{t,\text{trans}}$, $B_{t,\text{trans}}$.
Updated scale model $A_{t,\text{scale}}$, $B_{t,\text{scale}}$.

Translation estimation:

1: Extract sample $z_{t,\text{trans}}$ from I_t at $\boldsymbol{p}_{t-1}$ and s_{t-1}.
2: Compute correlation scores $y_{t,\text{trans}}$ using (4).
3: Set $\boldsymbol{p}_t$ to the target position that maximizes $y_{t,\text{trans}}$.

Scale estimation:

4: Extract sample $z_{t,\text{scale}}$ from I_t at $\boldsymbol{p}_t$ and s_{t-1}.
5: Compute correlation scores $y_{t,\text{scale}}$ using (4).
6: Set s_t to the target scale that maximizes $y_{t,\text{scale}}$.

Model update:

7: Extract samples $f_{t,\text{trans}}$ and $f_{t,\text{scale}}$ from I_t at $\boldsymbol{p}_t$ and s_t.
8: Update the translation model $A_{t,\text{trans}}$, $B_{t,\text{trans}}$ using (3).
9: Update the scale model $A_{t,\text{scale}}$, $B_{t,\text{scale}}$ using (3).

Figure 2.1: Algorithm of DSST taken from paper [18]

By the linearity property of the Fourier transform, the numerator Eq(2.3) of the learned filter can then equivalently be obtained by

$$A_t^l = \overline{G} F\{u_t^l\}$$

A projection matrix P_t is constructed by the learned template u_t. For projection of the low-dimensional features on to the low-dimensional subspace the projection matrix is used. The projection matrix P_t is d' * d, where d means the dimensionality of the compressed feature representation. P_t is obtained by minimizing the reconstruction error of the target template u_t

$$\epsilon = \sum_n \|u_t(n) - P_t^T P_t u_t(n)\|^2 \tag{2.6}$$

where, n, the index tuple, ranges over all elements in the template u_t. Eq (2.6) ut. Eq. (5) is minimized under the orthonormality constraint

$$P_t P_t^T = I$$

It can be solved by performing an eigenvalue decomposition (SVD) of the auto-correlation matrix

$$C_t = \sum_n u_t(n) u_t^T(n) \tag{2.7}$$

The rows of projection matrix are set to the d' eigen-vectors of C_t corresponding to the largest eigenvalues.

The compressed training sample

$$F_t' = Four\{P_t F_t\}$$

and the compressed target template

$$U_t' = F\{P_t u_t\}$$

are used to update the filter as the

$$A_t^{l\prime} = \overline{G} U_t^{l\prime} \tag{2.8}$$

$$B_t' = (1-\eta) B_{t-1}' + \eta \sum_{k=1} d' \overline{F_t^{k\prime}} F_t^{k\prime} \tag{2.9}$$

The linear operation of projection matrix is applied as an element-wise matrix multiplication

$$(P_t u_t)(n) = P_t u_t(n)$$

that projects the feature vector $u_t(n)$ R^d onto the rows of projection matrix. The equation (2.8) can be directly obtained from $U_t{}^{l\prime}$ because it linearly depends on f_t. Same cannot be said about Equation (2.9) since it is dependent on the auto-correlation of the samples. So a different projection matrix is required for the denominator. A good approximation of B_t is obtained by using the projection matrix for every frame t and using different projections for each term

$$\sum_{k=1} d' \overline{F_t^{k\prime}} F_t^{k\prime}$$

By applying the filter on the compressed sample

$$Z_t' = F\{P_{t-1} z_t\}$$

the correlation scores at the test sample z_t are obtained as

$$Y_t = \frac{\sum_{l=1} d' \overline{A_{t-1}^{l}{}'} Z_t^{l\prime}}{B_{t-1}' + \lambda} \tag{2.10}$$

Compressed scale filter

The number of elements $u_{t,trans}(n)$ in the template are greater than feature dimensionality for the translation filter used in DSST. On the other hand, for the scale filter, the reverse is true. In that scenario the rank of the auto correlation matrix from Eq(2.7) is smaller than or equal to the number of scales, i.e., $\text{rank}(C_{t,scale}) \leq S$.

Thus, without losing any information, the scale template $u_{t,scale}$ can be compressed to d' = S feature dimensions. The same is true for the training sample $f_{t,scale}$ of the scale filter.

This dimensionality reduction method is used for the scale filter by employing the following properties. Two projection matrices are computed and used to compress the sample and template without losing any information. Filter is updated by using these compressed versions. By linearity of Fourier transform the scale filter in Fourier domain is not affected by the dimensionality reduction since it can be fully reconstructed. By using a compressed test sample

$$z'_{t,scale} = P^{u}_{t-1,scale} z_{t,scale}$$

equation (2.10) is applied to estimate the scale correlations scores for the detection stage.

For less computational and memory efficiency, the auto-correlation matrices are not explicitly calculated but rather obtain the projection matrices $P_{t,scale}{}^{u}$ and $P_{f,scale}{}^{u}$ through a QR-factorization of $u_{t,scale}$ and $f_{t,scale}$ respectively. This does not affect the tracking output since the multi channel DCF approach is invariant to any change of orthonormal basis in the feature representation.

2.1.9 Mathematical operations

As this thesis involves a discussion on hardware implementation of the fDSST [18] algorithm. It is necessary to understand the involved mathematical operations in this algorithm, their nature, functionality and complexity. For this purpose this section deals with the introduction of these mathematical operations as well as shedding some light on their implementation. The major mathematical operations involved in the algorithms[18] include Singular value decomposition(SVD), QR Factorization, Discrete Fourier Transform(DFT), Fast Fourier Transform(FFT), and Inverse Discrete Fourier Transform(IDFT), FFT2, Image resizing, Hanning window, FHOG, circular shifting, Matrix multiplication and matrix transpose.

QR Factorization

QR factorization, also named as a QR decomposition or QU factorization is a form decomposition of a matrix B (B = QR) into a matrix product of QR i.e of an upper triangular matrix R and an orthogonal matrix Q. QR factorization is mostly used to solve the linear least squares problem and is the basis for a particular eigenvalue algorithm. After the QR factorization the two matrices created have these properties. Q^{T} x Q = I the identity matrix. If the dimension B is M x N then Q has the dimension of M x M and R has dimension of M x N.

It is implemented using Gram–Schmidt process. In this process the columns of the matrix are used. Let x1, x2, x3 denote the columns of the original matrices. Projection is found using

$$Proj_u x = \frac{u.x}{u.u} u$$

where "." represents inner product. In case of complex numbers the first term is conjugated. Than the Q matrix can be found by finding orthonormal columns of A from each other by using the projections as,

$$u_j = a_i - \sum_{i=1}^{j-1} Proj_{u_i} a_j$$

where u_i divided by its length make the columns of Q matrix. To find matrix R dot product of u_i (divided by its length) and column of original matrix is required.

Singular value decomposition(SVD)

The singular value decomposition of a M * N matrix of a matrix A is a factorization of the form USV. Here U is M * M real or complex matrix, S is M * N diagonal matrix with real non-negative values

and V is N * N unitary matrix. The diagonal elements of the matrix S are the singular values of A, while the columns of matrices U and V are left singular vectors and right singular vectors of A. The vector V is in transposed form.

The singular value decomposition can be calculated by using the following properties:

- The left singular vectors of matrix A are orthonormal eigenvectors of A * conjugate(A).
- The right singular vectors of matrix A are orthonormal eigenvectors of conjugate(A) * A.
- The non-negative singular values of matrix A i.e, the values on the diagonal of matrix S are the square roots of the eigenvalues of both conjugate(A) * A and A * conjugate(A).

Discrete Fourier Transform(DFT)

The discrete Fourier transform (DFT) transforms a series of equally-placed samples of a function into an exact length series of equally-placed samples of the discrete time Fourier transform (DTFT), which is a complex function of frequency. The length of the DTFT is the reciprocal of the duration of the input series.

It is calculated as

$$X_k = \sum_{n=0}^{N-1} x_n . e^{-\frac{j2\pi}{N} kn}$$

Inverse Discrete Fourier Transform(IDFT)

The inverse discrete Fourier transform (DFT) is a Fourier series. This uses the DTFT samples as coefficients of the sinusoids at the corresponding DTFT frequencies. It has the same equally-placed samples as the input series.

It is calculated as,

$$x_k = \frac{1}{N} \sum_{n=0}^{N-1} X_n . e^{\frac{j2\pi}{N} kn}$$

Discrete Fourier Transform2(DFT2)

This is the DFT of a matrix. First of all it takes the columns of matrix and calculates DFT. Than it takes the transpose of the result. It again performs the DFT of columns (hence this times rows).

Fast Fourier Transform(FFT)

This is the DFT of a matrix. This uses a technique called butterfly approach to calculate DFT in a fast manner for implementation.

Hanning window

In digital signal processing a window function is a function of mathematics that is zero outside of allocated interval, usually symmetric around the center of the interval. When another function is "multiplied" by a window function, the product is also zero-valued outside of allocated interval, except the overlap part. Hann window is one such window. It is also called cos window. It is used to realize Hann smoothing. It gives coefficients which are multiplied by the input series.

The coefficients are calculated as,

$$W_n = 0.5[1 - cos(\frac{2\pi n}{N})]$$

the length of the window is N + 1.

Image resizing

Resizing of an image means changing the size or its dimension, it can be the width only or the height alone or both of them. But, the aspect ratio of the input original image would be preserved i.e, not changed in the resized image. This is done with the help of interpolation of some sort. Most common is bilinear interpolation. A bilinear interpolation or INTER_LINEAR interpolation is an extension of linear interpolation for 2-dimensional rectangular interpolation.

The main idea is to do the linear interpolation first in one direction, and then in the other direction. Although each step on its on is linear, the interpolation as a whole is not a linear interpolation but a quadratic one.

In this interpolation method linear interpolation is utilised to get smooth scaling based on Neighbouring pixels of the image.

Linear Interpolation

This is an approach to approximate a random point between any two other points. Suppose there are two dots of different colors, blue and red. Suppose they are along a straight line, and there is a dot at an arbitrary point on this line. Objective is to find a suitable color for it. Let, its color value be x. The other two dots colors values are A and B respectively. Let L be the distance between A and B and the distance between A and x is l. Than the interpolation function [27] is

$$\frac{x - A}{l} = \frac{B - A}{L}$$

$$x = A + l * \frac{B - A}{L}$$

FHOG

Before understanding FHOG some terms are necessary to be introduced.[29]

GRADIENT The gradient vector of an image is defined as for each individual pixel, containing the information of the pixel color changes in both x direction and y direction. Its definition is in parallel with that of the gradient of multi-variable continuous function, which actually contains a vector having the partial derivatives of all the involved variables. The two main characteristics of an image gradient are its magnitue and direction.

HISTOGRAMOFGRADIENTS(HOG) An efficient method for extracting the features of the pixel colors, for creating a target recognition classifier, is named the Histogram of Oriented Gradients (HOG). With the help of image gradients here is how it works.

- Preparation of the image using color normalization and resizing.
- For each pixel the gradient vector is computed along with its direction and magnitude.
- The image is divided in many cells of 8 * 8 pixels. With in each of it, the 64 cells magnitude values are binned together and added cumulatively into nine buckets have direction unsigned. That is if a pixel's gradient vector direction lays between 2 buckets, it is split proportionally between them rather than assigning it to the close one. Thus, this gives robustness against distortions.

- Than a block is created of four histograms of 4 cells in 1-D vector by concatenating, which has 36 values. Then with the help of normalization this becomes unit weight.
- By concatenating all of these blocks the HOG feature vector is created.

Felzenszwalb's Algorithm Felzenszwalb suggested an algorithm for the purpose of dividing an image into similar areas utilizing a graph-based strategy. Since it is needed to define an area that potentially contains an object since usually there are multiple targets in an image. It is also the initialization method for Selective Search.

When there exist multiple objects in one image (true for almost every real-world photos), we need to identify a region that potentially contains a target object so that the classification can be executed more efficiently.

FHOG is used to efficiently compute the features of Felzenszwalb's HOG (FHOG). The HOG features that are computed have three times the number of Orients requested + 5 dimensions. Two times the number of Orients requested are channels that are contrast sensitive, four are for channels with texture, one contains zeros, while remaining channels are insensitive to contrast. The usual value is nine for each cell thus giving feature vector of 32 dimensions.

Circular shifting

It involves shifting the elements of a matrix or vector by number of positions as desired, but the last elements which is shifted goes to the start of the array. So size doesn't change. In case of a matrix it shifts the columns or rows.

Matrix multiplication

Matrix multiplication involves generating one matrix from two involved matrices. It is preformed as if A is an m * n matrix and B is an n * r matrix, than their product C = AB is an m * r matrix, in which the n elements in a row of A are multiplied by the n entries in a column of B and added to get an element of C.

2.1.10 Parameters for measuring image quality

Overlap Precision

The score for overlap precision is calculated using as percentage over all the frames included of which the intersection over union overlap exceeds a given threshold over the ground truth values. Using PASCAL evaluation criteria the threshold of overlap precision is set to 0.5 as suggested in [18].

Distance Precision

The score for distance precision is calculated as the percentage of all frames included of which the euclidean distance between centroids of ground truth and the tracker output is less than a given threshold. For this purpose twenty pixels are used as threshold as suggested in [18] .

PSNR

The score for PSNR(Peak/Pixel signal to noise ratio) is calculated using percentage over all the frames included. The difference of the real and expected value is taken and squared, for all the pixels in an image and summed together. Than the maximum value of the pixel is divided by this. To express it is decibels $\log_{10}$ is taken.

2.1.11 VIVADO HLS

A Xilinx software tool named Vivado Design Suite is developed for analysis and synthesis of HDL/hardware designs, along with Xilinx ISE with extra features for performing a SOC development and most importantly high-level synthesis.

It supports high level synthesis. Thus the code return in C++ will be converted to a HDL code. It allows with the help of pragmas to unroll, pipeline or partition loops and pieces of codes for optimization. It is very useful tool since it gives control over hardware with just using C/C++.

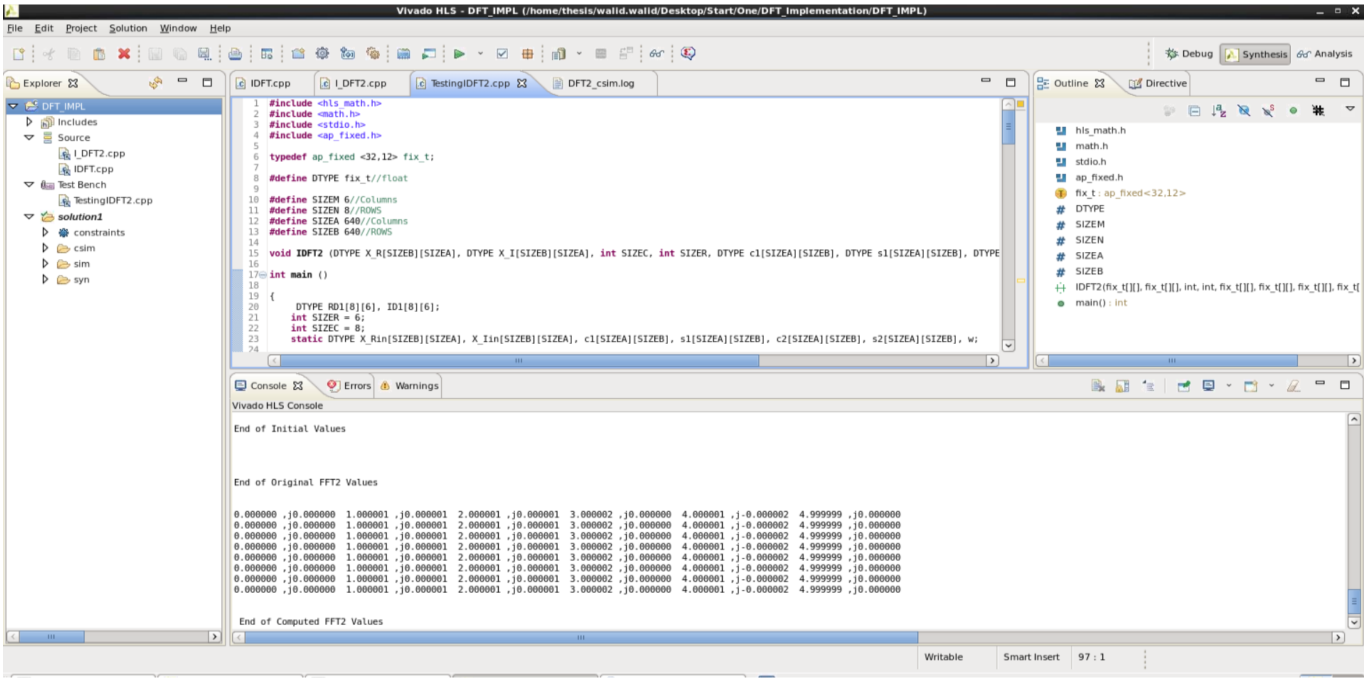


Figure 2.2: VIVADO HLS work environment

CHAPTER 3

Implementation of the algorithm

This chapter involves the fDSST algorithm [18] from a hardware implementation point of view. It goes through a sequence of steps in hardware from which the image passes, this is done for the sake of understanding the complexity of each block. The future chapters details the implementation strategies for these blocks.

The algorithm is implemented under the instructions mentioned in the paper [18]. The regularization parameter is fixed to

$$\lambda = 0.01$$

and the learning rate parameter is fixed to

$$\eta = 0.025$$

For desired correlation output g the standard deviation is fixed at 1/16 times the size of target in the dimension of translation. The padding around the image target to create the image patch is set to three times the original target size for the fDSST algorithm.

The scale factor a is set to 1.02. For this approach number of scales is set to 17, which is then interpolated to 33 to get the required scale. For desired correlation output g the standard deviation in dimension of the scale is fixed at 1/16 times the number of scales.

The PCA-HOG[21] is for representing image which is implemented according to the method given by [5]. The pixel-dense feature representation is achieved by using HOG computed on coarser feature grid for translation. The vector for feature is created by HOG 4 x 4 cells. The vectors of HOG are provided with the average gray scale value in the respective cell. These are then normalized to be in the range of [-1/2,1/2].

For computing the feature descriptor of the image patch for scale filter, firstly re-sizing is put into action. Resizing is done to a fixed size. he fixed size is basically the original target size. Then a 4 x 4 coarser cell is utilized for HOG extraction. T But when target area increases the limit (512 number of pixels), then a fixed size is maintained for it by keeping the aspect ratio unchanged. This is due to the fact, so that the maximum feature length is not greater than 992. After feature extraction, each of the extracted feature channel in the sample is passed through (i.e multiplied point wise) a Hann window.

This algorithm utilizes PCA for the reduction in the dimension of the translation filter. The original 32-dimensional HOG and its intensity combination is not utilized rather than its reduced version is used. For this approach, it is reduced to 18 dimensions. The other reduction scheme is used for the scale filter which reduces the dimension from d = 1000 to just S = 17 dimensions of the scale features.

3.0.1 Hardware blocks used in the algorithm

This section deals with the hardware blocks used in the algorithm. Also their sequence of application and complexity are considered. The block diagram of the hardware blocks involved is depicted here. It is drawn in two parts. Part 1 in figure 3.1 describes the initial steps of the algorithm and the Translation search part. Part 2 in figure 3.2 deals with the scale estimation block as well as the part belonging to translation filter update and scale filter update.

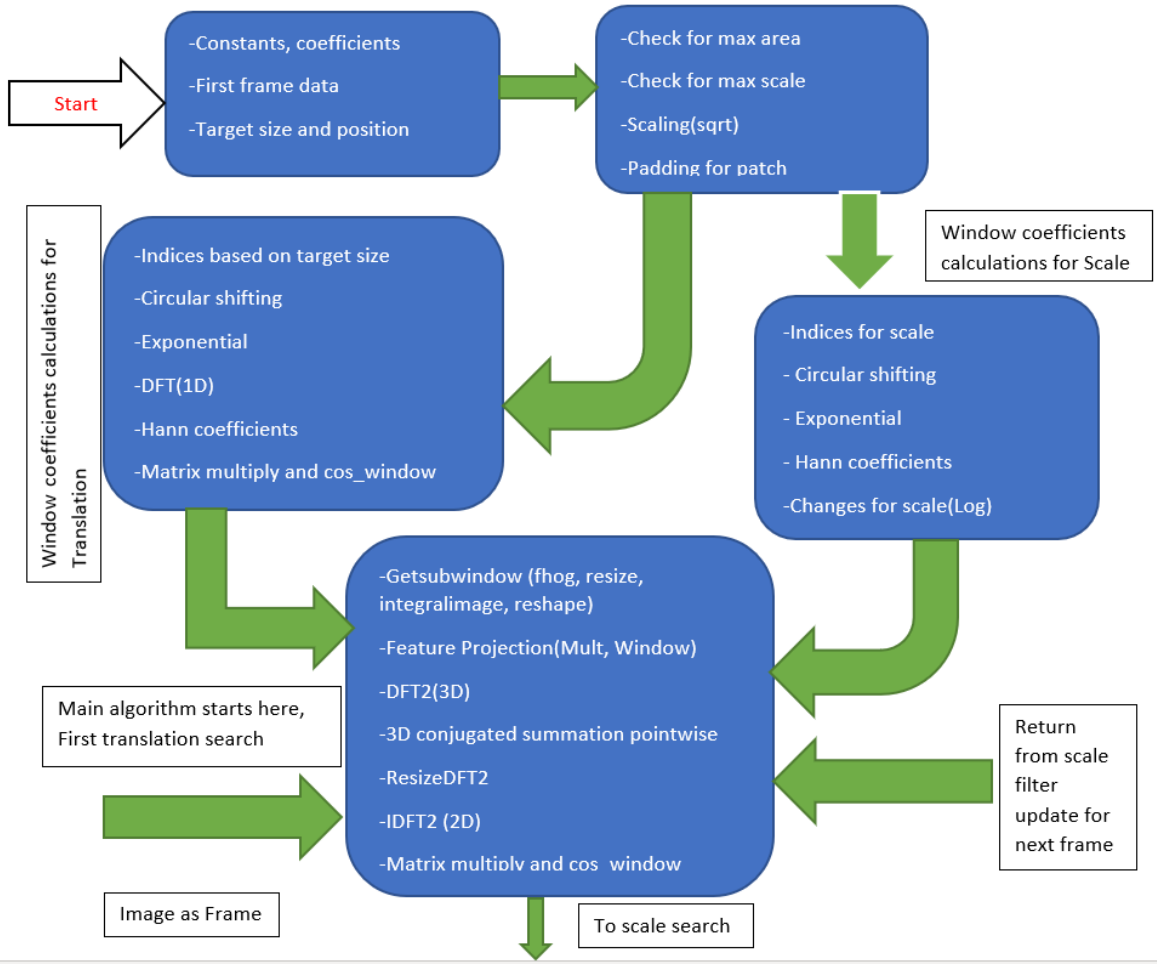


Figure 3.1: fDSST algorithm Block Diagram Part1

Input block

The algorithm starts at the first block as shown by red in the figure 3.1. The initial frame is given as input along with its corresponding initial target size, target position. Also the general constants of the algorithm are given as input from here. The Discrete Fourier Transform(DFT) and Hann window coefficients are also provided in the first block. Initial target size is given as height and width of the target [h,w] so a 1 x 2 vector. Its maximum values are 240 and 320 respectively.

Preparation Block

Than the second block in the top right corner of the figure 3.1 is entered. Here first of all an initial check on the dimensions of target are done to make sure to that they are in range i.e, less than the

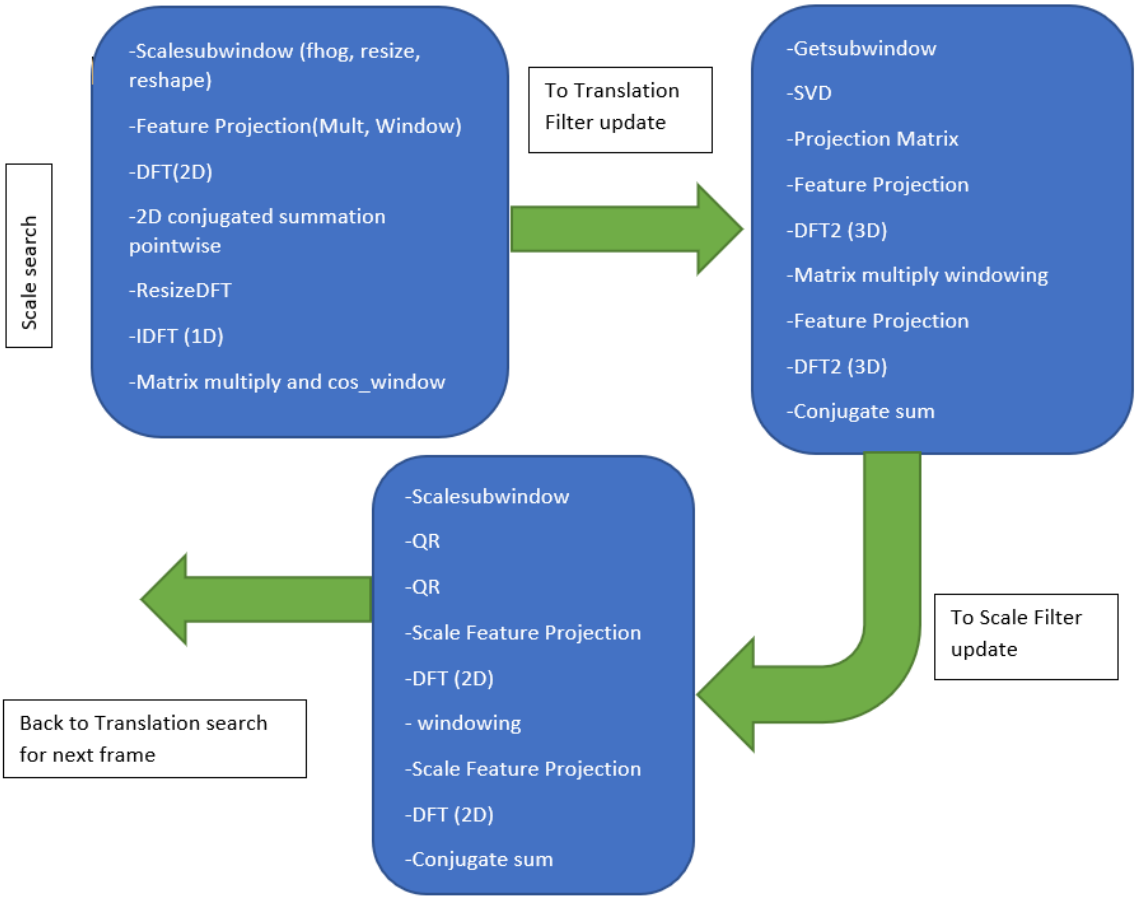


Figure 3.2: fDSST algorithm Block Diagram Part2

total maximum pixel area. Based on this the scale factor is decided accordingly using:

$$currentScaleFactor = sqrt(init\_target\_sz/translation\_model\_max\_area)$$

This involves a square root operation and division of floating point. Thus these two hardware units are to be implemented. Than based on this the target size is updated.

$$base\_target\_sz = target\_sz/currentScaleFactor$$

Similarly the same operations for the scale are applied in this block. Than the scale size is scaled accordingly. After this, for the extraction of the target from an image, a patch is created. Also window size is determined. Here a round off unit is required.

$$size = hls :: floor(base\_target\_sz * (1 + padding))$$

Then using feature ratio of 4 the new size is defined.

From here two tasks are performed in parallel. In the figure 3.1 the block on left is for translation window coefficients calculation and on the right is for scale window coefficients calculation.

Translation window preparation

To generate row indices and column indices, two vectors are defined based on initial size with the help of circular shifter. This blocks takes as input the height and width of target and generates circularly shifted indices. So the maximum dimension is based on initial target size. Then they are converted to matrices of initial size dimensions with repetition of elements in rows and columns. This performs transformation using for loops. So this transformation will affect the latency but this is only done for the first frame. Than to generate the input for the Discrete Fourier Transform(DFT) unit the elements of matrices are squared and raised to an exponential e = 2.71. This operation is applied on the whole matrix. Than the Discrete Fourier Transform2(DFT2) unit is exercised to created a window whose output is complex 2D matrix. This must be clear here that the maximum size possible here is 240 x 320 as it is the maximum number of target pixels in each frame.

Than hanning window is applied separately on width and height of the matrix i,e the rows and columns. The maximum size possible here is 240 or 320 as it is the maximum number of target pixels in each frame. Then to generate cos window matrix multiplication of the above two hann windows occur. Again the same maximum size.

Scale window preparation

From here on some necessary calculations occur for coefficients of scale, which are done once in the start and are independent on input target size. So it can be performed once and not included in algorithm. They are general constants and scale window coefficients. This is not true for a logarithm unit which has an input that is size dependent. But it only depends on the size of first frame target, so it can be taken out of the algorithm to save hardware and used as pre-computed values.

It is recommended to perform these two block at the start and store the results in a memory block in FPGA to save hardware.

The main algorithm starts from here. From here on, the required units and operations are repeated for each frame.

For the first frame there is no need to estimate target as it is already provided so jump to filter update part in figure 3.2 . Firstly the translation filter is updated.

Translation Filter update

The first major function in this block is getting sub-window. It involves patch image extraction. So it requires generating indices based on initial target size which is then used to extract the patch from the image. After that it is resized to the padded target size. Since padding is 3 so the maximum resized output dimension is 4 times the initial target size. So at maximum it can be 4 * 240 * 320. But it is a recommendation here that the try to pick target of maximum 120 * 160, so for FHOG part the dimensions are less to save heavy calculations. If target is small compared to the whole frame it is less computation hungry so less cost.

Then the fhog is applied. The maximum output is a 3D matrix with dimensions of initial target size * 32. As mentioned before for coarser grid the last matrix of fhog output is updated. on the fhog output it is desired to exercise the image integral unit. This is done by extracting along row and column each fourth element and joined. Which is then scaled down to so that the maximum value is less than 255. The maximum size of this scaled down output is initial target size. Then finally it passes through a reshaping hardware. Here the maximum output is a 2D matrix with rows = initial target size and columns 32. So it is a recommendation here that the try to pick target of maximum 120 * 160 so the dimensions are less to save hardware. If target is small compared to the whole frame it is less computation hungry so less cost.

The previous result is multiplied with its transpose which generates a 32 * 32 matrix. Then the result is decomposed using SVD decomposition of matrix. But maximum dimension here is 32 * 32

and out of which only 18(number of compressed dimension) columns are utilized in the projection matrix.

The second major function of this block is feature projection. This involves windowing of the cos window with projection matrix. Maximum output dimensions here are initial target size * 18 a 3D matrix. Then this is passed through a DFT2 unit. This is part is sort of the bottleneck of algorithm, as now it involves target size * 18 DFT calculations(also involving twice the matrix transpose). Thus the maximum can be 240 * 320 times 18. Then the output is conjugated and is windowed with the co-coefficients of Hann calculated earlier.

The second function is repeated with sub window output. Then the output is multiplied with its conjugated version element wise. The corresponding elements in the third dimension are summed together to get a 2D matrix as output. If it is not the first frame than it is update as

$$hf = (1 - interp\_factor) * hf + interp\_factor * new\_hf$$

Scale Filter update

After the translation filter, scale filter is updated. Again the major function is getting sub window for scales.It also involves patch extraction based on scales. It is equivalent to the translation version except it is repeated for 17 timed based on number of scales. The output has dimensions of scale's initial target size * 32.

After this, it is decomposed using QR factorization of matrix. But maximum dimensions here are 31 * (scale_model_size/4) * 17. This step is done twice.

Than the second major function feature projection for scale is applied. The maximum can be 17 * 17 for the DFT (2D). Than the output is conjugated and is windowed with the co-coefficients of Hann calculated earlier for scale. The second function is repeated with sub window output. The corresponding elements in the first dimension are summed together to get a 2D matrix as output.

Translation search

This first of all involves the same two major function steps as translation filter. Then third major function is resizing of 2D DFT result before taking IDFT2. This is done using linear interpolation. Again the maximum is 4 * the initial target size. That Inverse DFT2 is applied whose output has of same dimensions but real values.

Scale search

This first of all involves the same two major function steps as scale filter. Then third major function is resizing of 1D DFT result before taking IDFT. This is done using linear interpolation. Again the maximum is 31 * (scale_model_size/4) * 17. That Inverse DFT is applied whose output has of same dimensions but real values.

CHAPTER 4

Discrete Fourier Transform implementation

This chapter details with the implementation of discrete Fourier transform. This is in three basic versions.

- DFT (One dimensional 1D)
- DFT2 (Two dimensional 2D)
- DFT2 (Three dimensional 3D)i,e, repeated for fixed 18 times

Different approaches for DFT implementation exists. The best solution for hardware implementation is butterfly based DFT. This requires the DFT size to be a power of 2. So, butterfly approach can't be used since target selection is generic. So basic implementation based on its original formula is considered. Their implementation details are given as under:

4.1 Discrete Fourier Transform(DFT)

For the implementation of this block Vivado HLS is used as a tool. The basic DFT is calculated as

$$X_k = \sum_{n=0}^{N-1} x_n . e^{-\frac{j2\pi}{N} kn}$$

where x_n is the real input and X^k is the complex output. For the case of inverse Discrete Fourier Transform(IDFT) it is described as

$$x_k = \frac{1}{N} \sum_{n=0}^{N-1} X_n . e^{\frac{j2\pi}{N} kn}$$

In the case of this algorithm the maximum size of DFT or IDFT required are 17 and 33 respectively. The equation also involves the multiplication by coefficients. These coefficients are pre-computed and given at the input of DFT block as matrices. Also two separate vectors are given as inputs for real and imaginary inputs. These ports are bidirectional for saving resources i.e, input ports are re utilized for output. The core involves two main computational blocks. The first one uses 2 for loops, the inner loop implements the DFT equation. It involve 2 multiplication one subtraction and one addition. The same is also done here for the imaginary case. So computations are done twice. The outer loop iterates the inner one for all the DFT points. The second block involves one loop for assigning the final values to the output, so it is actually copying data.

As a general reference a FFT algorithm present in vivado library is synthesized with 1024 point butterfly, the results are shown in figure 4.1. The latency is 875 cycles operating at 100MHz. The resource consumption is high as it can be seen.

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.75	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
875	875	876	876	none

Detail

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	6	24	12481	9801
Memory	-	-	-	-
Multiplexer	-	-	-	5
Register	-	-	3	-
Total	6	24	12484	9806
Available	280	220	106400	53200
Utilization (%)	2	10	11	18

Figure 4.1: Vivado hlsfft library(1024 point FFT) implementation

After this the real DFT first version is implemented. This DFT unit will be used for DFT2 as well which has the maximum dimension of 320 * 320. So the DFT it self is designed for size 320. Then as a first approach, complex input and output numbers with only 2 vectors are used. One is for input and one is for output. After doing synthesis the achieved results are shown in figure 4.2. Its resource consumption is less but latency is high. To improve this other versions are developed.

As a second approach of this same case of complex input and output numbers, pipelining is used. The inner loop, the one implementing the DFT equation is pipelined. With this the output assignment loop is also pipelined. The results are shown in figure 4.3. It can be seen that the resources are less but the important result is improved latency.

To further decrease latency, another approach using two separate vectors one for real numbers and one for imaginary numbers is considered. For input and output both using the same vector to minimize resources. The results are shown in figure 4.4. It uses little more resources but latency is improved.

Now for the second case the previous approach is used but with pipelining and loop unrolling. The output assignment loop is unrolled by a factor of 2. The inner loop for DFT equation and the output assignment loops are pipelined. The results are shown in figure 4.5. It uses less resources and the latency is improved. The minimum latency is 803 clock cycles. So simple cases like one dimensional (1D) DFT which has the maximum size of 17 or 33 it will fall in this category and will be fast.

In the previous approach after trying with more unrolling there were no significant improvements in result. So another strategy is considered. Previous method is repeated but with fixed point datatype. The synthesis results are shown in figure 4.7. The maximum latency is almost half of the previous case.

Till now only DFT is considered. The IDFT also can be easily incorporated just by putting a check for it in the output assignment loop. When the check is true than instead of normal assignment,

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.41	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
1640002	1640002	1640003	1640003	none

Detail

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	94
FIFO	-	-	-	-
Instance	-	10	696	1422
Memory	2	-	0	0
Multiplexer	-	-	-	350
Register	-	-	367	-
Total	2	10	1063	1866
Available	280	220	106400	53200
Utilization (%)	~0	4	~0	3

Figure 4.2: DFT (320 * 320) implementation using complex matrices

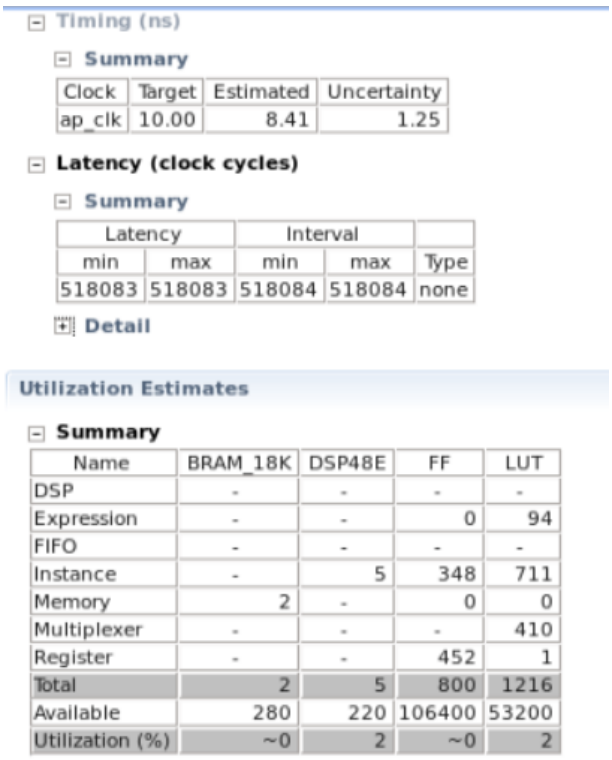
the normalized DFT values are assigned to the output. And the coefficients for IDFT instead of DFT will be passed as input. Hence same resource can perform both operations.

4.2 Discrete Fourier Transform2(DFT2)

This block involves transpose of matrices and DFTs. Firstly transpose the input matrix. Pass the rows (reality columns) to the 1 D DFT block for DFT computation. The result from DFT is again transposed and the rows are sent for DFT again. So it takes the DFT first along columns then along rows. In this case the maximum size of DFT and IDFT required is 240 * 320 and 4 * 240 * 320. Since transpose is involved, so for hardware implementation the maximum size would be 320. Thus it makes the maximum dimensions to be 320 * 320. Important to note here is that the coefficients are pre-computed and given in the input as a matrix. Also two separate matrices are given as inputs for real and imaginary. For saving resources ports are bidirectional meaning the same matrices are also used to writing the output result. The core involves 4 mains computation blocks. The first one uses two for loops, the inner loop implements the transpose of the matrices. The outer loop iterates it for all the rows. The second block involves DFT 1D. The result is transposed by the third block. The fourth is again DFT 1D. It is depicted in the figure 4.6

Again using the approach of DFT 1D, the loops involved in transpose are unrolled by a factor of 2 and also pipelined. The results are shown in figure 4.8. The latency is too high for this block.

But before moving forward one thing to be considered here is that this algorithm is for object tracking. In object tracking Usually the size of target is small as compared to the frames size. It suggested that if the maximum target size is set to half, than the DFT2 can be scaled to 120 * 160. This will save lot of computations. The previous approach is repeated for these dimensions i.e, 160 * 160. The results are shown in figure 4.9. The latency is much improved from the previous version.



Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.41	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
518083	518083	518084	518084	none

Detail

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	94
FIFO	-	-	-	-
Instance	-	5	348	711
Memory	2	-	0	0
Multiplexer	-	-	-	410
Register	-	-	452	1
Total	2	5	800	1216
Available	280	220	106400	53200
Utilization (%)	~0	2	~0	2

Figure 4.3: DFT (320 * 320) pipelined implementation of complex matrices

Now lets consider the IDFT2 block. The IDFT2 block has the maximum size of 4 * the initial target size. Using maximum size of 160 it is implemented for 640 i.e 4 * 160. The results are depicted in 4.10. These results are compared in next sections.

4.3 Discrete Fourier Transform2(3D)

This block uses DFT2 as a base unit. It performs the DFT2 function for 3 dimensional matrices. It simply calls the DFT2 function for a number of times. In this case the maximum size of DFT2 or IDFT2 required is 240 * 320 * 18 but suggested to be scaled to 160 * 160 *18. This will save lot of computations. Important to note here is that the co-coefficients are pre-computed and given in the input as a matrix. The core involves one main computation block. That uses one for loop, this loop implements the DFT2 of matrix. The main thing to note is the one major loop for computing 18 DFT2's. This is the real bottleneck of the algorithm.

4.4 Discrete Fourier Transform2 performance measurement

This sections deals with applying the test bench to check the correctness of the implemented algorithm. DFT is used as a block inside DFT2. As a result, this major block is used for checking the behaviour. The test data is taken from matlab implementation of the algorithm. On matlab, a 8 * 6 target size is selected from an image frame. This 8 * 6 matrix is traced along the algorithm, until it reach the input of DFT2. This input is taken as golden reference. Then matlab DFT2 is exercised with this data and output is sampled for comparison later. The real input and output of the DFT2 (3D) block from matlab implementation are shown in figures 4.7 and 4.8 respectively.

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.41	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
1639682	1639682	1639683	1639683	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	72
FIFO	-	-	-	-
Instance	-	16	982	2064
Memory	2	-	0	0
Multiplexer	-	-	-	307
Register	-	-	369	-
Total	2	16	1351	2443
Available	280	220	106400	53200
Utilization (%)	~0	7	1	4

Figure 4.4: DFT (320 * 320) implementation using two array for real and imaginary

Now in VIVADO HLS the DFT2 unit is exercised with matlab input data. It is simulated for fixed point. The fixed point is implemented for 32 bit with 20 bits of fractional part. The simulation output results are shown in 4.9. Also the relative difference in the result is compared with the golden matrix, and PSNR (Pixel/Peak Signal to Noise Ratio) is calculated and shown in table 4.10. The relative percentage value is small. This is very less justifying our use of fixed point. The PSNR value is negligible here because the difference is very small and PSNR calculation involves squaring so the results become negligible. This makes the PSNR values go to infinity.

4.5 Comparisons with other literature implementation

For comparing our results, two other implementations [4] [2] are selected and compared with our approach. The timing results from paper [4] which has implemented DFT in a FPGA are taken just as a reference to compare with our VIVADO results. They are shown in table 4.2. While our timing results are depicted in table 4.1. This paper implements DFT in 1D, and uses CORDIC to compute the twiddle factor. Authors have shown results for 50 Mhz clock. While our DFT runs at 100 MHz. Our approach exceeds the their timing results. Also in the resources section our approach 4.3 consumes less resources then in [4] in table 4.4. Mainly because of pipelining and also because in our approach the twiddle factors are pre calculated.

The timing results from [2] which has implemented FFT2 and FFT2 3D in FPGA are taken just as a reference to compare with our VIVADO results. They are shown in table 4.5. While our timing results are depicted in table 4.1. This paper implements FFT2 and FFT2 3D. There timing results

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.41	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
803	415523	804	415524	none

Detail

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	132
FIFO	-	-	-	-
Instance	-	5	348	711
Memory	4	-	0	0
Multiplexer	-	-	-	402
Register	-	-	555	2
Total	4	5	903	1247
Available	280	220	106400	53200
Utilization (%)	1	2	~0	2

Figure 4.5: DFT (320 * 320) pipelined implementation of separate arrays for real and imaginary

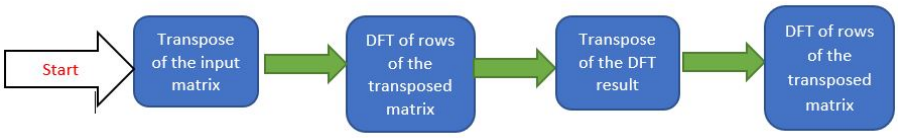


Figure 4.6: DFT2 block diagram

far exceeds our but it is important to note that their implementation is for FFT while our us based on DFT. The main insight given by these results is that the authors[2] used an approach to avoid transform. This is accomplished by using a fast SDRAM block. This allows burst transfer mode for accessing the columns. This approach can be utilised in our implementation to improve latency. On the other hand in the resources section our approach 4.3 consumes less resources then in [4]. It is displayed in table 4.6.

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	7.63	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
803	208163	804	208164	none

Detail

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	224
FIFO	-	-	-	-
Instance	-	16	0	0
Memory	4	-	0	0
Multiplexer	-	-	-	141
Register	-	-	619	3
Total	4	16	619	368
Available	280	220	106400	53200
Utilization (%)	1	7	~0	~0

Figure 4.7: DFT (320 * 320) pipelined implementation using fixed point

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.41	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
2564	266552324	2565	266552325	none

Detail

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	425
FIFO	-	-	-	-
Instance	4	5	945	1407
Memory	-	-	-	-
Multiplexer	-	-	-	528
Register	-	-	549	-
Total	4	5	1494	2360
Available	280	220	106400	53200
Utilization (%)	1	2	1	4

Figure 4.8: DFT2 (320 * 320) pipelined implementation

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	7.69	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
1284	17064004	1285	17064005	none

Detail

Instance

Loop

Utilization Estimates

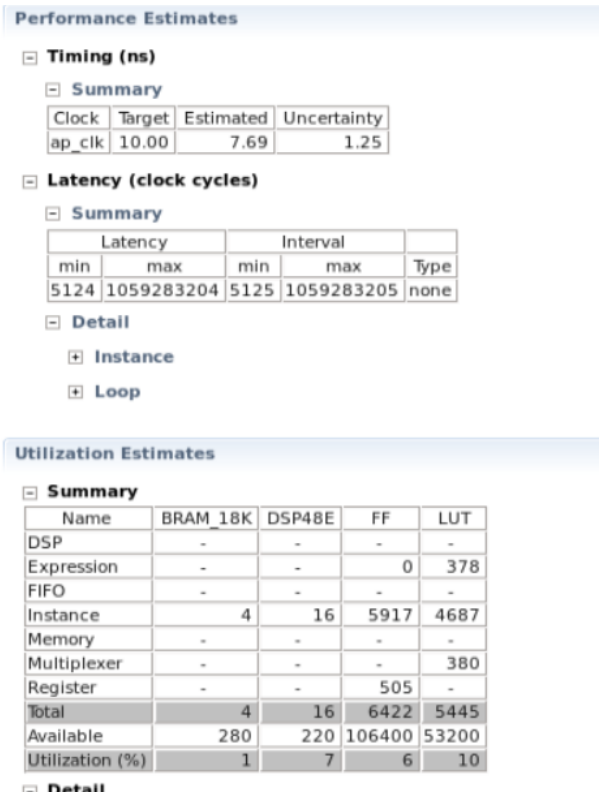
Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	319
FIFO	-	-	-	-
Instance	2	16	5887	4593
Memory	-	-	-	-
Multiplexer	-	-	-	344
Register	-	-	425	-
Total	2	16	6312	5256
Available	280	220	106400	53200
Utilization (%)	~0	7	5	9

Figure 4.9: DFT2 (160 * 160) pipelined implementation

Dimensions for implementation	Transform Type	Datatype	Pipeline	Number of clock cycles	Time
1024	FFT	–	–	875	7656.25 ns
320	DFT	(complex)	–	1640002	13.792 ms
320	DFT	(complex)	Yes	518083	4.3571 ms
320	DFT	float	–	1639682	13.79 ms
320	DFT	float	Yes	803(min)	6753.23 ns
320	DFT	float	Yes	415523(max)	3.495 ms
320	DFT	(fixed point)	Yes	803(min)	6753.23 ns
320	DFT	(fixed point)	Yes	208163(max)	1750.65 us
320 * 320	DFT2	(fixed point)	Yes	2546(min)	21411.86 ns
320 * 320	DFT2	(fixed point)	Yes	266552324(max)	2.241 ms
160 * 160	DFT2	(fixed point)	Yes	1284(min)	9873.96 ns
160 * 160	DFT2	(fixed point)	Yes	17064004(max)	131.222 ms
640 * 640	IDFT2	(fixed point)	Yes	5124(min)	39403.56 ns
640 * 640	IDFT2	(fixed point)	Yes	1059283204(max)	8145.888 ms

Table 4.1: Timing results of VIVADO based implementation of our approach



Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	7.69	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
5124	1059283204	5125	1059283205	none

Detail

Instance

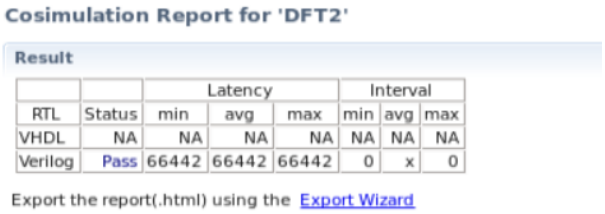
Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	378
FIFO	-	-	-	-
Instance	4	16	5917	4687
Memory	-	-	-	-
Multiplexer	-	-	-	380
Register	-	-	505	-
Total	4	16	6422	5445
Available	280	220	106400	53200
Utilization (%)	1	7	6	10

Figure 4.10: IDFT2 4*(160 * 160) pipelined implementation



Cosimulation Report for 'DFT2'

Result

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	66442	66442	66442	0	x	0

Export the report(.html) using the Export Wizard

Figure 4.11: DFT2 (160 * 160) pipelined implementation co-simulation result

Number of input sequences	Transform Type	Number of clock cycles	Time (ns)
10	DFT	24179	483580
12		28999	579980
20		96509	1930180

Table 4.2: Timing results take from [4] for DFT implementation in FPGA

Dimensions for implementation	Transform Type	Datatype	Pipeline	LUT	FF	DSP48E
1024	FFT	–	–	9806	12484	24
320	DFT	(complex)	–	1866	1065	10
320	DFT	(complex)	Yes	1216	800	5
320	DFT	float	–	2443	1351	16
320	DFT	float	Yes	1247	903	5
320	DFT	(fixed point)	Yes	368	619	16
320 * 320	DFT2	(fixed point)	Yes	2360	1494	5
160 * 160	DFT2	(fixed point)	Yes	5256	6312	16
640 * 640	IDFT2	(fixed point)	Yes	5445	6442	16

Table 4.3: Resourse utilization of VIVADO based implementation of our approach

No of input Sequence	Transform Type	Slice registers	Slice LUTs	FF LUT-FF pairs
10	DFT	655	616	1232
12	DFT	682	648	1285
20	DFT	798	776	1505

Table 4.4: Resource utilization results taken from [4] Spartan 3E

Shape	Image size	Row operations(ms)	Column local DFT(ms)	Total(ms)
Square	128 * 128	0.89	0.90	1.79
	256 * 256	3.01	3.04	6.05
	512 * 512	12.14	12.72	24.86
	1024 * 1024	50.21	52.42	102.63
	2048 * 2048	202.45	209.56	412.10
Rectangle	512 * 2048	50.34	52.37	102.71
	2048 * 512	50.11	50.47	102.58

Table 4.5: Timing results taken from [2]

Slices	DSP48E
25% (8273/33088)	53% (68/128)

Table 4.6: Resource utilization results take n from [2] for VIRtex-5

	0	1	2	3	4	5
0	0.0000 ,j0.0000	0.0000 ,j0.0000	0.0000 ,j0.0000	0.0000 ,j0.0000	0.0000 ,j0.0000	0.0000 ,j0.0000
1	0.0000 ,j0.0000	-0.0572 ,j0.0000	-0.1204 ,j0.0000	-0.0903 ,j0.0000	-0.0554 ,j0.0000	0.0000 ,j0.0000
2	0.0000 ,j0.0000	-0.1262 ,j0.0000	-0.3881 ,j0.0000	-0.3400 ,j0.0000	-0.1007 ,j0.0000	0.0000 ,j0.0000
3	0.0000 ,j0.0000	-0.2611 ,j0.0000	-1.1746 ,j0.0000	-0.9769 ,j0.0000	-0.2860 ,j0.0000	0.0000 ,j0.0000
4	0.0000 ,j0.0000	-0.3540 ,j0.0000	-1.0068 ,j0.0000	-1.0217 ,j0.0000	-0.3382 ,j0.0000	0.0000 ,j0.0000
5	0.0000 ,j0.0000	-0.2181 ,j0.0000	-0.4519 ,j0.0000	-0.4440 ,j0.0000	-0.1746 ,j0.0000	0.0000 ,j0.0000
6	0.0000 ,j0.0000	-0.0505 ,j0.0000	-0.1126 ,j0.0000	-0.1632 ,j0.0000	-0.0450 ,j0.0000	0.0000 ,j0.0000
7	0.0000 ,j0.0000	0.0000 ,j0.0000	0.0000 ,j0.0000	0.00000 ,j0.0000	0.0000 ,j0.0000	0.0000 ,j0.0000

Table 4.7: Real input from Matlab, dimensions are 8 * 6

	0	1	2	3	4	5
0	-8.3574 ,j0.0000	4.6297 ,j2.8766	-0.3754 ,j-1.0285	-0.1512 ,j0.0000	-0.3754 ,j1.0285	4.6297 ,j-2.8766
1	5.3115 ,j1.8092	-2.2532 ,j-2.8848	-0.1894 ,j0.7096	0.0815 ,j0.2814	0.7659 ,j-0.6668	-3.7162 ,j0.7514
2	-1.3945 ,j-1.0867	0.3183 ,j1.2039	0.4048 ,j-0.2384	-0.0026 ,j-0.2299	-0.5660 ,j-0.0026	1.2400 ,j0.3537
3	0.1300 ,j0.6420	0.1699 ,j-0.4108	-0.1718 ,j-0.0686	-0.1430 ,j0.1240	0.2843 ,j0.2181	-0.2695 ,j-0.5046
4	0.2634 ,j0.0000	-0.1192 ,j-0.1844	-0.1523 ,j0.1746	0.2796 ,j0.0000	-0.1523 ,j-0.1746	-0.1192 ,j0.1844
5	0.1300 ,j-0.6420	-0.2695 ,j0.5046	0.2843 ,j-0.2181	-0.1430 ,j-0.1240	-0.1718 ,j0.0686	0.1699 ,j0.4108
6	-1.3945 ,j1.0867	1.2400 ,j-0.3537	-0.5660 ,j0.0026	-0.0026 ,j0.2299	0.4048 ,j0.2384	0.3183 ,j-1.2039
7	5.3115 ,j-1.8092	-3.7162 ,j-0.7514	0.7659 ,j0.6668	0.0815 ,j-0.2814	-0.1894 ,j-0.7096	-2.2532 ,j2.8848

Table 4.8: Real output from Matlab, dimensions are 8 * 6

	0	1	2	3	4	5
0	-8.3575 ,j0.0000	4.6297 ,j2.8766	-0.3754 ,j-1.0283	-0.1511 ,j0.0000	-0.3754 ,j1.0283	4.6297 ,j-2.8766
1	5.3115 ,j1.8093	-2.2532 ,j-2.8850	-0.1894 ,j0.7097	0.0815 ,j0.2813	0.7657 ,j-0.6667	-3.7160 ,j0.7513
2	-1.3944 ,j-1.0867	0.3183 ,j1.2040	0.4048 ,j-0.2386	-0.0028 ,j-0.2299	-0.5660 ,j-0.0025	1.2400 ,j0.3537
3	0.1299 ,j0.6419	0.1698 ,j-0.4108	-0.1717 ,j-0.0684	-0.1429 ,j0.1239	0.2843 ,j0.2181	-0.2694 ,j-0.5047
4	0.2635 ,j-0.0000	-0.1192 ,j-0.1844	-0.1523 ,j0.1746	0.2795 ,j-0.0000	-0.1523 ,j-0.1746	-0.1192 ,j0.1845
5	0.1299 ,j-0.6419	-0.2694 ,j0.5047	0.2843 ,j-0.2181	-0.1429 ,j-0.1239	-0.1717 ,j0.0684	0.1698 ,j0.4108
6	-1.3944 ,j1.0867	1.2400 ,j-0.3537	-0.5660 ,j0.0025	-0.0028 ,j0.2299	0.4049 ,j0.2386	0.3183 ,j-1.2040
7	5.3115 ,j-1.8093	-3.7160 ,j-0.7513	0.7657 ,j0.6667	0.0815 ,j-0.2813	-0.1895 ,j-0.7097	-2.2532 ,j2.8850

Table 4.9: Real output from VIVADO HLS co-simulation(post synthesis), dimensions are 8 * 6

	0	1	2	3	4	5
0	0.0013 ,j0.0000	0.0002 ,j0.0006	0.0009 ,j0.0182	0.0661 ,j0.0312	0.0011 ,j0.0176	0.0000 ,j0.0001
1	0.0007 ,j0.0065	0.0007 ,j0.0076	0.0285 ,j0.0180	0.0211 ,j0.0415	0.0247 ,j0.0208	0.0043 ,j0.0079
2	0.0069 ,j0.0002	0.0127 ,j0.0064	0.0133 ,j0.0793	7.2688 ,j0.0001	0.0072 ,j3.6615	0.0029 ,j0.0031
3	0.0557 ,j0.0116	0.0478 ,j0.0093	0.0435 ,j0.2192	0.0576 ,j0.0927	0.0128 ,j0.0107	0.0461 ,j0.0264
4	0.0380 ,j100.0000	0.0022 ,j0.0349	0.0030 ,j0.0051	0.0361 ,j100.0000	0.0001 ,j0.0062	0.0002 ,j0.0334
5	0.0586 ,j0.0102	0.0472 ,j0.0267	0.0135 ,j0.0121	0.0570 ,j0.0904	0.0446 ,j0.2262	0.0461 ,j0.0095
6	0.0073 ,j0.0000	0.0031 ,j0.0072	0.0067 ,j3.5827	7.1741 ,j0.0007	0.0140 ,j0.0789	0.0109 ,j0.0068
7	0.0006 ,j0.0079	0.0041 ,j0.0102	0.0253 ,j0.0224	0.0176 ,j0.0387	0.0285 ,j0.0189	0.0001 ,j0.0082

* Pixel/Peak Signal to Noise Ratio for Real is inf and for imaginary inf *

Table 4.10: Relative difference in percentage between Golden Matrix and our approach, dimensions are 8 * 6

CHAPTER 5

QR Factorization implementation

This chapter details with the implementation strategies for QR Factorization of a matrix. Vivado HLS is used as a tool for this purpose. This factorization decomposes the Matrix B into matrices Q and R. If the dimension of original matrix B is M x N then Q has the dimension of M x M and R has dimension of M x N. Where R is an upper triangular matrix while Q is an orthogonal matrix.

The QR factorization as a preliminary approach can be implemented using Gram–Schmidt process. In this process the columns of the matrix are used. Let x1, x2, x3 denote the columns of the original matrix. Projection of a column is found by

$$Proj_u x = \frac{u.x}{u.u} u$$

where "." represents inner product or dot product of vectors, in this case the columns. In case of complex numbers the first term of the inner product or dot product is conjugated. After calculating the projection of all the columns of the original matrix, the Q matrix can be obtained by finding orthonormal columns of A from each other by using the projections as,

$$u_j = a_i - \sum_{i=1}^{j-1} Proj_{u_i} a_j$$

where u_i divided by its length make the columns of Q matrix. To find matrix R take the dot product of u_i (divided by its length) and columns of original matrix. It involves lengthy calculations which is not suitable for hardware.

Another better approach to implement QR factorization is by using Givens Rotations method. For this a series of Givens rotations are computed. Lets have a quick recall of given rotations. This is basically a rotation in a plane which is common to two of the coordinate axes. It is usually given by,

$$R = \begin{bmatrix} c & s \\ -s & c \end{bmatrix}$$

here c = cos(theta) and s = sin(theta). If this is left multiplied by a matrix than the element in the position of -s is zeroed. Theta is chosen as

$$theta = \frac{x^2}{\sqrt{x^2 + y^2}}$$

where y is the element to be zeroed and x is the top element of that column.

For three dimensional matrices, three given rotations exists for rotating along three axis. They are as under:

$$R3 = \begin{bmatrix} c & -s & 0 \\ s & c & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$R2 = \begin{bmatrix} c & 0 & -s \\ 0 & 1 & 0 \\ s & 0 & c \end{bmatrix}$$

$$R1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c & -s \\ 0 & s & c \end{bmatrix}$$

-s should be in the position of the element which has to be zeroed. And 1 should be at the axis around which it is needed to be rotated.

With the help of these given rotations from a Matrix B the upper triangular matrix R can be extracted by turning the respective elements to zero. QR decomposition has a property that Q^T * B = R. This means the product of rotation matrices gives Q^T. Hence the QR factorization. Compared to the Gram–Schmidt process this method is less computation hungry and also offers a lot of parallelism.

This approach is used by the authors [25] for the FPGA based implementation of the QR decomposition. Also the rotation of vectors in this is calculated by CORDIC algorithm with the help of additions and shifts. The one implementation provided in vivado library uses this approach but without the CORDIC part.

Now back to the algorithm [18]. The maximum dimensions required here depends on target size. The rows corresponding to scaled area while the columns are fixed to be 17. Rows depends on the target size by

$$y = sqrt(MaxArea/TargetArea)$$

$$rows = y * TargetArea$$

Which mounts up to huge dimensions. The number of columns are always 17 but the rows are greater than 1000.

Also another important thing to note is that from qr factorization in this algorithm only the matrix Q is used. The matrix R is not used. As a general idea the implementation of qr factorization from vivado HLS library is used. Two versions from VIVADO library are synthesized. They are shown in 5.2 and 5.1 and are synthesised for 320 * 17 dimensions. The results are compared with the implementation in FPGA provided by [25]. It is implemented for 4 * 4 matrix. So for a fair comparison the one in VIVADO library is also synthesized for 4 * 4 dimensions and DATAFLOW pragma is used to take the advantage of task level parallelism. The VIVADO based implementation results are in 5.3 and 5.4 while the timing and area details of the one implemented in FPGA by [25] is in 5.5. In timing analysis the implementation by [25] performs better than the one in vivado. Vivado based implementation works at 100MHz. In [25] some implementations work at about 400MHz which is really good. The latency of the VIVADO implementation in 5.3 of 544 cycles is comparable to the non-piplined one at 32 bits from 5.5 which is 2.5 times faster. The reason for high latency is that, that the one in 5.5 is implemented with fixed point and uses CORDIC while in VIVADO only floating point version is available. So it can achieve good performance by switching to fixed point. Area results are better for VIVADO specially in terms of number of DSP48E and FFs utilized.

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.47	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
229089	7989402	229090	7989403	dataflow

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	272	10	4288	6317
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	-	-
Total	272	10	4288	6317
Available	280	220	106400	53200
Utilization (%)	97	4	4	11

Figure 5.1: Vivado qr factorization Basic implementation

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.63	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
271812	10139988	271813	10139989	dataflow

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	302	16	7139	8718
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	-	-
Total	302	16	7139	8718
Available	280	220	106400	53200
Utilization (%)	107	7	6	16

Figure 5.2: Vivado qr factorization Alternate implementation

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.63	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
428	544	429	545	dataflow

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	4	16	6426	8024
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	-	-
Total	4	16	6426	8024
Available	280	220	106400	53200
Utilization (%)	1	7	6	15

Figure 5.3: Vivado qr factorization alternate 4 * 4 implementation

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.16	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
123	1491	124	1492	dataflow

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	4	10	3975	5835
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	-	-
Total	4	10	3975	5835
Available	280	220	106400	53200
Utilization (%)	1	4	3	10

Figure 5.4: Vivado qr factorization basic 4 * 4 implementation

Device	**Xilinx Virtex 6 XC6VLX240T -2**		
Word-Lenght	**16 bits**	**24 bits**	**32 bits**
CORDIC-Iterations	10	18	26
Cycles/Matrix	8	8	8
Multiplier A	**Dedicated Non-Pipelined DSP48E**		
Latency (cycles)	52	84	116
DSP48E	12(1%)	24(3%)	48(6%)
Max. Freq. (Mhz)	234.5	170.3	117.1
Slice Registers	2,402(1%)	5,888(1%)	10,844(3%)
Slice LUTs	2,498(1%)	6,109(4%)	11,337(7%)
Multiplier B	**Dedicated Pipelined DSP48E**		
Latency (cycles)	60	100	140
DSP48E	12(1%)	24(3%)	48(6%)
Stages Pipe. Mult.	2	4	6
Max. Freq. (Mhz)	421.1	398	377.6
Slice Registers	2,457(1%)	6,046(1%)	11,520(3%)
Slice LUTs	2,587(1%)	6,220(4%)	11,225(7%)
Multiplier C	**Const. Coef. Pipelined (without DSP48E)**		
Latency (cycles)	60	96	132
Stages Pipe. Mult.	2	3	4
Max. Freq. (Mhz)	421.1	398	377.6
Slice Registers	3,262(1%)	8,400(2%)	16,259(5%)
Slice LUTs	3,641(2%)	8,596(5%)	16,100(10%)

Figure 5.5: QR factorization 4 * 4 implementation by [25]

CHAPTER 6

SVD Decomposition implementation

This section details the implementation strategies for SVD Decomposition i.e, singular value decomposition of a matrix. Vivado HLS is used as a tool for this purpose. This decomposes the Matrix into matrices U, S and V. If the dimensions of a matrix B is M * N then the dimensions of matrix U are M * M, dimensions of matrix S are M * N and the dimensions of matrix V are N * N. Where U can be a real or a complex matrix, S is a diagonal matrix with real non-negative values and V is an unitary matrix. The diagonal elements of the matrix S are the singular values of original matrix, whiles the columns of matrices U and V are left singular vectors and right singular vectors of original matrix.

The singular value decomposition can be calculated by using the following properties:

- The left singular vectors of matrix B are orthonormal eigenvectors of B * conjugate(B).
- The right singular vectors of matrix B are orthonormal eigenvectors of conjugate(B) * B.
- The non-negative singular values of matrix B i.e, the values on the diagonal of matrix S are the square roots of the eigenvalues of both conjugate(B) * B and B * conjugate(B).

For implementing SVD there are many approaches like Jacobi method(double sided rotation); QR factorization and Hestenes-Jacobi (one-sided rotation) approach. Among them, the Jacobi algorithm is better because the parallelism in it can be utilized for a fast approach. If the matrix dimension is M than it is partitioned into M/2 2x2 sub matrices and solved. This approach in hardware is implemented in FPGA by the authors of [12]. The one available in VIVADO library uses this approach.

The maximum dimensions required here does not depend on target size. The number of columns and rows are always 32 so dimensions are 32 * 32. This simplifies SVD calculations.

Also only the U matrix is required for the algorithm not the other two. As a general idea the implementation of SVD factorization from vivado HLS library is used. It is available in two versions. The results are depicted in 6.1 and 6.2. But for both of them resource consumption is high. This will be improved by implementing SVD approach for calculating only the U matrix not the other two.

6.1 Comparison with other implementations

This section deals with the comparison of vivado based implementation with other solutions. The authors of [12] provide implementation based on Jacobi two sided algorithm. This is implemented for 4 * 4 fixed point. The resource consumption is shown in 6.3. The timing results are depicted in table. For comparison VIVADO based both solutions are synthesized for 4 * 4 architecture. The solution is only available for floating point. The results are shown in figures 6.5 and 6.4. The timing results are

compared in the table. Our approach is much slower but it is floating point based. So if implemented in fixed point results will improve.

For a more fair comparison our approach is tested with double floating point implementation of 30 * 30 matrix implemented by [19]. This is implemented in a work station which has an Intel core(TM)2 Quad, 2.5 GHz processor and 4GB DDR2 memory. But the fixed point approach is implemented in FPGA. The timing details and resources consumption are shown in 6.6 and 6.7. Also the vivado based 30 * 30 synthesized implementation are shown in figures 6.8 and 6.9. Our results are just 2 times slower than the fixed point implementation but are 3 times faster than double floating point implementation in software. The timing details are given in table.

Number of input sequences	Implementation	Number of clock cycles	Time
4 * 4	CORDIC based fixed point[12]	555 (11.244ns clock)	6.240 us
4 * 4	VIVADO basic	26601	223.98 us
4 * 4	VIVADO alternate	7378	105.726 us
30 * 30	VIVADO basic	1655401	23.722 ms
30 * 30	VIVADO alternate	5542501	46.668 ms
30 * 30	Jacobi floating point [19]	–	94 ms

Table 6.1: Timing results take from [12], [19] and Vivado based implementation of SVD

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.42	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
641	6637121	642	6637122	dataflow

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	6	60	8337	13061
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	-	-
Total	6	60	8337	13061
Available	280	220	106400	53200
Utilization (%)	2	27	7	24

Figure 6.1: Vivado SVD Decomposition Basic implementation

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	14.33	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
1999027	2003987	1999028	2003988	dataflow

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	16	41	8118	11447
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	-	-
Total	16	41	8118	11447
Available	280	220	106400	53200
Utilization (%)	5	18	7	21

Figure 6.2: Vivado SVD Decomposition Alternate implementation

Block Type	No of Blocks
32-bit Adders/Subtractors	26
32-bit Adder Tree	32
Counters	13
32-bit Accumulator	8
Flip-flops	834
Comparators	9
32 bit shifter arithmetic right	20
FSM	2

Figure 6.3: CORDIC based SVD Decomposition 4 * 4 implementation

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.42	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
81	26601	82	26602	dataflow

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	6	60	7958	13119
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	-	-
Total	6	60	7958	13119
Available	280	220	106400	53200
Utilization (%)	2	27	7	24

Figure 6.4: VIVADO based SVD Decomposition 4 * 4 basic implementation

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	14.33	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
7318	7378	7319	7379	dataflow

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	14	41	7945	10960
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	-	-
Total	14	41	7945	10960
Available	280	220	106400	53200
Utilization (%)	5	18	7	20

Figure 6.5: VIVADO based SVD Decomposition 4 * 4 alternate implementation

Matrix Dimension	Floating-Point	Fixed-Point (FPGA)
10	2.0e-4	3.57064e-3
20	3.0e-3	4.28464e-3
30	94.0e-3	12.54604e-3
40	484e-3	26.86384e-3

Figure 6.6: Fixed Jacobi SVD Decomposition implementation Execution time (sec)

Resources	Used	Available
Number of Slices	2756	4656
Number of Slice Flip Flops	3924	9312
Number of 4 input LUTs	5283	9312
Number of Block RAMs	8	20
Number of MULT18×18SIOs	12	20
Number of GCLKs	2	24

Figure 6.7: Fixed Jacobi SVD Decomposition implementation Resource used

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	14.33	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
1651051	1655401	1651052	1655402	dataflow

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	12	41	8243	11581
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	-	-
Total	12	41	8243	11581
Available	280	220	106400	53200
Utilization (%)	4	18	7	21

Figure 6.8: VIVADO based SVD Decomposition 30 * 30 basic implementation

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.42	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
601	5542501	602	5542502	dataflow

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	6	60	8329	13272
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	-	-
Total	6	60	8329	13272
Available	280	220	106400	53200
Utilization (%)	2	27	7	24

Figure 6.9: VIVADO based SVD Decomposition 30 * 30 alternate implementation

CHAPTER 7

Image Resize implementation

This section details the implementation strategies for Image Resizing i.e, the original image or image patch is resized to a new dimension. Vivado HLS is used as a tool for this purpose. The input and output dimensions are dependent on initial target size. So the maximum dimensions are 240 * 320.

As described before the linear interpolation function[27] is,

$$\frac{x - A}{l} = \frac{B - A}{L}$$

$$x = A + l * \frac{B - A}{L}$$

this is used for finding a new color or gray scale intensity x. A bi-linear interpolation or INTERLINEAR interpolation is an extension of linear interpolation for 2-dimensional rectangular interpolation. The main idea is to do the linear interpolation first in one direction, and then in the other direction. Although each step on its on is linear, the interpolation as a whole is not a linear interpolation but a quadratic one.

As it can be seen it is very simple and not computation hungry. This is also implemented in VIVADO HLS library but it is not generic to the image size. An image resizing algorithm based on bilinear interpolation is depicted in [26].

7.0.1 Generic resizing

The one implementation available in vivado library is utilised and made generic. The only trick to use is instead of taking the second matrix dimensions as new size, it is passed as another parameter. Then this is used as a new size. Note: When reading image only the parameter size should be read not the whole matrix. The solution implemented and the results are depicted in 7.1. A test bench is designed to exercise it. The input image is shown 7.2. It has the dimensions of 240 * 320. It is resized to 200 * 200. The output image is shown 7.3. The black area represents empty values so should not be read.

For comparison of results the same image is used and simulated in Matlab and VIVADO HLS. The timing and PSNR values are measured. The same function is repeated 100 times in Matlab and than averaged to find the execution time which was 1.3 ms. This result was achieved with running Matlab on intel corei7 with 16 GB RAM. This is then compared with the maximum time of our algorithm which is 758 us. So it is much faster than software implementation. The same image is resized to dimensions of 8 * 6 in both matlab and VIVADO. The Matlab output is used as refrence and the PSNR was measured. It was 47.783. The output of matlab and VIVADO are shown 7.1 and 7.2 respectively.

186	207	202	152	155	152
193	209	211	191	185	106
192	203	206	182	155	64
187	198	185	153	152	88
179	192	169	124	141	66
171	171	135	139	55	33
116	128	59	75	24	22
67	80	42	47	21	21

Table 7.1: Resized image 8 * 6 gray scale values from Matlab

192	211	204	151	165	151
195	209	211	188	190	65
193	204	205	206	188	95
190	196	197	159	167	127
180	190	190	73	149	64
172	175	157	193	29	25
158	107	34	99	23	22
44	49	16	74	22	21

Table 7.2: Resized image 8 * 6 gray scale values from VIVADO HLS

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.68	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
50	87292	51	87293	dataflow

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	2	16	5450	5515
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	-	-
Total	2	16	5450	5515
Available	280	220	106400	53200
Utilization (%)	~0	7	5	10

Figure 7.1: Generic Image resizing bilinear interpolation based implementation

Figure 7.2: Input image for resizing 240 * 320

Figure 7.3: Resized output image 200 * 200

CHAPTER 8

Integral Image and FHOG implementation

8.1 Image Integral

This section details with the implementation strategies for image integral of an image. Vivado HLS is sued as a tool for this purpose. The integral image at location (x, y) contains the sum of the pixels above and to the left of (x, y).

$$ImageIntegral = \sum_{i=0}^{m}\sum_{j=0}^{n} I(i,j)$$

Where I is the image and I(i,j) represents the gray scale value of the pixel. It is dependant on target image initial size. The rows of input matrix are fixed at 32 while the columns are the product of the target dimensions. The output dimensions are input dimension + 1. So the maximum dimension of input is 33 * (height*width + 1) which mounts up to huge dimensions. Another important thing to note is that as the input is gray scale image, the output is the summation of pixel values where each pixel has the maximum value of 255. This mounts up to large values which than needs to be scaled down. It is implemented in vivado hls. The results are shown in 8.2 for 241 * 321 and in 8.1 for 33 * 1001. Which are good enough considering it is synthesized for 32 * 1000 dimensions. An implementation of this approach is used in [34].

The results of VIVADO implementation are compared with the paper [6]. The timing of paper [6] results are shown in 8.3. To achieve a fair comparison VIVADO based approach is implemented for similar dimensions. Results for 360 * 240 in 8.4 and for 720 * 576 8.5. Our results for the comparison are shown in table 8.6. Our results are comparable only to the serial approach. But in terms of resources used out approach use consumes less. This paper [6] gives insight towards exploring the parallelism of this approach. Our results can be improved by using a more parallel approach on row calculations but again keeping an eye on resources.

8.2 FHOG

This section details the implementation strategies for fhog feature extractions of an image patch. Vivado HLS issued as a tool for this purpose.

Before understanding FHOG some terms are necessary to be introduced.[29]

GRADIENT The gradient vector of an image is defined as for each individual pixel, containing the information of the pixel color changes in both x direction and y direction. The two main characteristics of an image gradient are its magnitude and direction.

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.54	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
6	33166	7	33167	dataflow

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	1	-	119	171
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	-	-
Total	1	0	119	171
Available	280	220	106400	53200
Utilization (%)	~0	0	~0	~0

Figure 8.1: Image integral implemented for 33 * 1001 dimensions

HISTOGRAMOFGRADIENTS(HOG) An efficient method for extracting the features of the pixel colors, for creating a target recognition classifier, is named the Histogram of Oriented Gradients (HOG). This is done with the help of image gradients.

Felzenszwalb's Algorithm Felzenszwalb suggested an algorithm for the purpose of dividing an image into similar areas, utilizing a graph-based strategy. Since it is needed to define an area that potentially contains an object since usually there are multiple targets in an image.

In the algorithm [18] the FHOG is implemented with the maximum dimensions of the output as 240 * 320 * 32. That is the initial target size times 32. So the output is three dimensional matrix.

For implementing this, the paper [20] suggests an approach. It is based on fixed point implementation. The gradient magnitude and orientation is computed using CORDIC's vector translate feature. The magnitude of gradient is assigned to suitable bins based on the orientation gradient. Then an aggregate module is utilized for adding the 64 pixels 64 values of histogram bin by bin, so that the the final cell features can be sent to output. In the end, cell features contrast are normalized block wise. Where each block contains four cells. The algorithm implemented by the authors [20] in FPGA has the speed of 526 fps. As a general idea the results from the paper are shown here for implementation in 8.7.

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.54	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
7	7037	8	7038	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	1	-	109	159
Memory	-	-	-	-
Multiplexer	-	-	-	3
Register	-	-	3	-
Total	1	0	112	162
Available	280	220	106400	53200
Utilization (%)	~0	0	~0	~0

Figure 8.2: Image integral implemented for 241 * 321 dimensions

Image Size	Execution Time in Milliseconds		
	Serial	2-Rows Parallel	4-Rows Parallel
360 × 240	0.587	0.293	0.146
720 × 576	2.821	1.410	0.705
800 × 640	3.482	1.741	0.870
1280 × 720	6.269	3.134	1.567
1920 × 1080	14.106	7.053	3.526
2048 × 1536	21.399	10.699	5.349
2048 × 2048	28.532	14.266	7.133

Figure 8.3: Image integral timing results from [6]

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.54	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
6	88446	7	88447	dataflow

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	1	-	120	170
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	-	-
Total	1	0	120	170
Available	280	220	106400	53200
Utilization (%)	~0	0	~0	~0

Figure 8.4: Image integral implemented for 360 * 240 dimensions

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.54	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
6	418902	7	418903	dataflow

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	1	-	131	185
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	-	-
Total	1	0	131	185
Available	280	220	106400	53200
Utilization (%)	~0	0	~0	~0

Figure 8.5: Image integral implemented for 720 * 576 dimensions

SIZE	Implementation	latency	Time (us)
360 * 240	Serial	–	587
720 * 576	Serial	–	2821
360 * 240	2-row parallel	–	293
720 * 576	2-row parallel	–	1410
360 * 240	4-row parallel	–	146
720 * 576	4-row parallel	–	705
360 * 240	VIVADO based	88446	755.33
720 * 576	VIVADO based	418902	3577

Figure 8.6: Integral image Timing values for VIVADO implementation

Design	Block memory Kbits	Logic (in ALMs)	DSP blocks	Registers	Fmax (MHz)
HOG extractor	324 (8%)	7,922 (25%)	65 (75%)	14,787	49
Optimized HOG extractor	326 (8%)	8,610 (27%)	74 (85%)	17,697	162
Heterogeneous system	437 (11%)	12,138 (38%)	65 (75%)	21,715	69

Figure 8.7: HOG Extractor implementation details

CHAPTER 9

Miscellanies blocks implementation

This chapter deals with the implementation of the minor blocks used in the algorithm [18].

9.1 Hann Window

This section deals with the implementation of the Hann window used in the algorithm. The Hann window generates Hann coefficients to be used later for windowing. The maximum dimensions depend on the initial target size. So the maximum possible window size is 320. It is a pretty simple block. It is calculated as

$$W_n = 0.5[1 - cos(\frac{2\pi n}{N})]$$

It uses cos function for computations. This is used from VIVADO HLS library which is based on cordic algorithm. The implementation results are shown here. It is implemented in two versions. Pipelined in figure 9.2 and without pipeline in 9.1. It is synthesized for 160 size. It can be seen that the results of pipelined version are better. Also the simulation is performed to verify the results from matlab. The matlab results are shown in 9.2. Our results are in 9.2. And the post synthesis performance is in 9.3. For comparison the paper [22] suggests to use free scale CORDIC to implement the function. They were able to achieve latency of 7 with only 16 bit input samples. The minimum latency of our approach is 225 out of which 182 cycles of them are occupied by cos function. By using scaling free CORDIC this can be improved further.

9.2 Windowing

This section deals with the implementation of the windowing function used in the algorithm. The windowing basically multiplies point-wise two matrices. The maximum dimensions depend on the initial target size. So the maximum possible window size is 240 * 320. It is a pretty simple block. It is calculated as element by element multiplication.

VIVADO based implementation results are shown here. It is implemented in two versions. In 9.4 for without pipeline and in 9.5 for pipelined. It is synthesized for 160 size. It can be seen that the results of pipelined version are better. Also the simulation is performed to verify the results. The latency can be seen as it is run for 1*10 matrix in 9.6. This simply involves the point-wise multiplication. It is simply a matter of running multi objective optimization. It suits well for Pareto analysis. Run 2 objective optimization to find the Pareto points for minimum latency and minimum number of resources. Lets see what is Pareto analysis.

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.71	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
323	18999	324	19000	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	730
FIFO	-	-	-	-
Instance	0	10	2290	6011
Memory	-	-	-	-
Multiplexer	-	-	-	18
Register	-	-	369	-
Total	0	10	2659	6759
Available	280	220	106400	53200
Utilization (%)	0	4	2	12

Figure 9.1: Hann window implementation

For a nontrivial optimization problem with multiple objectives, a single solution can not be found that simultaneously optimizes each objective. In this scenario, the objective are conflicting, and a number of Pareto optimal solution exists (possibly infinite). If none of the objective can be optimized without degrading some of the other than such a solution is called Pareto optimal, nondominated, Pareto efficient or noninferior. All of the Pareto optimal solutions are considered equally well. Than the question remains to quantify a set of Pareto optimal solutions, in achieving the different goals, or a solution that satisfies the subjective preferences of a human decision maker.

Now back to our main topic. This analysis can be used and a Pareto point is chosen to reach our goals. The Pareto in this case will be done with maximum size of 160.

9.3 Circular shifting

This section deals with the implementation of the circular shifting function used in the algorithm. This basically shifts the elements of a vector from start to the end. The maximum dimensions depend on the initial target size. So the maximum possible size is 320. It is a pretty simple block.

The implementation results are shown here. It is implemented in two versions. Pipelined in 9.8 and without pipeline in 9.7. It is synthesized for 160 size. As can be seen from the result piplining makes no difference and latency will be equal to the array dimensions. A solution can be in using barrel shifter.

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	10.79	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
225	225	226	226	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	730
FIFO	-	-	-	-
Instance	-	12	3451	4729
Memory	-	-	-	-
Multiplexer	-	-	-	10
Register	-	-	383	19
Total	0	12	3834	5488
Available	280	220	106400	53200
Utilization (%)	0	5	3	10

Figure 9.2: Hann window piplined implementation

9.4 Reshaping2

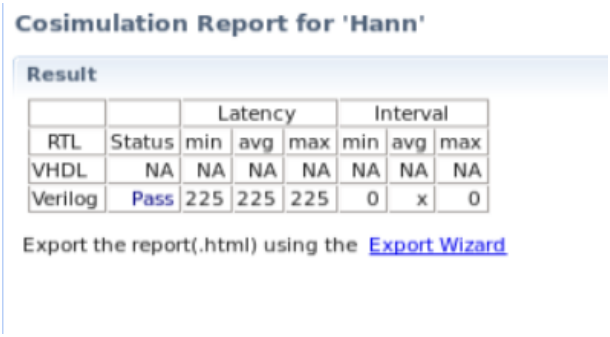
This section deals with the implementation of the reshaping function used in the algorithm. This basically copies the elements of a 3D matrix [R][C][S] to a 2D matrix with dimensions [R*C][S] from start to the end. The maximum dimensions depend on the initial target size. So the maximum possible window size 60 * 80 * 32 and for output (60 * 80) * 32. It is a pretty simple block.

The implementation results are shown in figure 9.9 for 60 * 80 * 32 and in 9.10 for 40 * 40 * 32. It is implemented in two versions. Pipelined in 9.11 and without pipeline. It is synthesized for 160 size that is 40 * 40 * 32. It can be seen that the results of pipelined version are better. Also the simulation is performed to verify the results.

This can be simply optimized by completely unrolling the loops. That is just using wires to connect the input matrix elements to respective outputs. But the whole matrix can not be read at once. A SDRAM memory can be used for increasing memory bandwidth.

9.5 Reshaping

This section deals with the implementation of the reshaping function used in the algorithm. This basically copies the elements of a 2D matrix [R][C] to a 1D matrix with dimensions [R*C][1] from start to the end. It deals with the reshaping of scale matrices. The maximum dimensions depend on the initial target size. But the maximum possible window size scaled model size and for output (scale model size) * 31. It is a pretty simple block. It is run 17 times and each time it just copies one 2D matrix into a column vector. After all the iterations it creates a 2D matrix.



Cosimulation Report for 'Hann'

Result

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	225	225	225	0	x	0

Export the report(.html) using the Export Wizard

Figure 9.3: Hann window VIVADO Post synthesis results for N=10

The implementation results are shown. It is implemented in two versions. Pipelined in 9.13 and without pipeline in 9.21. It can be seen that the results of pipelined version are better.

For optimization see the previous section.

9.6 Resize DFT2

This section deals with the implementation of the resizing function used in the algorithm. This basically copies the scaled elements of a original matrix [R][C] to a new matrix with dimensions [R*4][C*4] from start to the end while remaining entries are zero filled. The maximum dimensions depend on the initial target size. So the maximum possible window size (4*240) *(4*320). It is not a simple block.

The implementation results are shown. It is implemented in two versions. Pipelined in 9.15 and without pipeline in 9.14. It is synthesized for (4* 160) and (4 *160) size. It can be seen that the results of pipelined version are better. The latency can be seen as it is run for 32 * 32 for non pipelined 9.16 and for pipelined 9.17.

For optimization the solution strategies from Windowning and reshaping sections can be utilized. The important thing to note here is that the scaling is always by 16. Which is a power of 2. So it is equivalent to a shift left by 4. Which will save a huge amount of resources.

9.7 Resize DFT

This section deals with the implementation of the resizing function used in the algorithm. This basically copies the scaled elements of a original vector [R] to a new vector with dimensions [2*R] from start to the end while remaining entries are zero filled. The maximum dimensions depend on the scale interps which is fixed constant of the algorihm. So the maximum possible input size is 17 and output size is 33. It is a simple block.

The implementation results are shown. It is implemented in two versions. Pipelined in 9.19 and without pipelining in 9.18. It is synthesized for 160 size. It can be seen that the results of pipelined version are better. Also the simulation is performed to verify the results. The latency can be seen as it is run for vector size 9.20. See the optimization strategies for the previous block.

9.8 Comparison of Minor Blocks

The latency and implementation details of minor blocks are summarized in this section. They are given in the form of table in table 9.1. The pipelined also means loop unrolling of factor 2 involved.

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	6.08	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
230721	230721	230722	230722	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	54
FIFO	-	-	-	-
Instance	-	4	0	0
Memory	-	-	-	-
Multiplexer	-	-	-	22
Register	-	-	166	-
Total	0	4	166	76
Available	280	220	106400	53200
Utilization (%)	0	1	~0	~0

Figure 9.4: Window function implementation of size 160

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	6.63	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
12810	12810	12811	12811	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	110
FIFO	-	-	-	-
Instance	-	8	0	0
Memory	-	-	-	-
Multiplexer	-	-	-	40
Register	-	-	283	32
Total	0	8	283	182
Available	280	220	106400	53200
Utilization (%)	0	3	~0	~0

Figure 9.5: Window function pipelined implementation of size 160

Cosimulation Report for 'windowing'

Result

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	12810	12810	12810	0	x	0

Export the report(.html) using the Export Wizard

Figure 9.6: Window function post synthesis latency for N=10

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	5.15	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
161	161	162	162	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	100
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	81
Register	-	-	42	-
Total	0	0	42	181
Available	280	220	106400	53200
Utilization (%)	0	0	~0	~0

Figure 9.7: Circular shifting implementation for size 160

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	5.15	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
162	162	163	163	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	100
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	81
Register	-	-	43	-
Total	0	0	43	181
Available	280	220	106400	53200
Utilization (%)	0	0	~0	~0

Figure 9.8: Circular shifting piplined implementation for size 160

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	6.38	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
213885	213885	213886	213886	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	1	-	-
Expression	-	-	0	128
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	58
Register	-	-	154	-
Total	0	1	154	186
Available	280	220	106400	53200
Utilization (%)	0	~0	~0	~0

Figure 9.9: Reshaping a 3D matrix to 2D implementation for 60 * 80 * 32

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	5.42	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
105025	105025	105026	105026	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	136
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	52
Register	-	-	142	-
Total	0	0	142	188
Available	280	220	106400	53200
Utilization (%)	0	0	~0	~0

Figure 9.10: Reshaping a 3D matrix to 2D implementation for 40 * 40 * 32

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.53	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
25603	25603	25604	25604	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	309
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	91
Register	-	-	143	-
Total	0	0	143	400
Available	280	220	106400	53200
Utilization (%)	0	0	~0	~0

Figure 9.11: Reshaping a 3D matrix to 2D pipelined implementation for 40 * 40 * 32

Dimensions for implementation	Implemented Block	Latency type	Pipeline	Number of clock cycles (Latency)
160	Hann window	min	–	323
160	Hann window	max	–	18999
160	Hann window	min	Yes	225
160	Hann window	max	Yes	225
160	Window function	min	–	230721
160	Window function	max	–	230721
160	Window function	min	Yes	12810
160	Window function	max	Yes	12810
160	Circular Shifter	min	–	161
160	Circular Shifter	max	–	161
160	Circular Shifter	min	Yes	162
160	Circular Shifter	max	Yes	162
60 * 80 * 32	Reshaping (3D)	min	–	213885
60 * 80 * 32	Reshaping (3D)	max	–	213885
40 * 40 * 32	Reshaping (3D)	min	–	105025
40 * 40 * 32	Reshaping (3D)	max	–	105025
40 * 40 * 32	Reshaping (3D)	min	Yes	25603
40 * 40 * 32	Reshaping (3D)	max	Yes	25603
40 * 40	Reshaping (2D)	min	–	3281
40 * 40	Reshaping (2D)	max	–	3281
40 * 40	Reshaping (2D)	min	Yes	803
40 * 40	Reshaping (2D)	max	Yes	803
160 - 640	Resizing DFT2	min	–	430486
160 - 640	Resizing DFT2	max	–	430486
160 - 640	Resizing DFT2	min	Yes	211686
160 - 640	Resizing DFT2	max	Yes	211686
17 - 33	Resizing DFT	min	–	67
17 - 33	Resizing DFT	max	–	166
17 - 33	Resizing DFT	min	Yes	23
17 - 33	Resizing DFT	max	Yes	23

Table 9.1: Latency of VIVADO based implementation of minor blocks

S.N0.	Hann coefficient
1	0
2	0.1170
3	0.4132
4	0.7500
5	0.9698
6	0.9698
7	0.7500
8	0.4132
9	0.1170
10	0

Table 9.2: Hann window matlab results for N = 10

S.N0.	Hann coefficient
0	0.000000
1	0.116978
2	0.413176
3	0.750000
4	0.969846
5	0.969847
6	0.750000
7	0.413176
8	0.116978
9	0.000000

Table 9.3: Hann window vivado results for N = 10

S.N0.	A	B	C = A.*B
0	0.000000	0.000000	0.000000
1	1.000000	1.000000	1.000000
2	2.000000	2.000000	4.000000
3	3.000000	3.000000	9.000000
4	4.000000	4.000000	16.000000
5	5.000000	5.000000	25.000000
6	6.000000	6.000000	36.000000
7	7.000000	7.000000	49.000000
8	8.000000	8.000000	64.000000
9	9.000000	9.000000	81.000000

Table 9.4: Window function results for N = 10

3D Matrix [0]	Col 0	Col 1	Col 2	Col 3
Row 0	0.110000	1.650000	0.560000	0.006000
Row 1	0.453000	0.1760000	1.4300000	0.120000
3D Matrix [1]	Col 0	Col 1	Col 2	Col 3
Row 0	0.567400	1.287000	0.261000	0.1136000
Row 1	0.953000	0.1600000	1.4310000	0.123000

2D Matrix	Col 0	Col 1
Row 0	0.110000	0.567400
Row 1	1.650000	1.287000
Row 2	0.560000	0.261000
Row 3	0.006000	0.1136000
Row 4	0.453000	0.953000
Row 5	0.1760000	0.1600000
Row 6	1.4300000	1.4310000
Row 7	0.120000	0.123000

Table 9.5: Simulation results Reshaping Matrix 3 D for 2 * 4 * 2 size to Matrix 2D 8 * 2

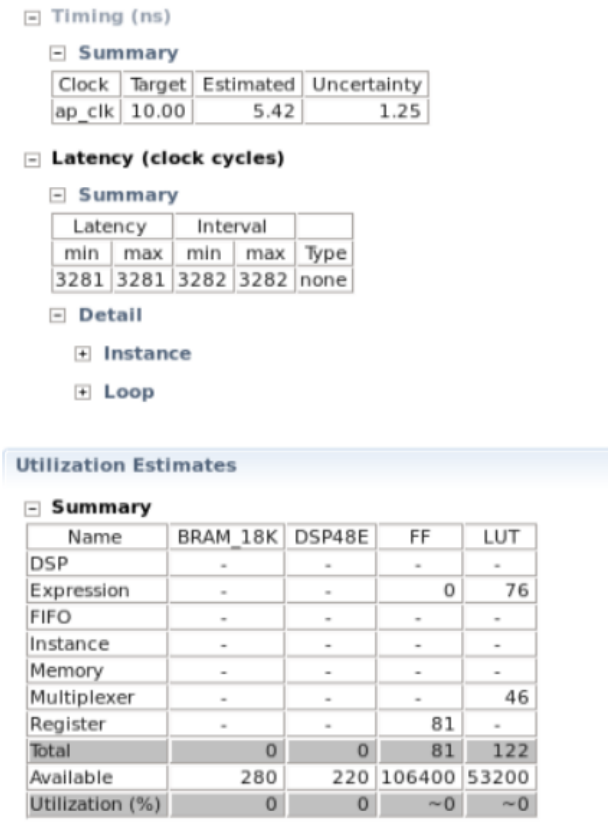
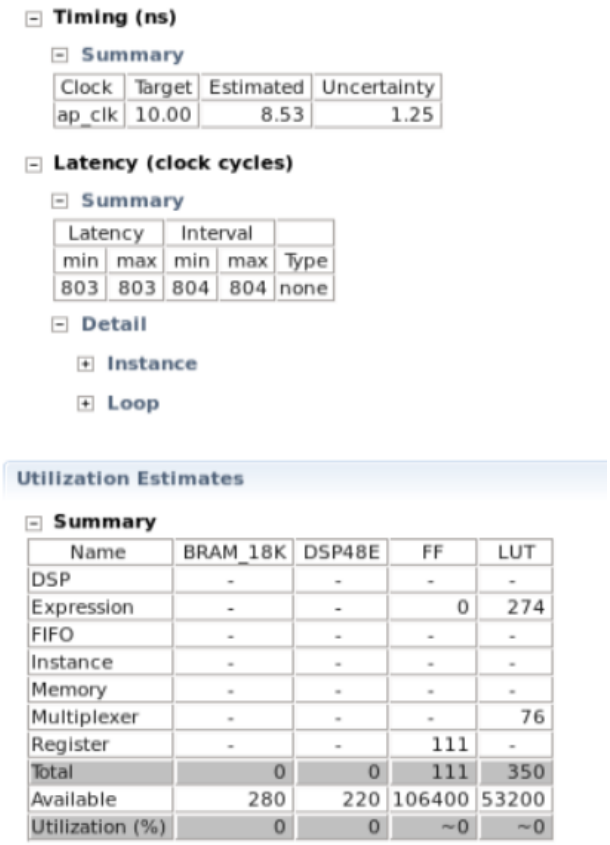


Figure 9.12: Reshaping of a 2D matrix to 1D column vector implementation



Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.53	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
803	803	804	804	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	274
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	76
Register	-	-	111	-
Total	0	0	111	350
Available	280	220	106400	53200
Utilization (%)	0	0	~0	~0

Figure 9.13: Reshaping of a 2D matrix to 1D column vector pipelined implementation

Input Vector having size of 1 * 17 to the DFT resizing																
0.0	1.0	2.0	3.0	4.0	5.0	6.0	7.0	8.0	9.0	10.0	11.0	12.0	13.0	14.0	15.0	16.0

Outpur Vector having size of 1 * 33 of the DFT resizing and scaling									
0.0	1.941176	3.882353	5.823529	7.764706	9.705882	11.647058	13.588235	15.529411	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	19.411764	21.352940	23.294117	25.235292
27.176470	29.117645	31.058823							

Table 9.6: Simulation results for Resizing DFT from 1 * 17 size to vector 1 * 33 and scaling

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.21	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
430486	440086	430487	440087	none

Detail

Instance

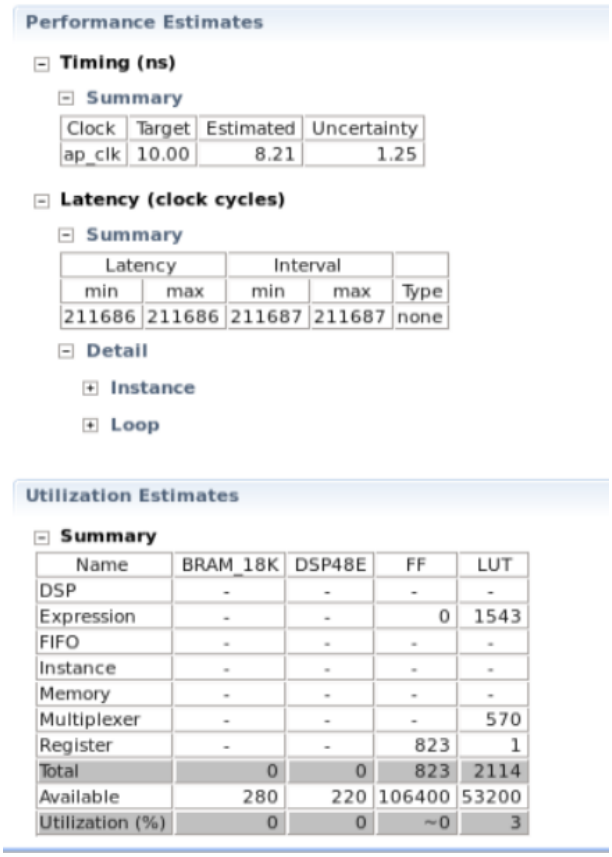
Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	1112
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	363
Register	-	-	665	-
Total	0	0	665	1475
Available	280	220	106400	53200
Utilization (%)	0	0	~0	2

Figure 9.14: Resizing DFT2 of a 2D matrix to 4 times its size implementation



Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.21	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
211686	211686	211687	211687	none

Detail

Instance

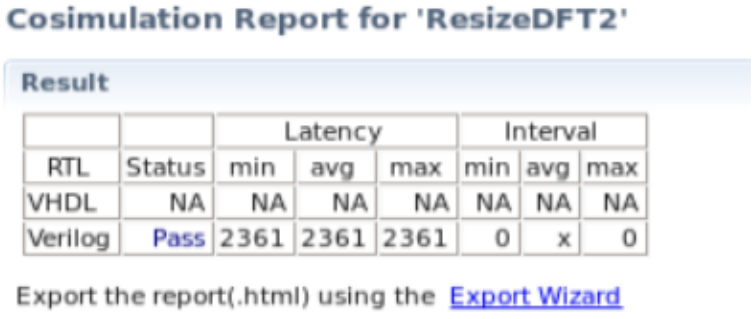
Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	1543
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	570
Register	-	-	823	1
Total	0	0	823	2114
Available	280	220	106400	53200
Utilization (%)	0	0	~0	3

Figure 9.15: Resizing DFT2 of a 2D matrix to 4 times its size pipelined implementation



Cosimulation Report for 'ResizeDFT2'

Result

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	2361	2361	2361	0	x	0

Export the report(.html) using the Export Wizard

Figure 9.16: Post synthesis timing from 32 * 32 Resizing DFT2

Cosimulation Report for 'ResizeDFT2'

Result

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	2317	2317	2317	0	x	0

Export the report(.html) using the Export Wizard

Figure 9.17: Post synthesis timing from 32 * 32 Resizing DFT2 pipelined

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.47	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
67	166	68	167	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	23
FIFO	-	-	-	-
Instance	-	8	0	0
Memory	-	-	-	-
Multiplexer	-	-	-	96
Register	-	-	163	-
Total	0	8	163	119
Available	280	220	106400	53200
Utilization (%)	0	3	~0	~0

Figure 9.18: Resizing DFT of a 1*17 vector to 1 * 33 implementation

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.47	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
23	23	24	24	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	157
FIFO	-	-	-	-
Instance	-	12	0	0
Memory	-	-	-	-
Multiplexer	-	-	-	197
Register	-	-	330	98
Total	0	12	330	452
Available	280	220	106400	53200
Utilization (%)	0	5	~0	~0

Figure 9.19: Resizing DFT of a 1*17 vector to 1 * 33 pipelined implementation

Cosimulation Report for 'ResizeDFT2_scale'

Result

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	22	22	22	0	x	0

Export the report(.html) using the Export Wizard

Figure 9.20: Post synthesis timing of Resizing DFT

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	5.42	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
3281	3281	3282	3282	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	76
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	46
Register	-	-	81	-
Total	0	0	81	122
Available	280	220	106400	53200
Utilization (%)	0	0	~0	~0

Figure 9.21: Simulation results for Resizing DFTn

CHAPTER 10

Conclusions

Real-time on field scale estimation of a target with accuracy is a research problem in visual tracking. It is important due to its applications in domains like surveillance, robotics and automation. In general, only the initial position of the object is known, and the trajectory of the object motion is desired to be traced. It becomes further difficult due to problems like fast motion of the objects, motion blur, size and scale variations. Existing algorithms estimate the size of target based on exhaustive scale search hence computationally expensive.

To tackle these problems, we used an algorithm [18] which is based on online explicit filtering based on target sampling at different scales. This thesis aims at FPGA or hardware based implementation of fast discriminative scale space tracking algorithm. The hardware created on FPGA will be fast and use resources to a minimum to decrease the cost.

For this the major mathematical blocks in this algorithm are studied and their best implementation approach for this algorithm is described. The best implementation approach is given in terms of the complexity and Dimensions of inputs involve, less area and fast operation. The whole algorithm is also depicted and the step by step operations involved, to better understand the decisions. Also, the synthesized version of Discrete Fourier Transform hardware is depicted. It is developed in different versions to detail their features. It include one dimensional Discrete Fourier Transform and inverse Discrete Fourier Transform. DFT2 which involves transposes and three dimensional DFT. Their timing behaviour is compared in the end.

VIVADO HLS tool is used for this purpose. It is the high level synthesis version of the VIVADO environment. The language used is synthesise able C++. The target code can be generated in both of the hardware descriptive languages i.e, verilog and VHDL. It is final targeted to be implemented in Xilinx FPGA Zed board. All codes are for this target FPGA board. Along with this VIVADO can be used for using brams and other interfaces.

Also the major blocks used in the algorithm are given study and their implementation details are discussed. They include QR factorization of a matrix, Singular value Decomposition of a matrix, Image resizing, Resizing a matrix, Image integral, FHOG. Along with that also major blocks some minor blocks are studied and implemented as well. They include Hann window, Windowing application, Resizing a vector, Circular shifting, reshaping a 3 dimensional matrix, reshaping a 2 dimensional matrix.

In the end some suggestions are suggested. In the block diagram it was described all the steps. The steps before the Translation search are only to be done for the first frame. It should be once performed than stored in the algorithm. This is to save the computational resources. Also as in visual object tracking the target size is usually smaller than the whole frame of image, so it will be more efficient and economical in terms of used resources. It is suggested to keep the area of target image at max half of the frame size for good performance.

For future works, the described methods for remaining blocks can be implemented in hardware to develop the whole working algorithm. Other implementation for major blocks can be utilised for different image sizes to incorporate them as well. Also a power consumption based analysis of the algorithm can be performed.

Bibliography

[1] A. Geiger, M. Lauer, C. Wojek, C. Stiller, and R. Urtasun, "3D traffic scene understanding from movable platforms,," in: *IEEE Trans. Pattern Anal. Mach. Intell.,* 36.5 (2014), pp. 1012–1025.

[2] Chi-Li Yu, Student Member, IEEE, Kevin Irick, Chaitali Chakrabarti, Senior Member, IEEE, and Vijaykrishnan Narayanan, Senior Member, IEEE. ""Multidimensional DFT IP Generator for FPGA Platforms"". In: *"IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS—I: REGULAR PAPERS "* ().

[3] D. S. Bolme, J. R. Beveridge, B. A. Draper, and Y. M. Lui, "Visual object tracking using adaptive correlation filters," in: *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.,* (2010), pp. 2544–2550.

[4] Debaprasad De, K. Gaurav Kumar, Archisman Ghosh, Anurup Saha, "FPGA Implementation of Discrete Fourier Transform Using CORDIC Algorithm". In: *AMSE JOURNALS-AMSE IIETA publication-2017-Series* 60.2 (2017), pp. 332–337.

[5] P. Dollar. *Piotr's Image and Video Matlab Toolbox (PMT),* URL: `http://vision.ucsd.edu/pdollar/toolbox/doc/index.html`.

[6] Shoaib Ehsan et al. "Integral Images: Efficient Algorithms for Their Computation and Storage in Resource-Constrained Embedded Vision Systems". In: *Sensors* 15 (July 2015), pp. 16804–16830. DOI: `10.3390/s150716804`.

[7] H. Galoogahi, T. Sim, and S. Lucey, "Multi-channel correlation filters," in: *Proc. IEEE Int. Conf. Comput. Vis.,* (2013), pp. 3072–3079.

[8] J. Henriques, J. Carreira, R. Caseiro, and J. Batista, "Beyond hard negative mining: Efficient detector learning via block-circulant decomposition," in: *Proc. IEEE Int. Conf. Comput. Vis.,* (2013), pp. 2760–2767.

[9] J. Henriques, R. Caseiro, P. Martins, and J. Batista, "Exploiting the circulant structure of tracking-by-detection with kernels," in: *Proc. 12th Eur. Conf. Comput. Vis.,* (2012), pp. 702–715.

[10] J. Henriques, R. Caseiro, P. Martins, and J. Batista, "High-speed tracking with kernelized correlation filters," in: *IEEE Trans. Pattern Anal. Mach. Intell.,* 37.3 (2015), pp. 583–596.

[11] J. Prokaj and G. Medioni, "Persistent tracking for wide area aerial surveillance," in: *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.,* (2014), pp. 1186–1193.

[12] Karthikeyan N, Sathyanarayanan S, Vamsi Krishna S, Shankar Balachandran. "FPGA implementation of Singular Value Decomposition," in: *IEEE Trans. Pattern Anal. Mach. Intell.,* 32.9 (2015), pp. 1627–1645.

[13] L. Sevilla-Lara, and E. G. Learned-Miller, "Distribution fields for tracking," in: *in Proc. IEEE Conf. Comput. Vis. Pattern Recognit.,* (2012), pp. 1910–1917.

[14] M. Danelljan, et al., "A low-level active vision framework for collaborative unmanned aircraft systems," in: *Proc. Eur. Conf. Comput. Vis. Workshop Comput. Vis. Veh. Technol. with Special Session Micro Aerial Veh.,* (2014), pp. 223–237.

[15] M. Danelljan, F. S. Khan, M. Felsberg, and J. van de Weijer, "Adaptive color attributes for real-time visual tracking," in: *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.,* (2014), pp. 1090–1097.

[16] M. Danelljan, G. Hager, F. S. Khan, and M. Felsberg, "Accurate scale estimation for robust visual tracking," in: *Proc. Brit. Mach. Vis. Conf.,* (2014), pp. 1–11.

[17] M. Kristan, R. Pflugfelder, A. Leonardis, and J. Matas, "The visual object tracking VOT2014 challenge results," in: *Proc. Eur. Conf. Comput. Vis. Workshop Visual Object Tracking Challenge,* (2014), pp. 191–217.

[18] Martin Danelljan, Gustav Hager, Fahad Shahbaz Khan, and Michael Felsberg. "Discriminative Scale Space Tracking". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2016).

[19] Ramanarayan Mohanty et al. "Design and Performance Analysis of Fixed-point Jacobi SVD Algorithm on Reconfigurable System". In: *IERI Procedia* 7 (Dec. 2014), pp. 21–27. DOI: 10.1016/j.ieri.2014.08.005.

[20] Vinh Ngo et al. "A High-Performance HOG Extractor on FPGA". In: (Jan. 2018).

[21] P. F. Felzenszwalb, R. B. Girshick, D. A. McAllester, and D. Ramanan, "Object detection with discriminatively trained part-based models," in: *IEEE Trans. Pattern Anal. Mach. Intell.,* 32.9 (2010), pp. 1627–1645.

[22] Rama Murali Krishna.P.G Rashmi.K.L. "FPGA Implementation of Efficient WindowArchitecture Design Using Completely Scaling – Free CORDIC Algorithm". In: *"IInternational Journal for Research in Applied Science Engineering Technology (IJRASET)* 3 (May 2015).

[23] S. Hare, A. Saffari, and P. Torr, "Struck: Structured output tracking with kernels," in: *Proc. Int. Conf. Comput. Vis.,* (2011), pp. 263–270.

[24] S. He, Q. Yang, R. Lau, J. Wang, and M.-H. Yang, "Visual tracking via locality sensitive histograms," in: *Proc. IEEE Conf. Comput. Vis.Pattern Recognit.,* (2013), pp. 2427–2434.

[25] Sergio D. Mu˜noz and Javier Hormigo. "High-Throughput FPGA Implementation of QR Decomposition". In: *IEEE TRANS. ON CIRCUITS AND SYSTEMS–II,* 62.9 (2015).

[26] Shraddha Mali. "Image Scaling Algorithm for Multimedia Applications". In: *International Journal of Science and Research (IJSR)* 32.9 (2010), pp. 1627–1645.

[27] Tech-Algorithm.com. *Bilinear Image Scaling,* URL: http://tech-algorithm.com/articles/bilinear-image-scaling/.

[28] V. N. Boddeti, T. Kanade, and B. V. K. V. Kumar, "Correlation filters for object alignment," in: *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.,* (2013), pp. 2291–2298.

[29] Lilian Weng. *Object Detection for Dummies Part 1: Gradient Vector, HOG, and SS.* URL: https://lilianweng.github.io/lil-log/2017/10/29/object-recognition-for-dummies-part-1.html. (accessed: 04.09.2019).

[30] X. Jia, H. Lu, and M.-H. Yang. "Visual tracking via adaptive structural local sparse appearance model," in: *Proc. IEEE Conf. Comput.Vis. Pattern Recognit.* (2012), pp. 1822–1829.

[31] Y. Li and J. Zhu, "A scale adaptive kernel correlation filter tracker with feature integration," in: *Proc. Eur. Conf. Comput. Vis. Workshop Visual Object Tracking Challenge,* (2014), pp. 254–265.

[32] Y. Wu, J. Lim, and M.-H. Yang, "Online object tracking: A benchmark," in: *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.,* (2013), pp. 2411–2418.

[33] Yun Zhe Cheong1, and Wei Jen Chew1. "The Application of Image Processing to Solve Occlusion Issue in Object Tracking". In: *School of Engineering, Taylor's University, 47500 Subang Jaya, Selangor, Malaysia* ().

[34] Jin Zhao. *Video/Image Processing on FPGA*. 2015.